Reference Guide

MatchUp® Object 3

32/64 BIT

Multiplatform

MELISSA DATA®

MatchUp Object

Reference Guide

Melissa Data Corporation

Melissa Data Corporation

22382 Avenida Empresa
Rancho Santa Margarita, CA 92688-2112

Phone: 1-800-MELISSA (1-800-635-4772)
Fax: 949-589-5211

E-mail: info@MelissaData.com
Internet: www.MelissaData.com

For the most recent version of this document, visit
http://www.melissadata.com/tech/matchup-object.htm

Document Code: DQTAPIMURG
Revision Number: 29042015.15

Dear Developer,

I would like to take this opportunity to introduce you to Melissa Data Corp. Founded in 1985, Melissa Data provides data quality solutions with emphasis on address and phone verification, postal encoding, and data enhancements.

We are a leading provider of cost-effective solutions for achieving the highest level of data quality for lifetime value. A powerful line of software, databases, components, and services afford our customers the flexibility to cleanse and update contact information using almost any language, platform, and media for point-of-entry or batch processing.

This manual will guide you through the properties and methods of our easy-to-use programming tools. Your feedback is important to me, so please don't hesitate to email your comments or suggestions to Ray@MelissaData.com.

I look forward to hearing from you.

Best Wishes,

[illegible]

Raymond F. Melissa

President/CEO

Table of Contents

Introduction to MatchUp Object 1

MatchUp Object Interfaces 2

Key Concepts 3

Entering Your MatchUp Object License 4

MatchCode List Interface 5

Matchcode List Interface Functions 5

Matchcodes and the Matchcode Editor 8

Matchcodes 8

Component Combinations 14

Blank Field Matching 16

Matchcode Mapping 19

Optimizing Matchcodes 23

Other Uses for Swap Matching 28

Using MatchUp Object with Non-U.S. Addresses 29

The Matchcode Editor 29

Starting the Matchcode Editor 29

The Matchcode Editor Interface 30

Read/Write Deduping 37

Read/Write Order of Operations 38

Read/Write Deduping Functions 42

Initialize the Read/Write Interface 44

Map Database Fields 51
Read Data and Build the Match Key 53
Process Records 56
Retrieve Dupe Data for Each Record 56

Incremental Deduping 62

Incremental Order of Operations 63
Using the Transaction Functions 66
Incremental Deduping Functions 67
Initialize the Incremental Interface 69
Map Database Fields 76
Read Data and Build the Match Key 78
Compare Record to Database 80
Add New Record to Key File 85
Transaction Methods 85

Hybrid Deduping 89

Clustering 89
Key Maintenance 89
Hybrid Order of Operations 90
Hybrid Deduping Functions 93
Initialize the Hybrid Interface 94
Map Database Fields and Build Keys 101
Build the Match Keys 102
Compare Records 104

Matchcode Interface 108

Order of Operations for Creating Matchcodes 108
Order of Operations for Reading Matchcodes 111

Matchcode Mapping Information 115
Matchcode Interface Functions 116
Initialize MatchUp Object 118
Create a New Matchcode 120
Retrieve Existing Matchcodes 121
General Matchcode Properties 122
Read Matchcode Component Information 123
Get Mapping Information 124
Change Matchcode Component Settings 126
Read Matchcode Component Settings 134
Add, Modify or Delete Matchcode Components 142
Save Changes to the Matchcode File 145

Appendix 147

Enumerations & International Deduping 147
Enumerations 147
International Deduping Considerations 151

Introduction to MatchUp Object

MatchUp Object is an extremely fast and powerful programmer's tool that can be integrated into custom applications to eliminate duplicate records. Because merge/purge and data quality initiatives go hand in hand, the powerful features of this tool fulfill the needs of many companies. Reducing printing costs, increasing response rates, maintaining an efficient database, and achieving better quality data are just some of the many benefits of the merge/purge process.

MatchUp Object allows developers to customize exactly how to merge and purge data to suit their business needs. This gives people the flexibility to integrate MatchUp at different points of their processes, from point of entry to batch processing on the back end.

MatchUp Object can find matches in any combination of over 35 different components — from common ones like address, city, state, ZIP™, name, and phone — to less-common elements, such as email address, company, gender, and social security number. Developers can even specify their own custom components. Each set of rules for matching is referred to as a matchcode, and a matchcode can apply up to 16 rules at a time. These rules are specified as combinations of components. A commonly used combination would be {Last Name + Street # + Street Name + ZIP Code™}, while another combination in the same matchcode would substitute PO Box™ for Street # and Street Name. With these options, the number of potential matching rules is limitless.

MatchUp Object is a very sophisticated tool. If record 1 and 3 match with combination #1 in a matchcode, and record 2 and 3 match using combination #2- MatchUp Object will use inferred matching and put records 1, 2, and 3 into the same group. It can split address, city/state/ZIP and name fields on the fly, as well as recognize phonemes like "ph" and "sh," nicknames (Liz, Beth, Betty, Elizabeth), and alternate spellings of names (Gene, Jean, Jeanne).

MatchUp Object can also handle nearly-exact strings of characters, such as "Lewis" vs. "Ewis," and "Palacino" vs. "Al Pacino" as well as initials such as "John Smith" to "J Smith." These are just a few examples of the powerful matching algorithms at your disposal: Exact Match; Phonetic; Soundex; Containment; Frequency; Frequency Near; Fast Near; Accurate Near; Vowels Only; Consonants Only; Alphas Only; and Numerics Only.

Speed also is an important feature of MatchUp Object. It can process an average of 10 to 50 million records per hour. MatchUp Object includes a 64-bit version to take advantage of newer processors and operating systems. The COM and .NET version of MatchUp Object eases integration with Microsoft languages.

MatchUp Object Features & Benefits

- Fast processing, about 10-50 million records per hour
- Extremely flexible and customizable
- 22 powerful matching algorithms
- Split name, address, and city/state/ZIP fields on the fly
- Easy to learn and use
- Sample Code provided in C#, VB.NET, C++, FoxPro, Java, SQL Server
- Free tech support

MatchUp Object Interfaces

This API provides the developer with three different interfaces, or deduping methods, allowing for maximum flexibility in selecting the best method for the application.

The Read/Write Deduper

The Read/Write Deduper checks an entire list of records against itself, flagging unique records and duplicates. This deduper is best for checking existing databases and purging duplicate data.

Detailed information on using the Read/Write deduper begins on page 37.

The Incremental Deduper

The incremental deduper checks a single record against a persistent historical key file. This enables an application to field incoming records, such as from a website or a customer service system, easily detect duplicate records, pointing the application to the original unique record.

Detailed information on using the Incremental deduper begins on page 62.

The Hybrid Deduper

The Hybrid Deduper allows greater flexibility in customizing the deduping process. This is done by turning the storage of the matchkeys to the developer. Through this you have the ability to select clusters of records to compare against the incoming record.

Detailed information on using the Hybrid deduper begins on page 89.

Key Concepts

The following concepts are essential to understanding the logic behind how MatchUp Object functions and successfully integrating the product into applications.

Match Keys

Match Keys are string tokens that represent a database record. They contain only enough information necessary to determine a record's unique or duplicate status.

Because they only contain a reduced portion of the data in the actual record, MatchUp Object is able to use these keys more efficiently than if it had to compare the complete record against every other record in the database.

Clustering

Once a matchcode key is generated for a given record, it can be compared to the keys of other records. Ideally, every record's key would be compared to every other record's key. This, however, is not practical in all but very trivial applications because the number of comparisons grows geometrically with the number of records processed. For example, a record set of 100 records requires 4,950 comparisons (99 + 98 +...). A larger set of 10,000 records requires 49,995,000 comparisons (9,999 + 9,998 +...). Large record sets would take prohibitive amounts of time to process.

So, the developers of MatchUp Object made the assumption that in order for two matchcode keys to be considered matching, there must be something in the keys that must match exactly. In many cases, this will be all or part of the ZIP/Postal Code. So what MatchUp Object does is only compare records that are (in this example) in the same ZIP or Postal Code. On the average (in the US using 5-digit ZIP codes), this will cut the average number of comparisons per record by a factor of thousands.

This concept is known as "break grouping," "clustering," "partitioning," or "neighborhood sorting." It is very likely that most, if not all other deduping programs have used some form of clustering method.

Here is an example set of matchcode keys using ZIP/Postal Code (5 characters), Last Name(4), First Name(2), Street Number(3), Street Name(5):

```
02346BERNMA49 GARD

02346BERNMA49 GARD

02357STARBR18 DAME

02357MILLLI123MAIN

03212STARMA18 DAME
```

When the deduping engine encounters this set of matchcode keys, it compares all the keys in “02346” (2 keys), then “02357” (2 keys), and finally “03212” (1 key). For this small set, 10 comparisons are turned into 2.

In reality, MatchUp Object’s clustering engine is a bit more complicated than this, but this description will aid in understanding its mechanics.

The second deduping engine removes the first component restrictions, allowing the user to create matching strategies with rule sets completely independent of each other. This eliminates having to run multiple passes, as was the case with previous versions.

Matchcodes

Matchcodes are sets of rules that MatchUp Object uses to determine how match keys are constructed and how much of the key is used for clustering.

A detailed explanation of matchcodes, as well as how to create and edit them using the Matchcode Editor application, begins on page 8.

Entering Your MatchUp Object License

The license string is a software key that unlocks the functionality of the component. Without this key, the object does not function. You set the license string using an environment variable called MD_LICENSE. If you are just trying out MatchUp Object and have a demo license, you can use the environment variable MD_LICENSE_DEMO for this purpose. This avoids conflicts or confusion if you already have active subscriptions to other Melissa Data object products.

In earlier versions of MatchUp Object, you would set this value with a call to the **SetLicenseString** function. Using an environment variable makes it much easier to update the license string without having to edit and re-compile the application.

It used to be necessary, even when employing an environment variable, to call the **SetLicenseString** function without passing the license string value. This is no longer true. MatchUp Object will still recognize the SetLicenseString function, but you should eventually remove any reference to it from your code.

Windows

Windows users can set environment variables by doing the following:

1. Select **Start** > **Settings**, and then click **Control Panel**.
2. Double-click **System**, and then click the **Advanced** tab.
3. Click **Environment Variables**, and then select either *System Variables* or *Variables for the user X*.

4. Click **New**.
5. Enter “MD_LICENSE” in the *Variable Name* box.
6. Enter the license string in the *Variable Value* box, and then click **OK**.

Please remember that these settings take effect only upon start of the program. It may be necessary to quit and restart the application to incorporate the changes.

Linux/Solaris/HP-UX/AIX

Unix-based OS users can simply set the license string via the following (use the actual license string, instead):

```
export MD_LICENSE=A1B2C3D4E5
```

If this setting is placed in the profile, remember to restart the shell.

MatchUp Object also used to employ its own environment variable, mdMatchUp_LICENSE. The MD_LICENSE variable is shared across the entire Melissa Data product line of programming tools. MatchUp Object will still use the old license variable for the time being, but you should transition to using MD_LICENSE as soon as possible.

MatchCode List Interface

The Matchcode Editor for Windows handles the task of creating and modifying matchcodes for many situations. The editor application will also run on a Linux system under WINE.

However, because MatchUp Object works across multiple platforms and not all users will have access to a Windows emulator, MatchUp Object also includes this MatchCode List interface which allows the developer to programmatically retrieve the list of available matchcodes in the MatchUp Object matchcode database.

Matchcode List Interface Functions

Initializing the Matchcode List Interface is simpler than for the other interfaces, since no license string is required.

SetPathToMatchUpFiles()

String value. This function accepts a string value indicating the file path to the folder containing the MatchUp Object files.

To provide maximum compatibility with Windows, three files are installed in your 'Common App Data' directory. For Windows Vista and Windows 7 the default location is "C:\ProgramData\MelissaDATA\MatchUp." For Windows XP the default location is "C:\Documents and Settings\All Users\Application Data\Melissa DATA\MatchUp." The location of this directory can be changed by users so please note this, as it can often be the source of issues when running the samples/demos.

Syntax

```
mcList->SetPathToMatchUpFiles(StringPathValue);
```

C

```
mdMUMatchcodeListSetPathToMatchUpFiles(mdMCList, char*);
```

COM

```
mcList.PathToMatchUpFiles = StringPathValue
```

InitializeDataFiles()

The InitializeDataFiles method opens the needed data files and prepares the MatchCode List Interface for use.

Syntax

```
ProgramStatus = mcList->InitializeDataFiles()
```

C

```
ProgramStatus =
   mdMUMatchcodeListInitializeDataFiles(mdMCList)
```

COM+/.NET

```
ProgramStatus = mcList.InitializeDataFiles()
```

GetInitializeErrorString()

Returns a descriptive string to describe the error from the InitializeDataFiles function when attempting to initialize the interface and retrieve the matchcode list.

Syntax

```
    StringValue = mcList->GetInitializeErrorString()
```

C

```
    StringValue =
       mdMUMatchcodeListGetInitializeErrorString(mdMCList)
```

COM+/.NET

```
    StringValue = mcList.GetInitializeErrorString()
```

GetMatchcodeCount()

Retrieves the number of matchcodes in the mdmatchup.mc database. The matchcode count allows you to programmatically create a loop using the returned count as the number of iterations required to retrieve all the present matchcode names.

Syntax

```
    integer = mcList->GetMatchcodeCount()
```

C

```
    integer = mdMUMatchcodeListGetMatchcodeCount(mdMCList)
```

COM+/.NET

```
    integer = mcList.MatchcodeCount
```

GetMatchcodeName()

Getmatchcodename can be called programmatically within a loop using the returned GetMatchcodeCount as the number of iterations required to retrieve all of the matchcode names in the current mdmatchup.mc database.

Syntax

```
    StringValue = mcList->GetMatchcodeName(int)
```

C

```
    StringValue = mdMUMatchcodeListGetMatchcodeName(mdMCList,
       int)
```

COM+/.NET

```
    StringValue = mcList.GetMatchcodeName(int)
```

Matchcodes and the Matchcode Editor

The first part of this chapter details the theory and practice of matchcodes. For information on the Matchcode Editor application, see the Matchcode Editor subsection.

Matchcodes

Matchcodes are sets of rules that tell MatchUp Object which data fields to consider when determining if two records are duplicates. MatchUp Object uses the matchcode to construct a "match key," a simplified string of characters that represents the information within the record, enough to determine if the given record is unique or a duplicate.

Matchcode Components

Each matchcode consists of one or more components which are specific data types that enable a developer to tell MatchUp Object which fields to use by programmatically mapping the fields in the real database to these data types.

The matchcode component should match the data type that MatchUp Object needs to build the match key, not necessarily the format found in the database. In other words, if the database contains full names (first and last) but only last names are needed for deduping, the matchcode would use the last name component. At the programming stage, where fields are mapped to specific components, the full name field would be mapped to the last name component. MatchUp Object is smart enough to parse the name and use only the information it needs.

The following table lists all of the available matchcode components (Data Types) in MatchUp Object.

Component	Description
Prefix	Prefix of a personal name (Mr, Mrs, Ms, Dr)
First Name	A first name
Middle Name	A middle name

Component	Description
Last Name	A last name
Suffix	A suffix from a personal name
Gender	Male/Female/Neutral
First/Nickname	A representative nickname for a first name
Middle/Nickname	A representative nickname for a middle name
Department/Title	A title and/or department name[1]
Company	A company name
Company Acronym	A company's acronym[2]
Street Number	The street number from an address line[3]
Street Pre-Directional	"South" in "3 South Main St"
Street Name	The street name from an address line
Street Suffix	An address suffix (St, Ave, Blvd)
Street Post-Directional	"North" in "3 Main St North"
PO Box	PO Boxes also include Farm Routes, Rural Routes, etc.
Street Secondary	Apartments, floors, rooms, etc.
Address	A single unparsed address line[4]
City	A city name. ZIP or Postal code is usually more accurate
State/Province	A state or province name
Zip9	A full ZIP + 4® code (9 digits)[5]
Zip5	The ZIP Code (5 digits)
Zip4	The +4 extension of a ZIP + 4 code (4 digits)
Postal Code (Canada)	A Canadian Postal Code
City (UK)	A city in the United Kingdom
County (UK)	A county in the United Kingdom
Postcode (UK)	A United Kingdom Postcode

Component	Description
Country	A country
Phone/Fax	A phone number[6]
E-Mail Address	An e-mail address[7]
Credit Card Number	A credit card number
Date	A date[8]
Numeric	A numeric field[9]
Proximity	Allows you to specify a maximum distance in miles between records in which a match will be possible.[10]
General	Any general information. ID, birthday, SSN, etc.

1. **Company, Company Acronym, Department/Title** — Frequently these components don't match exactly because of 'noise words' such as "the," "and," "agency," and so on. MatchUp strips these words from these components.
2. **Company Acronym** — MatchUp Object converts any multi-word company name into an acronym (for example, "International Business Machines" is squeezed into "IBM"). Single-word company names are left as they are. This conversion is done after noise words are removed.
3. **Street Address Components** — The seven street address components (Street Number, Street Pre-Directional, Street Name, Street Suffix, Street Post-Directional, PO Box, Street Secondary) are obtained by splitting up to three address lines. Note that PO Box and/or Street Secondary do not have to appear on their own line, or in a particular field. MatchUp's proprietary "street smart" splitter does all of the work.
4. **Full Address** — When using the Full Address component, you are at the mercy of every little deviation in data entry. Because MatchUp Object's street splitter is so powerful, it is preferable to use street address components instead of the Full Address in nearly all cases. The only exception may be when processing foreign addresses that don't conform very well to US, Canadian or UK addressing formats. This is discussed in more detail starting on page 151.
5. **Zip9, Zip5, Zip4, Canadian Postal Code** — MatchUp Object removes dashes and spaces from ZIP codes. When processing a mix of Canadian Postal Codes and US ZIP codes, use the Zip9 component.
6. **Phone Number** — MatchUp Object removes non-numeric characters from phone numbers. Leading '1-' and trailing extensions are stripped if present. Numbers lacking an area code are right justified so that the local dialing code and number are aligned with numbers having area codes. If a data table often has missing or inaccurate area codes (i.e., after a recent area code split), start at the 4th position of the phone number component. Do not use the right most 7 positions, as badly formatted extensions can sometimes cause the phone number to get coded improperly.
7. **E-Mail Address** — MatchUp Object removes illegal characters from e-mail addresses. Incomplete, changed, and commonly misspelled domain names are corrected using the Email Address data table.

8. **Date** — MatchUp Object allows you to specify a number of days for which a match will be possible if the records being compared fall within the set number of days apart.
9. **Numeric** — This allows you to specify an integer number for which a match will be possible if the record's unit difference falls within the set number.
10. **Proximity** — The proximity component requires you to map in Latitude / Longitude coordinates (Not determined by MatchUp. Can be determined by a product such as GeoCoder or Contact Verify) allowing you to match addresses within a maximum distance setting for this component.

Matchcode Component Properties

The matchcode components tell MatchUp Object which data types to use for creating the match key. The component properties tell MatchUp Object how much of the data to use and what parts.

Often, especially for potentially long fields like personal names and city or street names, MatchUp Object doesn't need the full contents of the field to determine if the field is a duplicate of another. Only ten characters or so will often be enough.

In another example, a database might include the area code and the phone number in some records and just the local number in others. By only considering seven characters of the field starting at position four, MatchUp Object has a better chance of detecting a duplicate.

1. **Data Type** — See the previous table starting on page 8.
2. **Label** — This is a line of text that describes the component. Not all fields allow the label to be edited. This is most useful for clarifying the contents of General fields that don't fit any of the other component types.
3. **Size** — This is the maximum number of characters from the field that MatchUp Object will use to build the match key. Sizing is done after all other properties are applied.
4. **Start** — This property determines where MatchUp Object begins counting when applying the Size property.
    - **Left** — Starts from the first character of the field. This is the most commonly used option.
    - **Right** — Starts from the last character of the field. In other words, if the data included a phone number of "949-589-5200" and the size was 7, MatchUp Object would use "5895200" for the match key.
    - **Position** — Starts from a specific position within the field.
5. **Fuzzy** — Fuzzy settings allow for matching of non-exact components. These options are mutually exclusive, so you can only select one at a time.

- **Phonetex** — (pronounced "Fo-NEH-tex") An auditory matching algorithm. It works best in matching words that sound alike but are spelled differently. It is an improvement over the Soundex algorithm described below.
- **Soundex** — An auditory matching algorithm originally developed by the Department of Immigration in 1917 and later adopted by the USPS. Although the Phonetex algorithm is measurably superior, the Soundex algorithm is presented for users who need to create a matchcode that emulates one from another application.
- **Containment** — Matches when one record's component is contained in another record. For example, "Smith" is contained in "Smithfield."
- **Frequency** — Matches the characters in one record's component to the characters in another without any regard to the sequence. For example "abcdef" would match "badcfe."
- **Fast Near** — A typographical matching algorithm. It works best in matching words that don't match because of a few typographical errors. Exactly how many errors is specified on a scale from 1(Loose) to 4(Tight). The Fast Near algorithm is a faster approximation of the Accurate Near algorithm described below. The trade-off for speed is accuracy; sometimes Fast Near will find false matches or miss true matches.
- **Accurate Near** — An implementation of the Levenshtein algorithm. It is a typographical matching algorithm. The Accurate Near algorithm produces better results than the Fast Near algorithm, but is slower.
- **Frequency Near** — Similar to Frequency matching except that you specify how many characters may be different between components.
- **Vowels Only** — Only vowels will be compared. Consonants will be removed.
- **Consonants Only** — Only consonants will be compared. Vowels will be removed.
- **Alphas Only** — Only alphabetic characters will be compared.
- **Numerics Only** — Only numeric characters will be compared. Decimals and signs are considered numeric.
- **MD Keyboard** — An algorithm developed by Melissa Data which counts keyboarding mis-hits with a weighted penalty based on the distance of the mis-hit and assigns a percentage of similarity between the compared strings.

6. **Fuzzy Advanced** —Please research the definitions of the following advanced algorithms before implementing in a matchcode.

- **Jaro** — Gathers common characters (in order) between the two strings, then counts transpositions between the two common strings.
- **Jaro-Winkler** — Just like Jaro, but gives added weight for matching characters at the start of the string (up to 4 characters).

- **n-Gram** — Counts the number of common sub-strings (grams) between the two strings. Substring size 'N', is currently defaulted as 2 in MatchUp.
- **Needleman-Wunch** — Similar to Accurate Near, except that inserts/deletes aren't weighted as heavily and as compensation for keyboarding mis-hits, not all character substitutions are weighted equally.
- **Smith-Waterman-Gotoh** — Builds on Needleman-Wunch, but gives a non-linear penalty for deletions. This effectively adds the 'understanding' that the keyboarder may have tried to abbreviate one of the words.
- **Dice's Coefficient** — Like Jaro, Dice counts matching n-Grams (discarding duplicate n-Grams).
- **Jaccard Similarity Coefficient** — Very similar to Dice's Coefficient with a slightly different calculation.
- **Overlap Coefficient** — Again, very similar to Dice's Coefficient with a slightly different calculation. String similarity algorithm based on a substring calculation.
- **Longest Common Substring** — Finds the longest common substring between the two strings.
- **Double MetaPhone** — Performs 2 different Phonetex-style transformations. Returns a value dependant on how many of the transformations match (ie, 1 versus 1, 1 versus 2, 2 versus 1, 2 versus 2).

7. **Distance** — This field is context sensitive, depending on the Data Type and Fuzzy algorithm.

- **Data Type**
    - Proximity - Distance in miles. Range: 0-4000
    - Numeric - Integer number.
    - Date - Number of days.
- **Fuzzy**
    - Fast Near - Number of typographical errors. Range: Loose(1) - Tight(4)
    - Accurate Near - Number of typographical errors. Range: Loose(1) -Tight(4)

The following use a percentage range of 0-100%, indicating the minimum percentage of similarity which will return a match between two strings.

- N-Gram
- Jaro
- Jaro-Winkler
- LCS
- Needleman-Wunch
- MD Keyboard

•Smith-Waterman-Gotoh

•Dice's Coefficient

•Jaccard Similarity Coefficient

•Overlap Coefficient

•Double MetaPhone

8. **Short/Empty Settings** — These settings control matching between incomplete or empty fields. They are not mutually exclusive, meaning that any combination of these settings may be selected.
   - **Initial Only** — Will match a full word to an initial (for example, "J" and "John").
   - **One Blank Field** — Will match a full word to no data (for example, "John" and "").
   - **Both Blank Fields** — Match this component if both records contain no data. This is a very important concept in creating matchcodes. See Blank Field Matching later in this chapter for more information.
9. **Swap** — Swap matching is the ability to compare one component to another component. For example, if you were to swap match a First Name component and a Last Name component, you could match "John Smith" to "Smith John." Swap matching is always defined for a pair of components. MatchUp allows you to specify up to 8 swap pairs (named "Pair A" through "Pair H"). It is strongly recommended that the other properties of both member components are identical. For more information on using Swap matching see "Other Uses for Swap Matching" on page 28.

Component Combinations

Every matchcode is composed of one or more combination of components. These columns represent different combinations of components which may detect a match between two records. A match found using any one of the combinations in a matchcode is considered a match. Programmers may think in terms of a series of OR conditions. Satisfying any one of them is considered a positive result.

MatchUp allows up to 16 different combinations of components per matchcode.

A good example of combinations would be a matchcode designed to catch last names as well as either street addresses or Post Office Box addresses.

- **Condition #1**: ZIP/PC, Last Name, Street Number, Street Name
- **Condition #2**: ZIP/PC, Last Name, PO Box

Such a matchcode might look like this:

Component	Size	1	2
ZIP/PC	5	X	X
Last Name	5	X	X
Street #	4	X	
Street Name	4	X	
PO Box	10		X

Columns 3 through 16 have been omitted for the sake of clarity. The trick to understanding this table is to look at the vertical columns of X's. For example, looking at column 1, there are X's in ZIP/PC, Last Name, Street #, and Street Name, indicating the goal of condition #1 exactly. In column 2 are X's in ZIP/PC, Last Name, and PO Box, matching condition #2.

For a more advanced example:

Component	Size	1	2	3	4
ZIP/PC	5	X	X	X	X
Last Name	5	X	X		
Company	10			X	X
Street #	4	X		X	
Street Name	4	X		X	
PO Box	10		X		X

This matchcode may produce matches if any one of following 4 conditions returns true:

- **Condition #1**: ZIP/PC, Last Name, Street Number, Street Name
- **Condition #2**: ZIP/PC, Last Name, PO Box
- **Condition #3**: ZIP/PC, Company, Street Number, Street Name
- **Condition #4**: ZIP/PC, Company, PO Box

This matchcode could be used on a list containing a mixture of both personal and company names and either street or PO Box addresses.

First Component Restrictions

MatchUp now has two deduping engines. The object will determine if one is either necessary or more efficient, and select that engine for usage. In some cases, processing will be faster if the traditional ReadWrite engine is selected because of First Component properties. These are:

1. It must appear in every combination.
2. It cannot use the following types of Fuzzy matching: Containment; Frequency; Fast Near; Frequency Near; Accurate Near. All others are allowed.
3. It cannot use Initial Only matching.
4. It cannot use One Blank Field matching.
5. It cannot use Swap Matching.

In other situations, you may have combinations where there are no common components. An example would be:

- **Condition #1**: ZIP/PC, Street Number, Street Name
- **Condition #2**: ZIP/PC, PO Box
- **Condition #3**: Proximity

In this case, MatchUp would determine that it needs to use its Intersecting Logic. This engine is required because there are no common components. Speed benchmarks may be surprisingly similar, but may in fact return more duplicates.

Blank Field Matching

This needs a special discussion, as its importance is often overlooked. As discussed above, if this property is on, then the absence of data in both records would indicate a match. If this property is off, then two records with missing data, but matching in every other way, will not match.

The following example demonstrates when Blank set to ON allows a match on a non-critical component. Setting Blank to OFF is recommended for a critical component.

Component	Size	Blank	1	2
ZIP/PC	5	Yes	X	X
Last Name	5	Yes	X	X
Street #	4	Yes	X	
Street Name	4	Yes	X	

Component	Size	Blank	1	2
PO Box	10	Yes		X

As described above, this produces the following combinations:

- **Condition #1**: ZIP/PC, Last Name, Street Number, Street Name
- **Condition #2**: ZIP/PC, Last Name, PO Box

For this example, take the following records:

Name:	Joe Smith	Suzi Smith
Address:	326 Main Street	405 Main St
City/State/PC:	Pembroke, MA 02066	Pembroke, MA 02066

The following matchcode keys would be generated:

Cond#	Zip/PC	Last Name	Street #	Street Name	PO Box
1	02066	SMITH	326	MAIN	
2	02066	SMITH	405	MAIN	

According to these matchcode keys, it is clear that these two records do not satisfy condition #1. But because blank field matching is selected, they do satisfy condition #2. The Zip/PC, Last Name, and PO Box are exactly the same. Therefore, the two records do match.

Obviously, this is not the correct result. Making one change to the matchcode:

Component	Size	Blank	1	2
ZIP/PC	5	Yes	X	X
Last Name	5	Yes	X	X
Street #	4	Yes	X	
Street Name	4	Yes	X	
PO Box	10	No		X

The same comparison is done for combination #2, but the match is disallowed this time because the matchcode now indicates that missing (blank) information is not allowed to figure in the matching condition.

Looking at another example (using the same matchcode):

Name:	Joe Smith	Suzi Smith
Address:	PO Box 123	PO Box 456
City/State/PC:	Pembroke, MA 02066	Pembroke, MA 02066

This pairing produces the following matchcode keys:

Cond#	Zip/PC	Last Name	Street #	Street Name	PO Box
1	02066	SMITH			123
2	02066	SMITH			456

This record has the same problem as before, but this time combination #1 is the cause. An even better matchcode would be:

Component	Size	Blank	1	2
ZIP/PC	5	Yes	X	X
Last Name	5	Yes	X	X
Street #	4	No	X	
Street Name	4	No	X	
PO Box	10	No		X

This is one matchcode that works well. There is one more possible tweak, however: turn on Both Blank Fields for the Street # component. Occasionally, MatchUp Object may encounter records such as:

Name:	Joe Notarangello	Suzi Notarangello
Address:	Oceanfront Estates	Oceanfront Est.
City/State/PC:	Pembroke, MA 02066	Pembroke, MA 02066

This reflects a trend in up-scale neighborhoods, where neither street address has a Street # component, though it is very likely these records should match.

So this new, improved matchcode will account for these situations:

Component	Size	Blank	1	2
ZIP/PC	5	Yes	X	X
Last Name	5	Yes	X	X

Component	Size	Blank	1	2
Street #	4	Yes	X	
Street Name	4	No	X	
PO Box	10	No		X

Matchcode Mapping

Matchcodes deal with the abstract. The components in a matchcode represent specific types of data, but they aren't directly linked to the fields in databases. Mapping creates the link between the data and the matchcode.

For example, take the following matchcode:

Component	Size	Fuzzy	1
Zip5	5	No	X
Last Name	5	No	X
First Name	5	No	X
Company	10	No	X

Add a database which contains the following fields:

NAME	Contains full names (“Mr. John Smith”).
COMPANY	Contains company names (“Melissa Data”).
ADD1	Contains first (primary) address line (“22382 Avenida Empresa”).
ADD2	Contains second (secondary) address line (“Suite 34”).
CSZ	Contains City/State/Zip (“Rancho Santa Margarita, CA 92688”).

An application must create a link between a database’s fields (Name, Company, Add1, Add2 and CSZ) and the matchcode components (Zip5, Last Name, First Name, Company).

With the example above, it may appear that the application will have to contain extensive splitting routines. This is not the case. All that is necessary is to tell MatchUp what type of data is in a specific field and the format of that data.

In the example above, an application would use the following matchcode mapping:

Matchcode Component	Database Field	Matchcode Mapping
Zip5	CSZ	CityStZip
Last Name	NAME	FullName
First Name	NAME	FullName
Company	COMPANY	Company

This mapping tells MatchUp that the 5-digit ZIP Code information is in a field named "CSZ" which is described as a field containing city, state, and ZIP Code information. The Last Name can be found in a field called "NAME" and is described as a full name field (which is a full name sequenced: Pre, FN, MN, LN, Suf). A complete list of possible matchcode mappings can be found on page 148.

Matchcode mappings follow five rules:

1. For every Matchcode Component, the application must specify a mapping. The only exception is described in rule 2.
2. Actual Address components names (such as Street Number, Street Pre-Directional, Street Name, Street Suffix, Street Post-Directional, PO Box, and Street Secondary) are not listed for mapping purposes. Instead, the names Address Line 1, Address Line 2, and Address Line 3 are used. The example below used four address components in the matchcode (Street #, Street Name, Street Secondary, PO Box). However, it only used two address lines.
3. If a matchcode uses any address components, Address Lines 1-3 will be listed after all other components regardless of where the address component appears in the matchcode. In the following example, the address components are listed before company in the matchcode, but Address Lines 1-3 are listed at the end (after company).
4. If a matchcode uses address components, Address Lines 1-3 will require at least one line to be mapped, but not all. If a database only has one address field, an application will only need to map Address 1 to that field. All other components must be mapped.
5. Address Lines should be mapped from the top down (Address Line 1, then 2, then 3).

Enhancing the matchcode in the previous example:

Component	Size	Fuzzy	1	2
Zip5	5	No	X	X
Last Name	5	No	X	X
First Name	5	No	X	X
Street #	5	No	X	
Street Name	5	No	X	
Street Secondary	12	No	X	
PO Box	10	No		X
Company	10	No	X	X

Again, MatchUp doesn't use the individual address components. They are replaced with Address 1, Address 2, and Address 3. So, the application would use the following Matchcode Mapping:

Matchcode Component	Database Field	Matchcode Mapping
Zip9	CSZ	CityStZip
Last Name	NAME	FullName
First Name	NAME	FullName
Company	COMPANY	Company
Address Line 1	ADD1	Address
Address Line 2	ADD2	Address
Address Line 3	(none)	

Note on Rule #1:

If a database does not contain a field for information called for by a component in a matchcode, such as company field in the above example, then that matchcode should not be used to dedupe that database.

Use a different matchcode or modify an existing matchcode, as outlined later in this chapter.

However, if a matchcode calls for last name, for example, and the database only has full name, then simply map the full name field to the last name and MatchUp Object will handle parsing the field.

Matchcode Mapping Using the API

All three of the MatchUp Object deduping interfaces (Incremental, Read/Write and Hybrid) have an **AddMapping** function. This is used to create mappings for the current instance of whatever deduper an application is using. For the last example above, call the function in the following way:

```
mu->ClearMapping();

mu->AddMapping(mu->CityStZip);

mu->AddMapping(mu->FullName);

mu->AddMapping(mu->FullName);

mu->AddMapping(mu->Company);

mu->AddMapping(mu->Address);

mu->AddMapping(mu->Address);
```

The value being passed to the function is an enumerated value of the type MatchcodeMapping.

Note that this code does not tell MatchUp Object anything about the database containing the data to be deduped. The application handles the data access separately and then passes the necessary fields to the deduper using the **AddField** function.

Changing Mappings

It is possible to change mappings in the middle of a session if, for example, an application has to handle two databases with different data structures. Continuing with the example from above, assume that the second database has the following structure:

Matchcode Component	Database Field	Matchcode Mapping
Zip9	CSZ	CityStZip
Last Name	LAST	LastName
First Name	FIRST	FirstName
Company	COMPANY	Company
Address Line 1	ADD1	Address
Address Line 2	ADD2	Address

Matchcode Component	Database Field	Matchcode Mapping
Address Line 3	(none)	

To use this mapping, the application would first have to call the **ClearMappings** function to remove the existing mappings and call the **AddMapping** function again to configure the new mapping.

```
mu->AddMapping(mu->CityStZip);

mu->AddMapping(mu->LastName);

mu->AddMapping(mu->FirstName);

mu->AddMapping(mu->Company);

mu->AddMapping(mu->Address);

mu->AddMapping(mu->Address);
```

Optimizing Matchcodes

Some matchcodes process much faster than others in spite of the fact that they detect the same matches. This section will assist in creating the most efficient matchcodes. This discussion is included so developers can better understand why certain things are done while optimizing.

Optimizing can make a significant difference in processing speed. 58-hour runs have been reduced to four hours simply by optimizing the matchcode.

It is important, however, that the developer verifies that a matchcode works in the intended way before attempting any optimizations. If a matchcode is not functioning properly, these optimizations will not help, and could quite possibly make the situation worse.

Component Sequence

As discussed in the previous section, data may process faster if the first component of a matchcode has certain properties:

- It must be used in every combination.
- It cannot use certain types of Fuzzy Matching: Containment; Frequency; Fast Near; Frequency Near; or Accurate Near (other types are okay, though).
- It cannot use Initial Only matching.
- It cannot use One Blank Field matching.
- It cannot use Swap matching.

If the matchcode's second component also follows these conditions, MatchUp Object will incorporate it into its clustering scheme (see Clustering on page 3 for more information on clustering). Additional components, if they follow in sequence (third, fourth, and so on), will be used if they, too, satisfy these conditions. Incorporating a component into a cluster greatly reduces the number of comparisons MatchUp Object has to perform which, in turn, speeds up your processing.

This is a simple example of optimization.

Component	Size	Fuzzy	Blank	1	2
ZIP/PC	5	No	Yes	X	X
Street #	5	No	Yes	X	
Street Name	5	No	No	X	
PO Box	10	No	No		X
Last Name	5	No	Yes	X	X

As shown here, MatchUp Object will only cluster by ZIP/PC. But note that the last component satisfies all the conditions listed earlier.

Component	Size	Fuzzy	Blank	1	2
ZIP/PC	5	No	Yes	X	X
Last Name	5	No	Yes	X	X
Street #	5	No	Yes	X	
Street Name	5	No	No	X	
PO Box	10	No	No		X

This simple optimization will produce significant improvements in speed. In general, if your matchcode requires multiple components to be used in all set combinations, place them before other components.

Fuzzy Algorithms

Fuzzy algorithms fall into two categories: early matching and late matching.

Early matching algorithms are algorithms where a string is transformed into a (usually shorter) representation and comparisons are performed on this result. In MatchUp, these transformations are performed during key generation (the **BuildKey** function in each interface), which means that the early matching algorithms pay a speed penalty once per record: as each record's key is built.

Late matching algorithms are actual comparison algorithms. Usually one string is shifted in one direction or another, and often a matrix of some sort is used to derive a result. These transformations are performed during key comparison. As a result, late matching algorithms pay a speed penalty every time a record is compared to another record. This may happen several hundred times per record.

Obviously, late matching is much slower than early matching. If a particular matchcode is very slow, changing to a faster fuzzy matching algorithm may improve the speed. Often, a faster algorithm will give nearly the same results, but it is a good idea to test any such change before processing live data.

The fuzzy algorithms, ranked from slowest to fastest:

Algorithm	**Late or Early**	**Speed (10=fastest)**
Jaro	Late	1
Jaro-Winkler	Late	1
n-Gram	Late	1
Needleman-Wunch	Late	1
Smith-Waterman-Gotoh	Late	1
Dice's Coefficient	Late	1
Jaccard Similarity Coefficient	Late	1
Overlap Coefficient	Late	1
Longest Common Substring	Late	1
Double Metaphone	Late	1
Accurate Near	Late	1
Fast Near	Late	3
Containment	Late	4
Frequency Near	Late	4
Frequency	Late	6
Phonetex	Early	7
Soundex	Early	8

Algorithm	Late or Early	Speed (10=fastest)
Vowels Only	Early	9
Numerics Only	Early	9
Consonants Only	Early	9
Alphas Only	Early	9
Exact	N/A	10

The speed values are only rough estimates.

Another benefit of using a faster fuzzy algorithm is that an application may be able to exploit the component sequence optimization shown earlier. All of the early matching algorithms satisfy the restrictions for first components.

Unnecessary Components

Components that are not used in any combinations (in other words, they have no X's in columns 1 through 16) are a sign of poor matchcode design.

Take the following matchcode:

Component	Size	Fuzzy	Blank	1	2
ZIP/PC	5	No	Yes	X	X
Last Name	5	No	Yes	X	X
First Name	5	No	Yes		
Street #	5	No	Yes	X	
Street Name	5	No	No	X	
PO Box	10	No	No		X

First name is not being used in any combination. Perhaps it was used in a combination that has since been removed from this matchcode, but it is no longer necessary.

Unnecessary Combinations

Take the following matchcode:

Component	Size	Fuzzy	Blank	1	2	3	4
ZIP/PC	5	No	Yes	X	X	X	X
Last Name	5	No	Yes	X	X	X	X
First Name	5	No	Yes	X	X		
Street #	5	No	Yes	X		X	
Street Name	5	No	No	X		X	
PO Box	10	No	No		X		X

Here are the four conditions for matching:

Condition #1:	ZIP/PC	Last Name	First Name	Street #	Street Name	
Condition #2:	ZIP/PC	Last Name	First Name			PO Box
Condition #3:	ZIP/PC	Last Name		Street #	Street Name	
Condition #4:	ZIP/PC	Last Name				PO Box

There is no match that will be detected by condition #1 that would not be detected by condition #3. Similarly, matches found by condition #2 will always be found by condition #4. In other words, condition 3 is a subset of condition 1, and condition 2 is a subset of condition 4. Subsets are rarely desirable.

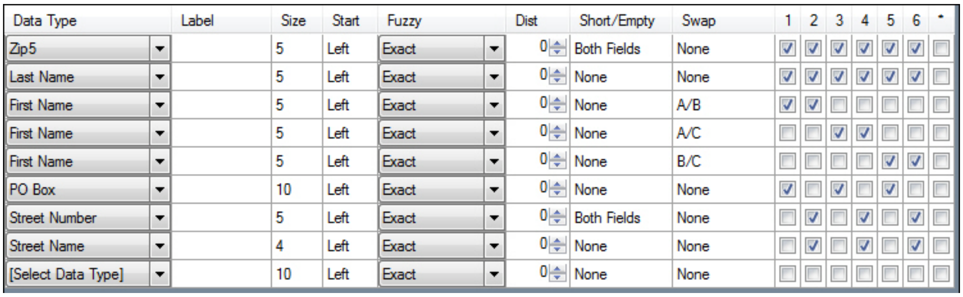
So either conditions 1 and 2 aren't needed or conditions 3 and 4 were a mistake. If conditions 1 and 2 are eliminated, the First Name component should also be removed, as it will not be needed.

Other Uses for Swap Matching

Swap matching is used to catch matches when two fields are flipped around. The most common occasion is catching the “John Smith” and “Smith John” records. But there are other uses:

Comparing Household Records

When there are two or three first or full names per record, a list provider may claim that every record is always “husband, wife, then children,” but records will read wife then child and husband):

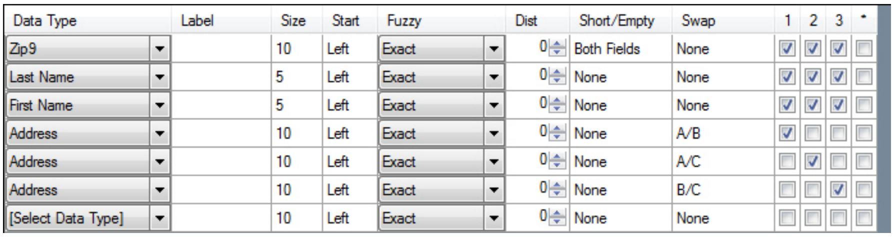


Data Type	Label	Size	Start	Fuzzy	Dist	Short/Empty	Swap	1	2	3	4	5	6	*
Zip5		5	Left	Exact	0	Both Fields	None	✓	✓	✓	✓	✓	✓	☐
Last Name		5	Left	Exact	0	None	None	✓	✓	✓	✓	✓	✓	☐
First Name		5	Left	Exact	0	None	A/B	✓	✓	☐	☐	☐	☐	☐
First Name		5	Left	Exact	0	None	A/C	☐	☐	✓	✓	☐	☐	☐
First Name		5	Left	Exact	0	None	B/C	☐	☐	☐	☐	✓	✓	☐
PO Box		10	Left	Exact	0	None	None	✓	☐	✓	☐	✓	☐	☐
Street Number		5	Left	Exact	0	Both Fields	None	☐	✓	☐	✓	☐	✓	☐
Street Name		4	Left	Exact	0	None	None	☐	✓	☐	✓	☐	✓	☐
[Select Data Type]		10	Left	Exact	0	None	None	☐	☐	☐	☐	☐	☐	☐

In the above example, select **Either component can match** for Swap Pairs A, B, and C.

Comparing up to Three Address Lines

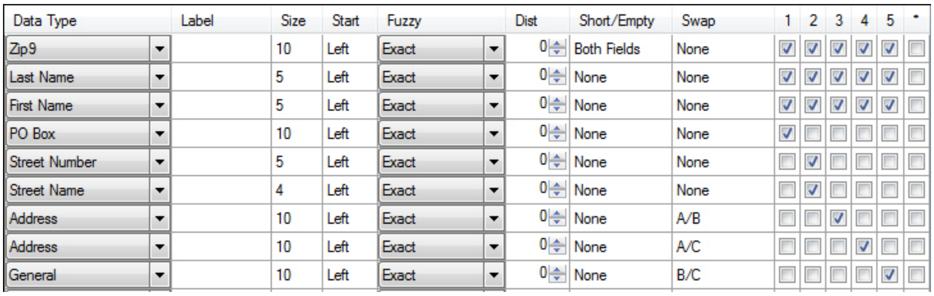
Although the address splitter works well in the US and Canada, some European countries can cause problems. A typical Euro-Matchcode will not use street split components and look at three address lines instead. The swap matching ensures that every address line is compared with every other address line.



Data Type	Label	Size	Start	Fuzzy	Dist	Short/Empty	Swap	1	2	3	*
Zip9		10	Left	Exact	0	Both Fields	None	✓	✓	✓	☐
Last Name		5	Left	Exact	0	None	None	✓	✓	✓	☐
First Name		5	Left	Exact	0	None	None	✓	✓	✓	☐
Address		10	Left	Exact	0	None	A/B	✓	☐	☐	☐
Address		10	Left	Exact	0	None	A/C	☐	✓	☐	☐
Address		10	Left	Exact	0	None	B/C	☐	☐	✓	☐
[Select Data Type]		10	Left	Exact	0	None	None	☐	☐	☐	☐

Again, select **Either component can match** for Swap Pairs A, B, and C.

Don’t always discard the street split component matchcodes because you are working with a foreign database. Sometimes the street splitter will yield usable results. Therefore, a combination of approaches will often work.



Data Type	Label	Size	Start	Fuzzy	Dist	Short/Empty	Swap	1	2	3	4	5	*
Zip9		10	Left	Exact	0	Both Fields	None	✓	✓	✓	✓	✓	
Last Name		5	Left	Exact	0	None	None	✓	✓	✓	✓	✓	
First Name		5	Left	Exact	0	None	None	✓	✓	✓	✓	✓	
PO Box		10	Left	Exact	0	None	None	✓					
Street Number		5	Left	Exact	0	None	None		✓				
Street Name		4	Left	Exact	0	None	None		✓				
Address		10	Left	Exact	0	None	A/B			✓			
Address		10	Left	Exact	0	None	A/C				✓		
General		10	Left	Exact	0	None	B/C					✓	

Using MatchUp Object with Non-U.S. Addresses

For more information on using MatchUp Object with addresses in Canada, the UK and other nations, see “International Deduping Considerations” on page 151.

The Matchcode Editor

The Matchcode Editor is a Windows-based application that creates and edits the matchcode file used by MatchUp Object. This program allows developers to customize copies of the original matchcodes that ship with MatchUp Object or create new matchcodes from scratch.

If you have ever used Melissa Data’s MatchUp software for Windows, you will already be familiar with the functionality of the Matchcode Editor.

Starting the Matchcode Editor

The default installation location for the MatchUpEditor executable is:

```
C:\Program Files\Melissa DATA\DQT\MatchUp\
```

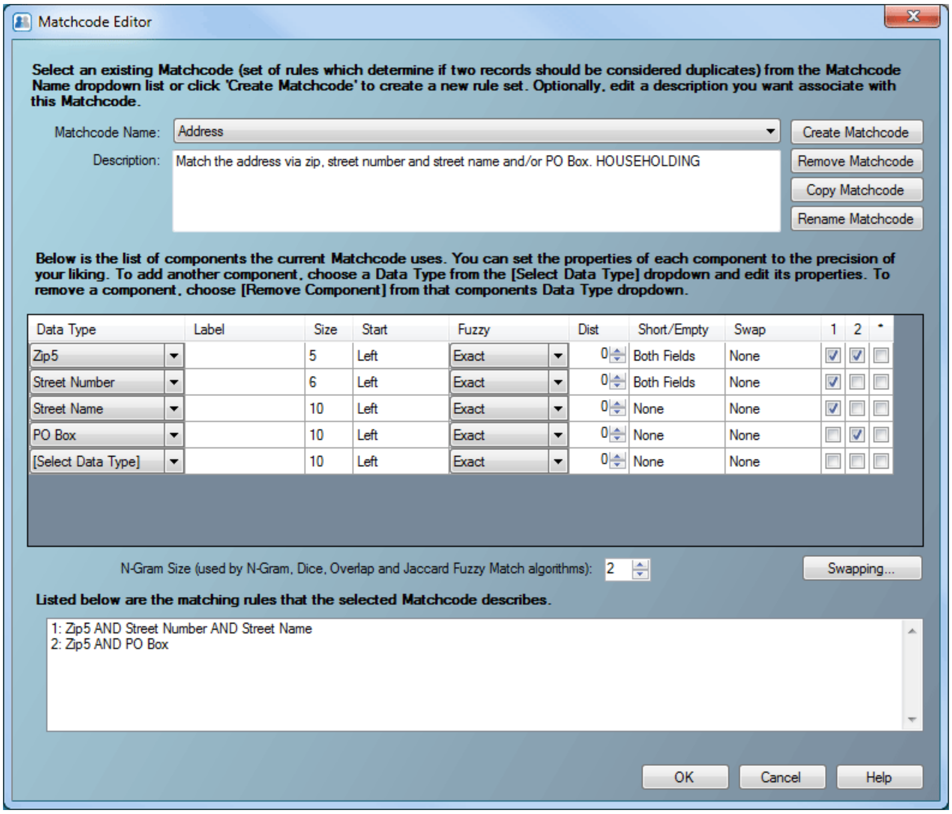
To run, specify the location of the mdMatchUp.mc matchcode file as a command-line parameter (ie, in the program’s shortcut), as in:

```
MatchUpEditor.exe “C:\programdata\Melissa Data\MatchUp”
```

Or you can alter the shortcut’s “Start in” location so that it starts in the same location as the mdMatchup.mc file. You can also run it directly from the Start menu, if you chose that as an installation option.

The Matchcode Editor Interface

The Matchcode Editor screen is divided into three distinct sections: a list of available matchcodes in the matchcode database; the properties of the selected Matchcode; and a description of the Matching Rules for the selected matchcode.


Matchcode Editor

Select an existing Matchcode (set of rules which determine if two records should be considered duplicates) from the Matchcode Name dropdown list or click 'Create Matchcode' to create a new rule set. Optionally, edit a description you want associate with this Matchcode.

Matchcode Name: Address

Description: Match the address via zip, street number and street name and/or PO Box. HOUSEHOLDING

Below is the list of components the current Matchcode uses. You can set the properties of each component to the precision of your liking. To add another component, choose a Data Type from the [Select Data Type] dropdown and edit its properties. To remove a component, choose [Remove Component] from that components Data Type dropdown.

Data Type	Label	Size	Start	Fuzzy	Dist	Short/Empty	Swap	1	2	*
Zip5		5	Left	Exact	0	Both Fields	None	✓	✓	
Street Number		6	Left	Exact	0	Both Fields	None	✓		
Street Name		10	Left	Exact	0	None	None	✓		
PO Box		10	Left	Exact	0	None	None		✓	
[Select Data Type]		10	Left	Exact	0	None	None			

N-Gram Size (used by N-Gram, Dice, Overlap and Jaccard Fuzzy Match algorithms): 2

Listed below are the matching rules that the selected Matchcode describes.

1: Zip5 AND Street Number AND Street Name
2: Zip5 AND PO Box

Matchcode Name

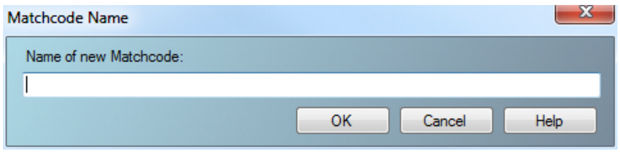
The top portion of the screen contains a drop-down menu of all the matchcodes found in the current matchcode file.

Below this is a **Description:** section that contains the description for the currently selected matchcode.

To the right are the **Create Matchcode**, **Remove Matchcode**, **Copy Matchcode**, and **Rename Matchcode** buttons with which you can create and modify matchcodes. Copying a current matchcode is often the best starting point for creating new matchcodes.

To add a new matchcode:

1. Click the **Create Matchcode** button.
2. Type a name for the new matchcode in the Matchcode Name dialog box and click **OK**.



3. The Matchcode editor presents a blank matchcode screen with no components.
4. Begin adding components. Once a Data Type is selected, click anywhere in the window, or press the Enter key. This will input that data type, and have another row appear that may be edited.

To remove an existing matchcode:

1. Select the matchcode to be deleted in the Matchcode Name: drop-down menu.
2. Click the **Remove Matchcode** button.
3. Click **Yes** in the Remove Matchcode dialog box to confirm the deletion.

To make a copy of an existing matchcode:

1. Select the matchcode to be copied in the Matchcode Name: drop-down menu.
2. Click the **Copy Matchcode** button.
3. Type a name for the new matchcode in the Matchcode Name dialog box and click **OK**.

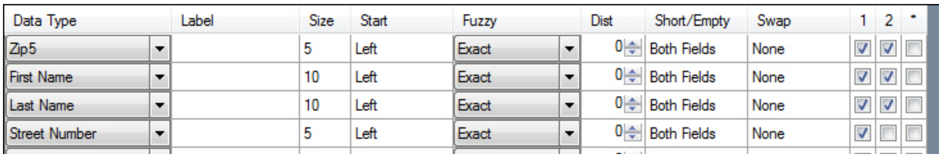
To rename an existing matchcode:

1. Select the matchcode to be renamed in the Matchcode Name: drop-down menu.
2. Click the **Rename Matchcode** button.

3. Type a new name for the matchcode in the Matchcode Name dialog box and click OK.

Matchcode List

Below the matchcode name is the Matchcode List section, a list of components used by the currently selected matchcode.



Data Type	Label	Size	Start	Fuzzy	Dist	Short/Empty	Swap	1	2	*
Zip5		5	Left	Exact	0	Both Fields	None	✓	✓	
First Name		10	Left	Exact	0	Both Fields	None	✓	✓	
Last Name		10	Left	Exact	0	Both Fields	None	✓	✓	
Street Number		5	Left	Exact	0	Both Fields	None	✓		

This list shows the basic settings for each combination.

- **Data Type** — The type of data used by this component. See the table on page 8 for a list of all available types.
- **Label** — (Optional) A description of the data found in this component. Not all component types use this field.
- **Size** — The maximum number of characters from this component to be used by this matchcode. If the data has fewer characters, it will be padded with spaces.
- **Start** — Sets where the current matchcode starts counting when selecting characters to use: the left (beginning); the right (end); a specific character position; or a specific word.
- **Fuzzy** — The type of matching to be used on the selected data type.
- **Distance** — Context sensitive, sets a range for specific data types or fuzzy matching.
- **Short/Empty** — These settings control matching between incomplete or empty fields.
- **Swap** — Swap matching is the ability to compare one component to another component.

For more information on the settings on this part of the dialogs, see "Matchcode Component Properties" on page 11.

Following these fields, to the right side of the list, there is a grid of editable check boxes that shows the combinations in which component is used.

To add a new component to the matchcode:

1. Click the **down arrow** to open the drop-down menu named [Select Data Type]. (There will always be a [Select Data Type] below the last defined matchcode component.)
2. Select the desired data type from the drop-down menu.

3. The new component is added as the last component in the matchcode.
4. Select the settings for the new component by clicking the field you want to change. See the sections below for more information on the controls within this dialog.

To remove a component from a matchcode:

1. Click the **down arrow** to open the drop-down menu of the component to be deleted.
2. Select **[Remove Component]** from the top of the list in the drop-down menu.
3. Once selected, click anywhere in the window, or press the Enter key. This will confirm the removal, and remove the component from the matchcode list.

To change the order of components in a matchcode:

1. Click and drag the name of the component.
2. Drag the component to the new position.

For more information on how combinations of components are used, see "Component Combinations" on page 14.

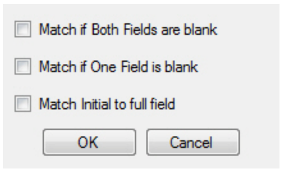
Matching Strategies (Fuzzy)

This setting controls what criteria the matchcode will use to determine how to compare this component of one match key to another match key.

Fuzzy Matching Strategies		
• Phonetex	• Vowels Only	• Needleman-Wunch
• Soundex	• Consonants Only	• Smith-Waterman-Gotoh
• Containment	• Alphas Only	• Dice's Coefficient
• Frequency	• Numerics Only	• Jaccard Similarity Coefficient
• Fast Near	• Jaro	• Overlap Coefficient
• Accurate Near	• Jaro-Winkler	• Longest Common Substring
• Frequency Near	• n-Gram	• Double MetaPhone

Short/Empty Settings

This setting controls whether blank or incomplete fields are considered matches to populated fields or other blank fields. These settings are not exclusive, so two or all three may be selected at one time.


Match if Both Fields are blank
Match if One Field is blank
Match Initial to full field
OK
Cancel

- **Match if both fields are blank** — If two records have the same empty component, that component will be counted as matching.
- **Match if one field is blank** — Allows matching missing data with the full data. For example, "Smith" matches "John Smith." However, two records with the same component missing will not match.
- **Match initial to full field** — Allows matching abbreviated data with the full data. For example, "J Smith" matches "John Smith."

Swap Match Pairs

The Swap Match section selects which combination belong to which swap pairs.

Swap Matching allows matching "John Smith" with "Smith John."

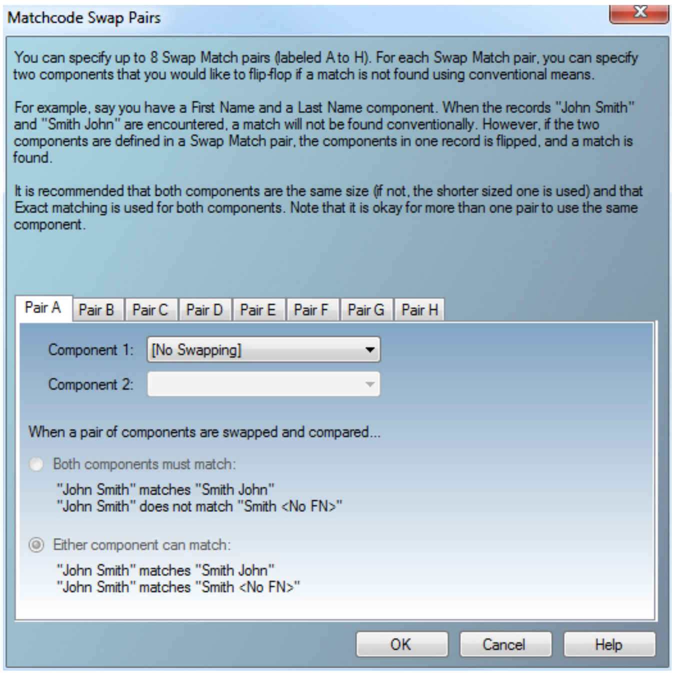
The components must be of the same size and should have the same set of matching options (for example, one can't use Phonetex the other SoundEx). Up to eight pairs, A through H, can be defined.

For more information on using swap pairs, see "Other Uses for Swap Matching" on page 28.

To configure a swap pair:

1. Click the **Swapping...** button.
2. The Matchcode Swap Pairs dialog will open.
3. First select the pair tab you desire to edit. Pair A is selected by default.

4. Select the two components that will be used for this swap pair by selecting them in their respective drop down menus.


Matchcode Swap Pairs

You can specify up to 8 Swap Match pairs (labeled A to H). For each Swap Match pair, you can specify two components that you would like to flip-flop if a match is not found using conventional means.

For example, say you have a First Name and a Last Name component. When the records "John Smith" and "Smith John" are encountered, a match will not be found conventionally. However, if the two components are defined in a Swap Match pair, the components in one record is flipped, and a match is found.

It is recommended that both components are the same size (if not, the shorter sized one is used) and that Exact matching is used for both components. Note that it is okay for more than one pair to use the same component.

Pair A | Pair B | Pair C | Pair D | Pair E | Pair F | Pair G | Pair H

Component 1: [No Swapping]

Component 2:

When a pair of components are swapped and compared...

Both components must match:
"John Smith" matches "Smith John"
"John Smith" does not match "Smith <No FN>"

[✓ Either component can match:]
"John Smith" matches "Smith John"
"John Smith" matches "Smith <No FN>"

5. Then select the swapping rule:

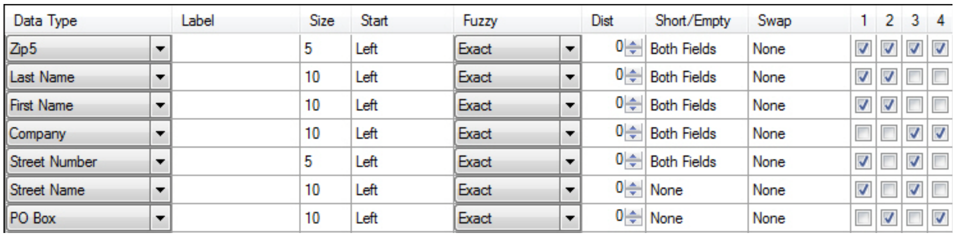
- **Both components must match** — The contents of both components must be a match according to fuzzy matching strategy in use for both components. "John Smith matches "Smith John" but not "Smith <blank>."
- **Either component can match** — At least one of the components must match. "John Smith matches both "Smith John" and "Smith <blank>."

6. Click **OK**.

Combinations

Use these check boxes to select which of the 16 possible combinations will use this component.

It is easier to visualize the effects of these boxes if you look at the list of matchcode components as well:



Data Type	Label	Size	Start	Fuzzy	Dist	Short/Empty	Swap	1	2	3	4
Zip5		5	Left	Exact	0	Both Fields	None	✓	✓	✓	✓
Last Name		10	Left	Exact	0	Both Fields	None	✓	✓	☐	☐
First Name		10	Left	Exact	0	Both Fields	None	✓	✓	☐	☐
Company		10	Left	Exact	0	Both Fields	None	☐	☐	✓	✓
Street Number		5	Left	Exact	0	Both Fields	None	✓	☐	✓	☐
Street Name		10	Left	Exact	0	None	None	✓	☐	✓	☐
PO Box		10	Left	Exact	0	None	None	☐	✓	☐	✓

It is important to note that each VERTICAL column of check marks designates one matchcode. For example, the illustration above shows a combination that is made up of 4 matchcodes:

1. Zip5, Last Name, First Name, Street Number, Street Name
2. Zip5, Last Name, First Name, PO Box
3. Zip5, Company, Street Number, Street Name
4. Zip5, Company, PO Box

Matching Rules

This section details the matching rules, depending on your selections under the Matchcode List.

Read/Write Deduping

Read/Write deduping is usually used for processing entire lists. It works in a manner similar to the way that the MatchUp software products does. A calling program passes an entire list to the Read/Write deduping engine one record at a time. When the entire list has been passed, the calling program tells the API to process the records. Then, the calling program retrieves each record, along with additional deduplication information, from the Read/Write deduper.

Read/Write deduping consists of the following steps:

1. One by one, the program sends a series of record data (ZIP/PC, Name, Address, etc) to the MatchUp API.
2. When completely done (1), the program sends a "process" command to the API.
3. The program retrieves the results for each record with deduplication information.

Order of Output Records

The program will send records in a particular sequence, either in record (raw) order, or maybe in a more sophisticated manner (by ZIP/PC, record type, and so on). MatchUp Object will not return the records in the same order. By default, records are output in cluster order. This order will be loosely based on the matchcode. For example, if the matchcode has Zip5 as its first component, output records will be more or less sorted by ZIP Code (but the developer should not count on this). If the application called the **SetGroupSorting** function, records in the same dupe group will be adjacent. Otherwise, duplicate records may or may not be adjacent (though they usually are near each other).

If a certain sequence is important (for example, records ordered in the same sequence they were input), sort the results after MatchUp Object has processed the data.

Data Lifetime

A Read/Write deduping session is relatively short-lived. Although the actual action of reading and writing records may take time (hours or days), the process is strictly defined into three distinct steps. The key file does not persist beyond this point. Because of this, Read/Write deduping is not usually the choice for ongoing or online processes.

Record Identity

Because MatchUp Object does not read or write directly to the database, some mechanism must be provided so that the application can match each record back to the original data source. The **SetUserInfo** function allows the application to pass an unique identifier for each record.

Read/Write Order of Operations

Using the Read/Write deduper is pretty straight forward. This section will outline the basic steps and then show an example of the programming logic for a typical implementation of the Read/Write deduper.

1. Initialize the Read/Write deduper.

   After creating an instance of the Read/Write deduper, point the object toward its supporting data file, select a matchcode and key file to use, and initialize these files.
2. Create field mappings.

   In order to build a key to be written to the key file, the Read/Write deduper needs to know which types of data the application will be passing to the deduper and in what order.
3. Read the records from the database.

   Loop through the master database and get the data fields needed to build a key, according to the mappings defined in step 2.
4. Build a match key for each record.

   This consists of passing the actual data to the deduper in the same order used when creating the field mapping. After passing the necessary fields (usually a small subset of the fields from each record) via the **AddField** function, the deduper uses this information to generate a match key.
5. Write each match key to the key file.

   The **WriteRecord** function stores each match key in a temporary key file.
6. Process the keys.

   After building the keys, calling the **Process** function loops through the keys and compares them to each other.
7. Loop through the records and read the deduping data for each one.

   The **ReadRecord** function loops through the entire set of deduped records and allows the application to read information on the record's duplicate/unique status, the number of duplicates for each record and the record dupe group.

The following section outlines a common implementation of the Read/Write deduper, using pseudocode for maximum clarity. Working sample programs in several programming languages can be found on the MatchUp Object install disc and more can be downloaded from the MatchUp Object support page on the Melissa Data website.

Step 1: Initialize the Read/Write deduper

After creating an instance of the Read/Write deduper, point the object toward its supporting data file, select a matchcode and key file to use, and initialize these files.

First, create a new instance of the Read/Write deduper.

```
SET mu = NEW mdMUReadWrite
```

In order to successfully initialize this new instance, the application must point it toward its data files and supply a valid license string.

```
CALL mu.SetLicenseString with LicenseString

CALL mu.SetPathToMatchUpFiles with PathToMatchUpFiles
```

Before initialization, the application must specify which matchcode and key file will be used for the current deduping operation.

```
CALL mu.SetMatchcodeName with MatchCodeName

CALL mu.SetKeyFile with PathToKeyFile
```

If all of the above have been set correctly, calling the InitializeDataFiles function should return a value of ErrorNone. If it does not, call the GetInitializeErrorString function to determine the reason for the failure to initialize.

```
CALL mu.InitializeDataFiles RETURNING ProgramStatusResult

IF ProgramStatusResult is not 0 THEN
    PRINT "Initialization Error: " + mu.GetInitializeErrorString
    EXIT ROUTINE
END IF
```

If the initialization was successful, the application can call the following functions to display version and expiration information about the instance of MatchUp Object currently in use on the local computer.

```
PRINT "Confirming Initialization: " + mu.GetInitializeErrorString

PRINT "Build Number: " + mu.GetBuildNumber

PRINT "Database Date: " + mu.GetDatabaseDate

PRINT "Database Expiration Date: " + mu.GetDatabaseExpirationDate

PRINT "License Expiration Date: " + mu.GetLicenseExpirationDate
```

Step 2: Create field mappings

Field mappings define which types of data the Read/Write deduper is expecting. In this case, the selected matchcode looks for a five-digit ZIP Code, a first name, a last name and a street address.

```
CALL mu.ClearMappings
```

After clearing any mappings from a previous use of the Read/Write deduper, call the **AddMapping** function once for each field being considered.

```
CALL mu.AddMapping with mu.Zip5 RETURNING mapOK

CALL mu.AddMapping with mu.First RETURNING mapOK

CALL mu.AddMapping with mu.Last RETURNING mapOK

CALL mu.AddMapping with mu.Address RETURNING mapOK
```

Step 3: Loop through database records and build keys

The Read/Write deduper builds a temporary key file out of the data from the database. To do this, the application loops through each record and pulls the data from the fields that match the mappings made above.

```
FOR EACH Record in database

    Read Zip5, FirstName, LastName, StreetAddress, userInfo fields from
    database
```

After pulling the data from the database, pass it to the Read/Write deduper with the **AddField** function. The application must do this in the same order that it mapped the data types in the step above.

Even if the fields in a database do not exactly match the components required by the matchcode, MatchUp Object is able to extract only the information it needs. For example, if the database only contained a full name field, that field could be passed twice and MatchUp Object would recognize the first names and last names and only use the parts it needed.

After passing each set of fields, call the **BuildKey** function to create the match key according to the mappings and the current matchcode.

```
CALL mu.ClearFields

CALL mu.AddField with Zip5

CALL mu.AddField with FirstName

CALL mu.AddField with LastName

CALL mu.AddField with StreetAddress

CALL mu.BuildKey
```

The UserInfo is a unique identifier for each record. The application will need this to later match the deduping information to the original records.

```
CALL mu.SetUserInfo with userInfo
```

The **WriteRecord** function adds the current key and UserInfo to the key field.

```
CALL mu.WriteRecord
```

Repeat for every record in the current data set.

```
NEXT Record
```

Step 4: Begin processing the records

The **Process** function switches the Read/Write deduper from writing new keys to comparing the stored keys to each other.

```
CALL mu.Process
```

Step 5: Examine the processed records

At this point, loop through the processed records, and get information on each record's unique/duplicate status and how many duplicates of each record exist in the data set.

WHILE mu.ReadRecord does not return 0

Each call to the **ReadRecord** function advances the deduper to the next record and populates the fields returned by the functions below.

```
PRINT "Record: " + mu.GetUserInfo
PRINT "Key: " + mu.GetKey
PRINT "Dupe Group: " + mu.GetDupeGroup
PRINT mu.GetCount + " records in this dupe group."
PRINT "This is record #" + mu.GetEntry + " in this dupe group."
```

The Results property indicates whether the record is unique, a record with duplicates, or a duplicate of another record.

```
CASE mu.GetResults Contains
    MS03 :PRINT "This record is a duplicate."
    MS02 :PRINT "This record has duplicates."
    MS01 :PRINT "This record is unique."
ENDCASE
```

The result codes returned by the **GetResults** function also indicate which combination or combinations defined by the matchcode produced the hit. In addition to the status code, other potential result codes correspond to a specific combination number, so the application needs to use logical AND operation for each bit to actually make use of this information.

```
CALL mu.GetResults Contains
MS06 Match: Rule 1   Matched another record by matchcode combination 1
```

```
MS07 Match: Rule 2  Matched another record by matchcode combination 2
MS08 Match: Rule 3  Matched another record by matchcode combination 3
MS09 Match: Rule 4  Matched another record by matchcode combination 4
MS10 Match: Rule 5  Matched another record by matchcode combination 5
MS11 Match: Rule 6  Matched another record by matchcode combination 6
MS12 Match: Rule 7  Matched another record by matchcode combination 7
MS13 Match: Rule 8  Matched another record by matchcode combination 8
MS14 Match: Rule 9  Matched another record by matchcode combination 9
MS15 Match: Rule 10 Matched another record by matchcode combination 10
MS16 Match: Rule 11 Matched another record by matchcode combination 11
MS17 Match: Rule 12 Matched another record by matchcode combination 12
MS18 Match: Rule 13 Matched another record by matchcode combination 13
MS19 Match: Rule 14 Matched another record by matchcode combination 14
MS20 Match: Rule 15 Matched another record by matchcode combination 15
MS21 Match: Rule 16 Matched another record by matchcode combination 16
```

Read/Write Deduping Functions

The following is a master list of the function in the Read/Write deduper interface.

Initialize the Read/Write Interface

The following functions prepare the Read/Write deduper for use and link it to its supporting data files.

SetPathToMatchUpFiles 44
SetLicenseString 44
SetMatchcodeName 46
SetMatchcodeObject 46
SetKeyFile 47
SetGroupSorting 47
InitializeDataFiles 48
GetInitializeErrorString 49
GetBuildNumber 49
GetDatabaseDate 50
GetDatabaseExpirationDate 50
GetLicenseExpirationDate 51

Map Database Fields

Before generating match keys for the database records, the application must supply the Read/Write deduper with information about what sort of data it will be handling.

ClearMappings 51
AddMapping 52

Read Data and Build the Match Key

The following functions take the real data being compared and construct a match key according to the mappings defined with the above functions and the matchcode defined when the Read/Write deduper was initialized.

ClearFields 53
AddField 53
BuildKey 54
SetKey 54
SetUserInfo 55
WriteRecord 55

Process Records

This single function takes every record passed via the WriteRecord function and determines which are unique, which have duplicates, and which are duplicates.

Process 56

Retrieve Dupe Data for Each Record

The functions in this section cycle through each record processed and return output unique/duplicate information.

ReadRecord 56
GetStatusCode 57
GetCount 57
GetDupeGroup 57
GetEntry 58
GetCombinations 58
GetKey 59
GetUserInfo 59
GetResults 59

Initialize the Read/Write Interface

The following functions prepare the Read/Write deduper for use and link it to its supporting data files.

SetPathToMatchUpFiles

String value. This function accepts a string value containing the path to the folder containing the MatchUp Read/Write data files. It must be called before calling the InitializeDataFiles function.

To provide maximum compatibility with Windows, three files are installed in your 'Common App Data' directory. For Windows Vista and Windows 7 the default location is "C:\ProgramData\MelissaDATA\MatchUp." For Windows XP the default location is "C:\Documents and Settings\All Users\Application Data\Melissa DATA\MatchUp." The location of this directory can be changed by users so please note this, as it can often be the source of issues when running the samples/demos.

Syntax

```
mdMU->SetPathToMatchUpFiles(char)
```

C

```
mdMUReadWriteSetPathToMatchUpFiles(mdMU, char)
```

COM+/.NET

```
mdMU.PathToMatchUpFiles = string
```

SetLicenseString

Passes the license string required for MatchUp Object to function. Required only if the environment variable method is not used.

Each customer is issued a license string when purchasing MatchUp Object or renewing a subscription. This string must be passed to this function to unlock the functionality of MatchUp Object.

The license string is normally set using an environment variable, either MD_LICENSE or MD_LICENSE_DEMO. Calling SetLicenseString is an alternative method for setting the license string, but applications developed for a production environment should only use the environment variable.

When using an environment variable, it is not necessary to call the **SetLicenseString** function.

For more information on setting the environment variable, see "Entering Your MatchUp Object License" on page 4.

Using an environment variable makes it much easier to update the license string without having to edit and re-compile the application.

When using an environment variable, it is still necessary to call the **SetLicenseString** function, but it is not necessary to pass the license string to the function. Instead, simply call the function and pass an empty string as the parameter.

Windows

Windows users can set environment variables by doing the following:

- Select **Start** > **Settings**, and then click **Control Panel**.
- Double-click **System**, and then click the **Advanced** tab.
- Click **Environment Variables**, and then select either **SystemVariables** or **Variables for the user X**.
- Click New.
- Enter “MDMATCHUP_LICENSE” in the Variable Name box.
- Enter the license string in the Variable Value box and then click **OK**.

Please remember that these settings take effect only upon start of the program. It may be necessary to quit and restart the development environment to incorporate the changes.

Linux/Solaris/HP-UX/AIX

Unix-based OS users can simply set the license string via the following:
export MDMATCHUP_LICENSE=A1B2C3D4E5 (not the actual license string).
After putting this setting in the profile, remember to restart the shell.

Input Parameters

A string value containing an empty string or, if necessary, a valid license string for MatchUp Object.

Return Values

This function returns an integer value. A value of 1 indicates a valid license string, 0 an invalid or empty string,

Syntax

```
int = mdMU->SetLicenseString(char)
```

C

```
int = mdMUReadWriteSetLicenseString(mdMU, char)
```

COM+/.NET

```
integer = mdMU.SetLicenseString string
```

SetMatchcodeName

This function selects the matchcode to use for the current Read/Write deduping operation. The **SetMatchcodeName** function accepts a string value that must match the name of an existing matchcode in the current matchcode file.

Syntax

```
    mdMU->SetMatchcodeName(char)
```

C

```
    mdMUReadWriteMatchcodeName(mdMU, char)
```

COM+/.NET

```
    mdMU.MatchcodeName = string
```

SetMatchcodeObject

This functions selects the matchcode to use for the current Read/Write deduping operation. It largely duplicates the purpose of the **SetMatchcodeName** function, but instead of accepting a character value containing the name of a matchcode in the current matchcode file, this function accepts a Matchcode object created using the Matchcode Editing interface.

Because this function requires the use of a separate interface to create the Matchcode object variable, it is usually simpler to use the **SetMatchcodeName** function.

It is possible, however, to use this function to build a new matchcode on the fly using the Matchcode Editing interface. Unless a specific application demands such flexibility, it is usually much simpler to add a new matchcode to the matchcode file and call it using the **SetMatchcodeName** function.

Syntax

```
    mdMU->SetMatchcodeObject(mdMUMatchcode)
```

C

```
    mdMUReadWriteSetMatchcodeObject(mdMU, mdMUMatchcode)
```

COM+/.NET

```
    mdMU.MatchcodeObject = mdMUMatchcode
```

SetKeyFile

This function selects the name and file path for the key file that will be used for the current Read/Write deduping operation.

Every instance of the Read/Write deduper creates a new key file for each session. Any existing key file with the same name is overwritten. If more than one instance of the Read/Write deduper is running on either the same computer or the same network, make certain that they do not point to the same key file. If one instance overwrites the key file being used by another instance, it can cause the second instance to fail.

Syntax

```
mdMU->SetKeyFile(char)
```

C

```
mdMUReadWriteSetKeyFile(mdMU, char)
```

COM+/.NET

```
mdMU.KeyFile = string
```

SetGroupSorting

This function sets the Read/Write deduper to return processed records in dupe group order. By default, the Read/Write deduper returns records sorted by their match key. This should return records in the same dupe group together or close to each other.

Passing a boolean True value to this function will cause the Read/Write deduper to return the processed records sorted into dupe groups.

The additional processing can increase the time needed to dedupe a large list and it is often possible to use the information returned by the Read/Write deduper to sort records into this order programmatically.

Syntax

```
mdMU->SetGroupSorting()
```

C

```
mdMUReadWriteSetGroupSorting(mdMU)
```

COM+/.NET

```
mdMU.GroupSorting
```

InitializeDataFiles

The InitializeDataFiles function opens the needed data files and prepares the MatchUp Object for use. Before calling this function, the application must have successfully called the SetLicenseString and SetPathToMatchUpFiles functions.

Check the return value of the GetInitializeErrorString function to retrieve the result of the initialization call. Any result other than "No Error" means the initialization failed for some reason.

Return Value

Returns a value of the enumerated type ProgramStatus.

Value		Reason
0	ErrorNone	No error - initialization was successful.
1	ErrorConfigFile	Could not find mdMatchUp.dat.
2	ErrorLicenseExpired	The License String has expired.
3	ErrorDatabaseExpired	The database has expired.
4	ErrorMatchcodeNotSpecified	No matchcode was specified.
5	ErrorMatchcodeNotFound	Specified Matchcode does not exist.
6	ErrorInvalidMatchcode	The specified matchcode is not valid.
7	ErrorKeyFile	The specified key file was not found.

If any other value other than NoError is returned, check the GetInitializeErrorString function to see the reason for the error.

Syntax

```
ProgramStatus = mdMU->InitializeDataFiles()
```

C

```
int = mdMUReadWriteInitializeDataFiles(mdMU)
```

COM+/.NET

```
ProgramStatus = mdMU.InitializeDataFiles
```

GetInitializeErrorString

Returns a descriptive string to describe the error from the InitializeDataFiles function.

The possible strings returned by this function are:

```
"No error"
"Could not find mdMatchUp.dat."
"The License String has expired."
"The database has expired."
"No matchcode was specified."
"Specified Matchcode does not exist."
"The specified matchcode is not valid."
"The specified key file was not found."
```

Return Value

The **GetInitializeErrorString** function returns a string describing the error caused when the **InitializeDataFiles** function cannot be called successfully.

Syntax

```
char = mdMU->GetInitializeErrorString()
```

C

```
char = mdMUReadWriteGetInitializeErrorString(mdMU)
```

COM+/.NET

```
string = mdMU.GetInitializeErrorString
```

GetBuildNumber

The **GetBuildNumber** function returns the current development release build number of MatchUp Object.

Input Parameters

None.

Return Value

The **GetBuildNumber** function returns the current development release build number of the MatchUp Object.

Syntax

```
char = mdMU->GetBuildNumber()
```

C

```
char = mdMUReadWriteGetBuildNumber(mdMU)
```

COM+/.NET

```
string = mdMU.GetBuildNumber
```

GetDatabaseDate

The GetDatabaseDate function returns a string value that represents the revision date of the MatchUp Object data files.

Input Parameters

None.

Return Value

The GetDatabaseDate function returns a string value that represents the date of the MatchUp Object data files.

Syntax

```
    char = mdMU->GetDatabaseDate()
```

C

```
    char = mdMUReadWriteGetDatabaseDate(mdMU)
```

COM+/.NET

```
    string = mdMU.GetDatabaseDate
```

GetDatabaseExpirationDate

Returns a string value containing the expiration date of the current database file.

Input Parameters

None.

Return Value

Returns a string value indicating the expiration date of the current database file (mdMatchUp.dat).

Syntax

```
    char = mdMU->GetDatabaseExpirationDate()
```

C

```
    char = mdMUReadWriteGetDatabaseExpirationDate(mdMU)
```

COM+/.NET

```
    string = mdMU.GetDatabaseExpirationDate
```

GetLicenseExpirationDate

Returns a string value containing the expiration date of the current license string. After this date, MatchUp Object will no longer function.

Input Parameters

None.

Return Value

Returns a string value that indicates the expiration date of the license string.

Syntax

```
    char = mdMU->GetLicenseExpirationDate()
```

C

```
    char = mdMUReadWriteGetLicenseExpirationDat(mdMU)
```

COM+/.NET

```
    string = mdMU.GetLicenseExpirationDate
```

Map Database Fields

Before generating match keys for the database records, the application must supply the Read/Write deduper with information about what sort of data it will be handling.

ClearMappings

This function clears any existing field mappings. It is a good idea to call this function before beginning to map fields, especially if the application may be required to perform multiple deduping operations in a single session.

Syntax

```
    mdMU->ClearMappings()
```

C

```
    mdMUReadWriteClearMappings(mdMU)
```

COM+/.NET

```
    mdMU.ClearMappings
```

AddMapping

This function selects the types of fields that will be used to build the match key and the order in which they will be added using the **AddField** function.

The function accepts an enumerated value of the type MatchcodeMapping. It tells the Read/Write deduper which data types will be used for this deduping operation and in what order they will be passed to the deduper when passing data using the **AddField** function.

The data types used must contain the data expected by the matchcode being used, but it does not have to be an exact match. For example, if the matchcode requires a five-digit ZIP Code but the data in the list uses a single "City/State/ZIP" field, simply add the CityStZip mapping and pass the full string to the **AddField** function later. MatchUp Object is smart enough to use only the information it needs.

In another example, a matchcode calls for both last name and first name but database contains only full names. The application would simply apply the FullName mapping twice and pass the full name data twice to the **AddField** function.

To demonstrate the above:

```
mdMU->AddMapping(mdMU.CityStZip)    // uses only ZIP Code
mdMU->AddMapping(mdMU.FullName)    // uses last name only
mdMU->AddMapping(mdMU.FullName)   // uses first name only
mdMU->AddMapping(mdMU.Address)
```

For a list of the possible values, see "MatchcodeMapping" on page 148.

The function returns a non-zero value if the mapping is allowed by the selected matchcode, false if the mapping caused an error.

Syntax

```
int = mdMU->AddMapping(mdMU.MatchcodeMapping)
```

C

```
int = mdMUReadWriteAddMapping(mdMU,
   mdMU.mdMatchUpMatchmodeMapping)
```

COM+/.NET

```
integer = mdMU.AddMapping(mdMU.MatchcodeMapping)
```

Read Data and Build the Match Key

The following functions take the real data being compared and construct a match key according to the mappings defined with the above functions and the matchcode defined when the Read/Write deduper was initialized.

ClearFields

This function clears all values from previous calls to the **AddField** or **ReadRecord** function. The application should call this function after calling the WriteRecord function, before the first call to the AddField function, or before each call to the ReadRecord function.

Syntax

```
mdMU->ClearFields()
```

C

```
mdMUReadWriteClearFields(mdMU)
```

COM+/.NET

```
mdMU.ClearFields
```

AddField

This function passes the contents of a field from a database to the deduper prior to calling the BuildKey function.

Fields must be passed to this function in the same order that the corresponding data types were mapped using the AddMapping function.

The following example expands on the previous AddMapping example. The matchcode uses five-digit ZIP codes, last and first names, in that order, and the street addresses. The list includes only a single “City/ST/ZIP" and a single full name field.

```
mdMU->AddField("Rancho Santa Margarita, CA 92688")
mdMU->AddField("Raymond F. Melissa")
mdMU->AddField("Raymond F. Melissa")
mdMU->AddField("22382 Avenida Empresa")
```

The deduper would use only the ZIP Code from the first field, the last name from the second AddField and first name from the third AddField.

Syntax
`mdMU->AddField(char)`
C
`mdMUReadWriteAddField(mdMU, char)`
COM+/.NET
`mdMU.AddField(string)`

BuildKey

This function builds a match key using information passed via the **AddField** function. The information passed via calls to the **AddField** function and, using the mapping defined by the **AddMapping** function and the pattern defined by the matchcode being used, builds a match key.

A match key is a character string built according to a pattern defined by the current matchcode, consisting only of enough information to determine if the current record is unique or has a duplicate within the key file.

For example, let's assume the matchcode called for a five-digit ZIP Code, first ten characters of a last name, first ten of a first name, a street number and the first ten characters of a street name. The current record is for Raymond F. Melissa at 22382 Avenida Empresa in the 92688 ZIP Code. The match key would be:

```
92688MELISSA   RAYMOND   22382EMPRESA
```

Because "Empresa" is only seven characters, the key would be padded with three spaces at the end.

Syntax
`mdMU->BuildKey()`
C
`mdMUReadWriteBuildKey(mdMU)`
COM+/.NET
`mdMU.BuildKey`

SetKey

This function accepts a match key before calling the **ReadRecord** function.

The **BuildKey** function creates a key from input data. If, however, the match keys are already stored in the source database, use this function to pass the keys to the deduper before calling MatchRecord.

Syntax

```
mdMU->SetKey(char)
```

C

```
mdMUReadWriteSetKey(mdMU, char)
```

COM+/.NET

```
mdMU.Key = string
```

SetUserInfo

This function accepts a character value that uniquely identifies each record in a set of data.

The character value passed to this function must be unique for every record. This enables the application to associate the match key in the key file to the corresponding record in the list.

Syntax

```
mdMU->SetUserInfo(char)
```

C

```
mdMUReadWriteSetUserInfo(mdMU, char)
```

COM+/.NET

```
mdMU.SetUserInfo = string
```

WriteRecord

This function creates a record of the current key and user info and writes it to the key file.

The WriteRecord function requires that either the **BuildKey** or **SetKey** function, plus the SetUserInfo function, have previously been called. This function writes the information stored by those functions to a new record in the current key file.

The application cannot call this function after the Process function has been called.

Syntax

```
mdMU->WriteRecord()
```

C

```
mdMUReadWriteWriteRecord(mdMU)
```

COM+/.NET

```
mdMU.WriteRecord
```

Process Records

This single function takes every record passed via the WriteRecord function and determines which are unique, which have duplicates, and which are duplicates.

Process

This function switches the Read/Write deduper from write mode to read mode. After calling this function, the application can no longer add more records to the key file with the **WriteRecord** function.

The **Process** function will pass every record though the Read/Write deduping logic to determine which records are unique, which have duplicates, and which are duplicates.

Syntax

```
mdMU->Process()
```

C

```
mdMUReadWriteProcess(mdMU)
```

COM+/.NET

```
mdMU.Process
```

Retrieve Dupe Data for Each Record

The functions in this section cycle through each record processed and return output unique/duplicate information.

ReadRecord

This function reads the next line from the key file and populates the relevant fields with duplicate status information.

The ReadRecord function reads the next record from the key file, if there is another record, and populates the fields used by the following functions: **GetResults**; **GetCount**; **GetDupeGroup**; **GetEntry**; **GetKey**; **GetUserInfo**.

This function cannot be called until after the application has called the Process function.

If there are no more records, this returns an integer value of zero.

Syntax

```
    int = mdMU->ReadRecord()
```

C

```
    int = mdMUReadWriteReadRecord(mdMU)
```

COM+/.NET

```
    integer = mdMU.ReadRecord
```

GetStatusCode

This function is deprecated. You should use the GetResults function instead.

GetCount

This function returns an integer value indicating the total number of matching records in this dupe group.

If there were matches detected during processing, the GetCount function will return a integer value equalling the number of duplicate keys found.

Syntax

```
    int = mdMU->GetCount()
```

C

```
    int = mdMUReadWriteGetCount(mdMU)
```

COM+/.NET

```
    integer = mdMU.Count
```

GetDupeGroup

This function returns a long integer value indicating the group of duplicate records that the current key matches.

Every unique record (one with no duplicates) will have a unique “Dupe Group” number. Any duplicate record will be assigned the same number. This function returns the Dupe Group number of a matching record in the key file.

Syntax

```
    long = mdMU->GetDupeGroup()
```

C

```
    long = mdMUReadWriteGetDupeGroup(mdMU)
```

COM+/.NET

```
    long = mdMU.DupeGroup
```

GetEntry

Returns an integer value indicating where the current record would fall within the order of its dupe group.

If the **ReadRecord** function detected at least one duplicate, this function will return an integer value that indicates where the current record falls within its dupe group. If this is the sixth matching record found, this function will return a 6.

Syntax

```
int = mdMU->GetEntry()
```

C

```
int = mdMUReadWriteGetEntry(mdMU)
```

COM+/.NET

```
integer = mdMU.Entry
```

GetCombinations

This function is deprecated. You should use the GetResults function instead.

This function returns a long integer value that can be used to determine which combinations defined in the current matchcode were matched by the last record processed by the ReadRecord function.

Each matchcode may contain as many as 16 different combinations of data types that may be used to detect a match. A matching record may match more than one combination.

This function returns a long integer that can be used to determine which combination produced the match, if the ReadRecord function detected a matching key.

Syntax

```
long = mdMU->GetCombinations()
```

C

```
long = mdMUReadWriteGetCombinations(mdMU)
```

COM+/.NET

```
long = mdMU.Combinations
```

GetKey

This function returns the match key used by the last call to the ReadRecord function.

The **GetKey** returns the match key created by the last call to the **BuildKey** function and used by the last call to the **ReadRecord** function.

Syntax

```
    char = mdMU->GetKey()
```

C

```
    char = mdMUReadWriteGetKey(mdMU)
```

COM+/.NET

```
    string = mdMU.Key
```

GetUserInfo

This function returns a character value containing the value passed to the SetUserInfo function. It returns the unique identifier associated with the record being checked by the Read/Write deduper.

The application will need this information if the application has to match the current matchkey back to an original data source.

Syntax

```
    char = mdMU->GetUserInfo()
```

C

```
    char = mdMUReadWriteGetUserInfo(mdMU)
```

COM+/.NET

```
    string = mdMU.UserInfo
```

GetResults

This function returns a comma-delimited string of four-character codes that detail the output disposition of the last call to the ReadRecord function. It will also contain the result code of any matchcode combination which contributed to the present record matching other records in its dupe group.

The **GetResults** function is intended to replace the **GetStatusCode** and **GetCombinations** functions, providing a single source of information about the last MatchRecord function call and eliminating the need to perform bitwise operations on the GetCombinations return value to determine which matchcode combinations contributed to the record matching other records in its Dupe Group.

The function returns one or more of the following codes in a comma-delimited list:

MatchUp Object: Result Codes

Code	Short Description	Long Description
MS01	Unique Record	The reocrd did not match any other records.
MS02	Has Duplicates	The record matched other records and was tagged as the output record.
MS03	Is Duplicate	The record matched other records and was tagged as a duplicate.
MS04	Record Suppressed	The source record was suppressed.
MS05	Record Not Intersected	The source record was not intersected.
MS06	Match: Rule 1	Records were matched by matchcode combination 1.
MS07	Match: Rule 2	Records were matched by matchcode combination 2.
MS08	Match: Rule 3	Records were matched by matchcode combination 3.
MS09	Match: Rule 4	Records were matched by matchcode combination 4.
MS10	Match: Rule 5	Records were matched by matchcode combination 5.
MS11	Match: Rule 6	Records were matched by matchcode combination 6.
MS12	Match: Rule 7	Records were matched by matchcode combination 7.
MS13	Match: Rule 8	Records were matched by matchcode combination 8.
MS14	Match: Rule 9	Records were matched by matchcode combination 9.
MS15	Match: Rule 10	Records were matched by matchcode combination 10.
MS16	Match: Rule 11	Records were matched by matchcode combination 11.
MS17	Match: Rule 12	Records were matched by matchcode combination 12.
MS18	Match: Rule 13	Records were matched by matchcode combination 13.
MS19	Match: Rule 14	Records were matched by matchcode combination 14.
MS20	Match: Rule 15	Records were matched by matchcode combination 15.
MS21	Match: Rule 16	Records were matched by matchcode combination 16.
MS30	Suppressor Record	The lookup record suppressed a source record.

MatchUp Object: Result Codes		
MS31	Intersector Record	The lookup record intersected a source record.

Syntax

```
    StringValue = object->GetResults();
```

C

```
    StringValue = mdMatchUpGetResults(object);
```

COM

```
    StringValue = object.Results
```

Incremental Deduping

Incremental deduping is usually used for real-time data entry validation. For example, a call center data-entry system where an operator would like to determine whether or not the caller is an existing customer. At any time, a calling program can pass the incremental deduping engine the contents of a record; the engine will then report as to whether or not this record is a dupe, and if so, which record or records it matches.

Incremental deduping consists of the following steps:

1. The program processes a record and sends the specific information (ZIP/PC, Name, Address, etc) to MatchUp Object.
2. Based on previous records sent to the API, it reports whether or not the record from the first step matches any of these previous records.
3. Optionally, the application can tell MatchUp Object to add this record to its database for consideration in future comparisons.

The Historical Database

The incremental deduping engine relies heavily on a historical database that it maintains. The lifetime of this database is as long as necessary (seconds, days, even years). This database is constructed and maintained by MatchUp Object, so it can determine whether or not an incoming record matches other records fairly quickly.

Multi-User/Multi-Thread Considerations

Incremental deduping is unique in that multiple users or multiple processes can access the same historical database simultaneously. The API maintains a locking system to ensure that competing processes don't collide. In order for two processes to work in this fashion, the initialization function for each process must specify the same historical database (a.k.a. “key file”).

Transaction-Based Processing

The Incremental deduper interface of MatchUp Object features the option of using transaction-based operations on the historical database. This enables an application to process multiple calls to the AddRecord function as one, speeding up processing of large lists.

Incremental Order of Operations

Using the Incremental deduper is pretty straightforward. This section will outline the basic steps and then show an example of the programming logic for a typical implementation of the Incremental deduper.

1. Initialize the Incremental deduper.

   After creating an instance of the Incremental deduper, point the object toward its supporting data file, select a matchcode and key file to use, and initialize these files.

2. Create field mappings.

   In order to build a key to compare to the key file, the Incremental deduper needs to know which types of data the program will be passing to the deduper and in what order.

3. Read the record from the data source.

   This can be a new address passed from a website, a single record from a newly acquired list or data source, to be compared against the master list.

4. Build a match key for the incoming record.

   This consists of passing the actual data to the deduper in the same order used when creating a field mapping. After passing the necessary fields (usually a small subset of the fields from each record) via the AddField function, the Incremental deduper uses this information to generate a match key.

5. Compare the match key to the key file.

   The **MatchRecord** function searches the key file for any keys that match the new record. If it finds a match, it provides information on the duplicate records in the key file.

6. Write new records to the key file.

   The new key, whether or not it is unique, can then be written to the key file, so it can be used for future deduping operations. The program code must also write the new address record to the database separately.

The following section outlines a common implementation of the Incremental deduper. We are using pseudocode for maximum clarity. Working sample programs in several programming languages can be found on the MatchUp Object install disc and more can be downloaded from the support page on the Melissa Data website.

Step 1: Initialize the Incremental deduper

After creating an instance of the Incremental deduper, point the object toward its supporting data file, select a matchcode and key file to use, and initialize these files.

First, create a new instance of the Incremental deduper.

```
SET mu = NEW mdMUIncremental
```

In order to successfully initialize this new instance, point it toward its data files and supply a valid license string.

```
CALL mu.SetLicenseString with LicenseString

CALL mu.SetPathToMatchUpFiles with PathToMatchUpFiles
```

Before initialization, specify which matchcode and key file will be used for the current deduping operation.

```
CALL mu.SetMatchcodeName with MatchCodeName

CALL mu.SetKeyFile with PathToKeyFile
```

If all of the above have been set correctly, calling the InitializeDataFiles function should return a value of NoError. If it does not, call the GetInitializeErrorString function to determine the reason for the failure to initialize.

```
CALL mu.InitializeDataFiles RETURNING initResult

IF initResult is not NoError THEN
    PRINT "Initialization Error: " + mu.GetInitializeErrorString
    EXIT PROGRAM
END IF
```

If the initialization was successful, call the following functions to display version and expiration information about the instance of MatchUp Object currently in use on the local computer.

```
PRINT "Confirming Initialization: " + mu.GetInitializeErrorString
PRINT "Build Number: " + mu.GetBuildNumber
PRINT "Database Date: " + mu.GetDatabaseDate
PRINT "Database Expiration Date: " + mu.GetDatabaseExpirationDate
PRINT "License Expiration Date: " + mu.GetLicenseExpirationDate
```

Step 2: Create field mappings

Field mappings define a valid incoming type of data for each matchcode component. For example, a typical matchcode may include a five-digit ZIP Code, a last name, and a street address. But the data coming in, however, may contain the city, state, and ZIP as a single character field and the person's full name as a single field as well.

Even if the fields in a database do not exactly match the components required by the matchcode, MatchUp Object is able to extract only the information it needs.

```
CALL mu.ClearMappings
```

After clearing any mappings from a previous use of the Incremental deduper, call the AddMapping function once for each field being considered.

```
CALL mu.AddMapping with mu.CityStZip RETURNING mapOK
CALL mu.AddMapping with mu.FullName RETURNING mapOK
CALL mu.AddMapping with mu.Address RETURNING mapOK
```

Step 3: Get the record from the data source

Regardless of the source, the object only needs to read the fields containing the data that the Incremental deduper needs for comparison.

```
READ Record FROM database RETURNING userInfo, CityStateZip,
    TheFullName, StreetAddress
```

The userInfo field is any identifying character string that is unique to the current record.

Note that the Incremental deduper does not handle database input and output. This must be done programmatically using whatever database interface is being used.

Step 4: Build the match key for the current record

We'll use the data from the previous step to construct a match key to use in the next step.

```
CALL mu.ClearFields
```

After clearing any data from a previous use of the Incremental deduper, call the **AddField** function once for each field being considered.

```
CALL mu.AddField with CityStateZip
CALL mu.AddField with TheFullName
CALL mu.AddField with StreetAddress
```

Pass the userInfo to the deduper using the **SetUserInfo** function.

```
CALL mu.SetUserInfo with userInfo
```

The **BuildKey** function constructs the match key out of the information passed via the **AddField** function calls.

```
CALL mu.BuildKey
```

Step 5: Compare the match key to the key file

The **MatchRecord** function compares the match key to the key file and determines if the key already exists in the key file.

```
CALL mu.MatchRecord
```

Check the **GetResults** function to determine if the **MatchRecord** call produced a match, meaning that the current record is a duplicate to another one in the key file.

```
CALL mu.GetResults RETURNING ResultsCodes
```

If the record is a duplicate, this code retrieves information about the other duplicate records in the database.

```
IF ResultsCodes contains "MS03" THEN // Record is a duplicate
```

```
        PRINT "This record is in the database."
        PRINT "There are " + mu.GetCount + " records in dupe group #" +
            mu.GetDupeGroup
        PRINT "It matches these records:"
        WHILE mu.NextMatchingRecord
        PRINT "#" + mu.GetEntry + " is Record: " + mu.GetUserInfo
    ENDWHILE
```

“Dupe Group” is a number that gets assigned to each unique record. Duplicate records are assigned the same number.

Step 6: Add the new key to the key file

In this example, duplicate records are being rejected while unique records are being added to the database. Depending on the end user needs, the program may handle duplicate records differently.

```
    ELSE
        CALL mu.AddRecord
        Add New Record to master database
    ENDIF
```

The **AddRecord** function only adds the new key to the key file. To add the data to the database, the developer would need to implement that in code, according to whatever database engine is in use.

Using the Transaction Functions

The transaction functions allow MatchUp Object to delay writing changes to the historical database until all records have been compared and all duplicates detected. After a call to the **BeginTransaction** function, the Incremental deduper will cache all calls to the **AddRecord** function until a call to the **CommitTransaction** function, which writes all changes to the keyfile in a single operation, significantly speeding up processing.

If any errors are detected, the **RollbackTransaction** function flushes all **AddRecord** function calls since the call to the **BeginTransaction** function and no changes will be written to the historical database.

Below is a simplified example of how transactions work with the Incremental deduper.

```
    CALL BeginTransaction
    FOR EACH Record in DatabaseTable
        READ Record
        Build Match Key
        CALL MatchRecord
        IF Record Is Unique THEN
            CALL AddRecord
        ENDIF
```

```
NEXT Record
IF ERROR
    CALL RollbackTransaction
    Return
ENDIF
CALL CommitTransaction
```

Incremental Deduping Functions

The following is a master list of the functions in the Incremental deduper interface.

Initialize the Incremental Interface

The following functions prepare the Incremental deduper for use and link it to its supporting data files.

SetPathToMatchUpFiles 69
SetLicenseString 69
SetMatchcodeName 71
SetMatchcodeObject 71
SetMustExist 72
SetKeyFile 72
InitializeDataFiles 73
GetInitializeErrorString 74
GetBuildNumber 74
GetDatabaseDate 75
GetDatabaseExpirationDate 75
GetLicenseExpirationDate 76

Map Database Fields

Before generating match keys for the records in the database, the code must supply the Incremental deduper with information about what sort of data it will be handling.

ClearMappings 76
AddMapping 77

Read Data and Build the Match Key

The following functions take the real data being compared and construct a match key according to the mappings defined with the above functions and the matchcode specified when the Incremental deduper was initialized.

ClearFields 78
AddField 78

BuildKey 79
SetKey 80
SetUserInfo 80

Compare Record to Database

The following functions compare the new key with the existing key file and, if a duplicate is found, return information about the duplicate records in the file.

MatchRecord 80
GetStatusCode 81
NextMatchingRecord 81
GetCount 82
GetDupeGroup 82
GetEntry 82
GetCombinations 83
GetKey 83
GetUserInfo 83
GetResults 83

Add New Record to Key File

The Incremental deduper does not handle reading or writing data to the database. This function only adds the new match key to the key file for future comparisons.

AddRecord 85

Transaction Methods

The following functions enable the Incremental deduper to use transactions, processing multiple calls to the AddRecord function before committing the changes to the key file.

BeginTransaction 86
CommitTransaction 86
RollbackTransaction 88

Initialize the Incremental Interface

The following functions prepare the Incremental deduper for use and link it to its supporting data files.

SetPathToMatchUpFiles

String value. This function accepts a string value containing the path to the folder containing the MatchUp data files. It must be called before calling the InitializeDataFiles function.

To provide maximum compatibility with Windows, three files are installed in your 'Common App Data' directory. For Windows Vista and Windows 7 the default location is "C:\ProgramData\MelissaDATA\MatchUp." For Windows XP the default location is "C:\Documents and Settings\All Users\Application Data\Melissa DATA\MatchUp." The location of this directory can be changed by users so please note this, as it can often be the source of issues when running the samples/demos.

Syntax

```
    mdMU->SetPathToMatchUpFiles(char)
```

C

```
    mdMUIncrementalSetPathToMatchUpFiles(mdMU, char)
```

COM+/.NET

```
    mdMU.PathToMatchUpFiles = string
```

SetLicenseString

Passes the license string required for MatchUp Object to function.

Each customer is issued a license string when purchasing MatchUp Object or renewing a subscription. This string must be passed to this function to unlock the functionality of MatchUp Object.

The license string is normally set using an environment variable, either MD_LICENSE or MD_LICENSE_DEMO. Calling SetLicenseString is an alternative method for setting the license string, but applications developed for a production environment should only use the environment variable.

When using an environment variable, it is not necessary to call the SetLicenseString function.

For more information on setting the environment variable, see "Entering Your MatchUp Object License" on page 4.

Using an environment variable makes it much easier to update the license string without having to edit and re-compile the application.

Windows

Windows users can set environment variables by doing the following:

- Select Start > Settings, and then click Control Panel.
- Double-click System, and then click the Advanced tab.
- Click Environment Variables, and then select either System Variables or Variables for the user X.
- Click New.
- Enter "MD_LICENSE" in the Variable Name box.
- Enter the license string in the Variable Value box and then click OK.

Please remember that these settings take effect only upon start of the program. It may be necessary to quit and restart the development environment to incorporate the changes.

Linux/Solaris/HP-UX/AIX

Unix-based OS users can simply set the license string via the following:

- export MD_LICENSE=A1B2C3D4E5
  (not the actual license string).

After putting this setting in the profile, remember to restart the shell.

Input Parameters

A string value containing an empty string or, if necessary, a valid license string for MatchUp Object.

Return Values

This function returns an integer value. A value of 1 indicates a valid license string, 0 an invalid or empty string,

Syntax

```
int = mdMU->SetLicenseString(char)
```

C

```
int = mdMUIncrementalSetLicenseString(mdMU, char)
```

COM+/.NET

```
integer = mdMU.SetLicenseString string
```

SetMatchcodeName

This function selects the matchcode to use for the current Incremental deduping operation. It accepts a string value that must match the name of an existing matchcode in the current matchcode file.

Syntax

```
    mdMU->SetMatchcodeName(char)
```

C

```
    mdMUIncrementalSetMatchcodeName(mdMU, char)
```

COM+/.NET

```
    mdMU.MatchcodeName = string
```

SetMatchcodeObject

This function selects the matchcode to use for the current Incremental deduping operation.

This function largely duplicates the purpose of the **SetMatchcodeName** function, but instead of accepting a character value containing the name of a matchcode in the current matchcode file, this function accepts a Matchcode object created using the Matchcode Editing interface.

Because this function requires a separate interface to create the Matchcode object variable, it is normally easier to use the **SetMatchcodeName** function.

It is possible, however, to use this function to build a new matchcode on the fly using the Matchcode Editing interface. Unless a specific application demands such flexibility, it is usually much simpler to add a new matchcode to the matchcode file using the Matchcode Editor and call it using the **SetMatchcodeName** function.

Syntax

```
    mdMU->SetMatchcodeObject(mdMUMatchcode)
```

C

```
    mdMUIncrementalSetMatchcodeObject(mdMU, mdMUMatchcode)
```

COM+/.NET

```
    mdMU.MatchcodeObject = mdMUMatchcode
```

SetMustExist

This function determines whether or not the path specified by the SetKeyFile function must point to an existing key file.

If this option is set to true, initialization of MatchUp Object will fail if the path specified in the **SetKeyFile** function does not point to an existing key file.

If this option is false, and the path specified in the **SetKeyFile** function does not point to an existing key file, a new empty key file will be created.

Syntax
`mdMU->SetMustExist(bool)`
C
`mdMUIncrementalSetMustExist(mdMU, bool)`
COM+/.NET
`mdMU.MustExist = boolean`

SetKeyFile

This function selects the name and file path for the key file that will be used for the current Incremental deduping operation.

If the **SetMustExist** function has been set to True, the string value passed to the **SetKeyFile** function must contain a valid path to an existing key file.

If the **SetMustExist** function has been set to False, MatchUp Object will create an empty key file if none is found during initialization.

Syntax
`mdMU->SetKeyFile(char)`
C
`mdMUIncrementalSetKeyFile(mdMU, char)`
COM+/.NET
`mdMU.KeyFile = string`

InitializeDataFiles

The InitializeDataFiles function opens the needed data files and prepares the MatchUp Object for use.

Before calling this function, the code must have successfully called the **SetLicenseString**, **SetMatchcodeName** (or **SetMatchcodeObject**) and **SetPathToMatchUpFiles** functions.

Check the return value of the **GetInitializeErrorString** function to retrieve the result of the initialization call. Any result other than “No Error” means the initialization failed for some reason.

Return Value

Returns a value of the enumerated type ProgramStatus.

Value		Reason
0	ErrorNone	No error - initialization was successful.
1	ErrorConfigFile	Could not find mdMatchUp.dat.
2	ErrorLicenseExpired	The License String has expired.
3	ErrorDatabaseExpired	The database has expired.
4	ErrorMatchcodeNotSpecified	No matchcode was specified.
5	ErrorMatchcodeNotFound	Specified Matchcode does not exist.
6	ErrorInvalidMatchcode	The specified matchcode is not valid.
7	ErrorKeyFile	The specified key file was not found.

If any other value other than ErrorNone is returned, check the **GetInitializeErrorString** function to see the reason for the error.

Syntax

```
    ProgramStatus = mdMU->InitializeDataFiles()
```

C

```
    ProgramStatus = mdMUIncrementalInitializeDataFiles(mdMU)
```

COM+/.NET

```
    ProgramStatus = mdMU.InitializeDataFiles
```

GetInitializeErrorString

Returns a descriptive string to describe the error from the **InitializeDataFiles** function. The possible strings returned by this function are:

```
No error
Could not find mdMatchUp.dat.
The License String has expired.
The database has expired.
No matchcode was specified.
Specified Matchcode does not exist.
The specified matchcode is not valid.
The specified key file was not found.
```

Return Value

The **GetInitializeErrorString** function returns a string describing the error caused when the **InitializeDataFiles** function cannot be called successfully.

Syntax

```
    char = mdMU->GetInitializeErrorString()
```

C

```
    char = mdMUIncrementalGetInitializeErrorString(mdMU)
```

COM+/.NET

```
    string = mdMU.GetInitializeErrorString
```

GetBuildNumber

The **GetBuildNumber** function returns the current development release build number of MatchUp Object.

Input Parameters

None.

Return Value

The **GetBuildNumber** function returns the current development release build number of the MatchUp Object.

Syntax

```
    char = mdMU->GetBuildNumber()
```

C

```
    char = mdMUIncrementalGetBuildNumber(mdMU)
```

COM+/.NET

```
    string = mdMU.GetBuildNumber
```

GetDatabaseDate

The GetDatabaseDate function returns a string value that represents the revision date of the MatchUp Object data files.

Input Parameters

None.

Return Value

The **GetDatabaseDate** function returns a string value that represents the date of the MatchUp Object data files.

Syntax

```
    char = mdMU->GetDatabaseDate()
```

C

```
    char = mdMUIncrementalGetDatabaseDate(mdMU)
```

COM+/.NET

```
    string = mdMU.GetDatabaseDate
```

GetDatabaseExpirationDate

Returns a string value containing the expiration date of the current database file.

Input Parameters

None

Return Value

Returns a string value indicating the expiration date of the current database file (mdMatchUp.dat).

Syntax

```
    char = mdMU->GetDatabaseExpirationDate()
```

C

```
    char = mdMUIncrementalGetDatabaseExpirationDate(mdMU)
```

COM+/.NET

```
    string = mdMU.GetDatabaseExpirationDate
```

GetLicenseExpirationDate

Returns a string value containing the expiration date of the current license string. After this date, MatchUp Object will no longer function.

Input Parameters

None

Return Value

Returns a string value that indicates the expiration date of the license string passed to the **SetLicenseString** function.

Syntax

```
    char = mdMU->GetLicenseExpirationDate()
```

C

```
    char = mdMUIncrementalGetLicenseExpirationDate(mdMU)
```

COM+/.NET

```
    string = mdMU.GetLicenseExpirationDate
```

Map Database Fields

Before generating match keys for the records in the database, the code must supply the Incremental deduper with information about what sort of data it will be handling.

ClearMappings

This function clears any existing field mappings.

It is a good idea to call this function before beginning to map fields, especially if the application may be required to perform multiple deduping operations in a single session.

Syntax

```
    mdMU->ClearMappings()
```

C

```
    mdMUIncrementalClearMappings(mdMU)
```

COM+/.NET

```
    mdMU.ClearMappings
```

AddMapping

This function selects the types of fields that will be used to build the match key and the order in which they will be added using the **AddField** function.

The function accepts an enumerated value of the type MatchcodeMapping. It tells the Incremental deduper which data types will be used for this deduping operation and in what order they will be passed to the deduper when passing data using the **AddField** function.

The data types used must contain the data expected by the matchcode being used, but it does not have to be an exact match. For example, if the matchcode requires a five-digit ZIP Code but the database contains a single “City/State/ZIP” field, simply add the CityStZip mapping and pass the full string to the AddField function later. MatchUp Object is smart enough to use only the information it needs.

In another example, a matchcode calls for both last name and first name but the database contains only full names. Simply apply the FullName mapping twice and pass the full name data twice to the AddField function.

Applying the two above examples to a matchcode that uses 5-digit ZIP codes, street addresses, last and first names, in that order, use the following mappings:

```
mapOK = mdMU->AddMapping(mdMU.CityStZip)// uses only ZIP Code
mapOK = mdMU->AddMapping(mdMU.FullName) // uses last name only
mapOK = mdMU->AddMapping(mdMU.FullName) // uses first name only
mapOK = mdMU->AddMapping(mdMU.Address)
```

For a list of the possible values, see the table on page 148.

The function returns a non-zero value if the mapping is allowed by the selected matchcode, false if the mapping caused an error.

Syntax

```
int = mdMU->AddMapping(mdMU.MatchcodeMapping)
```

C

```
int = mdMUIncrementalAddMapping(mdMU,
mdMU.mdMatchUpMatchmodeMapping)
```

COM+/.NET

```
integer = mdMU.AddMapping(mdMU.MatchcodeMapping)
```

Read Data and Build the Match Key

The following functions take the real data being compared and construct a match key according to the mappings defined with the above functions and the matchcode specified when the Incremental deduper was initialized.

ClearFields

This function clears all values from previous calls to the AddField function.

To ensure that no extraneous information carried over from one record to the next, call this function after calling the **BuildKey** function or before the first call to the **AddField** function.

Syntax
`mdMU->ClearFields()`
C
`mdMUIncrementalClearFields(mdMU)`
COM+/.NET
`mdMU.ClearFields`

AddField

This function passes the contents of a field from a database to the deduper while building a match key.

This function passes a component of list data to the deduper prior to calling the **BuildKey** function.

Fields must be passed to this function in the same order that the corresponding data types were mapped using the **AddMapping** function.

The following example expands on the **AddMapping** example on the previous page. The matchcode uses five-digit ZIP codes, the street addresses, last and first names, in that order. The database contains a single "City/ST/ZIP" and a single full name field.

```
mdMU->AddField("Rancho Santa Margarita, CA 92688")
mdMU->AddField("Raymond F. Melissa")
mdMU->AddField("Raymond F. Melissa")
mdMU->AddField("22382 Avenida Empresa")
```

The deduper would use only the ZIP Code from the first AddField mapping, the last name from the second mapping, the first name from the third, etc.

Syntax

```
    mdMU->AddField(char)
```

C

```
    mdMUIncrementalAddField(mdMU, char)
```

COM+/.NET

```
    mdMU.AddField(string)
```

BuildKey

This function builds a match key using information passed via the **AddField** function.

This function takes the information passed via calls to the **AddField** function and, using the mapping defined by the **AddMapping** function and the pattern defined by the matchcode being used, builds a match key.

A match key is a character string built according to a pattern defined by the current matchcode, consisting only of enough information to determine if the current record is unique or has a duplicate within the key file.

For example, let's assume the matchcode called for a five-digit ZIP Code, first ten characters of a last name, a street number and the first ten characters of a street name. The current record is for Raymond F. Melissa at 22382 Avenida Empresa in the 92688 ZIP Code. The match key would be:

```
92688MELISSA   RAYMOND   22382EMPRESA
```

Because "Empresa" is only seven characters, the key would be padded with three spaces at the end.

Syntax

```
    mdMU->BuildKey()
```

C

```
    mdMUIncrementalBuildKey(mdMU)
```

COM+/.NET

```
    mdMU.BuildKey
```

SetKey

This function accepts a match key before calling the **MatchRecord** function.

The **BuildKey** function creates a key from input data. If, however, the match keys are already stored in the source database, use this function to pass the keys to the deduper before calling **MatchRecord**.

Syntax

```
mdMU->SetKey(char)
```

C

```
mdMUIncrementalSetKey(mdMU, char)
```

COM+/.NET

```
mdMU.Key = string
```

SetUserInfo

This function accepts a character value that uniquely identifies each record in a set of data.

The character value passed to this function must be unique for every record. This enables the developer to associate the match key in the key file to the corresponding record in the database.

Syntax

```
mdMU->SetUserInfo(char)
```

C

```
mdMUIncrementalSetUserInfo(mdMU, char)
```

COM+/.NET

```
mdMU.SetUserInfo = string
```

Compare Record to Database

The following functions compare the new key with the existing key file and, if a duplicate is found, return information about the duplicate records in the file.

MatchRecord

This function compares the current match key to the keys in the key file and determines if this key matches a record that is already in the file.

If it is not a duplicate, a typical program would call the **AddRecord** function to add this key to the current key file.

If it is a duplicate, the developer can use the **GetKey**, **GetCount**, **GetDupeGroup**, **GetResults** and **GetEntry** functions to gather information about the existing duplicate records in the file.

Syntax

```
    mdMU->MatchRecord()
```

C

```
    mdMUIncrementalMatchRecord(mdMU)
```

COM+/.NET

```
    mdMU.MatchRecord
```

GetStatusCode

This function is deprecated. Use GetResults.

NextMatchingRecord

This function recalls the match data about the next record in the key file that matches the current search key.

After the **MatchRecord** function has detected a match between the input record, use the **NextMatchingRecord** function to loop through all of the matching records in the key file, returning the match data for each record.

This function returns a true value if there is another matching record, false if there are no more matching records, so an application can use a WHILE loop to repeatedly call the **GetEntry**, **GetResults**, **GetDupeGroup** and **GetKey** functions for each matching record.

Syntax

```
    int = mdMU->NextMatchingRecord()
```

C

```
    int = mdMUIncrementalNextMatchingRecord(mdMU)
```

COM+/.NET

```
    integer mdMU.NextMatchingRecord
```

GetCount

This function returns an integer value indicating the number of records in the key file that matched the current key.

If there were matches detected during a call to the **MatchRecord** function, the **GetCount** function will return a integer value equalling the number of duplicate keys found.

Syntax

```
int = mdMU->GetCount()
```

C

```
int = mdMUIncrementalGetCount(mdMU)
```

COM+/.NET

```
integer = mdMU.Count
```

GetDupeGroup

This function returns a long integer value indicating the group of duplicate records that the current key matches.

Every unique record (one with no duplicates) will have a unique “Dupe Group” number. Any duplicate record will be assigned the same number. This function returns the Dupe Group number of a matching record in the key file.

Syntax

```
long = mdMU->GetDupeGroup()
```

C

```
long = mdMUIncrementalGetDupeGroup(mdMU)
```

COM+/.NET

```
long = mdMU.DupeGroup
```

GetEntry

Returns an integer value indicating where the current record would fall within the order of its dupe group.

This function will return an integer value that indicates where the current record falls within its dupe group. If this is the sixth matching record found, this function will return a 6.

Syntax

```
int = mdMU->GetEntry()
```

C

```
int = mdMUIncrementalGetEntry(mdMU)
```

COM+/.NET

```
integer = mdMU.Entry
```

GetCombinations

This function is deprecated. Use GetResults.

GetKey

This function returns the match key created by the last call to the **BuildKey** function and used by the last call to the **MatchRecord** function.

Syntax

```
    char = mdMU->GetKey()
```

C

```
    char = mdMUIncrementalGetKey(mdMU)
```

COM+/.NET

```
    string = mdMU.Key
```

GetUserInfo

This function returns a character value containing the value passed to the **SetUserInfo** function.

This function returns the unique identifier associated with the record being checked by the Incremental deduper.

The Incremental deduper will need this information to match the current match key back to an original data source.

Syntax

```
    char = mdMU->GetUserInfo()
```

C

```
    char = mdMUIncrementalGetUserInfo(mdMU)
```

COM+/.NET

```
    string = mdMU.UserInfo
```

GetResults

This function returns a comma-delimited string of four-character codes that detail the output disposition of the last call to the MatchRecord function. It will also contain the result code of any matchcode combination which contributed to the present record matching other records in its dupe group.

The **GetResults** function is intended to replace the **GetStatusCode** and **GetCombinations** functions, providing a single source of information about the last MatchRecord function call and eliminating the need to perform bitwise operations on the

GetCombinations return value to determine which matchcode combinations contributed to the record matching other records in its Dupe Group.

The function returns one or more of the following codes in a comma-delimited list:

MatchUp Object: Result Codes

Code	Short Description	Long Description
MS01	Unique Record	The reocrd did not match any other records.
MS02	Has Duplicates	The record matched other records and was tagged as the output record.
MS03	Is Duplicate	The record matched other records and was tagged as a duplicate.
MS04	Record Suppressed	The source record was suppressed.
MS05	Record Not Intersected	The source record was not intersected.
MS06	Match: Rule 1	Records were matched by matchcode combination 1.
MS07	Match: Rule 2	Records were matched by matchcode combination 2.
MS08	Match: Rule 3	Records were matched by matchcode combination 3.
MS09	Match: Rule 4	Records were matched by matchcode combination 4.
MS10	Match: Rule 5	Records were matched by matchcode combination 5.
MS11	Match: Rule 6	Records were matched by matchcode combination 6.
MS12	Match: Rule 7	Records were matched by matchcode combination 7.
MS13	Match: Rule 8	Records were matched by matchcode combination 8.
MS14	Match: Rule 9	Records were matched by matchcode combination 9.
MS15	Match: Rule 10	Records were matched by matchcode combination 10.
MS16	Match: Rule 11	Records were matched by matchcode combination 11.
MS17	Match: Rule 12	Records were matched by matchcode combination 12.
MS18	Match: Rule 13	Records were matched by matchcode combination 13.
MS19	Match: Rule 14	Records were matched by matchcode combination 14.
MS20	Match: Rule 15	Records were matched by matchcode combination 15.
MS21	Match: Rule 16	Records were matched by matchcode combination 16.

MatchUp Object: Result Codes		
MS30	Suppressor Record	The lookup record suppressed a source record.
MS31	Intersector Record	The lookup record intersected a source record.

Syntax

```
    StringValue = object->GetResults();
```

C

```
    StringValue = mdMatchUpGetResults(object);
```

COM

```
    StringValue = object.Results
```

Add New Record to Key File

The Incremental deduper does not handle reading or writing data to the database. This function only adds the new match key to the key file for future comparisons.

AddRecord

The **AddRecord** function appends the key generated by the most recent call to the **BuildKey** function to the current file. The typical application would use this function to add a new unique record to the key file if no duplicate was found by the last call to the **MatchRecord** function.

Note: This function does not add the current record to the database. It merely appends a new key to the key file.

Syntax

```
    mdMU->AddRecord()
```

C

```
    mdMUIncrementalAddRecord(mdMU)
```

COM+/.NET

```
    mdMU.AddRecord
```

Transaction Methods

The following functions enable the Incremental deduper to use transactions, processing multiple calls to the **AddRecord** function before committing the changes to the key file.

BeginTransaction

This function tells MatchUp Object to wrap subsequent multiple calls to the **AddRecord** function with a transaction block.

This will greatly speed up processing when adding large numbers of records in the Incremental processor because Records will not be physically written to the Incremental database until the **CommitTransaction** function is called. The transaction functions are used in the same way that BEGIN, COMMIT and ROLLBACK are used in SQL.

Even though the keys have not been permanently added, Records will still be matched properly. However, other running processes that may be matching against the same database WILL NOT see these new records until after a call to the CommitTransaction function. Thus, transaction processing should not be used if multiple threads, processes, users, or machines are accessing the same Incremental database.

The **BeginTransaction** function returns true when transaction processing is successfully initialized.

Syntax

```
    BooleanValue = mdMU->BeginTransaction()
```

C

```
    BooleanValue = mdMUIncrementalBeginTransaction(mdMU)
```

COM+/.NET

```
    BooleanValue = mdMU.BeginTransaction
```

CommitTransaction

This function tells MatchUp Object to commit, or add, the previous calls to the **AddRecord** function to the key file since the **BeginTransaction** function was called.

This will greatly speed up processing when adding large numbers of records in the Incremental processor because Records will not be physically written to the Incremental database until the **CommitTransaction** function is called. The transaction functions are used in the same way that BEGIN, COMMIT and ROLLBACK are used in SQL.

Even though the keys have not been permanently added, Records will still be matched properly. However, other running processes that may be matching against the same database WILL NOT see these new records until after the call to the **CommitTransaction** function. Thus, transaction processing should not be used if multiple threads, processes, users, or machines are accessing the same Incremental database.

The **CommitTransaction** function returns true when an existing transaction has successfully completed.

Syntax

```
    BooleanValue = mdMU->CommitTransaction()
```

C

```
    BooleanValue = mdMUIncrementalCommitTransaction(mdMU)
```

COM+/.NET

```
    BooleanValue = mdMU.CommitTransaction
```

RollbackTransaction

This function enables you to roll back or erase the previous calls to the **AddRecord** function from the last call **BeginTransaction** function. A boolean value of true is returned if a successful rollback is completed.

Even though the RollbackTransaction will ensure that the keys have not been permanently added, Records will still be matched properly. Therefore, other running processes that may be matching against the same database will not see these new records. Thus, transaction processing SHOULD NOT be used if multiple threads/processes/users/machines are accessing the same Incremental database.

The **RollbackTransaction** function may also be used when an error is raised with an input or master database, prompting you to gracefully rollback the key file to the point before the current problematic data was processed.

Syntax

```
BooleanValue = mdMu->RollbackTransaction()
```

C

```
BooleanValue = mdMUIncrementalRollbackTransaction(muMU)
```

COM+/.NET

```
BooleanValue = mdMU.RollbackTransaction
```

Hybrid Deduping

The Hybrid deduper differs from the Incremental and Read/Write dedupers in that it does not maintain a key file of its own. It is up to the developer to maintain a list of match keys to use for deduping operations. This increases the flexibility of the Hybrid deduper but at the expense of programming complexity.

The main advantage of Hybrid deduping is that it allows the developer to build smaller lists of match keys on the fly and quickly compare records to a small subset of the database.

Clustering

The concept of Clustering, outlined in the first chapter, is essential to the Hybrid deduper. Unlike the other dedupers, where the clustering is taking place behind the scenes, the Hybrid deduper allows the developer to use clustering to compare a record against only a small portion of a list.

The Hybrid deduper uses the concept of a cluster size, which is the maximum number of characters at the beginning of a key that can be used to group a number of keys into smaller groups that can be compared against each other. For example, a cluster size of 5 means that the first five characters of a match key are used to create the clusters.

In other words, only the records where the first five characters of the match key for one record are identical to the first five characters of the match key for another record are considered when performing a Hybrid deduping operation.

Key Maintenance

Unlike the other interfaces, the Hybrid deduper does not automatically handle the read/ write operations to a key file. While this forces the developer to do more work, it allows a great deal of flexibility in how match keys are stored and handled.

In the previous example, with a cluster size of 5, if the match keys are stored in a field within a SQL database, a cluster could be built quickly by performing a SELECT query where the first five characters of the match key field matches the first five characters of the match key for the new record.

While this gives the developer far more flexibility, it also requires a great deal more coding and a greater understanding of certain MatchUp concepts.

Hybrid Order of Operations

Using the Hybrid deduper is not as straightforward as the other interfaces, as it puts greater burden on the developer to handle storage and management of match keys.

This section will outline the basic steps and then show an example of the programming logic for a typical implementation of the Hybrid deduper.

1. Initialize the Hybrid deduper.

   After creating an instance of the Hybrid deduper, point the object toward its supporting data file, select a matchcode to use, and initialize these files.
2. Create field mappings.

   In order to build keys to compare, the Hybrid deduper needs to know which types of data the program will be passing to the deduper and in what order.
3. Build a master list of keys.

   Each record must have a match key so the Hybrid deduper can select a cluster of records or check for duplicates. This consists of passing the data used in record comparison from each record to the deduper in the same order used when creating a field mapping. After passing the necessary fields (usually a small subset of the fields from each record) via the **AddField** function, the Hybrid deduper uses this information to generate a match key.
4. Build a match key for the new address record.

   Repeat the step above to create a match key for the record to be compared against the cluster.
5. Build the cluster list.

   Cycle through the master key list, extract only those records where the first part of the match key equals the first part of the match key for the new record.
6. Compare the match key to the cluster list.

   Loop through the cluster key file for any keys that match the new record. If it finds a match, the **CompareKey** function indicates a match.

The following section outlines a common implementation of the Hybrid deduper. We are using pseudocode for maximum clarity. Working sample programs in several programming languages can be found on the MatchUp Object install disc and more can be downloaded from the MatchUp Object support page on the Melissa Data website.

Step 1: Initialize the Hybrid deduper

After creating an instance of the Hybrid deduper, point the object toward its supporting data file, select a matchcode and key file to use, and initialize these files.

First, create a new instance of the Hybrid deduper.

```
SET mu = NEW mdMUHybrid
```

In order to successfully initialize this new instance, point it toward its data files and supply a valid license string. Also, select a matchcode, by name, before initializing.

```
CALL mu.SetLicenseString with LicenseString
CALL mu.SetPathToMatchUpFiles with DataPath
CALL mu.SetMatchcodeName with MatchcodeName
```

If all of the above have been set correctly, calling the InitializeDataFiles function should return a ProgramStatus value of ErrorNone. If it does not, call the GetInitializeErrorString function to determine the reason for the failure to initialize.

```
CALL mu.InitializeDataFiles RETURNING ProgramStatus

IF ProgramStatus is not ErrorNone THEN
    CALL mu.GetInitializeErrorString RETURNING ErrorMsg
    Display ErrorMsg
    Exit Routine
END IF
```

If the initialization was successful, call the following functions to display version and expiration information about the instance of MatchUp Object currently in use on the local computer.

```
PRINT "Confirming Initialization: " + mu.GetInitializeErrorString
PRINT "Build Number: " + mu.GetBuildNumber
PRINT "Database Date: " + mu.GetDatabaseDate
PRINT "Database Expiration Date: " + mu.GetDatabaseExpirationDate
PRINT "License Expiration Date: " + mu.GetLicenseExpirationDate
```

Step 2: Create field mappings

Field mappings define which types of data the Hybrid deduper is expecting. For example, a typical matchcode may include a five-digit ZIP Code, a last name, and a street address. The data coming in, however, may contain the city, state, and ZIP as a single character field and the person's full name as a single field as well.

As long as MatchUp Object knows what kind of data is being passed to it, the object is smart enough to pull what it needs from the data supplied to it.

```
CALL mu.ClearMappings
```

After clearing any mappings from a previous use of the Hybrid deduper, call the **AddMapping** function once for each field being considered.

```
CALL mu.AddMapping with mu.Zip9
CALL mu.AddMapping with mu.First
CALL mu.AddMapping with mu.Last
CALL mu.AddMapping with mu.Address
```

Step 3: Create Master Key File

Unlike the Incremental and Read/Write dedupers, the Hybrid deduper requires the developer to maintain a list of keys for the deduping operation. In this example, the keys are stored in a text file generated on the fly.

```
Open KeyFile as text file for writing
```

Each record is read from the database, converted to a match key and written to the text file.

```
FOR EACH Record in Database
    Read Zip9, FirstName, LastName, StreetAddress fields from database
    CALL mu.ClearFields
    CALL mu.AddField with Zip9
    CALL mu.AddField with FirstName
    CALL mu.AddField with LastName
    CALL mu.AddField with StreetAddress
    CALL mu.BuildKey
    CALL mu.GetKey RETURNING Key
    Write Key to KeyFile
NEXT

Close KeyFile
```

Step 4: Create the Match Key for the Input Data

The next step is to take the record that is to be checked and create a match key for it.

```
GET Zip9, FirstName, LastName, StreetAddress from data source

CALL mu.ClearFields
CALL mu.AddField with Zip9
CALL mu.AddField with FirstName
CALL mu.AddField with LastName
CALL mu.AddField with StreetAddress
CALL mu.BuildKey
CALL mu.GetKey RETURNING Key
```

Step 5: Create the Cluster List

Use the key generated in the last step to select only those records where the first part of the match key matches the same part of the match key for the record to be checked. The size of the portion of the match key to be checked is determined by the **GetClusterSize** function.

```
CALL mu.GetKeySize RETURNING KeySize
CALL mu.GetClusterSize RETURNING ClusterSize
SET ClusterKey = Left part of Key, size = ClusterSize
```

ClusterKey is a string, with a length equalling ClusterSize, used to match the first part of the match key from the input record. Cycle through the key list and create a cluster of only those records that match the cluster key to a new text file.

```
Open KeyFile for reading
FOR EACH Record in KeyFile
    Read MasterKey
    IF First ClusterSize characters of MasterKey = ClusterKey THEN
        ADD Record to Cluster
    END IF
NEXT
Close KeyFile
```

Step 6: Check Input Record Against Cluster List

With the cluster list built, check the whole key for the input record against each line of the cluster list, using the CompareKey function to determine if there was a match.

```
FOR EACH Record in Cluster
    Read MatchKey
    CALL mu.CompareKey with MasterKey, MatchKey RETURNING NoError
    IF NoError is True
        PRINT MasterKey matches MatchKey
    END IF
NEXT
```

Hybrid Deduping Functions

The following is a master list of the functions in the Hybrid deduper interface.

Initialize the Hybrid Interface

The following functions prepare the Hybrid deduper for use and link it to its supporting data files.

SetPathToMatchUpFiles 94
SetLicenseString 95
SetMatchcodeName 96
SetMatchcodeObject 97

InitializeDataFiles 97
GetInitializeErrorString 98
GetBuildNumber 99
GetDatabaseDate 99
GetDatabaseExpirationDate 100
GetLicenseExpirationDate 100

Map Database Fields and Build Keys

Before generating match keys for the records in the database, the code must supply the Hybrid deduper with information about what sort of data it will be handling.

ClearMappings 101
AddMapping 101

Build the Match Keys

The following functions gather the input data and use it to generate match keys according to the mapping defined above and the selected matchcode.

ClearFields 102
AddField 102
BuildKey 103
GetKey 104

Compare Records

Use the following functions to determine how much of each match key will be used to select records for the cluster and compare the input data to the keys in the cluster.

GetKeySize 104
GetClusterSize 105
CompareKeys 105
GetResults 106

Initialize the Hybrid Interface

The following functions prepare the Hybrid deduper for use and link it to its supporting data files.

SetPathToMatchUpFiles

This function accepts a string value containing the path to the folder containing the MatchUp Hybrid data files.

This function must be called before calling the InitializeDataFiles function.

To provide maximum compatibility with Windows, three files are installed in your 'Common App Data' directory. For Windows Vista and Windows 7 the default location is "C:\ProgramData\MelissaDATA\MatchUp." For Windows XP the default location is "C:\Documents and Settings\All Users\Application Data\Melissa DATA\MatchUp." The location of this directory can be changed by users so please note this, as it can often be the source of issues when running the samples/demos.

Syntax

```
mdMU->SetPathToNameFiles(char)
```

C

```
mdMUHybridSetPathToMatchUpFiles(mdMU, char)
```

COM+/.NET

```
mdMU.PathToMatchUpFiles = string
```

SetLicenseString

Passes the license string required for MatchUp Object to function.

Each customer is issued a license string when purchasing MatchUp Object or renewing a subscription. This string must be passed to this function to unlock the functionality of MatchUp Object.

The license string is normally set using an environment variable, either MD_LICENSE or MD_LICENSE_DEMO. Calling **SetLicenseString** is an alternative method for setting the license string, but applications developed for a production environment should only use the environment variable.

When using an environment variable, it is not necessary to call the **SetLicenseString** function.

For more information on setting the environment variable, see Entering Your MatchUp Object License on page 4 of this guide.

Using an environment variable makes it much easier to update the license string without having to edit and re-compile the application.

Windows

Windows users can set environment variables by doing the following:

•Select Start > Settings, and then click Control Panel.

•Double-click System, and then click the Advanced tab.

•Click Environment Variables, and then select either System Variables or Variables for the user X.

- Click New.
- Enter “MD_LICENSE” in the Variable Name box.
- Enter the license string in the Variable Value box and then click OK.

Please remember that these settings take effect only upon start of the program. It may be necessary to quit and restart the development environment to incorporate the changes.

Linux/Solaris/HP-UX/AIX

Unix-based OS users can simply set the license string via the following:

export MD_LICENSE=A1B2C3D4E5 (not the actual license string).

After putting this setting in the profile, remember to restart the shell.

Input Parameters

A string value containing an empty string or, if necessary, a valid license string for MatchUp Object.

Return Values

This function returns an integer value. A value of 1 indicates a valid license string, 0 an invalid or empty string.

Syntax

```
int = mdMU->SetLicenseString(char)
```

C

```
int = mdMUHybridSetLicenseString(mdMU, char)
```

COM+/.NET

```
integer = mdMU.SetLicenseString string
```

SetMatchcodeName

The **SetMatchcodeName** function accepts a string value that must match the name of an existing matchcode in the current matchcode file.

Syntax

```
mdMU->SetMatchcodeName(char)
```

C

```
mdMUHybridMatchcodeName(mdMU, char)
```

COM+/.NET

```
mdMU.MatchcodeName = string
```

SetMatchcodeObject

This functions selects the matchcode to use for the current Hybrid deduping operation.

This function largely duplicates the purpose of the **SetMatchcodeName** function, but instead of accepting a character value containing the name of a matchcode in the current matchcode file, this function accepts a Matchcode object created using the Matchcode Editor interface.

Because this function requires the use of a separate interface to create the Matchcode object variable, it is normally simpler to use the **SetMatchcodeName** function.

It is possible, however, to use this function to build a new matchcode on the fly using the Matchcode Editor interface. Unless a specific application demands such flexibility, it is usually much simpler to add a new matchcode to the matchcode file and call it using the **SetMatchcodeName** function.

Syntax

```
mdMU->SetMatchcodeObject(mdMUMatchcode)
```

C

```
mdMUHybridSetMatchcodeObject(mdMU, mdMUMatchcode)
```

COM+/.NET

```
mdMU.MatchcodeObject = mdMUMatchcode
```

InitializeDataFiles

The InitializeDataFiles function opens the needed data files and prepares the MatchUp Object for use.

Before calling this function, the application must have successfully called the **SetLicenseString** and **SetPathToMatchUpFiles** functions.

Check the return value of the **GetInitializeErrorString** function to retrieve the result of the initialization call. Any result other than “ErrorNone” means the initialization failed for some reason.

Return Value

Returns a value of the enumerated type ProgramStatus.

Value		Reason
0	ErrorNone	No error - initialization was successful.
1	ErrorConfigFile	Could not find mdMatchUp.dat.
2	ErrorLicenseExpired	The License String has expired.

Value		Reason
3	ErrorDatabaseExpired	The database has expired.
4	ErrorMatchcodeNotSpecified	No matchcode was specified.
5	ErrorMatchcodeNotFound	Specified Matchcode does not exist.
6	ErrorInvalidMatchcode	The specified matchcode is not valid.

If any other value other than ErrorNone is returned, check the **GetInitializeErrorString** function to see the reason for the error.

Syntax

```
    ProgramStatus = mdMU->InitializeDataFiles()
```

C

```
    ProgramStatus = mdMUHybridInitializeDataFiles(mdMU)
```

COM+/.NET

```
    ProgramStatus = mdMU.InitializeDataFiles
```

GetInitializeErrorString

Returns a descriptive string to describe the error from the **InitializeDataFiles** function.

The possible strings returned by this function are:

```
No error
Could not find mdMatchUp.dat.
The License String has expired.
The database has expired.
No matchcode was specified.
Specified Matchcode does not exist.
The specified matchcode is not valid.
```

The **GetInitializeErrorString** function returns a string describing the error caused when the **InitializeDataFiles** function cannot be called successfully.

Syntax

```
    char = mdMU->GetInitializeErrorString()
```

C

```
    char = mdMUHybridGetInitializeErrorString(mdMU)
```

COM+/.NET

```
    string = mdMU.GetInitializeErrorString
```

GetBuildNumber

The **GetBuildNumber** function returns the current development release build number of MatchUp Object.

Input Parameters

None.

Return Value

The **GetBuildNumber** function returns the current development release build number of the MatchUp Object.

Syntax

```
    char = mdMU->GetBuildNumber()
```

C

```
    char = mdMUHybridGetBuildNumber(mdMU)
```

COM+/.NET

```
    string = mdMU.GetBuildNumber
```

GetDatabaseDate

The **GetDatabaseDate** function returns a string value that represents the revision date of the MatchUp Object data files.

Input Parameters

None.

Return Value

The **GetDatabaseDate** function returns a string value that represents the date of the MatchUp Object data files.

Syntax

```
    char = mdMU->GetDatabaseDate()
```

C

```
    char = mdMUHybridGetDatabaseDate(mdMU)
```

COM+/.NET

```
    string = mdMU.GetDatabaseDate
```

GetDatabaseExpirationDate

Returns a string value containing the expiration date of the current database file.

Input Parameters

None

Return Value

Returns a string value indicating the expiration date of the current database file (mdMatchUp.dat).

Syntax

```
    char = mdMU->GetDatabaseExpirationDate()
```

C

```
    char = mdMUHybridGetDatabaseExpirationDate(mdMU)
```

COM+/.NET

```
    string = mdMU.GetDatabaseExpirationDate
```

GetLicenseExpirationDate

Returns a string value containing the expiration date of the current license string. After this date, MatchUp Object will no longer function.

Input Parameters

None

Return Value

Returns a string value that indicates the expiration date of the license string passed to the **SetLicenseString** function.

Syntax

```
    char = mdMU->GetLicenseExpirationDate()
```

C

```
    char = mdMUHybridGetLicenseExpirationDate(mdMU)
```

COM+/.NET

```
    string = mdMU.GetLicenseExpirationDate
```

Map Database Fields and Build Keys

Before generating match keys for the records in the database, the code must supply the Hybrid deduper with information about what sort of data it will be handling.

ClearMappings

This function clears any existing field mappings.

It is a good idea to call this function before beginning to map fields, especially if the application is required to perform multiple deduping operations in a single session.

Syntax

```
mdMU->ClearMappings()
```

C

```
mdMUHybridClearMappings(mdMU)
```

COM+/.NET

```
mdMU.ClearMappings
```

AddMapping

This function selects the types of fields that will be used to build the match key and the order in which they will be added using the **AddField** function.

The function accepts an enumerated value of the type MatchcodeMapping. It tells the Hybrid deduper which data types will be used for this deduping operation and in what order they will be passed to the deduper when passing data using the **AddField** function.

The data types used must contain the data expected by the matchcode being used, but it does not have to be an exact match. For example, if the matchcode requires a five-digit ZIP Code but the database contains a single “City/State/ZIP" field, simply add the CityStZip mapping and pass the full string to the **AddField** function later. MatchUp Object is smart enough to use only the information it needs.

In another example, a matchcode calls for both last name and first name but database contains only full names. The application would simply apply the FullName mapping twice and pass the full name data twice to the **AddField** function.

Let’s apply the two above examples to a matchcode that uses 5-digit ZIP codes, street addresses, last and first names, in that order.

```
mdMU->AddMapping(mdMU.CityStZip)  // uses only ZIP Code
mdMU->AddMapping(mdMU.FullName)   // uses last name only
mdMU->AddMapping(mdMU.FullName)   // uses first name only
mdMU->AddMapping(mdMU.Address)
```

For a list of the possible values, see the table on page 148.

The function returns a non-zero value if the mapping is allowed by the selected matchcode, false if the mapping caused an error.

Syntax

```
int = mdMU->AddMapping(mdMU.MatchcodeMapping)
```

C

```
int = mdMUHybridAddMapping(mdMU,
   mdMU.mdMatchUpMatchmodeMapping)
```

COM+/.NET

```
integer = mdMU.AddMapping(mdMU.MatchcodeMapping)
```

Build the Match Keys

The following functions gather the input data and use it to generate match keys according to the mapping defined above and the selected matchcode.

ClearFields

Use this function before the first call to **AddField** function for each record or after calling the **BuildKey** function.

Syntax

```
mdMU->ClearFields()
```

C

```
mdMUHybridClearFields(mdMU)
```

COM+/.NET

```
mdMU.ClearFields
```

AddField

This function passes the field data to the deduper while building a match key.

This function passes a component of data to the deduper prior to calling the **BuildKey** function.

Fields must be passed to this function in the same order that the corresponding data types were mapped using the **AddMapping** function.

The following example expands on the **AddMapping** function example on the previous page. The matchcode uses five-digit ZIP codes, last and first names, and the street addresses, in that order. The file includes only a single “City/ST/ZIP" and a single full name field.

```
mdMU->AddField("Rancho Santa Margarita, CA 92688")
mdMU->AddField("Raymond F. Melissa")
mdMU->AddField("Raymond F. Melissa")
mdMU->AddField("22382 Avenida Empresa")
```

The deduper would use only the ZIP Code from the first field, the last name from the second and first name from the third.

Syntax

```
mdMU->AddField(char)
```

C

```
mdMUHybridAddField(mdMU, char)
```

COM+/.NET

```
mdMU.AddField(string)
```

BuildKey

This function takes the information passed via calls to the **AddField** function and, using the mapping defined by the **AddMapping** function and the pattern defined by the matchcode being used, builds a match key.

A match key is a character string built according to a pattern defined by the current matchcode, consisting only of enough information to determine if the current record is unique or has a duplicate within the key file.

For example, let’s assume the matchcode called for a five-digit ZIP Code, first ten characters of a last name, a street number and the first ten characters of a street name. The current record is for Raymond F. Melissa at 22382 Avenida Empresa in the 92688 ZIP Code. The match key would be:

92688MELISSA   RAYMOND   22382EMPRESA

Because “Empresa” is only seven characters, the key would be padded with three spaces at the end.

Syntax

```
mdMU->BuildKey()
```

C

```
mdMUHybridBuildKey(mdMU)
```

COM+/.NET

```
mdMU.BuildKey
```

GetKey

This function returns a string value containing the match key generated by the most recent call to the **BuildKey** function.

Use this function to recall the most recently generated match key before writing it to a key file or passing it to the **CompareKey** function.

Syntax

```
    char = mdMU->GetKey()
```

C

```
    char = mdMUHybridGetKey(mdMU)
```

COM+/.NET

```
    string = mdMU.Key
```

Compare Records

Use the following functions to determine how much of each match key will be used to select records for the cluster and compare the input data to the keys in the cluster.

GetKeySize

This function returns an integer value indicating the number of characters in a key generated using the matchcode selected using the **SetMatchcodeName** function.

This function can be useful for determining field sizes or how much memory will need to be allocated, if the programming language requires the developer to handle memory management.

Syntax

```
    int = mdMU->GetKeySize()
```

C

```
    int = mdMUHybridGetKeySize(mdMU)
```

COM+/.NET

```
    integer = mdMU.KeySize
```

GetClusterSize

This function returns an integer value indicating the maximum size of the portion of the match key that can be used for clustering.

Use this function to determine how much of the key to use for comparison when building the cluster file, a subset of keys from your master database.

Syntax

```
    int = mdMU->GetClusterSize()
```

C

```
    int = mdMUHybridGetClusterSize(mdMU)
```

COM+/.NET

```
    integer = mdMU.ClusterSize
```

CompareKeys

This function compares two match keys and returns a long integer value indicating whether they match.

If the **CompareKeys** function does not find a match, the return value will be zero. If there was a match, this function returns an unsigned long integer value indicating which combination or combinations within the matchcode produced the match.

Each bit of the integer value matches a specific combination. Use a logical AND operation to determine if a particular combination produced the match. For example:

If (mdMU->GetCombination AND 0x8000) THEN Print "Combo 16 Matched"

Although this function does not return an enumerated value, it does use the same values as the MatchcodeCombination enumeration used by the Matchcode Editing interface.

For a list of these values see the table on page 147.

The **CompareKeys** function does not return any information about dupe groups or the number of duplicate records. This is because the Hybrid deduper does not keep track of matching records (it is up to the programmer to do this).

Syntax

```
    long = mdMU->CompareKeys()
```

C

```
    long= mdMUHybridCompareKeys(mdMU)
```

COM+/.NET

```
    long = mdMU.CompareKeys
```

GetResults

This function returns a comma-delimited string of four-character codes that detail the output disposition of the last call to the CompareKeys function. It will also contain the result code of any matchcode combination which contributed to the present record matching other records in its dupe group.

The **GetResults** function is intended to add new functionality of a returned Status Code and matchcode Combinations previously available by evaluating the CompareKeys return value, providing a single source of information about the last CompareKeys function call, and eliminating the need to perform bitwise operations on the return value to determine which matchcode combinations contributed to the record matching the other match key.

The function returns one or more of the following codes in a comma-delimited list:

MatchUp Object: Result Codes

Code	Short Description	Long Description
MS01	Unique Record	The reocrd did not match any other records.
MS02	Has Duplicates	The record matched other records and was tagged as the output record.
MS03	Is Duplicate	The record matched other records and was tagged as a duplicate.
MS04	Record Suppressed	The source record was suppressed.
MS05	Record Not Intersected	The source record was not intersected.
MS06	Match: Rule 1	Records were matched by matchcode combination 1.
MS07	Match: Rule 2	Records were matched by matchcode combination 2.
MS08	Match: Rule 3	Records were matched by matchcode combination 3.
MS09	Match: Rule 4	Records were matched by matchcode combination 4.
MS10	Match: Rule 5	Records were matched by matchcode combination 5.
MS11	Match: Rule 6	Records were matched by matchcode combination 6.
MS12	Match: Rule 7	Records were matched by matchcode combination 7.
MS13	Match: Rule 8	Records were matched by matchcode combination 8.
MS14	Match: Rule 9	Records were matched by matchcode combination 9.
MS15	Match: Rule 10	Records were matched by matchcode combination 10.

MatchUp Object: Result Codes		
MS16	Match: Rule 11	Records were matched by matchcode combination 11.
MS17	Match: Rule 12	Records were matched by matchcode combination 12.
MS18	Match: Rule 13	Records were matched by matchcode combination 13.
MS19	Match: Rule 14	Records were matched by matchcode combination 14.
MS20	Match: Rule 15	Records were matched by matchcode combination 15.
MS21	Match: Rule 16	Records were matched by matchcode combination 16.
MS30	Suppressor Record	The lookup record suppressed a source record.
MS31	Intersector Record	The lookup record intersected a source record.

Syntax

```
    StringValue = object->GetResults();
```

C

```
    StringValue = mdMatchUpGetResults(object);
```

COM

```
    StringValue = object.Results
```

Matchcode Interface

The Matchcode Editor for Windows will handle the task of creating and modifying matchcodes for many situations. The editor application will also run on a Linux system under WINE.

However, because MatchUp Object works across multiple platforms and not all users will have access to a Windows emulator, MatchUp Object also includes an interface for creating, modifying and viewing matchcodes programmatically on any system.

The Matchcode Interface also enables applications to create and use matchcodes on the fly, if this becomes advantageous to do.

Creating matchcodes programmatically is more complicated and advanced than using the Windows GUI editor. The Windows Matchcode Editor handles error checking and enforces many of the rules described in the chapter on matchcodes, while the Matchcode Interface returns the necessary error codes to detect such problems but requires that the developer implement the necessary error handling.

Using the Matchcode Interface requires a more thorough understanding of matchcodes and how they are used by MatchUp Object. We recommend carefully reading the beginning of chapter 6 before attempting to use the features in this chapter.

Order of Operations for Creating Matchcodes

Using the Matchcode Interface is not overly complicated but there are many options that must be considered for matchcodes and matchcode components.

This section will outline the basic steps and then show an example of the programming logic for a typical implementation of the Matchcode Interface.

1. Initialize the Matchcode Interface and set the data path.

   The Matchcode Interface does not require a license string so this step is much simpler than with the deduper interfaces.
2. Create a new matchcode.

   The **CreateNewMatchcode** function creates a new, blank matchcode for editing.

3. Create new matchcode components.

   Matchcode components are created as class variables. Create an instance of the MatchcodeComponent for each component.
4. Set the options for each matchcode component.

   Use the functions of the matchcode component class to select the options for the component type, size, matching strategy, swap pair, and to which combinations the component belongs.
5. Add the components to the new matchcode.

   Use the **AddMatchcodeItem** function to add the component to the new matchcode. At this point, the Matchcode Interface checks the component for errors.
6. Save the changes to the matchcode file.

   The Matchcode Interface can either save the changes to the original matchcode file or to a new copy of the file.

The following section shows a simplified code sample, written in pseudocode, showing the creation of a basic matchcode using the Matchcode Interface.

Step 1: Initialize the Matchcode Interface

Initialization consists of creating a new instance of the Matchcode class and connecting that instance to the data files. The Matchcode Interface does not require a license string, so many of the functions found in the deduper interfaces are not present here.

```
SET mc = NEW mdMUMatchcode
CALL mc.SetPathToMatchUpFiles with PathToMatchUpFiles
IF mc.ErrorNone == mc.InitializeDataFiles THEN

    PRINT "Initializing DataFiles..."
    PRINT "Confirm Init Non-Error:  " + mc.GetInitializeErrorString
ELSE

    PRINT "Init Error: " + mc.GetInitializeErrorString
ENDIF
```

Step 2: Create a new matchcode

A new matchcode requires a name. This program forces the users to keep entering a name until a valid name is entered.

```
REPEAT
    PRINT "Enter New Matchcode Name: "
    GET INPUT NewMatchCodeName
    CALL mc.CreateNewMatchcode with NewMatchCodeName RETURNING Created
    IF Created = 0 THEN PRINT "Could Not Create Matchcode!"
UNTIL Created <> 0
```

Step 3: Create new matchcode components

Matchcode components are created as instances of the MatchcodeComponent class. Another approach would be to create an array or simply create and add each component as part of a loop.

```
SET mcComp, mcComp2, mcComp3, mcComp4, mcComp5 = NEW
    mdMUMatchcodeComponent
```

Step 4: Set the options for each matchcode component

Use the functions of the matchcode component class to select the options for the component type, size, matching strategy, swap pair, and to which combinations the component belongs.

```
CALL mcComp.SetComponentType WITH MatchCodeComponentType.Zip5
CALL mcComp.SetStart WITH MatchcodeStart.Left
CALL mcComp.SetFuzzy WITH MatchcodeFuzzy.Exact
CALL mcComp.SetSwap WITH MatchcodeSwap.NoSwap
CALL mcComp.SetFieldMatch WITH MatchcodeFieldMatch.NoFieldMatch
CALL mcComp.SetSize WITH 5
CALL mcComp.SetCombination WITH (MatchcodeCombination.Combo1 OR
    MatchcodeCombination.Combo2)
```

Step 5: Add the components to the new matchcode

Use the **AddMatchcodeItem** function to add the component to the new matchcode. At this point, the Matchcode Editing Matchcode Interface allows you to check for errors, when attempting to add this components.

```
CALL mc.AddMatchcodeItem WITH mcComp RETURNING mcCompAdded
IF mcCompAdded <> 0 THEN PRINT "Component Added."
```

Step 6: Repeat for each component

Repeat steps 4 & 5 for each component.

```
CALL mcComp2.SetComponentType WITH MatchCodeComponentType.Last
CALL mcComp2.SetStart WITH MatchcodeStart.Left
CALL mcComp2.SetFuzzy WITH MatchcodeFuzzy.AccurateNear
CALL mcComp2.SetNear WITH 1
CALL mcComp2.SetSwap WITH MatchcodeSwap.NoSwap
CALL mcComp2.SetFieldMatch WITH mcComp.BothBlankMatch
CALL mcComp2.SetSize WITH 7
CALL mcComp2.SetCombination WITH (MatchcodeCombination.Combo1 OR
    MatchcodeCombination.Combo2)
CALL mc.AddMatchcodeItem WITH mcComp2 RETURNING mcCompAdded
IF mcCompAdd <> 0 THEN PRINT "Component Added"
```

Repeat until all five components have been created and added to the current matchcode.

Step 7: Save the changes to the matchcode file

The Matchcode Editing Matchcode Interface can either save the changes to the original matchcode file or to a new copy of the file.

```
CALL mc.Save
```

Order of Operations for Reading Matchcodes

Reading the matchcode file is simpler in that it requires no thought about the rules for matchcodes, but it does require some programming to translate the values returned into meaningful information.

1. Initialize the Matchcode Interface and set the data path.

   The Matchcode Interface does not require a license string so this step is much simpler than with the deduper interfaces.
2. Retrieve the matchcode.

   The **FindMatchcode** function loads the specified matchcode into memory.
3. Begin cycling through every component in the matchcode.

   The Matchcode Interface returns the number of components in the current matchcode. Use this number to loop through all of the components, assigning each component into to a MatchcodeComponent class variable.
4. Retrieve the component settings.

   Call the MatchcodeComponent functions that return the settings for each component and, if necessary, translate them into meaningful information.

These functions would normally be used in conjunction with those for creating and modifying matchcodes but, for simplicity and clarity, this section will concentrate solely on showing how to retrieve the matchcode and component information.

Step 1: Initialize the Matchcode Interface

Initialization consists of creating a new instance of the Matchcode class and connecting that instance to the data files. The Matchcode Interface does not require a license string so many of the functions found in the deduper interfaces are not present here.

```
SET mc = NEW mdMUMatchcode
CALL mc.SetPathToMatchUpFiles with PathToMatchUpFiles

IF mc.ErrorNone == mc.InitializeDataFiles THEN
```

```
        PRINT "Initializing DataFiles..."
        PRINT "Confirm Init Non-Error:  " + mc.GetInitializeErrorString
    ELSE
        PRINT "Init Error: " + mc.GetInitializeErrorString
    ENDIF
```

Step 2: Retrieve a matchcode

The **FindMatchcode** function requires the name of an existing matchcode in the current matchcode file.

```
    PRINT "Enter Existing Matchcode to Look Up: "
    INPUT MatchcodeName
    CALL mc.FindMatchcode WITH MatchcodeName RETURNING errorCode
    IF errorCode IS NOT 1 THEN
        PRINT "Matchcode can not be OPENED: " + errorCode
        END ROUTINE
    END IF
```

Step 3: Begin cycling through every component in the matchcode

Use the **GetMatchcodeItemCount** function, determine how many components are present in the current matchcode.

```
    FOR MatchcodeItem = 1 TO mc.GetMatchcodeItemCount
        CALL mc.GetMatchcodeItem with MatchcodeItem RETURNING mcComp
```

Step 4: Retrieve the component settings

Begin calling the MatchcodeComponent functions to return the settings for each matchcode.

```
        CALL mcComp.GetLabel RETURNING Label
        IF LABEL IS NOT EMPTY THEN PRINT LABEL
        CALL mcComp->GetComponentType RETURNING ComponentTypeName
        CASE ComponentTypeName OF
            1:  Type = "Prefix"
            2:  Type = "First"
            3:  Type = "Middle"
            4:  Type = "Last"
            5:  Type = "Suffix"
            6:  Type = "Gender"
            7:  Type = "FirstNickname"
            8:  Type = "MiddleNickname"
            9:  Type = "Title"
            10: Type = "Company"
            11: Type = "CompanyAcronym"
            12: Type = "StreetNumber"
            13: Type = "StreetPreDir"
            14: Type = "StreetName"
            15: Type = "StreetSuffix"
            16: Type = "StreetPostDir"
            17: Type = "POBox"
```

This is an incomplete list, but with a separate case for each component type, the program determines the component type and displays the name.

```
            Other: Type = "UNDETERMINED"
        ENDCASE
        PRINT "Type:" + Type
```

Retrieve and display the size of the current component.

```
        CALL mcComp->GetSize RETURNING Size : Print "Size: " + Size
```

Repeat the same basic procedure for the Component starting position.

```
        CALL mcComp->GetStart RETURNING ComponentStart
        CASE OF ComponentStart
             0x08: Start = "Left"
             0x10: Start = "Right"
             0x20: Start = "Pos:"
             0x40: Start = "Word:"
            Other: Start = "UNKNOWN"
        ENDCASE
        PRINT "Start: " + Start
```

Repeat again for the fuzzy matching rule for the current component. The following is an incomplete list.

```
        CALL mcComp->GetFuzzy RETURNING ComponentFuzzy
        CASE OF ComponentFuzzy
            0x0000 : Fuzzy  = "Exact"
            0x0001 : Fuzzy  = "SoundEx"
            0x0002 : Fuzzy  = "Phonetex"
            0x0004 : Fuzzy  = "Containment"
            0x0008 : Fuzzy  = "Frequency"
            0x0010 : Fuzzy  = "FastNear"
            0x0020 : Fuzzy  = "AccrNear"
            0x0040 : Fuzzy  = "Vowels"
            0x0080 : Fuzzy  = "Consonants"
            0x0100 : Fuzzy  = "Alphas"
            0x0200 : Fuzzy  = "Numerics"
            0x0400 : Fuzzy  = "FreqNear"
             Other: Fuzzy  = "UNKNOWN"
        ENDCASE
        PRINT "Fuzzy Matching: " + Fuzzy
```

Repeat again for the blank field matching rules.

```
Call mcComp->GetFieldMatch RETURNING ComponentField
CASE OF ComponentField
        0: FieldMatch = "NO Blank"
   0x0100: FieldMatch = "BothBlank"
   0x0200: FieldMatch = "OneBlank"
   0x0400: FieldMatch = "Initial"
   0x0300: FieldMatch = "Both/One"
   0x0500: FieldMatch = "Both/Init"
   0x0600: FieldMatch = "Init/One"
   0x0700: FieldMatch = "Both/One/Init"
    Other: FieldMatch = "UNKNOWN"
ENDCASE
PRINT "Blank Field Matching: " + Field
```

The process for getting the information about which combinations the component belongs to is somewhat more complicated. It involves using a logic AND operation to compare the value returned by the **GetCombination** function to each of the possible values shown in the table on page 147.

```
CALL mcComp->GetCombination RETURNING ComponentCombos

SET CombinationsList to empty string

IF ComponentCombos AND 0x0001 THEN
    Concatenate "1" TO CombinationsList
ELSE
    Concatenate "." TO CombinationsList
END IF

IF ComponentCombos AND 0x0002 THEN
    Concatenate "2" TO CombinationsList
ELSE
    Concatenate "." TO CombinationsList
END IF

IF ComponentCombos AND 0x0004 THEN
    Concatenate "3" TO CombinationsList
ELSE
    Concatenate "." TO CombinationsList
END IF
```

This snippet of code adds a digit for each combination that uses the component. Otherwise, it adds a period to represent a blank. Repeat the above structure for each possible value from the table on page 147.

```
IF ComponentCombos AND 0x8000 THEN
    Concatenate "F" TO CombinationsList
ELSE
    Concatenate "." TO CombinationsList
```

```
        END IF

        PRINT "This component is used for these combinations: " +
        CombinationsList
```

Repeat for each component in the matchcode.

```
    ENDFOR
```

Matchcode Mapping Information

In addition to information about the components used by a matchcode, the Matchcode Interface can also return information about the required mapping for each matchcode. These can be different from the Component mapping types because the component type tells you the data type which will be used to match records, while the matchcode mapping tells the API the format of the incoming data.

One use for this would be for an application to retrieve the information from a matchcode and dynamically create the mappings based on that information.

Use the **SetMatchcodeObject** function instead of the **SetMatchcodeName** function to set the matchcode used by the deduper.

The following illustrates a few of the many possible matchcode mappings required by a respective component type.

```
    CASE MatchCode.ComponentType.PrefixType: CALL mdMU.AddMapping WITH
        mm.FullName
    CASE MatchCode.ComponentType.FirstType: CALL mdMU.AddMapping WITH
        mm.FullName
    CASE MatchCode.ComponentType.LastType: CALL mdMU.AddMapping WITH
        mm.FullName
    CASE MatchCode.ComponentType.SuffixType: CALL mdMU.AddMapping WITH
        mm.FullName
    CASE MatchCode.ComponentType.FirstNicknameType: CALL mdMU.AddMapping
        WITH mm.FullName
```

Keep in mind that these are not all the matchcode mapping targets. For a full list of these targets, see “MatchcodeMappingTarget” on page 149. If a matchcode contains component types that can not be extracted from the database you want to process, that matchcode should not be used for that process.

Matchcode Interface Functions

The following is a master list of the functions in the Matchcode Interface.

Initialize MatchUp Object

Initializing the Matchcode Interface is simpler than the other interfaces, since no license string is required.

SetPathToMatchUpFiles 118
InitializeDataFiles 119
GetInitializeErrorString 120

Create a New Matchcode

The single function in this section, part of the Matchcode class, creates a new, blank matchcode that can be populated with matchcode components and saved to the current matchcode file.

CreateNewMatchcode 120

Retrieve Existing Matchcodes

These functions are used to retrieve a specific matchcode from the matchcode file.

FindMatchcode 121
GetMatchcodeName 122

General Matchcode Properties

These functions help with defining various Matchcode properties.

GetDescription 122
SetDescription 122
GetNGram 123
SetNGram 123

Read Matchcode Component Information

The functions in this section retrieve the number of components in a given matchcode and retrieve the contents of a specific matchcode component.

GetMatchcodeItemCount 124
GetMatchcodeItem 124

Get Mapping Information

Mapping information is different from component information, revealing the order and mapping types that should be used when creating the mappings in any one of the deduper interfaces.

GetMappingItemCount 124
GetMappingItemType 125
GetMappingItemLabel 125

Change Matchcode Component Settings

The functions in the section set the values for the various settings of a matchcode component object. They can be used to construct new matchcode components when adding them to a matchcode or to change the settings of an existing component. Every function in this section requires a variable based on the mdMatchcodeComponent class.

SetComponentType 126
SetSize 126
SetLabel 127
SetWordCount 127
SetStart 127
SetStartPos 128
SetTrim 129
SetFuzzy 129
SetNear 130
SetNearDbl 131
SetFieldMatch 131
SetCombination 132
SetSwap 133

Read Matchcode Component Settings

The following functions read and return the settings from a specific matchcode component variable.

GetComponentType 134
GetSize 134
GetLabel 135
GetWordCount 135
GetStart 136
GetStartPos 137
GetTrim 137
GetFuzzy 138
GetNear 138

GetNearDbl 139
GetFieldMatch 139
GetCombination 140
GetSwap 141

Add, Modify or Delete Matchcode Components

The functions in this section add, insert, update, or delete matchcode components from the current Matchcode object.

AddMatchcodeItem 142
InsertMatchcodeItem 143
ChangeMatchcodeItem 144
DeleteMatchcodeItem 145

Save Changes to the Matchcode File

The functions in this section save changes to the current matchcode file, either back to the original default file or to a new file.

Save 145
SaveToFile 146
RenameMatchcode 146
DeleteMatchcode 146

Initialize MatchUp Object

Initializing the Matchcode Interface is simpler than the other interfaces, since no license string is required.

SetPathToMatchUpFiles

String value. This function accepts a string value indicating the file path to the folder containing the MatchUp Object files. This function must be called before calling the InitializeDataFiles function.

To provide maximum compatibility with Windows, three files are installed in your ‘Common App Data’ directory. For Windows Vista and Windows 7 the default location is “C:\ProgramData\MelissaDATA\MatchUp.” For Windows XP the default location is “C:\Documents and Settings\All Users\Application Data\Melissa DATA\MatchUp.”

The location of this directory can be changed by users so please note this, as it can often be the source of issues when running the samples/demos.

Syntax

```
    mdMC->SetPathToMatchUpFiles(char)
```

C

```
    mdMUMatchcodeSetPathToMatchUpFiles(mdMC, char)
```

COM+/.NET

```
    mdMC.PathToMatchUpFiles = string
```

InitializeDataFiles

The InitializeDataFiles method opens the needed data files and prepares the MatchUp Object for use.

Before calling this method, you must have successfully called **SetPathToMatchUpFiles** function.

Check the return value of the GetInitializeErrorString method to retrieve the result of the initialization call. Any result other than "No Error" means the initialization failed for some reason.

Return Value

Returns a value of the enumerated type ProgramStatus.

Value		Reason
0	ErrorNone	No error - initialization was successful.
5	ErrorMatchcodeNotFound	Specified Matchcode does not exist.

If any other value other than NoError is returned, check the GetInitializeErrorString method to see the reason for the error.

Syntax

```
    ProgramStatus = mdMC->InitializeDataFiles()
```

C

```
    ProgramStatus = mdMUMatchcodeInitializeDataFiles(mdMC)
```

COM+/.NET

```
    ProgramStatus = mdMC.InitializeDataFiles
```

GetInitializeErrorString

Returns a descriptive string to describe the error from the **InitializeDataFiles** function. The possible strings returned by this method are:

```
No Error
Could not open mdName.dat
Matchcode not found
```

Return Value

The **GetInitializeErrorString** function returns a string describing the error caused when the **InitializeDataFiles** function cannot be called successfully.

Syntax

```
     char = mdMC->GetInitializeErrorString()
```

C

```
     char = mdMUMatchcodeGetInitializeErrorString(mdMC)
```

COM+/.NET

```
     string = mdMC.GetInitializeErrorString
```

Create a New Matchcode

The single function in this section, part of the Matchcode class, creates a new, blank matchcode that can be populated with matchcode components and saved to the current matchcode file.

CreateNewMatchcode

This function creates a new, blank matchcode, represented by the current instance of the Matchcode class.

This function accepts a single character string, the name for the newly created matchcode. This name must be unique. It cannot be used for another matchcode in the same matchcode file.

If the function successfully creates a new matchcode, it returns a non-zero integer value. A zero value means that there was an error, most likely because the matchcode name was already in use.

Syntax

```
    int = mdMC->CreateNewMatchcode(MatchcodeName)
```

C

```
    int = mdMUMatchcodeCreateNewMatchcode(mdMC, MatchcodeName)
```

COM+/.NET

```
    integer = mdMC.CreateNewMatchcode (MatchcodeName)
```

Retrieve Existing Matchcodes

These functions are used to retrieve a specific matchcode from the matchcode file.

FindMatchcode

This function populates the current instance of the Matchcode object with the settings of the matchcode specified in the character string passed to the function. It accepts a single character string as its input parameter. This must be the name of an existing matchcode in the current matchcode file.

If the matchcode name is valid (represents an existing matchcode), this function returns an integer value of 1.

Syntax

```
    int = mdMC->FindMatchcode(MatchcodeName)
```

C

```
    int = mdMUMatchcodeFindMatchcode(mdMC, MatchcodeName)
```

COM+/.NET

```
    integer = mdMC.FindMatchcode(MatchcodeName)
```

GetMatchcodeName

This function returns a string value containing the name of the current matchcode, assuming one has been loaded via the **FindMatchcode** function or created with the **CreateNewMatchcode** function.

Syntax

```
    char = mdMC->GetMatchcodeName()
```

C

```
    char = mdMUMatchcodeGetMatchcodeName(mdMC)
```

COM+/.NET

```
    string = mdMC.MatchcodeName
```

General Matchcode Properties

These functions help with defining various Matchcode properties.

GetDescription

Retrieves a matchcode's user-specified description associated with this matchcode.

Syntax

```
    StringValue =mdMC->GetDescription()
```

C

```
    StringValue = mdMUMatchcodeGetDescription(mdMUMatchcode)
```

COM+/.NET

```
    StringValue = mdMC.Description
```

SetDescription

Allows the user to assign a description to the matchcode.

For example, it may describe what the matchcode evaluates or the type of process the matchcode is used in. When viewing the matchcode in the matchcode editor, the description will be present along with the actual properties of the matchcode.

Syntax

```
    mdMC->SetDescription(StringValue)
```

C

```
    mdMUMatchcodeSetDescription(mdMUMatchcode, StringValue)
```

COM+/.NET

```
    mdMC.Description = StringValue
```

GetNGram

Retrieves a matchcode's N-gram setting.

Since a matchcode may contain multiple components each using a different fuzzy algorithm, many of which require an N-gram setting, the N-gram setting is applied to all relevant components. In other words, the N-gram is set at the matchcode level, not the component level.

Syntax

```
integer = mdMC->GetNGram()
```

C

```
integer = mdMUMatchcodeGetNGram(mdMUMatchcode)
```

COM+/.NET

```
integer = mdMC.NGram
```

SetNGram

Sets a matchcode's N-gram setting.

Since a matchcode may contain multiple components each using a different fuzzy algorithm, many of which require an N-gram setting, the N-gram setting is applied to all relevant components. In other words, the N-gram is set at the matchcode level, not the component level.

Syntax

```
mdMC->SetNGram(integer)
```

C

```
mdMUMatchcodeSetNGram(mdMUMatchcode, integer)
```

COM+/.NET

```
mdMC.NGram = integer
```

Read Matchcode Component Information

The functions in this section retrieve the number of components in a given matchcode and retrieve the contents of a specific matchcode component.

GetMatchcodeItemCount

This function returns the number of components in the current Matchcode object. The return value is an integer indicating the number of MatchcodeComponent objects in the current Matchcode object.

Syntax

```
int = mdMC->GetMatchcodeItemCount()
```

C

```
int = mdMUMatchcodeGetMatchcodeItemCount(mdMC)
```

COM+/.NET

```
integer = mdMC.MatchcodeItemCount
```

GetMatchcodeItem

This function returns the MatchcodeComponent object located at the position indicated by the integer value passed to the function.

Syntax

```
MatchcodeComponent = mdMC->GetMatchcodeItem(int)
```

C

```
mdMUMatchcodeComponent =
mdMUMatchcodeGetMatchcodeItem(mdMC, int)
```

COM+/.NET

```
MatchcodeComponent = mdMC.MatchcodeItem(integer)
```

Get Mapping Information

Mapping information is different from component information, revealing the order and mapping types that should be used when creating the mappings in any one of the deduper interfaces.

GetMappingItemCount

This function returns an integer value showing the number of mappings required for the current matchcode.

Mapping items differ from matchcode components, mostly in how street address lines are represented. Components include the individual address components that are used to construct the match key, such as street number and street name.

The same components are represented by the mapping items for address lines (address1, address2 and address3). No matter what order the components appear in the matchcode, the address lines mapping items appear at the end of the list of mapping items.

Syntax

```
int = mdMC->GetMappingItemCount()
```

C

```
int = mdMUMatchcodeGetMappingItemCount(mdMC)
```

COM+/.NET

```
integer = mdMC.MappingItemCount
```

GetMappingItemType

This function returns the specific type of a mapping item specified by an integer value.

For a complete list of these values, see “MatchcodeMappingTarget” on page 149.

The return value is a variable of type MatchcodeMappingTarget that indicates the type of mapping item found at the position indicated by the integer value passed to the function.

Syntax

```
MatchcodeMappingTarget = mdMC->GetMappingItemType(int)
```

C

```
mdMUMatchcodeMappingTarget =
mdMUMatchcodeGetMappingItemType (mdMC, int)
```

COM+/.NET

```
MatchcodeMappingTarget = mdMC.MappingItemType(integer)
```

GetMappingItemLabel

This function returns a character string containing the label, if any, of the mapping item specified by an integer value.

If the specified mapping item does not have a label, this function returns the name of the mapping item type.

Syntax

```
char = mdMC->GetMappingItemLabel(int)
```

C

```
char= mdMUMatchcodeGetMappingItemLabel(mdMC, int)
```

COM+/.NET

```
string = mdMC.MappingItemLabel(integer)
```

Change Matchcode Component Settings

The functions in this section set the values for the various settings of a matchcode component object. They can be used to construct new matchcode components when adding them to a matchcode, or to change the settings of an existing component. Every function in this section requires a variable based on the mdMatchcodeComponent class.

SetComponentType

This function specifies the type for the current MatchcodeComponent object.

The only parameter for this function is an enumerated value of the type MatchcodeComponentType. A chart showing all possible values for this enumeration appears on page 150.

Syntax

```
mdMCC->SetComponentType(MatchcodeComponentType)
```

C

```
mdMUMatchcodeComponentSetComponentType(mdMCC,
   mdMUMatchcodeComponentType)
```

COM+/.NET

```
mdMCC.ComponentType = MatchcodeComponentType
```

SetSize

This function sets how many characters from the source data will be used by the current MatchcodeComponent.

This integer value sets the number of characters that this component will use from the related field from each record. If the field is longer than this value, the data will be truncated. If the field is shorter, it will be padded with spaces.

Size is only applied to a piece of data after all other component properties have been considered.

Syntax

```
mdMCC->SetSize(int)
```

C

```
mdMUMatchcodeComponentSetSize(mdMCC, int)
```

COM+/.NET

```
mdMCC.Size = integer
```

SetLabel

This function assigns a label to the current MatchcodeComponent object.

Not all components accept a label. For example, none of the street address components (Street number, street name and so on) can have a label since they are not used for mapping.

Components that are not assigned a label will return the name of their component type. If a label is passed to a component that cannot use one, the component will ignore it.

Syntax
`mdMCC->SetLabel(char)`
C
`mdMUMatchcodeComponentSetLabel(mdMCC, char)`
COM+/.NET
`mdMCC.Label = string`

SetWordCount

This function sets the maximum number of words used by the current MatchcodeComponent object.

The maximum number of words offers further control over the amount of data used by each component. If this function is set to 1, then MatchUp Object will take every character up to, but not including, the first space.

If the first word is shorter than the value passed to the SetSize function, the data will still be truncated at that character, regardless of the setting from this function.

Syntax
`mdMCC->SetWordCount(int)`
C
`mdMUMatchcodeComponentSetWordCount(mdMCC, int)`
COM+/.NET
`mdMCC.WordCount = integer`

SetStart

This function sets the starting point used by the current MatchcodeComponent object, Left, Right, Character or Word. It controls where MatchUp Object starts counting when applying the component size and accepts an enumerated value of the type MatchcodeStart.

If the selected value is either StartAtPos or StartAtWord, the application will need to call the **SetStartPos** function.

Name	Value	Description
Left	0x08	The default. MatchUp Object starts counting from the beginning of the field.
Right	0x10	MatchUp Object starts counting backwards from the end of the field.
StartAtPos	0x20	MatchUp Object starts counting from the character position indicated by the SetStartPos function.
StartAtWord	0x40	MatchUp Object starts counting from the word indicated by the SetStartPos function.

Syntax

```
    mdMCC->SetStart(MatchcodeStart)
```

C

```
    mdMUMatchcodeComponentSetStart(mdMCC, mdMUMatchcodeStart)
```

COM+/.NET

```
    mdMCC.Start = MatchcodeStart
```

SetStartPos

This function sets the specific character position or word used as the starting point, when StartAtPos or StartAtWord are passed to the **SetStart** function.

This function is required if the selection from the **SetStart** function is either StartAtPos or StartAtWord.

It sets either the character position or the word where MatchUp Object starts counting when adding a field to a match key.

For example, if the value passed to SetStartPos is 2 and the **SetStart** function is set to StartAtWord, MatchUp Object will start at the second word.

Syntax

```
    mdMCC->SetStartPos(int)
```

C

```
    mdMUMatchcodeComponentSetStartPos(mdMCC, int)
```

COM+/.NET

```
    mdMCC.StartPos = integer
```

SetTrim

This function enables or disables trimming of blank spaces from the beginning or end of field data through an enumerated value of the type MatchcodeTrim.

For most applications, this function will be set to All Trim, which trims excess blank spaces from both the start and end of a field before adding to a match key.

Name	Value
LeftTrim	0x02
RightTrim	0x04
AllTrim	0x06

Syntax

```
    mdMCC->SetTrim(MatchcodeTrim)
```

C

```
    mdMUMatchcodeComponentSetTrim(mdMCC, mdMUMatchcodeTrim)
```

COM+/.NET

```
    mdMCC.Trim = MatchcodeTrim
```

SetFuzzy

The function selects the matching algorithm used when comparing this MatchcodeComponent. It accepts an enumerated value of the type MatchcodeFuzzy.

Name	Value	Name	Value
Exact	0x0000	NGram	0x0800
SoundEx	0x0001	Jaro	0x1000
Phonetex	0x0002	JaroWinkler	0x2000
Containment	0x0004	LCS	0x4000
Frequency	0x0008	NeedlemanWunsch	0x8000
FastNear	0x0010	MDKeyboard	0x10000
AccurateNear	0x0020	SmithWatermanGotoh	0x20000
VowelsOnly	0x0040	Dice	0x40000

Name	Value	Name	Value
ConsonantsOnly	0x0080	Jaccard	0x80000
AlphasOnly	0x0100	Overlap	0x100000
NumericsOnly	0x0200	DoubleMetaphone	0x200000
FrequencyNear	0x0400		

For a detailed explanation of the various matching strategies, see the section beginning on page 11.

Syntax

```
    mdMCC->SetFuzzy(MatchcodeFuzzy)
```

C

```
    mdMUMatchcodeComponentSetFuzzy(mdMCC, mdMUMatchcodeFuzzy)
```

COM+/.NET

```
    mdMCC.Fuzzy = MatchcodeFuzzy
```

SetNear

This function sets the degree of precision used when the **SetFuzzy** function is set to Fast Near, Accurate Near, or Frequency Near.

The integer value from 1 to 4 sets how many differences are allowed before two keys are no longer considered a match when one of the Near matching strategies is selected with the **SetFuzzy** function.

Syntax

```
    mdMCC->SetNear(int)
```

C

```
    mdMUMatchcodeComponentSetNear(mdMCC, int)
```

COM+/.NET

```
    mdMCC.Near = integer
```

SetNearDbl

This function sets the minimum percentage of similarity which will return a match between two strings when the SetFuzzy function is set to Proximity, N-Gram, Jaro, Jaro Winkler, LCS, Needleman, MDKeyboard, Smith Waterman, Dice's Coefficient, Jaccard, Overlap Coefficient, or DoubleMetaphone algorithm.

The double value from 100 to 0 sets the minimum threshold percent similarity between two keys which will be considered a match when one of the NearDbl matching strategies is selected with the SetFuzzy function.

Syntax

```
mdMCC->SetNearDbl(double)
```

C

```
mdMUMatchcodeComponentSetNearDbl
   (mdMUMatchcodeComponent, double)
```

COM+/.NET

```
mdMCC.NearDbl(double)
```

SetFieldMatch

This function determines how MatchUp Object handles blank or partial fields when applying a matchcode. It accepts an enumerated value of the type MatchcodeFieldMatch.

Name	Value
NoFieldMatch	0x0000
BothBlankMatch	0x0100
OneBlankMatch	0x0200
InitialMatch	0x0400

These selections are not mutually exclusive. In order to select more than one, you will need to use a logical OR operation to combine multiple options and pass that value to the function.

Some languages, such as C++, do not easily handle using logical operation on enumerations. In these cases, it may be necessary to cast the enumerated values as integers and then combine them using the OR operation.

When working with the first component of a matchcode, keep in mind the special rules for first components on page 16. The first component cannot use Initial Only matching or One Blank Field matching.

For details on the different types of blank or partial field matching, see page 14.

Syntax

```
    mdMCC->SetFieldMatch(MatchcodeFieldMatch)
```

C

```
   mdMUMatchcodeComponentSetFieldMatch(mdMCC,
      mdMUMatchcodeFieldMatch)
```

COM+/.NET

```
    mdMCC.FieldMatch = MatchcodeFieldMatch
```

SetCombination

This function selects which combinations in the current matchcode will use this component. It accepts an enumerated value of the type MatchcodeCombination.

Name	Value	Name	Value
Combo1	0x0001	Combo9	0x0100
Combo2	0x0002	Combo10	0x0200
Combo3	0x0004	Combo11	0x0400
Combo4	0x0008	Combo12	0x0800
Combo5	0x0010	Combo13	0x1000
Combo6	0x0020	Combo14	0x2000
Combo7	0x0040	Combo15	0x4000
Combo8	0x0080	Combo16	0x8000

These selections are not mutually exclusive. In order to select more than one, you will need to use a logical OR operation to combine multiple options and pass that value to the function.

Some languages, such as C++, do not easily handle using logical operation on enumerations. In these cases, it may be necessary to cast the enumerated values as integers and then combine them using the OR operation.

Syntax

```
    mdMCC->SetCombination(MatchcodeCombination)
```

C

```
   mdMUMatchcodeComponentSetCombination(mdMCC,
      mdMUMatchcodeCombination)
```

COM+/.NET

```
    mdMCC.Combination = MatchcodeCombination
```

SetSwap

This function selects the swap pair or swap pairs to which this MatchcodeComponent object belongs and accepts an enumerated value of the type MatchcodeSwap.

Name	Value
NoSwap	0x00
SwapA	0x01
SwapB	0x02
SwapC	0x04
SwapD	0x08
SwapE	0x10
SwapF	0x20
SwapG	0x40
SwapH	0x80

These selections are not mutually exclusive. In order to select more than one, you will need to use a logical OR operation to combine multiple options and pass that value to the function.

Some languages, such as C++, do not easily handle using logical operation on enumerations. In these cases, it may be necessary to cast the enumerated values as integers and then combine them using the OR operation.

Syntax

```
mdMCC->SetSwap(MatchcodeSwap)
```

C

```
mdMUMatchcodeComponentSetSwap(mdMCC, mdMUMatchcodeSwap)
```

COM+/.NET

```
mdMCC.Swap = MatchcodeSwap
```

Read Matchcode Component Settings

The following functions read and return the settings from a specific matchcode component variable.

GetComponentType

This function returns the component type of the current MatchcodeComponent object.

The return value for this function is an enumerated value of the type MatchcodeComponentType. A chart showing all possible values for this enumeration appears on page 150.

Syntax

```
MatchcodeComponentType = mdMCC->GetComponentType()
```

C

```
mdMUMatchcodeComponentType=
   mdMUMatchcodeComponentGetComponentType(mdMCC)
```

COM+/.NET

```
MatchcodeComponentType = mdMCC.ComponentType
```

GetSize

This function returns how many characters from the source data will be used by the current MatchcodeComponent.

This integer value shows the number of characters that this component will use from the related field from each record. If the field is longer than this value, the data will be truncated. If the field is shorter, it will be padded with spaces.

Size is only applied to a piece of data after all other component properties have been considered.

Syntax
`int = mdMCC->GetSize()`
C
`int = mdMUMatchcodeComponentGetSize(mdMCC)`
COM+/.NET
`integer = mdMCC.Size`

GetLabel

This function returns the label, if any, of the current MatchcodeComponent object.

Not all components accept a label. For example, none of the street address components (Street number, street name, and so on) use a label because they are not used for mapping.

Components that are not assigned a label will return the name of their component type.

Syntax
`char = mdMCC->GetLabel()`
C
`char = mdMUMatchcodeComponentGetLabel(mdMCC)`
COM+/.NET
`string = mdMCC.Label`

GetWordCount

This function returns the maximum number of words used by the current MatchcodeComponent object.

The maximum number of words offers further control over the amount of data used by each component. If this function is set to 1, then MatchUp Object will take every character up to, but not including, the first space.

If the first word is shorter than the value passed to the **SetSize** function, then the data will still be truncated at that character, regardless of the setting returned by this function.

Syntax

```
    int = mdMCC->GetWordCount()
```

C

```
    int = mdMUMatchcodeComponentGetWordCount(mdMCC)
```

COM+/.NET

```
    integer = mdMCC.WordCount
```

GetStart

This function returns an enumerated value of the type MatchcodeStart that shows where MatchUp Object starts counting when applying the component size.

If the selected value is either StartAtPos or StartAtWord, the application will need to call the **GetStartPos** function to discover what starting word or character position is being used.

Name	Value	Description
Left	0x08	The default. MatchUp Object starts counting from the beginning of the field.
Right	0x10	MatchUp Object starts counting backwards from the end of the field.
StartAtPos	0x20	MatchUp Object starts counting from the character position indicated by the **SetStartPos** function.
StartAtWord	0x40	MatchUp Object starts counting from the word indicated by the **SetStartPos** function.

Syntax

```
    MatchcodeStart = mdMCC->GetStart()
```

C

```
    mdMUMatchcodeStart= mdMUMatchcodeComponentGetStart(mdMCC)
```

COM+/.NET

```
    MatchcodeStart = mdMCC.Start
```

GetStartPos

This functions returns the specific character position or word used as the starting point, when the SetStart function is set to Position or Word.

It will return an integer value when the **SetStart** function has been set to either StartAtPos or StartAtWord.

It returns either the character position or the word where MatchUp Object starts counting when adding a field to a match key.

Syntax

```
    int = mdMCC->GetStartPos()
```

C

```
    int = mdMUMatchcodeComponentGetStartPos(mdMCC)
```

COM+/.NET

```
    integer = mdMCC.StartPos
```

GetTrim

This function returns an enumerated value of the type MatchcodeTrim, showing whether the current matchcode will trim beginning or ending spaces from the data before performing other operations upon it.

For most applications, this function will return the value for All Trim, which trims excess blank spaces from both the start and end of a field before adding to a match key.

Name	Value
LeftTrim	0x02
RightTrim	0x04
AllTrim	0x06

Syntax

```
    MatchcodeTrim = mdMCC->GetTrim()
```

C

```
    mdMUMatchcodeTrim = mdMUMatchcodeComponentGetTrim(mdMCC)
```

COM+/.NET

```
    MatchcodeTrim = mdMCC.Trim
```

GetFuzzy

This function returns an enumerated value of the type MatchcodeFuzzy used when comparing this MatchcodeComponent.

Name	Value	Name	Value
Exact	0x0000	NGram	0x0800
SoundEx	0x0001	Jaro	0x1000
Phonetex	0x0002	JaroWinkler	0x2000
Containment	0x0004	LCS	0x4000
Frequency	0x0008	NeedlemanWunsch	0x8000
FastNear	0x0010	MDKeyboard	0x10000
AccurateNear	0x0020	SmithWatermanGotoh	0x20000
VowelsOnly	0x0040	Dice	0x40000
ConsonantsOnly	0x0080	Jaccard	0x80000
AlphasOnly	0x0100	Overlap	0x100000
NumericsOnly	0x0200	DoubleMetaphone	0x200000
FrequencyNear	0x0400		

For a detailed explanation of the various matching strategies, see “Matchcode Component Properties” on page 11.

Syntax

```
MatchcodeFuzzy = mdMCC->GetFuzzy()
```

C

```
mdMUMatchcodeFuzzy = mdMUMatchcodeComponentGetFuzzy(mdMCC)
```

COM+/.NET

```
MatchcodeFuzzy = mdMCC.Fuzzy
```

GetNear

This function returns the degree of precision used when the **SetFuzzy** function is set to Fast Near, Accurate Near or Frequency Near.

The integer value from 1 to 4 shows how many differences are allowed before two keys are no longer considered a match when one of the Near matching strategies is selected with the **SetFuzzy** function.

Syntax

```
int mdMCC->GetNear()
```

C

```
int = mdMUMatchcodeComponentGetNear(mdMCC)
```

COM+/.NET

```
integer = mdMCC.Near
```

GetNearDbl

This function returns the minimum percentage of similarity which will return a match between two strings when the SetFuzzy function is set to Proximity, N-Gram, Jaro, Jaro Winkler, LCS, Needleman, MDKeyboard, Smith Waterman, Dice's Coefficient, Jaccard, Overlap Coefficient, or DoubleMetaphone algorithm.

The double value from 100 to 0 shows the minimum threshold percent similarity between two keys which will be considered a match when one of the NearDbl matching strategies is selected with the SetFuzzy function.

Syntax

```
double = mdMCC->GetNearDbl()
```

C

```
double = mdMUMatchcodeComponentGetNearDbl
   (mdMUMatchcodeComponent)
```

COM+/.NET

```
double = mdMCC.NearDbl
```

GetFieldMatch

This function returns an enumerated value of the type MatchcodeFieldMatch, which determines how MatchUp Object handles blank or partial fields when applying a matchcode.

Name	Value
NoFieldMatch	0x0000
BothBlankMatch	0x0100
OneBlankMatch	0x0200

Name	Value
InitialMatch	0x0400

These selections are not mutually exclusive. In order to determine which settings are being used, you will need to use a logical AND operation to check the return value against each of the above values.

Some languages, such as C++, do not easily handle using logical operation on enumerations. In these cases, it may be necessary to cast the return values as an integer before using the AND operation to check the values.

For details on the different types of blank or partial field matching, see page 14.

Syntax

```
MatchcodeFieldMatch = mdMCC->GetFieldMatch()
```

C

```
mdMUMatchcodeFieldMatch =
   mdMUMatchcodeComponentGetFieldMatch(mdMCC)
```

COM+/.NET

```
MatchcodeFieldMatch = mdMCC.FieldMatch
```

GetCombination

This function shows which combinations in the current matchcode will use this component. It returns an enumerated value of the type MatchcodeCombination.

Name	Value	Name	Value
Combo1	0x0001	Combo9	0x0100
Combo2	0x0002	Combo10	0x0200
Combo3	0x0004	Combo11	0x0400
Combo4	0x0008	Combo12	0x0800
Combo5	0x0010	Combo13	0x1000
Combo6	0x0020	Combo14	0x2000
Combo7	0x0040	Combo15	0x4000
Combo8	0x0080	Combo16	0x8000

These selections are not mutually exclusive. In order to determine which settings are being used, you will need to use a logical AND operation to check the return value against each of the above values.

Some languages, such as C++, do not easily handle using logical operation on enumerations. In these cases, it may be necessary to cast the return values as an integer before using the AND operation to check the values.

Syntax

```
    MatchcodeCombination = mdMCC->GetCombination()
```

C

```
   mdMUMatchcodeCombination =
      mdMUMatchcodeComponentGetCombination(mdMCC)
```

COM+/.NET

```
    MatchcodeCombination = mdMCC.Combination
```

GetSwap

This function shows which swap pairs in the current matchcode will use this component. It accepts an enumerated value of the type MatchcodeSwap.

Name	Value
NoSwap	0x00
SwapA	0x01
SwapB	0x02
SwapC	0x04
SwapD	0x08
SwapE	0x10
SwapF	0x20
SwapG	0x40
SwapH	0x80

These selections are not mutually exclusive. In order to determine which settings are being used, you will need to use a logical AND operation to check the return value against each of the above values.

Some languages, such as C++, do not easily handle using logical operation on enumerations. In these cases, it may be necessary to cast the return values as an integer before using the AND operation to check the values.

Syntax

```
    MatchcodeSwap = mdMCC->GetSwap()
```

C

```
    mdMUMatchcodeSwap = mdMUMatchcodeComponentGetSwap(mdMCC)
```

COM+/.NET

```
    MatchcodeSwap = mdMCC.Swap
```

Add, Modify or Delete Matchcode Components

The functions in this section add, insert, update, or delete matchcode components from the current Matchcode object.

AddMatchcodeItem

This function adds a MatchcodeComponent object to the current Matchcode.

The **AddMatchcodeItem** function accepts a MatchcodeComponent object as its only argument and adds this component as the last component for this matchcode.

To add a component at any position other than the last component, use the **InsertMatchcodeItem** function instead. To modify an existing matchcode, use the **ChangeMatchcodeItem** function.

This function returns an enumerated value of the type MatchcodeStatus that indicates if the component was successfully added to the matchcode and, if not, the reason for the error.

Name	Value
MCNoError	0
MCFirstComponentFuzzyOptions	1
MCFirstComponentNoSwapPair	2
MCDataTypeNoFuzzy	3
MCComponentFuzzyIncorrectSize	4

Name	Value
MCDataTypeNoMaximumNumberWords	5
MCDataTypeNoStartRightOrWordOrPos	6
MCIncorrectMaximumNumberWords	7
MCNearOutOfRange	8
MCFirstComponentNotUsedInEveryCondition	9
MCCannotChangeFirstComponent	10
MCInvalidSwapPair	11

Syntax

```
MatchcodeStatus = mdMC-
   >AddMatchcodeItem(MatchcodeComponent)
```

C

```
mdMUMatchcodeStatus = mdMUMatchcodeAddMatchcodeItem(mdMCC,
   mdMUMatchcodeComponent)
```

COM+/.NET

```
MatchcodeStatus =
   mdMC.AddMatchcodeItem(MatchcodeComponent)
```

InsertMatchcodeItem

This function adds a MatchcodeComponent to a specific position in the component order of the current Matchcode object.

Use this function to add a new MatchcodeComponent object to the current matchcode in any position other than the very last. The **InsertMatchcodeItem** function accepts two arguments, the MatchcodeComponent object to be added and an integer value indicating the position where the component is to be inserted. The integer value can be from one to the number of components currently stored in the current Matchcode object.

This function returns an enumerated value of the type MatchcodeStatus that indicates if the component was successfully added to the matchcode and, if not, then the reason for the error. For a list of the possible values, see the table on page 150.

Syntax

```
    MatchcodeStatus = mdMC-
        >InsertMatchcodeItem(MatchcodeComponent, int)
```

C

```
    mdMUMatchcodeStatus =
        mdMUMatchcodeInsertMatchcodeItem(mdMCC,
        mdMUMatchcodeComponent, int)
```

COM+/.NET

```
    MatchcodeStatus =
        mdMC.InsertMatchcodeItem(MatchcodeComponent, integer)
```

ChangeMatchcodeItem

This function replaces the MatchcodeComponent object at a specific position in the component order of the current Matchcode object.

Use this function to replace an existing MatchcodeComponent object with a modified or new component. The **ChangeMatchcodeItem** function accepts two arguments: the MatchcodeComponent object to be added; and an integer value indicating the position where the component is to be replaced. The integer value can be from one to the number of components currently stored in the current Matchcode object.

This function returns an enumerated value of the type MatchcodeStatus that indicates if the component was successfully added to the matchcode and, if not, then the reason for the error. For a list of the possible values, see the table on page 150.

Syntax

```
    MatchcodeStatus = mdMC-
        >ChangeMatchcodeItem(MatchcodeComponent, int)
```

C

```
    mdMUMatchcodeStatus =
        mdMUMatchcodeChangeMatchcodeItem(mdMCC,
        mdMUMatchcodeComponent, int)
```

COM+/.NET

```
    MatchcodeStatus =
        mdMC.ChangeMatchcodeItem(MatchcodeComponent, integer)
```

DeleteMatchcodeItem

Use this function to remove a specific MatchcodeComponent object from a Matchcode object. The **DeleteMatchcodeItem** function accepts a single argument, the integer value indicating the position where the component is to be deleted. The integer value can be from one to the number of components currently stored in the current Matchcode object.

This function returns an enumerated value of the type MatchcodeStatus that indicates if the component was successfully added to the matchcode and, if not, then the reason for the error. For a list of the possible values, see the table on page 150.

Syntax

```
MatchcodeStatus = mdMC->DeleteMatchcodeItem(int)
```

C

```
mdMUMatchcodeStatus =
   mdMUMatchcodeDeleteMatchcodeItem(int)
```

COM+/.NET

```
MatchcodeStatus = mdMC.DeleteMatchcodeItem(integer)
```

Save Changes to the Matchcode File

The functions in this section save changes to the current matchcode file, either back to the original default file or to a new file.

Save

Use this function to save the current matchcode to the default matchcode file used by MatchUp Object. If an existing matchcode was edited, then that matchcode will be overwritten with the current Matchcode object. If the current Matchcode object was newly created using the **CreateNewMatchcode** function, then this matchcode will be added to the current file.

Syntax

```
mdMC->Save()
```

C

```
mdMUMatchcodeSave(mdMCC)
```

COM+/.NET

```
mdMC.Save
```

SaveToFile

Use this function to save the current matchcode to a new copy of the current matchcode file in a location specified by a character string that contains a valid path to an existing directory and valid filename.

Syntax

```
mdMC->SaveToFile(char)
```

C

```
mdMUMatchcodeSaveToFile(mdMCC, char)
```

COM+/.NET

```
mdMC.SaveToFile(string)
```

RenameMatchcode

Allows you to change a matchcode's name.

This function can be used when you want to edit an existing matchcode and have that new functionality reflected in the matchcode name.

Syntax

```
int = mdMC->RenameMatchcode(StringValue)
```

C

```
int = mdMUMatchcodeRenameMatchcode(mdMUMatchcode,
   StringValue)
```

COM+/.NET

```
int = mdMC.RenameMatchcode(StringValue)
```

DeleteMatchcode

Delete a matchcode.

Calling this function will permanently remove the matchcode from the MatchUp mdMatchUp.mc matchcode database.

Syntax

```
int = mdMC->DeleteMatchcode()
```

C

```
int = mdMUMatchcodeDeleteMatchcode(mdMUMatchcode)
```

COM+/.NET

```
int = mdMC.DeleteMatchcode()
```

Appendix

Enumerations & International Deduping

Enumerations

The following section lists values for the global enumerations used by all interfaces of MatchUp Object.

MatchcodeCombinations

This enumeration is used by the Matchcode Editor interface to set or read the combinations for which particular field is used. These values can also be used by the Incremental and Read/Write interfaces to determine which combinations were matched when a duplicate record is found.

Name	Hex value	Decimal Value	Name	Hex value	Decimal Value
Combo1	0x0001	1	Combo9	0x0100	256
Combo2	0x0002	2	Combo10	0x0200	512
Combo3	0x0004	4	Combo11	0x0400	1,024
Combo4	0x0008	8	Combo12	0x0800	2,048
Combo5	0x0010	16	Combo13	0x1000	4,096
Combo6	0x0020	32	Combo14	0x2000	8,192
Combo7	0x0040	64	Combo15	0x4000	16,384
Combo8	0x0080	128	Combo16	0x8000	32,768

MatchcodeMapping

The MatchcodeMapping enumeration is used by the **AddMapping** function in the Incremental, Read/Write, and Hybrid dedupers.

Name	Value	Name	Value
Prefix	1	CityStZip	20
Gender	2	Country	21
First	3	CanadianPostalCode	22
MixedFirst	4	UKCity	23
Middle	5	UKCounty	24
Last	6	UKPostcode	25
MixedLast	7	UKCityCountyPC	26
Suffix	8	Phone	27
FullName	9	EMail	28
InverseName	10	CreditCard	29
GovernmentInverseName	11	General	30
Title	12	Latitude	40
Company	13	Longitude	41
Address	14	Date	42
City	15	Numeric	43
State	16	Address1	250
Zip9	17	Address2	251
Zip5	18	Address2	252
Zip4	19		

MatchcodeMappingTarget

This enumeration is used by the Matchcode Interface to read the type of mapping item required at the position indicated by the integer value passed to the GetMappingItemType function.

After a successful call to GetMappingItemCount, you can find the required Matchcode Mapping for each item by calling the GetMappingItemType function, which returns an enumerated value. This function is available because matchcodes using address components often contain a differnet number of components than the number of required mappings.

Name	Value	Name	Value
PrefixType	1	CountryType	18
FirstType	2	CanadianPCType	19
MiddleType	3	UKCityType	20
LastType	4	UKCountyType	21
SuffixType	5	UKPCType	22
GenderType	6	PhoneType	23
FirstNicknameType	7	EMailType	24
MiddleNicknameType	8	CreditCardType	25
TitleType	9	GeneralType	26
CompanyType	10	Address1Type	28
CompanyAcronymType	11	Address2Type	29
AddressType	12	Address3Type	30
CityType	13	LatitudeType	34
StateType	14	LongitudeType	35
Zip9Type	15	DateType	36
Zip5Type	16	NumericType	37
Zip4Type	17		

MatchcodeComponentType

The MatchcodeComponentType enumeration is the only parameter of the **SetComponentType** function and the return value of the GetComponentType function, both in the Matchcode API.

Name	Value	Name	Value
Prefix	1	Address	19
First	2	City	20
Middle	3	State	21
Last	4	Zip9	22
Suffix	5	Zip5	23
Gender	6	Zip4	24
FirstNickname	7	Country	28
MiddleNickname	8	CanadianPC	29
Title	9	UKCity	30
Company	10	UKCounty	31
CompanyAcronym	11	UKPC	32
StreetNumber	12	Phone	33
StreetPreDir	13	EMail	34
StreetName	14	CreditCard	35
StreetSuffix	15	General	36
StreetPostDir	16	GeoDistance	38
POBox	17	Date	39
Secondary	18	Numeric	40

International Deduping Considerations

Foreign Character Translation

Foreign characters are translated into English equivalents. For example, “Ç” is converted to “C.” All translations are based on the assumption that your data was entered with the 1252 (Windows Latin 1) code page.

Canadian Users

MatchUp recognizes Canadian provinces and postal codes. In fact, it will abbreviate province names to their two letter abbreviation automatically.

MatchUp does handle the “QC” province abbreviation for Quebec, and “PQ” entries are automatically changed to “QC.”

In Canada, ”5-20 Main Street” means “20 Main Street, Apt 5,” but in the US, it means ”5 Main Street, Apt 20.” When deduping, MatchUp uses the contents of the ZIP/Postal code as a basis to determine a record's country of origin, and splits this type of address accordingly.

When creating matchcodes for use with Canadian Postal Codes, use the Postal Code component. However, if a database is a mix of US and Canadian records, use Zip9 as the component type. Zip9 will not adversely affect processing of Canadian records. The goal is to prevent the deduper from trying to extract a ZIP + 4 from a Canadian Postal Code.

United Kingdom Users

MatchUp can recognize United Kingdom Cities, Counties, and Postal codes. When creating matchcodes for use with United Kingdom addresses, use the Postal code (UK) component. Depending on requirements, consider using the City (UK) and County (UK) components. The Postal code component is structured in the following format: AADDIII, where AA is the Postal code Area (left justified), DD is the Postal code district (right justified), and III is the Inward Code (left justified). Extra spaces and dashes are removed as this structuring is done, so the size of this component is always 7.

Like any other matchcode component, a portion of the Postal code can always be compared by reducing its size and/or starting at a specific position. For example, starting at position 5 for a size of 3 will compare just the Inward code.

MatchUp’s street splitter will not split United Kingdom street addresses as well as Canadian and US addresses. Usually, a matchcode containing a mix of split address components and full address components is a good way to get the benefit of the street splitter (which often does perform well), along with a full-address match for backup. MatchUp Object includes the United Kingdom Address matchcode to be used as a starting point to build on.

International Users

MatchUp was designed to work with US and Canadian addresses, and performs well with addresses from other English speaking countries.

The main obstacle with international records is with the Street Splitter. Try doing a test run with one of the default matchcodes. If the street splits are not working well, use the full address when creating a matchcode instead of using the components (such as street number, street name, etc.).

Often, users have had success when combining the full address and street splitter. For example, here's an international version of one of the default matchcodes:

Component	Size	Start	Fuzzy	Short/Empty	1	2	3
General	10	Left	No	Both Empty	X	X	X
Last Name	5	Left	No	Both Empty	X	X	X
First Name	3	Left	No	Both Empty	X	X	X
PO Box	10	Left	No	No	X		
Street #	4	Left	No	Both Empty		X	
Street Name	4	Left	No	No		X	
Full Address	20	Left	No	No			X