m209 Documentation

Release 0.1

Brian Neal

July 23, 2013

CONTENTS

Author Brian Neal <bgneal@gmail.com>

Version 0.1

Date July 23, 2013

Home Page https://bitbucket.org/bgneal/m209/

License MIT License (see LICENSE.txt)

Documentation http://m209.readthedocs.org/

Support https://bitbucket.org/bgneal/m209/issues

CHAPTER

ONE

INTRODUCTION

The M-209 is a mechanical cipher machine used by the US military during World War II and up to the Korean War. The M-209 is also known as the CSP-1500 by the US Navy. The M-209 is an example of a Hagelin device, a family of mechanical cipher machines created by Swedish inventor Boris Hagelin, where it is known as the C-38.

`m209` is a complete M-209 simulation library and command-line application written in Python 3. `m209` is historically accurate, meaning that it can exchange messages with an actual M-209 cipher machine.

It is hoped that this library will be useful to M-209 enthusiasts, historians, and students interested in cryptography.

`m209` strives to be Pythonic, easy to use, and comes with both unit tests and documentation. `m209` is a library for building applications for encrypting and decrypting M-209 messages. `m209` also ships with a simple command-line application that can encrypt & decrypt messages for scripting and experimentation.

CHAPTER

TWO

DOCUMENTATION

Contents:

2.1 Tutorials

2.1.1 Command-line Tutorial

In order for two parties to exchange M-209 messages, each must set up their device in exactly the same manner. This was accomplished by publishing key lists in code books which were distributed to end users. A code book instructed users on what key list to use on any given day in a given month. Each key list detailed the numerous wheel pin and lug settings that needed to be made for a given day. Because there are so many settings, the `m209` utility allows users to store key lists in a key file for convenience. So let us first create a key file that holds 30 key lists:

```
$ m209 keygen -n 30
```

This command randomly creates 30 key lists and stores them in a file called `m209keys.cfg` by default. We did not specify a starting key list indicator, so 30 random ones were chosen. The first 13 lines of our new key file are displayed below:

```
$ head -n 13 m209keys.cfg
[AB]
lugs = 0-4*4 0-5*6 1-0*10 2-0*2 3-0 3-5*2 3-6 4-5
wheel1 = BDFGIKRSTUWX
wheel2 = BCEJKLORSUX
wheel3 = CFHJKLMQSTU
wheel4 = ABCDHIJMOPRTU
wheel5 = BCEFINPS
wheel6 = ACDEHJN
check = GZWUU SFYQN NFAKK FXSEN FAFMF B

[AK]
lugs = 0-4*2 0-5*9 0-6 1-0*3 1-2 1-5 1-6*2 3-0*8
wheel1 = ABDEFHIJMQSUXZ
```

Note: If you are following along at home, you'll probably get different output than what is shown here. This is because the key lists are generated at random, and it is very unlikely that your key list will match mine!

Here we can see that the first key list in our file has the indicator `AB` (shown in square brackets), and we can see the settings for the lugs and six wheels. This notation is explained later (see *Key list file format*). Also included is a so-called check string. Because there are so many settings, it is quite error-prone to set up an M-209. This check string

allows the operator to verify their work. After configuring the M-209 with the given settings, the operator can set the six key wheels to AAAAAA, then encipher the letter A 26 times. If the message that appears on the paper tape matches the check string, the operator knows the machine is set up correctly for the day.

After the key list AB, the key list AK starts, and so on for all 30 key lists.

Now that we have created a key file, we can encrypt our first message. The m209 utility has many options to let you have fine control over the various encryption parameters. These are explained in detail later. If you omit these parameters they are simply chosen at random. Here is the simplest example of encrypting a message:

```
$ m209 encrypt -t "THE PIZZA HAS ARRIVED STOP NO SIGN OF ENEMY FORCES STOP"
IIPDU FHLMB LASGD KTLDO OSRMZ PWGEB HYMCB IKSPT IUEPF FUHEO NQTWI VTDPC GSPQX IIPDU FHLMB
```

What just happened here? Since we did not specify a key file, the default m209keys.cfg was used. Since we did not specify a key list indicator, one was chosen randomly from the key file. Other encryption parameters, explained later, were also randomly chosen. Next, the message given on the command-line was encrypted using the standard US Army procedure described in *References* (see [5] and [7]). This resulted in the encrypted message, which is displayed in 5-letter groups. Notice that the first and last 2 groups are identical. These are special indicators that tell the receiver how to decrypt the message. In particular note that the last 2 letters in the second and last groups are MB. This is the key list indicator and tells the receiver what key list was used. The remaining groups in the middle make up the encrypted message.

Astute M-209 enthusiasts will note that our message included spaces. Actual M-209 units only allow the input of the letters A through Z. Whenever a space was needed, the operator inserted the letter Z. The m209 utility automatically performs this substitution for convenience.

Let's suppose our message was then sent to our recipient, either by courier, Morse code over radio, or in the modern age, email or even Twitter. In order for our receiver to decrypt our message they must also have the identical key list named MB. We will assume for now that our key file, m209keys.cfg was sent to our receiver earlier in some secure manner. The receiver then issues this command:

```
$ m209 decrypt -t "IIPDU FHLMB LASGD KTLDO OSRMZ PWGEB HYMCB IKSPT IUEPF FUHEO NQTWI VTDPC GSPQX IIPD
THE PI  A HAS ARRIVED STOP NO SIGN OF ENEMY FORCES STOP
```

Here again, since no key file was explicitly specified, the file m209keys.cfg was used. The file was searched for the key list MB. Then the standard Army procedure was followed, making use of the indicator groups to decrypt the message, which is displayed as output.

But wait, what happened to our Pizza? Why are the Z's missing? This is how an actual M-209 operates. Recall that an operator must substitute a letter Z whenever a space is needed. The M-209 helpfully replaces the letter Z in the decrypt output with a space as an aid to the operator. As a side effect, legitimate uses of the letter Z are blanked out. Usually it is clear from context what has happened, and the operator has to put the Z's back into the message before passing it up the chain of command.

It may also happen that the original message did not fit perfectly into an even number of 5-letter groups. In that case the encrypted message would be padded with X characters according to procedure. Upon decrypt, these X characters would appear as garbage characters on the end of the message. The receiving operator would simply ignore these letters. Note that our message did not exhibit this behavior.

This is all you need to know to start creating your own M-209 messages! For more details, consult the *Command-line Reference*.

2.1.2 Library Tutorial

Here is one way to perform the encrypt and decrypt operations from the command-line tutorial, above. In order to produce the same output, we explicitly specify the encryption parameters: the key list, the external message indicator, and the system indicator. These parameters are explained in *References* [5] & [7].

```
"""Example of how to perform an encrypt operation using the standard
procedure. Assumes a key file named m209keys.cfg exists in the current directory
and contains the key list with indicator MB.

"""
from m209.procedure import StdProcedure
from m209.keylist.config import read_key_list

key_list = read_key_list('m209keys.cfg', 'MB')
if key_list:
    proc = StdProcedure(key_list=key_list)
    plaintext = "THE PIZZA HAS ARRIVED STOP NO SIGN OF ENEMY FORCES STOP"
    msg = proc.encrypt(plaintext, spaces=True, ext_msg_ind='PDUFHL', sys_ind='I')
    print(msg)
else:
    print("Key list MB not found")
```

This program outputs:

```
IIPDU FHLMB LASGD KTLDO OSRMZ PWGEB HYMCB IKSPT IUEPF FUHEO NQTWI VTDPC GSPQX IIPDU FHLMB
```

A decrypt is just a bit more complicated. After constructing a `StdProcedure` object, you hand it the encrypted message to analyze. The procedure object examines the groups in the message and extracts all the indicators. These are returned as a `DecryptParams` named tuple which indicates, amongst other things, what key list is required. It is then up to you to obtain this key list somehow. Here we use the `read_key_list()` function to do so. After installing the key list into the procedure object with `set_key_list()`, you can finally call `decrypt()`:

```
"""Example of how to perform a decrypt operation using the standard
procedure. Assumes a key file named m209keys.cfg exists in the current directory
and contains the key list with indicator MB.

"""
from m209.procedure import StdProcedure
from m209.keylist.config import read_key_list

msg = ('IIPDU FHLMB LASGD KTLDO OSRMZ PWGEB HYMCB IKSPT IUEPF FUHEO NQTWI VTDPC'
       ' GSPQX IIPDU FHLMB')

proc = StdProcedure()
params = proc.set_decrypt_message(msg)
key_list = read_key_list('m209keys.cfg', params.key_list_ind)
if key_list:
    proc.set_key_list(key_list)
    plaintext = proc.decrypt()
    print(plaintext)
else:
    print("Key list '{}' not found".format(params.key_list_ind))
```

This program prints:

```
THE PI  A HAS ARRIVED STOP NO SIGN OF ENEMY FORCES STOP
```

2.2 Command-line Reference

2.2.1 Overview

The `m209` command-line utility performs three functions:

- Creates key list files
- Encrypts text, either given on the command line or read from a file
- Decrypts text, either given on the command line or read from a file

These functions are implemented as sub-commands. To see the list of sub-commands and options common to all sub-commands, use the `-h` or `--help` option:

```
$ m209 --help
usage: m209 [-h] [-l {debug,info,warning,error,critical}]
            {encrypt,enc,decrypt,dec,keygen,key} ...

M-209 simulator and utility program

optional arguments:
  -h, --help            show this help message and exit
  -l {debug,info,warning,error,critical}, --log {debug,info,warning,error,critical}
                        set log level [default: warning]

list of commands:
  type m209 {command} -h for help on {command}

  {encrypt,enc,decrypt,dec,keygen,key}
    encrypt (enc)       encrypt text from file or command-line
    decrypt (dec)       decrypt text from file or command-line
    keygen (key)        generate key list
```

The `-l` / `--log` options control the verbosity of output. Currently only the `keygen` sub-command makes use of this option.

Each sub-command has an alias for those who prefer shorter commands.

2.2.2 Keygen sub-command

`keygen`, or `key` for short, is the sub-command that pseudo-randomly creates key list files for use by the `encrypt` and `decrypt` sub-commands, as well as by the `m209` library routines.

Help on the `keygen` sub-command can be obtained with the following invocation:

```
$ m209 keygen --help
usage: m209 keygen [-h] [-z KEY_FILE] [-o] [-s XX] [-n NUMBER]

Generate key list file

optional arguments:
  -h, --help            show this help message and exit
  -z KEY_FILE, --key-file KEY_FILE
                        path to key list file [default: m209keys.cfg]
  -o, --overwrite       overwrite key list file if it exists
  -s XX, --start XX     starting indicator; if omitted, random indicators are
                        used
```

```
  -n NUMBER, --number NUMBER
                        number of key lists to generate [default: 1]
```

The options for `keygen` are described below.

-z or --key-file This option names the key list file. If not supplied, this defaults to `m209keys.cfg`. Note that the other sub-commands also have this option, and they too use the same default value.

-o or --overwrite This switch must be present if the key list file already exists. It provides confirmation that the user wants to overwrite an existing file. If the key list file already exists, and this option is not supplied, this sub-command will exit with an error message and the original key list file will be unchanged.

-s or --start This option sets the starting indicator for the key list file. Key list indicators are two letters in the range `AA` to `ZZ`. For example, `keygen` can be told to create 3 key lists, starting with indicator `AA`. In this case the key lists `AA`, `AB`, and `AC` would be written to the file. If this parameter is omitted, `keygen` picks indicators at random. Key list indicators simply name the key list, and are placed in the leading and trailing groups of encrypted messages to tell the receiver which key list was used to create the message. Both sender and receiver must have the same key list (name and contents) to communicate.

-n or --number This option specifies the number of key lists to generate. The default value is 1.

Note: The algorithm the `keygen` sub-command uses to generate key lists is based on the actual US Army procedure taken from the 1944 manual. This procedure is somewhat loosely specified and a lot is left up to the soldier creating the key list. The `keygen` algorithm is ad-hoc and uses simple heuristics and random numbers to make decisions. Occasionally this algorithm may fail to generate a key list that meets the final criteria defined in the manual. If this happens an error message will be displayed and no key list file will be created. It is suggested to simply run the command again as it is not likely to happen twice in a row.

Keygen examples

To generate 30 key lists in the default key list file (`m209keys.cfg`) with random indicators, and overwriting the key list file if it exists:

```
$ m209 keygen -o -n 30
$ m209 key --overwrite --number=30
```

To generate 5 key lists that sequentially start with the indicator `BN` in the key list file `m209/keys/november/keys.cfg`:

```
$ m209 keygen -z m209/keys/november/keys.cfg -s BN -n 5
```

2.2.3 Encrypt sub-command

`encrypt`, or `enc`, is the sub-command used to encrypt text. To get help on the `encrypt` command, type the following:

```
$ m209 encrypt -h
usage: m209 encrypt [-h] [-z KEY_FILE] [-f FILE] [-t TEXT] [-k XX] [-e ABCDEF]
                    [-s S]

Encrypt text from a file or command-line

optional arguments:
  -h, --help            show this help message and exit
  -z KEY_FILE, --key-file KEY_FILE
```

```
                        path to key list file [default: m209keys.cfg]
  -f FILE, --file FILE  path to plaintext file or - for stdin
  -t TEXT, --text TEXT  text string to encrypt
  -k XX, --key-list-ind XX
                        2-letter key list indicator; if omitted a random one
                        is used
  -e ABCDEF, --ext-ind ABCDEF
                        6-letter external message indicator; if omitted a
                        random one is used
  -s S, --sys-ind S     1-letter system indicator; if omitted a random one is
                        used

Either the -f/--file or -t/--text arguments must be supplied
```

The options to the `encrypt` command are described below.

`-z` or `--key-file` This option names the key list file. If not given, the default of `m209keys.cfg` is used.

`-f` or `--file` This option specifies the file that contains the text to encrypt. If the filename is given as `-` then input is read directly from `stdin`. Note that either this option or the `-t` option must be specified, but not both.

`-t` or `--text` This option specifies the text to encrypt on the command-line. Depending upon your system, you'll probably have to quote or escape your text. Note that you must either specify this option or the `-f` option, but not both.

`-k` or `--key-list-ind` This option specifies the two-letter key list indicator to use. Valid values range from `AA` to `ZZ`. The key list with this indicator must be in the key list file given by the `-z` option. If this option is omitted, a key list from the key list file is chosen at random.

`-e` or `--ext-ind` This option specifies the six-letter external message indicator, which is an encryption parameter as explained in the 1944 manual (see *References* [5] & [7]). Each letter must exist on the key wheels from left to right. If this option is omitted, an external message indicator is chosen at random.

`-s` or `--sys-ind` This option specifies the one-letter system indicator, which is an encryption parameter as explained in the 1944 manual (see *References* [5] & [7]). This must be a letter from `A` to `Z`. If not given, one is chosen at random.

Note: An actual M-209 can only accept the letters `A-Z`. When using an actual M-209, space characters must be input as the letter `Z`. Numbers must typically be spelled out as words or some other agreed-upon convention. Likewise with punctuation. To make encryption more convenient, the `m209` program will accept spaces and automatically convert them to the letter `Z`. Lowercase letters will automatically be converted to uppercase. All other characters will be silently dropped from the input. This applies to both text read on the command-line with the `-t` option and text read from files (`-f`).

Encrypt examples

To encrypt a simple string on the command-line using the default key file and random encryption parameters:

```
$ m209 encrypt -t "Rendezvous at zero seven thirty"
BBEPH SSLBY RKHWO OBAJB VYQEQ NJHGV FWRCJ UZHMB PXXXX BBEPH SSLBY
```

To save the encrypted text to a file:

```
$ m209 encrypt -t "Rendezvous at zero seven thirty" > secret.txt
```

To read the contents of a file and encrypt it, saving it to a new file:

```
$ m209 enc -f message.txt > secret.txt
```

To explicitly specify encryption parameters, and read text from `stdin`:

```
$ cat message.txt | m209 enc --file=- -k SU -e ZQGMFO -s A
```

2.2.4 Decrypt sub-command

`decrypt`, or `dec`, is the sub-command used to decrypt text. To get help on the `decrypt` command, type the following:

```
$ m209 decrypt --help
usage: m209 decrypt [-h] [-z KEY_FILE] [-f FILE] [-t TEXT]

Decrypt text from a file or command-line

optional arguments:
  -h, --help            show this help message and exit
  -z KEY_FILE, --key-file KEY_FILE
                        path to key list file [default: m209keys.cfg]
  -f FILE, --file FILE  path to ciphertext file or - for stdin
  -t TEXT, --text TEXT  text string to decrypt

Either the -f/--file or -t/--text arguments must be supplied
```

The options to the `decrypt` command are described below.

`-z` or `--key-file` This option names the key list file. If not given, the default of `m209keys.cfg` is used.

`-t` or `--file` This option specifies the file that contains the text to decrypt. If the filename is given as `-` then input is read directly from `stdin`. Note that either this option or the `-t` option must be specified, but not both.

`-t` or `--text` This option specifies the text to decrypt on the command-line. Depending upon your system, you'll probably have to quote or escape your text. Note that you must either specify this option or the `-f` option, but not both.

Note: The first and last 2 groups of an encrypted message contain the information needed to decrypt the message: the system indicator, the external message indicator, and the key list indicator. If the key list file given to the decrypt command does not contain the key list used to encrypt the message, then the message cannot be decrypted and an error message will be displayed.

Decrypt examples

To decrypt a simple string on the command-line using the default key file:

```
$ m209 decrypt -t "BBEPH SSLBY RKHWO OBAJB VYQEQ NJHGV FWRCJ UZHMB PXXXX BBEPH SSLBY"
RENDE VOUS AT  ERO SEVEN THIRTYXSJQ
```

To save the decrypted text to a file:

```
$ m209 decrypt -t "BBEPH SSLBY RKHWO OBAJB VYQEQ NJHGV FWRCJ UZHMB PXXXX BBEPH SSLBY" > msg.txt
```

To read the contents of a file and decrypt it, saving it to a new file:

```
$ m209 dec -f secret.txt > msg.txt
```

To decrypt from `stdin`:

```
$ cat secret.txt | m209 dec -f -
RENDE VOUS AT  ERO SEVEN THIRTYXSJQ
```

Note: In this example, the last group of the encrypted message only has one letter. It was padded out to five letters with `X`'s by the encryption process, and thus four "garbage" letters appear at the end in the decrypted output.

Note also that the `Z` in `RENDEZVOUS` and `ZERO` were converted to spaces by the decrypt process.

In both of these cases the operator would have to "fix up" the message before passing it up the chain of command.

2.3 Library Reference

This section of the documentation is aimed at developers who wish to use the `m209` library as part of their own application. This documentation covers the major classes and functions.

2.3.1 Exceptions

The `m209` library defines an exception hierarchy, rooted at the `M209Error` class. These exceptions are briefly described below.

class `m209.`**`M209Error`**
The base exception in the hierarchy. It inherits from the built in Python `Exception` class.

class `m209.drum.`**`DrumError`**
Inherits from `M209Error`. This exception is used to report drum related errors.

class `m209.key_wheel.`**`KeyWheelError`**
Inherits from `M209Error`. This exception is used to report key wheel related errors.

class `m209.keylist.generate.`**`KeyListGenError`**
Inherits from `M209Error`. This is public exception, used to report errors during the key list generation process.

class `m209.procedure.`**`ProcedureError`**
Inherits from `M209Error`. This is public exception, used to report errors during `StdProcedure` operations.

2.3.2 Key lists

Key lists are represented as a named tuple called `KeyList`.

class `m209.keylist.`**`KeyList`**(*indicator*, *lugs*, *pin_list*, *letter_check*)
As a named tuple, `KeyList` has the following attributes:

- `indicator` - the string name for the `KeyList`; must be 2 letters in the range `AA` - `ZZ`
- `lugs` - a string representing the drum lug settings; see below
- `pin_list` - a list of six strings which represent key wheel pin settings; see below
- `letter_check` - a string representing the letter check used to verify operator settings; if unknown this can be `None` or an empty string

Lug settings format

Drum lug settings are often conveniently represented as strings consisting of at most 27 whitespace-separated pairs of integers separated by dashes. For example:

```
lugs = '1-0 2-0 2-0 0-3 0-5 0-5 0-5 0-6 2-4 3-6'
```

Each integer pair must be in the form `m-n` where m & n are integers between 0 and 6, inclusive. Each integer represents a lug position where 0 is a neutral position, and 1-6 correspond to key wheel positions. If m & n are both non-zero, they cannot be equal.

If a string has less than 27 pairs, it is assumed all remaining bars have both lugs in the neutral positions, i.e. `0-0`.

The order of the pairs within the string does not matter.

To reduce typing and to aid in readability, an alternate shortcut notation is supported:

```
lugs = '1-0 2-0*2 0-3 0-5*3 0-6 2-4 3-6'
```

Any pair that is suffixed by `*k`, where k is a positive integer, means there are `k` copies of the preceeding lug pair combination. For example, these two strings describe identical drum configurations:

```
lugs1 = '2-4 2-4 2-4 0-1 0-1'
lugs2 = '2-4*3 0-1*2'
```

Key wheel pin settings

Key wheel pin settings are represented as iterables of letters whose pins are slid to the "effective" position (to the right). Letters not appearing in this sequence are considered to be in the "ineffective" position (to the left). If None or empty, all pins are set to be ineffective.

Examples:

```
all_ineffective = ''
wheel1 = 'ABDEFHIJMQSUXZ'
wheel2 = 'EINPQRTVXZ'
wheel3 = 'DEFGIKNOSUX'
wheel4 = 'BFGJKRS'
wheel5 = 'ABCDFGHIJMPS'
wheel6 = 'ADEFHIJKN'
```

Key list example

An example of using the `KeyList` is:

```
from m209.keylist import KeyList

key_list1 = KeyList(
       indicator='AA',
       lugs='0-4 0-5*4 0-6*6 1-0*5 1-2 1-5*4 3-0*3 3-4 3-6 5-6',
       pin_list=[
           'FGIKOPRSUVWYZ',
           'DFGKLMOTUY',
           'ADEFGIORTUVX',
           'ACFGHILMRSU',
           'BCDEFJKLPS',
           'EFGHIJLMNP'
```

```
    ],
    letter_check='QLRRN TPTFU TRPTN MWQTV JLIJE J')
```

Key list file I/O

Key lists can be stored in files in config file ("INI") style format using functions found in the `m209.keylist.config` module.

`m209.keylist.config.`**`read_key_list`**(*fname*[, *indicator=None*])

Reads key list information from the file given by `fname`.

Searches the config file for the key list with the given indicator. If found, returns a `KeyList` object. Returns `None` if not found.

If `indicator` is `None`, a key list is chosen from the file at random.

`m209.keylist.config.`**`write`**(*fname*, *key_lists*)

Writes the key lists to the file named `fname` in config file format.

`key_lists` must be an iterable of `KeyList` objects.

Key list file format

An example key list file in config file format is presented below. The label for each section of the file is the key list indicator.

```
[CA]
lugs = 0-5*5 0-6*2 1-0*7 1-2 1-3*3 1-6 2-0 3-0*3 3-5*2 3-6 4-5
wheel1 = ABCDFGHJLOPRVWYZ
wheel2 = BCDEIJKPQSUVX
wheel3 = ACDGLNQRSTUV
wheel4 = FGHIJNQRSU
wheel5 = DEIJOQS
wheel6 = BCDEILMNOP
check = RGPRO RTYOO TWYSN GXTPF PNWIH P

[CD]
lugs = 0-4*4 0-5 1-0*7 1-2*2 1-4*3 2-0*2 2-4*2 2-6*2 3-0*4
wheel1 = AEFHIKMPQRSUVZ
wheel2 = ABFGHINORSUVZ
wheel3 = BDEHJKLMNOQRSU
wheel4 = CDEFGHJKMRU
wheel5 = FGHIJOQS
wheel6 = EGIJKLP
check = ZRLWL YRMIZ RZOPN UWMVZ DVGPM H
```

Generating key lists

The `m209` library contains a function to pseudo-randomly generate a key list that is based on the procedure described in the 1944 M-209 manual (see *References* [4]).

`m209.keylist.generate.`**`generate_key_list`**(*indicator*[, *lug_selection=None*[, *max_lug_attempts=MAX_LUG_ATTEMPTS*[, *max_pin_attempts=MAX_PIN_ATTEMPTS*]]])

The only required parameter is `indicator`, the two-letter indicator for the key list.

If successful, a `KeyList` object is returned.

If a KeyList could not be generated a KeyListGenError exception is raised.

The algorithm is heuristic-based and makes random decisions based upon the 1944 procedure. The actual procedure is loosely specified in the manual, and much is left up to the human operator. It is possible that the algorithm cannot find a solution to meet the key list requirements specified in the manual, in which case it simply tries again up to some set of limits. These limits can be tweaked using the optional parameters to the algorithm. If no solution is found after exhausting the limits, a KeyListGenError is raised.

The optional parameters are:

- lug_selection - a list of 6 integers used to drive the lug settings portion of the algorithm. If not supplied, a list of 6 integers is chosen from data tables that appear in the 1944 manual. For more information on the requirements for these integers, see the manual.
- max_lug_attempts - the maximum number of times to attempt to create lug settings before giving up
- max_pin_attempts - the maximum number of times to attempt to generate key wheel pin settings before giving up

2.3.3 M209 Class

Naturally, the m209 library includes a class that simulates a M-209 converter. The M209 class allows you to experiment with all moving parts of an M-209, including encrypting and decrypting messages. Keep in mind there is a higher level class, StdProcedure, that encapsulates all the steps of the standard encrypting and decrypting operations, including generating indicators and placing or removing them from messages. However if you need lower-level access or you are inventing your own procedures, you would use the M209 class directly.

class m209.converter.**M209**([*lugs=None*[, *pin_list=None*]])

The M209 class takes the following optional arguments.

Parameters

- **lugs** – either a lug settings list or string as per set_drum_lugs()
- **pin_list** – a list of six strings each formatted as per *Key wheel pin settings*

M209 objects have the following attributes.

letter_count
This attribute represents the *letter counter* functionality. It is an integer that is incremented after every letter is encrypted or decrypted. It may be set to any integer value or examined at any time.

M209 objects support the following methods.

set_pins(*n*, *effective_pins*)
Sets the pin settings on the specified key wheel n.

Parameters

- **n** – an integer between 0-5, inclusive. Key wheel 0 is the left-most wheel and wheel 5 is the right-most.
- **effective_pins** – an iterable of letters whose pins are slid to the "effective" position (to the right). See *Key wheel pin settings*.

set_all_pins(*pin_list*)
Sets all key wheel pins according to the supplied pin list.

Parameters **pin_list** – must either be None or a 6-element list of strings where each string element is as described in *Key wheel pin settings*. If None, all pins in all key wheels are moved to the ineffective position.

set_drum_lugs(*lug_list*)
Sets the drum lugs according to the given `lug_list` parameter.

If `lug_list` is `None` or empty, all lugs will be placed in neutral positions.

Otherwise, the `lug_list` can either be a list or a string.

If `lug_list` is passed a list, it must be a list of 1 or 2-tuple integers, where each integer is between 0-5, inclusive, and represents a 0-based key wheel position. The list can not be longer than 27 items. Only lug bars with lugs in non-neutral positions need be listed. Lug bars with one lug in a non-neutral position are represented by a 1-tuple. Bars with 2 non-netural lugs are represented as a 2-tuple.

If `lug_list` is passed as a string, it is assumed to be in key list format as described in *Lug settings format*.

Example:

```
m = M209()
m.set_drum_lugs('1-0 2-0*2 0-3 0-5*3 0-6 2-4 3-6')

# or equivalently
m.set_drum_lugs([(0, ), (1, ), (1, ), (2, ), (4, ), (4, ), (4, ), (5, ), (1, 3), (2, 5)])
```

set_key_wheel(*n*, *c*)
Set key wheel `n` to the letter `c`.

Parameters

- **n** – an integer between 0-5 where key wheel 0 is the leftmost key wheel, and 5 is the rightmost
- **c** – a 1-letter string valid for key wheel `n`

Raises KeyWheelError if `c` is not valid for wheel `n`

set_key_wheels(*s*)
Set the key wheels from left to right to the six letter string `s`.

Raises KeyWheelError if any letter in `s` is not valid for the corresponding key wheel

set_random_key_wheels()
Sets the six key wheels to random letters.

Returns a string of length six representing the new key wheel settings

get_settings()
Returns the current key settings.

Returns a named tuple of `(lugs, pin_list)` representing the current key settings. `lugs` will be in string format.

encrypt(*plaintext*[, *group=True*[, *spaces=True*]])
Performs an encrypt operation on the given plaintext and returns the encrypted ciphertext as a string.

Parameters

- **plaintext** – the text string to encrypt
- **group** – if `True`, the ciphertext string will be grouped into 5-letter groups, separated by spaces
- **spaces** – if `True`, space characters in the input plaintext will automatically be treated as `Z` characters. Otherwise spaces in the plaintext will raise an `M209Error`.

Returns the ciphertext as a string

decrypt(*ciphertext*[, *spaces=True*[, *z_sub=True*]])

Performs a decrypt operation on the given ciphertext and returns the decrypted plaintext as a string.

Parameters

- **ciphertext** – the text string to decyrpt
- **spaces** – if `True`, spaces will be allowed in the input ciphertext and ignored. Otherwise space characters will raise an `M209Error`. This is useful if the input ciphertext is in 5-letter groups, separated by spaces.
- **z_sub** – if `True`, `Z` characters in the output plaintext will be replaced by space characters, just like an actual M-209.

Returns the plaintext as a string

Example:

```
>>> from m209.converter import M209
>>> m = M209()
>>> m.set_drum_lugs('1-0 2-0*2 0-3 0-5*3 0-6 2-4 3-6')
>>> pin_list = [
...     'FGIKOPRSUVWYZ',
...     'DFGKLMOTUY',
...     'ADEFGIORTUVX',
...     'ACFGHILMRSU',
...     'BCDEFJKLPS',
...     'EFGHIJLMNP'
... ]
>>> m.set_all_pins(pin_list)
>>> m.set_key_wheels('FFEGJP')
>>> ct = m.encrypt('THE PIZZA HAS ARRIVED')
>>> ct
'QBCHU WCCDI YFNCH LOZJY G'
>>> m.set_key_wheels('FFEGJP')
>>> pt = m.decrypt(ct)
>>> pt
'THE PI  A HAS ARRIVED'
```

2.3.4 StdProcedure Class

The `StdProcedure` class encapsulates the encrypting and decrypting procedures outlined in *References*. In particular, see references [5] and [7]. This class takes care of the high level details of inserting various message indicators into an encrypted message, and stripping them off during decrypt. These message indicators tell the recipient what key list and initial key wheel settings to use when configuring their M-209 for decrypt.

class `m209.procedure.`**StdProcedure**([*m_209=None*[, *key_list=None*]])

Parameters

- **m_209** – an instance of a `M209` can optionally be provided to the procedure object. If `None` the procedure object will create one for internal use.
- **key_list** – an instance of a `KeyList` can be provided if known ahead of time. Before an encrypt or decrypt operation can be performed, a key list must be provided. This can be done after object creation via `set_key_list()`. Note that the `letter_check` attribute of the `KeyList` is not accessed by the procedure object, and can be `None` if not known.

Before an `encrypt()` operation can be performed, a valid key list must be installed, either during procedure object construction, or by the `set_key_list()` method.

Decrypt operations are performed in a 3-step process.

1. First, a call to the `set_decrypt_message()` method passes the message to be decrypted to the procedure and establishes the parameters to be used for the actual `decrypt()` operation. These decrypt parameters are returned to the caller.
2. The caller can examine the decrypt parameters to determine which key list must be installed before a successful `decrypt()` operation can be carried out. The caller may call `get_key_list()` to examine the currently installed key list. It is up to the caller to obtain the required key list and install it with `set_key_list()`, if necessary. This is done by ensuring the installed key list indicator matches the `key_list_ind` field of the decrypt parameters.
3. Finally `decrypt()` can be called. If the procedure does not have the key list necessary to decrypt the message, a `ProcedureError` is raised.

`StdProcedure` objects have the following methods:

get_key_list()

Returns the currently installed `KeyList` object or `None` if one has not been set

set_key_list(*key_list*)

Establishes the `KeyList` to be used for future `encrypt()` and `decrypt()` operations

Parameters **key_list** – the new `KeyList` to use

encrypt(*plaintext*[, *spaces=True*[, *ext_msg_ind=None*[, *sys_ind=None*]]])

Encrypts a plaintext message using the installed `KeyList` and by following the standard procedure. The encrypted text with the required message indicators are returned as a string.

Parameters

- **plaintext** – the input string to be encrypted
- **spaces** – if `True`, space characters in the input plaintext are allowed and will be replaced with `Z` characters before encrypting
- **ext_msg_ind** – this is the external message indicator, which, if supplied, must be a valid 6-letter string of key wheel settings. If not supplied, one will be generated randomly.
- **sys_ind** – this is the system indicator, which must be a string of length 1 in the range `A` - `Z`, inclusive. If `None`, one is chosen at random.

Returns the encrypted text with the required message indicators

Raises ProcedureError if the procedure does not have a `KeyList` or the input indicators are invalid

set_decrypt_message(*msg*)

Prepare to decrypt the supplied message.

Parameters **msg** – the messsage to decrypt. The message can be grouped into 5-letter groups separated by spaces or accepted without spaces.

Returns a `DecryptParams` named tuple to the caller (see below)

The `DecryptParams` named tuple has the following attributes:

- `sys_ind` - the system indicator
- `ext_msg_ind` - the external message indicator

- `key_list_ind` - the key list indicator
- `ciphertext` - the cipher text with all indicators removed

The caller should ensure the procedure instance has the required `KeyList` before calling `decrypt()`. The `key_list_ind` attribute of the returned `DecryptParams` named tuple identifies the key list that should be installed with `set_key_list()`.

decrypt()

Decrypt the message set in a previous `set_decrypt_message()` call. The resulting plaintext is returned as a string.

Returns the decrypted plaintext as a string

Raises ProcedureError if the procedure instance has not been previously configured with the required `KeyList` via `set_key_list()`

Here is a simple interactive example of performing an encrypt operation. Here we choose a random key list from our key list file, and use random indicators:

```
>>> from m209.keylist.config import read_key_list
>>> from m209.procedure import StdProcedure
>>>
>>> key_list = read_key_list('m209keys.cfg')
>>> proc = StdProcedure(key_list=key_list)
>>> ct = proc.encrypt('ORDER THE PIZZA AT TWELVE HUNDRED HOURS')
>>> ct
'YYGBM ENNHT VBMTJ PEEFV JWLUU PAFTS VOHEA QEPEQ OKVUA XDAUX YYGBM ENNHT'
>>>
```

The first and last two groups of this message contain the indicators. Here we can see the system indicator was `Y`, the external message indicator is `GBMENN`, and the key list indicator is `HT`.

An example session for decrypting the above message might look like:

```
>>> proc = StdProcedure()
>>> ct = 'YYGBM ENNHT VBMTJ PEEFV JWLUU PAFTS VOHEA QEPEQ OKVUA XDAUX YYGBM ENNHT'
>>> params = proc.set_decrypt_message(ct)
>>> params
DecryptParams(sys_ind='Y', ext_msg_ind='GBMENN', key_list_ind='HT', ciphertext='VBMTJ PEEFV JWLUU PAF
>>> key_list = read_key_list('m209keys.cfg', params.key_list_ind)
>>> proc.set_key_list(key_list)
>>> pt = proc.decrypt()
>>> pt
'ORDER THE PI  A AT TWELVE HUNDRED HOURS '
>>>
```

CHAPTER

THREE

REQUIREMENTS

m2 0 9 is written in Python 3, specifically Python 3.3. At this time it will not run on Python 2.x.

m2 0 9 has no other requirements or dependencies.

CHAPTER

FOUR

INSTALLATION

m209 is available on the Python Package Index (PyPI).

You can install it using pip:

```
$ pip install m209                  # install
$ pip install --upgrade m209        # upgrade
```

You may also download an archive file of the latest code by visiting the m209 Bitbucket page. Alternatively if you use Mercurial, you can clone the repository with the following command:

```
$ hg clone https://bitbucket.org/bgneal/m209
```

If you did not use pip (you downloaded or cloned the code yourself), you can install with:

```
$ cd where-you-extracted-m209
$ python setup.py install
```

To run the unit tests:

```
$ cd where-you-extracted-m209
$ python -m unittest discover -b
```

CHAPTER

FIVE

SUPPORT & SOURCE

All support takes place at the m209 Bitbucket page. Please enter any feature requests or bugs into the issue tracker.

CHAPTER

SIX

REFERENCES

All of the resources listed below were useful to me in the creation of the `m209` library. In particular, I want to thank Mark J. Blair for his detailed explanations of the M-209's operation and procedures. The official training film was also highly instructive.

1. M-209 at Wikipedia
2. Mark J. Blair's Converter M-209-B
3. 1942 M-209 Manual
4. 1944 M-209 Manual
5. Official M-209 Training Film - This is a 4 video YouTube playlist of an actual 1940's era US War Department training film. Demonstrates the M-209 and operational procedures. Very interesting!
6. Transcript of Training Film - Transcript of the above film.
7. Mark J. Blair's M-209 Cipher Machine Group - Informal club for M-209 enthusiasts. Includes detailed explanations of the device and how to use it. Very useful.
8. Dirk Rijmenants' M-209 Simulator - Graphical M-209 simulator
9. Mark J. Blair's Hagelin project suite at GitHub - M-209 simulator written in C++
10. Jean-François Bouchaudy's Crypto Pages - Includes another Python-based M-209 simulator and a M-209 challenge. In French.
11. The C-38 / M-209 Cipher Machine - Another M-209 page. This one has useful info on creating key lists and a C-38 simulator written in C.

CHAPTER

SEVEN

INDICES AND TABLES

- *genindex*
- *search*