C library to convert between mapcodes and latitude/longitude

Version 2.2

1. Converting a coordinate into a mapcode

```
int encodeLatLonToSingleMapcode(    // find SHORTEST mapcode
        char*  result,
        double lat,
        double lon,
        int    territoryCode,
        int    extraDigits)
```

input:

- **lat:** latitude in degrees, capped to -90.0 to 90.0 by method
- **lon:** longitude in degrees, wrapped to -180.0 to 180.0 by method
- **territoryCode:** territory (1-999) to encode in
- **extraDigits:** the number of high-precision extra "digits" to add to the generated mapcode. The preferred default is 0). See section 2.2 below.

output:

- **return value**: number of results (0 or negative if no mapcode was found)
- **result:** a string representing the full mapcode found, including territory code; the caller must allocate the memory for the result string (MAX_MAPCODE_RESULT_LEN).

Example code:

```
double lat = 52.376514;
double lon =  4.908542;
const char* countryabbr = "NLD";

char result[MAX_MAPCODE_RESULT_LEN];
int tc = getTerritoryCode(countryabbr, 0);
int n = encodeLatLonToSingleMapcode(result, lat, lon, tc, 0);
if (n > 0)
  printf("%0.6f,%0.6f has mapcode %s\n", lat, lon, result);
else
  printf("No results\n");
```

Output:

```
52.376514,4.908542 has mapcode NLD 49.4V
```

Thus, **convertTerritoryNameToCode** is used to determine the territory code **tc** for "NLD". Then **encodeLatLonToSingleMapcode** is used to determine the mapcode for the coordinate 52.376514,4.908542.

If you are not sure of the territory, and/or if you are interested in alternative (longer) mapcodes for the same coordinate in the *same* territory, you can also generate *all* possible mapcodes for a coordinate:

```
int encodeLatLonToMapcodes(    // find ALL mapcodes
        Mapcodes*    mapcodes,
        double       lat,
        double       lon,
        int          territoryCode,
        int          extraDigits)
```

where

```
typedef struct {
        int count;
        char mapcodes[count][MAX_MAPCODE_RESULT_LEN];
} Mapcodes;
```

input:

- **mapcodes:** a structure capable of storing the results. The x-th result will be in mapcodes.mapcode[x] and will have a territory name followed by a space followed by a mapcode, or just the mapcode if it is international.
- **lat:** latitude in degrees, capped to -90.0 to 90.0 by method
- **lon:** longitude in degrees, wrapped to -180.0 to 180.0 by method
- **territoryCode:** territory (1-999), or pass 0 to encode in ALL possible territories)
- **extraDigits:** the number of high-precision extra "digits" to add to the generated mapcode. The preferred default is 0. See section 2.2 below.

output:

- **return value**: the number of results

Note: *For legacy's sake, the old, non-threadsafe routine* ***encodeLatLonToMapcodes*** *which returns* ***char****'s is also still available (method name suffixed* ***_Deprecated****), but it uses global static storage (overwritten at each call) and returns pairs of strings. Use the new method instead.*

Example code:

```
int i;
double lat = 51;
double lon =  5;
int precision = 0; // normal precision
Mapcodes m; // storage
encodeLatLonToMapcodes(&m, lat, lon, 0, precision);

printf("%d mapcodes for %0.6f,%0.6f:\n", m.count, lat, lon);
for (i = 0; i < m.count; i++)
  printf("%s\n", m.mapcode[i] );
```

Output:

```
4 mapcodes for 51.000000,5.000000:
BEL C17.DF3
BEL P6CG.MRQ
FRA P6CG.MRQ
VHXP4.M457
```

Since 0 was passed as territory, mapcodes are generated for *any territory for which the mapcode rectangle contains the coordinate*. This includes not only Belgium (where the coordinate belongs), but also France.
The mapcodes are grouped per territory. The territories themselves are listed in no particular order, except that the international mapcode is always the last code in the list.

Furthermore, there are *two* mapcodes generated in Belgium, both correct. The first mapcode generated in a particular territory is always the shortest mapcode in that territory, and is always the mapcode that would be generated by **encodeLatLonToMapcodes** for that territory.

Finally, note that passing 0 as territory is guaranteed to return at least one result (an international mapcode).

2. Converting a mapcode into a coordinate

```
int decodeMapcodeToLatLon(
        double*     lat,
        double*     lon,
        const char* mapcode,
        int         territoryCode)
```

input:

- **mapcode:** a "free-form" user input, which may be recognized if it is of the form
    - **[xxx[-yyy]] PPP.QQQ [-RR]**
        - xxx is a country code (2 or 3 characters)
        - yyy is an state code (2 or 3 characters)
        - PPP : mapcode prefix (between 2 and 5 characters)
        - QQQ: mapcode postfix (between 2 and 4 characters)
        - RR: high-precision addendum (one or two characters)
- **territoryCode:** territory (or 0), to help disambiguate the user input if that proves necessary; If you pass 0, the mapcode may fail to decode, or (if it is ambiguous) will be disambiguated randomly.
    - **Examples:**
        - user input "US-CA XX.XX" needs no disambiguation and returns a location in California
        - user input "IN XX.XX" can be decoded without a tc, but it is unpredictable whether it decodes to a location in Indiana, USA or in the Ingushetia, Russia.
        - user input "XX.XX" is impossible to decode unless a valid tc is provided

output:

- returns nonzero in case of error. Otherwise, **lat** and **lon** are filled with the decoded coordinates. Latitudes are always in the range [-90, 90] and longitudes in range [-180, 180>.

Example code:

```
const char* userinput = "NLD 49.4V";
double lat, lon;
int err = decodeMapcodeToLatLon(&lat, &lon, userinput, 0);
if (err)
  printf("\"%s\" is not a valid mapcode\n", userinput);
else
  printf("\"%s\" represents %0.6f,%0.6f\n", userinput, lat, lon);
```

Output:

```
"NLD 49.4V" represents 52.376514,4.908543
```

Note that the above piece of code will only "accidentally" handle *ambiguous* partial mapcodes correctly. For example, **userinput="IN VY.HV"** will *either* be interpreted

as an abbreviation of "US-IN VY.HV" *or* of "RU-IN VY.HV", and thus *either* produce a coordinate in Indiana, USA, *or* in Ingushetia, Russia.
Passing a "default context" improves the chances of ambiguous user input to be interpreted "as the user intended". Thus, if one builds a system that is mostly going to be used in the USA, the following hard-codes that preference (i.e. "when in doubt, assume the "USA"):

```
int defaultcontext = getTerritoryCode("USA",0);
const char* userinput = "IN VY.HV";
double lat, lon;
int err = decodeMapcodeToLatLon(&lat, &lon, userinput,
            defaultcontext);
if (err)
  printf("\"%s\" is not a valid mapcode\n", userinput);
else
  printf("\"%s\" represents %0.6f,%0.6f\n", userinput, lat, lon);
```

Output:

```
"IN VY.HV" represents 39.727950,-86.118444
```

A more sophisticated system could of course make much better assumptions, for example based on the GPS coordinate of the user, or the current cursor position on a world map that is being displayed to the user.

In an interactive system, the *best* way to handle ambiguity is probably to always use the most recent *successful, explicitly stated* context as default.
For example, suppose you remembered the previous *correctly* interpreted user input:

```
const char* previous_successful_input = "RU-IN DK.CN0";
```

then the following code snippet will correctly interpret the completely abbreviated mapcode "49.4V" as being in the same state (i.e. RU-IN):

```
int previouscontext = getTerritoryCode(
                        previous_successful_input,0);
const char* userinput = "D6.58";
int err = decodeMapcodeToLatLon(&lat, &lon, userinput,
            previouscontext);
```

Output:

```
"D6.58" represents 43.259275,44.771980
```

which is in Ingushetia Republic. And in fact, had we written

```
userinput="AL D6.58"
```

this would generate the output

```
"AL D6.58" represents 51.977856,85.935367
```

which is in the Russian republic of Altai: because of the context RU-IN, the territory has been interpreted as RU-AL, instead of the equally likely US-AL (Alabama, USA) or BR-AL (Alagoas, Brazil).

Here is an example that decodes consecutive user inputs, some of them wildly ambiguous. With the exception of the very first input, all are probably interpreted as the user intended:

```
  int context = -1; // no initial context

  const char* userinput[] = {
    "49.4V",
    "US-IN 49.4V",
    "49.4V",
    "AL 49.4V",
    "RU-IN 49.4V",
    "AL 49.4V",
    "NLD XXX.YYY",
    "49.4V",
    "CCCCC.CCCC",
    "49.4V",
    NULL
  };

  int i;
  for (i = 0; userinput[i] != 0; i++) {
    double lat, lon;
    int err = decodeMapcodeToLatLon(&lat, &lon, userinput[i],
            context);
    if (err)
      printf("\"%s\" is not a valid mapcode\n", userinput[i]);
    else {
      int c;
      printf("%12s represents %0.6f,%0.6f\n", userinput[i], lat,lon);
      c=getTerritoryCode(userinput[i], 0);
      if (c >= 1 && c < MAX_MAPCODE_TERRITORY_CODE)
        context = c;
    }
  }
```

Output:

```
"49.4V" is not a valid mapcode
 US-IN 49.4V represents 39.783750,-86.198832
       49.4V represents 39.783750,-86.198832
    AL 49.4V represents 33.532750,-86.836184
 RU-IN 49.4V represents 43.249285,44.741354
    AL 49.4V represents 51.967866,85.899261
 NLD XXX.YYY represents 51.204537,5.541607
       49.4V represents 52.376514,4.908543
  CCCCC.CCCC represents -16.326209,-48.016850
       49.4V represents 52.376514,4.908543
```

Explanation:

```
"49.4V" is not a valid mapcode
```
Since there is no previous context, this ambiguous mapcode can simply not be interpreted. For this reason, it may be a good idea to choose a better default context than previouscontext=-1 (e.g. based on the user's GPS position).

```
US-IN 49.4V represents 39.783750,-86.198832
```
A complete and unambiguous mapcode, it results in a coordinate in Indiana, USA.

```
49.4V represents 39.783750,-86.198832
```
Since the previous input was in Indiana, this incomplete mapcode is encoded in the same context.

```
AL 49.4V represents 33.532750,-86.836184
```
Since the previous input was in the Indiana, USA, the context "AL" is interpreted as another state in the USA, and thus results in a coordinate in Alabama (rather than, say, the state of Alagoas in Brazil).

```
RU-IN 49.4V represents 43.249285,44.741353
```
A complete and unambiguous mapcode, it results in a coordinate in Ingushetia, Russia.

```
AL 49.4V represents 51.967866,85.899261
```
Unlike two inputs back, AL 49.4V is now interpreted in Russia (the Altai Republic) rather than in the USA (Alabama), since the most recent context was Russian.

```
NLD XXX.YYY represents 51.204536,5.541607
```
A complete and unambiguous mapcode, it results in a coordinate in The Netherlands.

```
49.4V represents 52.376514,4.908542
```
Since the previous input was in The Netherlands, this time 49.4V is interpreted in The Netherlands as well.

```
CCCCC.CCCC represents -16.326209,-48.016851
```
A complete and unambiguous *international* mapcode. Although it decodes to a coordinate somewhere in Brazil, the mapcode does not **explicitly** specify Brazil as a context. Therefore, the context for future inputs will remain "The Netherlands".

```
49.4V represents 52.376514,4.908542
```
Since the most recent **explicit** context was in The Netherlands, this ambiguous mapcode is now also interpreted in The Netherlands.

2.1 Recognizing an input as a mapcode

Sometimes you may wish to allow a user to input something in a "general" search box – an address, a coordinate, a mapcode, or something else.
The following routine is efficient and lightweight, and recognizes if a user input looks like it is *intended* as a mapcode. For example:

"NLD 503.XX2"

is intended as a mapcode, while

"St. Jacobs Street 45, London"

is not. Since anything that looks like a mapcode is *very unlikely* to represent anything else, we would recommend to handle (i.e. decode) anything that looks like a mapcode as a mapcode. If it fails to decode, you could still try to interpret as something else, but as has been said: it is very unlikely it *does* represent something else.

Note: the routine can not guarantee that the input represents a *valid* mapcode. For example, the input "XXX XX.XX" will pass although XXX is not a valid territory.

```
int compareWithMapcodeFormat(
        const char*    string ,
        int            includesTerritory)
```

input:

- **string:** a "free-form" user input
- **includesTerritory:** an integer: if you pass 1, full mapcodes (including *optional* territory context) will be recognized, i.e. inputs of the form
  [whitespace] [xxx[-yyy]] [whitespace] PPP.QQQ [-RR] [whitespace]
  If you pass 0, only "proper" mapcodes will be recognized:
  [whitespace] PPP.QQQ [-RR] [whitespace]

output:

- returns 0 if the string looks like a full/proper mapcode
- return negative in case of error (the special value -999 is returned if the string looks like a *partial* mapcode, i.e. might become a valid mapcode if some more characters were added).

2.2 Extra precision

Mapcodes are intended for easy, daily use. They were therefore made short, and no more precise than is necessary to be useful at the "human" scale: accurate to within a few meters – or put another way: *in*accurate by *up to* several meters.

For special applications, mapcodes can be generated with higher accuracy, by appending extra letters. One letter extra decreases the worst-case inaccuracy to less than 162 centimeters (70 cm on average), two letters decreases it to less than 25 centimeters (13 cm on average), four letters to less than a centimeter.

As an example, consider coordinate 52.3765, 4.90858. When encoded, it yields mapcode **NLD 49.4V**. This mapcode *decodes* back into 52.376514, 4.908543, a coordinate which is 2.48 meters off to the west, and 1.56 meters too far north (in combination, the mapcode is 2.93 meters away from the original coordinate). When we encode the same coordinate with an extra digit, we get **NLD 49.4V-L**, a mapcode that *decodes* into 52.376508, 4.908575, only 95 centimeters off. With *two* extra digits, we get a mapcode that is about 5.7 centimeters off.

Mapcode:	decodes into:	error vs original coordinate:
49.4V	52.376514, 4.908543	2.93 meter
49.4V-L	52.376491, 4.908574	0.95 meter
49.4V-LX	52.376497, 4.908584	0.06 meter

Note that this is just an example. Had we encoded 52.376514, 4.908543, the mapcode **49.4V** would already be precise to 2.5 centimeters. It is the *maximum* error that is reduced by adding extra letters to a mapcode.

PLEASE NOTE: the above may make it seem that it is a good idea to *always* add extra letters. This would defeat the core purpose of the mapcode system, which is to be accurate *enough* for daily, human-scale use. The high-precision extension was made for very special applications only.

See section 4 about distance-related routines.

3. Routines related to territories

The mapcode system is based on an official code table, which in turn is based on the ISO 3166 standards.
For efficiency, these codes need to be converted into internal "territory codes". For these, the following three support routines are relevant:

```
int getTerritoryCode(
        const char*     alphaCode,
        int             parentTerritoryCode)
```

- **alphaCode:** a string starting with the "ISO standard" code of a country or a state, such as "USA", "CA", "US-CA", or "USA-CA"
- **parentTerritoryCode:** optional territory code (1-999) to help the routine choose when the abbreviation is ambiguous, which can happen if the abbreviation is of a state and the state's country is omitted (For example, "AL" might mean either "US-AL" or "BR-AL"). Pass 0 if not available.
- **Returns**: the territory code (1-999), or negative if not no match was found.

```
char *getTerritoryAlphaCode(
        char *result,
        int     territoryCode,
        int     format)
```

- **Result** – a string to store the result in (capable of storing at least MAX_ISOCODE_LEN characters *plus* a zero-terminator); Returns an empty string if territoryCode is illegal.
- **territoryCode** – a territory code (1-999).
- **format** – specifies the format of the return value:
  - 0 – return in unambiguous full format: "XXX" for a country, "XX-YY" for a state
  - 1 – short format: "XXX" or "XX"; especially when a 2 letter (state) code is returned, although always unique within its country, it may not be unique in the world
- **Returns:** a pointer to **result**.

Note: for legacy's sake, the following routine, which is not threadsafe, is also still available:

const char* convertTerritoryCodeToIsoName(territoryCode,format)

It is similar to ***getTerritoryAlphaCode*** *except that it uses global storage which is overwritten by the next call.*

int **getParentCountryOf(**int **territoryCode)**

- **territoryCode** – territory code (1-999)
- returns:
  - the parent country of territoryCode
  - -1 if territoryCode is invalid or not a subdivision

4. Routines related to distance

double **distanceInMeters(**double **lat1,** double **lon1,** double **lat2,** double **lon2)**

- **lat1, lon1**: a latitude/longitude (in degrees)
- **lat2, lon2**: another latitude/longitude (in degrees)
- returns: distance between the coordinates, in meters

NOTE: only correct for coordinates that are within a few miles of each other

double **maxErrorInMeters(**int **extraDigits)**

- **extraDigits:** the number of high-precision "digits" in a mapcode (see section 2.2).
- returns: worst-case distance in meters between the original coordinate and the decode location of the mapcode.

5. Routines related to Unicode and/or foreign alphabets

char *convertToRoman(
char *asciibuf,
int maxlength,
const UWORD* string)

Mapcodes may be specified in other alphabets. This routine converts a 16-bit **string** (i.e. a zero-terminated string of 16-bit Unicode characters) into a roman (8-bit) equivalent – which can then be offered to an **encode** routine**.**
The result is stored in **asciibuf** (which must be at least **maxlen** in size) as a zero-terminated 8-bit string. A pointer to the result is returned.

Note: for legacy's sake, we also support the following routine, which returns a pointer to a static buffer that is rewritten on each call:
const char* decodeToRoman(const UWORD* string);

UWORD* convertToAlphabet(
UWORD* unibuf,
int maxlength,
const char *string,
int alphabet)

This routine converts a zero-terminated (ascii) **string** into a zero-terminated Unicode string in the specified **alphabet**. The result is stored in **unibuf** (which must be at least **maxlength** in size). A pointer to the result is returned.

For example, this routine can convert

PQ.RS to **नप.भम** (Hindi alphabet)
PQ.RS to **РФ.ЯЦ** (Russian alphabet)

At the date of writing, the following 4 alphabets have been approved for official use, the others have not yet officially been approved.
0 = roman,
2 = Cyrillic,
4 = Hindi,
12 = Gurumukhi

Note: for legacy's sake, we also support the following routine, which returns a pointer to a static buffer that is rewritten on each call:
const UWORD* encodeToAlphabet(const char* mapcode, int alphabet);

6. Data changes in version 2

Since version 2.0.0, coordinates are not rounded to the nearest 1,000,000th of a degree (a millionth of a degree roughly equals 11 centimeters). This will seldom affect daily life, but decoding an old mapcode may yield differences in the 6th coordinate decimal (i.e. on the 11-centimeter scale), and in edge cases, encoding a coordinate may yield a different mapcode than before (one that is a few centimeters *closer* to the original coordinate than the mapcode produced with the old code).

As part of the application process for the International Standards Organisation, a thorough check was done on all 16,000 territory/population-density rectangles defined in 2001. Some new code ranges were *added*, mostly for remote islands, which means the new rectangles can produce mapcodes that are not recognized on older systems. However, a few adjustments had to be made which **break compatibility:** a mapcode generated by the new system would decode to a *different coordinate* on an old system, and vice versa. This is true for the changes listed below under "out-sized cells".

1) out-sized cells – The mapcode database divides the territories of the world into cells of at about 10 x 10 meters. The worst-case error in such cell is at the corners (about 7.1 meters from the center). Our check yielded about a dozen cases (out of 16,300) where the cells exceded 10.5 x 10.5 meters and the error could exceed 7.5 meters. These records were adjusted.

This adjustments **breaks compatibility** for the following "mainland" mapcodes:

a. Codes of the form xx.xx for the small **town of Altaysk**, Altai Republic, Russia (RU-AL)
b. Codes of the form xx.xx for the micro-state **San Marino**
c. Codes of the form xxxx.xxx in the **Xinjiang Uyghur province**, China
d. 7-letter codes for **Inner Mongolia**
e. 6-letter codes for **state of Chihuahua**, Mexico
f. 6-letter codes for **Bangladesh** and **Romania**
g. Codes of the form xxx.xxxx codes in the far north of **Sakha Republic, Russia**
h. Codes of the form xx.xx for the **town of Fargo,** in North Dakota, USA
i. Codes of the form xx.xxx for the town of **Toronto** in Canada
j. The *optional* 7-letter code range 6xx.xxxx for **Andaman and Nicobar**, India (a state fully covered by 6-character mapcodes)
k. 7-letter codes for **Sudan** and **South Sudan**, which are now contiguous (both countries were furthermore given optional 8-character codes)

and also on the following island territories:

l. Changed the 5-letter codes for **Reunion Island (REU)** and added optional codes of the form xxx.xxx
m. Replaced optional codes of the form xxxx.xxx for **Saint Helena, Ascension and Tristan da Cunha** (SHN) by optional codes of the form xxxx.xxxx (note: all land area is covered by shorter codes); Added code range K0.000-PZ.ZZZ to cover **Cough Island**
n. Changed codes of the form **xxx.xxx** for the **Maldives (MDV)**; added *optional* 7-letter codes

o. Changed codes of the form **xx.xxx** codes for **Saint Vincent and the Grenadines (VCT);** added *optional* 6-letter codes
p. Changed the mapcodes for the **islands of Kiribati (KIR)**
q. Changed codes of the form **xxxx.xx** for the **South China Sea islands** of the **Hainan province (CN-HI)**
r. Changed codes of the form **xx.xxx** for the **o'Ahu island** of Hawaii (US-HI)
s. Changed codes of the form **xx.xxx** for the **British Virgin Islands (VGB)**
t. Improved 4-letter codes for **Wallis and Futuna** (WLF) to cover almost all of Wallis
u. Adjusted 5-letter codes for **Turks and Caicos Islands** (TCA) to include all land area
v. Adjusted 5-letter codes for **Comoros** (COM) to include all land area
w. Adjusted 6-letter codes for **Solomon Islands** (SLB) to include all land area

2) Missing islands, atolls and rocks (extra code ranges only)
Code ranges were added to cover islands, atolls and rocks that were missing from the borders of certain island nations. All codes from old mapcode systems are correctly recognized, but the *new* code ranges are not recognized by by old mapcode systems.

a. ASM (American Samoa) – Code range H0.000-L6.ZPC added for **Rose Atoll**; Optional codes of the form xxxx.xxx added to cover all land area
b. VIR (US Virgin Islands) - Codes of the form xxx.xx added to include **Savana and French Cap Cay**
c. FSM (Deferated States of Micronesia) – codes of the form xxx.xxxx added to include the **Nukuoro and Tokodakaaka Atolls**
d. MUS (Mauritius) – code range X00.000-XZZ.ZZZ added to cover some **atolls north of Cargados Carajos**
e. SGS (South Georgia and the South Sandwich Islands) – code range P000.00-RZZZ.ZZ added to cover **Black Rock.**
f. TWN (Taiwan) – added code range Y00.000-YZZ.ZZZ to cover **Agincourt, Pinnacle, and Craig islands**
g. EST (Estonia) – added code range X00.000-XZZ.ZZZ to cover **Vaindloo island**
h. GUF (French Guiana) – added code range B000.00-CZZZ.ZZ to include **Isle du Grand Connetable and Ile du Diable**
i. PRT (Portugal) – added code ranges S000.00-SZZZ.ZZ and N000.000-NZZZ.ZZZ to cover some **islands far south of Madeira**
j. KOR (South koreea) – added code range Z000.00-ZZZZ.ZZ to include the **Dongdo-ri islands**
k. NZL (New Zealand) – added *optional* code range L000.001-MZZZ.ZZZ to provide 7-letter optional equivalents for all 6 letter mapcodes
l. JPN (Japan) - added range V000.001-WZZZ.ZZZ to cover the **Liancourt Rocks, Oshima Island and Aramiko Island**
m. ALA (Aaland Islands) – optional borders extended to cover **Lökharu island**

n. MDG (Madagascar) – added code range S000.01-SZZZ.ZZ to cover the **western sand banks**; Optional 8-character codes added to cover **coastal waters**

o. ZAF (South Africa) – added code range M00.000Y-MZZ.ZZZZ to cover **Marion Island and Prince Edward island**

p. Mexico – added code range 800.00A0 – 8ZZ.ZZZZ to include the **Arrecife Alacranes islands**

3**) Other improvements (extra code ranges only):**

a. HUN (Hunagry) – added code range 70.00A0 - DZ.TCZK so that the whole south of the country has mapcodes of the same form (xx.xxxx); the mapcodes of the old forms are of course still available.

b. BGR (Bulgaria) – added code range L0.0000-MZ.ZZZZ so that the whole south of the country has mapcodes of the same form (xx.xxxx); the mapcodes of the old forms are of course still available.

c. BEN (Benin) - added code range V0.0000-ZZ.ZZZZ so that the whole north of the country has mapcodes of the same form (xx.xxxx); the mapcodes of the old forms are of course still available.

d. SUR (Suriname) - added code range Y0.0003-ZZ.ZZZY so that the whole south of the country has mapcodes of the same form (xx.xxxx); the mapcodes of the old forms are of course still available.

e. US-IL (Illinois, USA) - added code range X0.0002-ZZ.ZZZZ so that the whole south of the state has mapcodes of the same form (xx.xxxx); the mapcodes of the old forms are of course still available.

f. US-NY (New York State, USA) - added code range Z00.000-ZZY.ZZY so that every location in the state has a 6-letter code; the mapcodes of the old forms are of course still available.

g. SYR (Syria) – the country rectangle was incorrectly marked "optional" – for the south-east desert, 7-letter mapcodes are not optional.

h. PHL (Phillippines) – added code range of the forms xxx.xxxx and xxxx.xxxx to include Saluag Island and Francis Reef

i. DNK (Denmark) – added code range Z0.0000-ZZ.ZZZZ to include Falster Islands's southernmost point

j. PER (Peru) – added code range of the forms xxx.xxxx and xxxx.xxxx to include the easternmost point

k. BR-AC (Acre, Brazil) - added codes of the form xxxx.xxx as alternative for the 8-letter codes for the northern jungles

l. For Mexico, India, Australia, Brazil, the USA and Russia, national mapcodes are also available as regional mapcodes (i.e. within the states and subdivisions). For the following subdivisions, the rectangles were slightly enlarged to assure all locations within the subdivision are enclosed: MX-DIF, MX-GRO, MX-VER, IN-PB, IN-HR, IN-TN, IN-PY, AU-NSW, AU-NT, AU-SA, AU-VIC, AU-QLD, BR-SP, BR-RS, US-NV, RU-AD, RU-AST, RU-VLA, RU-KRS, RU-TA, RU-TT, RU-RYA, RU-SAM, RU-PSK, RU-KDA and RU-PO

m. Added 6-letter codes for Gibraltar (GIB) that overlap with Spain mapcodes; added 6-letter and 7-letter codes for San Marino (SMR) that overlap with Italy mapcodes; added 7-letter codes for Isle of Man, Jersey and Guernsey that overlap with GBR mapcodes

6.2. Data changes in version 2.2

It was discovered that there were a few micro-degree gaps between the rectangles that define a territory. For example, in Sierra Leone, one subarea ended at latitude 8.526879 and the next started at 8.526880. Since coordinates are not rounded to 6 decimals any more, locations that fell inside this 11-centimeter-wide gap, such as (8.5268795, -12), had no Sierra Leone mapcode! This required a fix to the data.

Effects: for mapcodes of the forms affected (see table), there will be up to 11 centimeter difference between the way an old system decodes such a mapcode, and the coordinate generated by a new mapcode system.

Territory	**Affects mapcodes of the form:**
Antarctica	xxxx.xxxx
Austria	**B**xx.xxx , **C**xx.xxx
Brazil	**PR** xxxx.xx
Bulgaria	**J**xx.xxx
Congo-Kinshasa	xxxx.xxx
Croatia	xxx.xxx
Czech Republic	**8**xx.xxx
Dominican Republic	**Z**xx.xxx
French Guiana	**9**xx.xxx , **D**xx.xxx
Ghana	xxx.xxx
India	**3**xxx.xxx , **4**xxx.xxx, **D**xx.xxxx, **BR** xxx.xxx, **TN** 9xx.xxx, **JH** xxx.xxx , **JH** xx.xxxx , **GJ** Zxxx.xx , **UP** 7xxx.xx
Iran	xxxx.xxx
Liberia	**C**xx.xxx , **G**xx.xxx
Malawi	**1**xx.xxx
Mexico	xxxx.xxx
Moldova	Mxx.xxx
Pakistan	**3**xxx.xxx , **4**xxx.xxx
Panama	**3**xx.xxx
Saudi Arabia	**2**xxx.xxx , **N**xxx.xxx
Sierra Leone	xxx.xxx , **3**x.xxxx
Tajikistan	xx.xxxx
USA	**WV** xxx.xxx , **AR** xx.xxxx , **NC** xxx.xxx , **NY** xxx.xxx , **NY** xxxx.xx , **FL** xxx.xxx , **AK 2**xxx.xxx , **AK 3**xxx.xxx

In a few cases, larger gaps were discovered. Rather than breaking compatibility with old mapcodes beyond 11 centimeters, new sub-territories were added.

Effects: old systems will not recognize mapcodes of the forms listed below:

Territory	**New mapcodes of the form:**
Croatia	**Z**x.xxxx
Japan	**Z**xxx.xxx
Congo-Kinshasa	**8**xx.xxxx
India	**AS Z**xx.xxx, **AS T**xx.xxx, **BR Z**xx.xxx , **8**xx.xxxx
USA	**TX X**xxx.xxx, **TX Z**xxx.xxx
Mexico	**9**xx.xxxx
Xinjiang Uyghur, China	**W**xxx.xxx

C library version history

1.25 initial release to the public domain.
1.26 added alias OD ("Odisha") for Indian state OR ("Orissa").
1.27 improved (faster) implementation of the function isInArea.
1.28 bug fix for the needless generation of 7-letter alternatives
to short mapcodes in large states in India.
1.29 also generate country-wide alternative mapcodes for states.
1.30 updated the documentation and extended it with examples and suggestions.
1.31 added lookslikemapcode().
1.32 added coord2mc1();fixed 1.29 so no country-wide alternative is produced
in edge cases; prevent FIJI failing to decode at exactly 180 degrees.
1.33 fix to not remove valid results just across the edge of a territory
improved interface readability and renamed methods to more readable forms.
1.4 renamed API to more appropriate convention (.h support legacy calls)
1.40 added extraDigits parameter so that high-precision mapcodes can be generated.
1.41 added the India state Telangana (IN-TG), until 2014 a region in Adhra Pradesh.
1.5 made threadsafe versions of encoding/decoding routines

2.0 added up to 8-character high-precision support (10 micron accuracy), made several changes to the raw data (see chapter 6.1).
2.1 Rewrote fraction floating points to integer arithmetic; significant speed improvements; added source code to test the library calls.
2.2 fixed micro-degree gap issue in data (see chapter 6.2)

Index