ISO/TC 211 N 4037

2015-05-28

Number of pages: 51

ISO/TC 211
Geographic information/Geomatics
Secretariat: SN (Norway)

Document type:	Other committee document
Title:	**Draft new work item proposal, The Map Code standard**
Status:	For information and consideration by the PMG and at the 40th ISO/TC 211 plenary meeting in Southampton 2015-06-11/12.
Source:	NMB of the Netherlands
Expected action:	Info
Email to secretary:	bjs@standard.no
Committee URL:	http://isotc.iso.org/livelink/livelink/open/tc211 and http://www.isotc211.org

NEW WORK ITEM PROPOSAL	
Closing date for voting	Reference number (to be given by the Secretariat)
Date of circulation	**ISO/TC** **/ SC** N 4037
Secretariat	☐ Proposal for new PC

A proposal for a new work item within the scope of an existing committee shall be submitted to the secretariat of that committee with a copy to the Central Secretariat and, in the case of a subcommittee, a copy to the secretariat of the parent technical committee. Proposals not within the scope of an existing committee shall be submitted to the secretariat of the ISO Technical Management Board.

The proposer of a new work item may be a member body of ISO, the secretariat itself, another technical committee or subcommittee, or organization in liaison, the Technical Management Board or one of the advisory groups, or the Secretary-General.

The proposal will be circulated to the P-members of the technical committee or subcommittee for voting, and to the O-members for information.

IMPORTANT NOTE: Proposals without adequate justification risk rejection or referral to originator.
Guidelines for proposing and justifying a new work item are contained **in Annex C of the ISO/IEC Directives, Part 1.**

☐ The proposer has considered the guidance given in the Annex C during the preparation of the NWIP.

Proposal (to be completed by the proposer)

Title of the proposed deliverable.
(in the case of an amendment, revision or a new part of an existing document, show the reference number and current title)

English title the Mapcode Standard

French title
(if available)

Scope of the proposed deliverable.

The Mapcode system defines a set of grids on the surface of the earth, a unique code for each grid cell, and algorithms to convert the code of a grid cell into the lattitude and longitude of the center point of that grid cell; determine, for any lattitude and longitude, the code of the grid cell(s) that contain the coordinate. The system was designed tot provide a very short, unique, easy to remember, easy to communicate code for any location on earth, in essence matching the capabilities of the lattitude/longitude system, with the provision that it intended to be accurate only on the "human scale".

Purpose and justification of the proposal*

Acces to Global Positioning Systems has become ubiquitous, built into practically every mobile phone, every car, and more and more devices. As a result, more and more entities in the world ar now easily identifiable and reachable by their physical location (their lattitude and longitude), rather than their location DESCRIPTION (such as address).

However, a short CODE is preferable to a long, technical- and complicated -looking lattitude/longitude coordinate in many situations in daily life (e.g. on business cards, or as part of an address on an envelope, or to exchange in an email, or to communicate by voice, or to remember). A code that is convertible into a lattitude/longitude would be best, having the advantages of both systems (the power of lattitude/longitude and the simplicity of a code).

**The reason for requiring justification statements with approval or disapproval votes is primarily to collect input on market or stakeholder needs, and on market relevance of the proposal, to benefit the development of the proposed ISO standard(s). Any NSB vote in relation to a proposal for new work may result in significant commitments of resources by all parties (NSBs, committee leaders and delegates/experts) or may have significant implications for ISO's relevance in the global community. It is especially important that NSBs consider and express why they vote the way they do. In addition, it is felt that it would be useful for ISO and its committees to have documentation as to why the NSBs feel a proposal has market need and market relevance. Therefore, please ensure that your justifying statements with your approval or disapproval vote convey the reason(s) why your national consensus does or does not support the market need and/or global relevance of the proposal.*

If a draft is attached to this proposal,:

Please select from one of the following options (note that if no option is selected, the default will be the first option):

☒ Draft document will be registered as new project in the committee's work programme (stage 20.00)
☒ Draft document can be registered as a Working Draft (WD – stage 20.20)
☐ Draft document can be registered as a Committee Draft (CD – stage 30.00)
☐ Draft document can be registered as a Draft International Standard (DIS – stage 40.00)

Is this a Management Systems Standard (MSS)?

☐ Yes ☒ No

NOTE: if Yes, the NWIP along with the Justification study (see Annex SL of the Consolidated ISO Supplement) must be sent to the MSS Task Force secretariat (tmb@iso.org) for approval before the NWIP ballot can be launched.

Indication(s) of the preferred type or types of deliverable(s) to be produced under the proposal.

☒ International Standard ☐ Technical Specification ☐ Publicly Available Specification ☐ Technical Report

Proposed development track ☐ 1 (24 months) ☐ 2 (36 months - default) ☒ 3 (48 months)

Known patented items (see ISO/IEC Directives, Part 1 for important guidance)

☐ **Yes** ☐ **No** **If "Yes", provide full information as annex**

A statement from the proposer as to how the proposed work may relate to or impact on existing work, especially existing ISO and IEC deliverables. The proposer should explain how the work differs from apparently similar work, or explain how duplication and conflict will be minimized.

See the very extended version 1.0 of the Mapcode Standard

A listing of relevant existing documents at the international, regional and national levels.

ISO 3166 - Codes for representation of countries and their subdivisions - Part 1: Country Codes

PRC GB/T 2260 - standard for territory abbreviations in China

WGS84 - standard for representing Earth surface locations by lattitude/longitude

A simple and concise statement identifying and describing relevant affected stakeholder categories (including small and medium sized enterprises) and how they will each benefit from or be impacted by the proposed deliverable(s)

This MapCode is already used in different GPS-systems and devices by big international companies - TomTom, NOKIA

Liaisons: **A listing of relevant external international organizations or internal parties (other ISO and/or IEC committees) to be engaged as liaisons in the development of the deliverable(s).**	**Joint/parallel work:** **Possible joint/parallel work with:** ☐ IEC (please specify committee ID) ☐ CEN (please specify committee ID) ☒ Other (please specify) UPU
A listing of relevant countries which are not already P-members of the committee. Kenia, xxx	
Preparatory work (at a minimum an outline should be included with the proposal) ☒ A draft is attached ☐ An outline is attached ☐ An existing document to serve as initial basis The proposer or the proposer's organization is prepared to undertake the preparatory work required ☒ Yes ☐ No	
Proposed Project Leader (name and e-mail address) Kewal Shienmar; kewal.shienmar@mapcode.com	**Name of the Proposer** (include contact information) NEN - Wiene Fokkinga, consultant Construction & Instalation; wiene.fokkinga@nen.nl 00 31 - 15 - 2690 322; 00 31 - 6 - 3333 0347
Supplementary information relating to the proposal ☒ This proposal relates to a new ISO document; ☐ This proposal relates to the adoption as an active project of an item currently registered as a Preliminary Work Item; ☐ This proposal relates to the re-establishment of a cancelled project as an active project. Other:	

Annex(es) are included with this proposal (give details)

☒ Annex A- Handling non-latin alphabets and works; Annex B - Encoding and Decoding Mapodes; Annex C - Territory Codes

The Mapcode Standard

Version 1.0

By P. A. Geelen, 19 May 2015

The Mapcode Standard

Version 1.0
By P. A. Geelen, 19 May 2015

i. Justification

The Mapcode system defines a set of grids on the surface of the earth, a unique code for each grid cell, and algorithms to

- convert the code of a grid cell into the latitude and longitude of the center point of that grid cell;
- determine, for any latitude and longitude, the code of the grid cell(s) that contain the coordinate

The system was designed to provide a very short, unique, easy to remember, easy to communicate code for any location on earth, in essence matching the capabilities of the latitude/longitude system, with the proviso that it intended to be accurate only on the "human scale".

Why such codes are needed

Access to Global Positioning Systems has become ubiquitous, built into practically every mobile phone, every car, and more and more devices. As a result, more and more entities in the world are now easily identifiable and reachable by their physical location (their latitude and longitude), rather than their location DESCRIPTION (such as their address).

However, a short CODE is preferable to a long, technical- and complicated-looking latitude/longitude coordinate in many situations in daily life (e.g. on business cards, or as part of an address on an envelope, or to exchange in an email, or to communicate by voice, or to remember). A code that is convertible into a latitude/longitude would be best, having the advantages of both systems (the power of latitude/longitude and the simplicity of a code).

What makes a good code

When deciding between the many different ways in which a conversion between codes and coordinates *could* be defined, the following criteria are especially important:

- is the system applicable everywhere on earth, in all countries, in all languages and in all alphabets (without sacrificing the advantages of a code system dedicated to a particular country)
- is the code easy to pronounce, to communicate
- is the code easy to write (not requiring complicated symbols, preferring few-stroke characters above multi-stroke characters)
- is the code easy to type (or are SHIFT and ALT keys needed)
- is the code short (shorter codes are easier to use, to remember, to speak, to write or type)

- is the code recognizable for what it is (e.g. when written as part of an address, can it not be confused easily with the addressee, a street name, a house number; when typed in a Google search box, can it not easily be confused with a word or a number; for example, abcdefg@xyz.de is easily recognized as an email address, simply because of its structure)

The mapcode system was designed to satisfy the above criteria in as optimal a way as possible. The mapcode system was furthermore designed to lower the barriers to world-wide adoption as much as possible, by making it a free standard allowing unconditional and unrestricted use.

ii. Scope

The mapcode system defines a set of grids on the surface of the earth, a unique code for each grid cell, and algorithms to

- convert the code of a grid cell into the latitude and longitude of the center point of that grid cell;
- determine, for any latitude and longitude, the code of the grid cell(s) that contain the coordinate

The mapcode system provides a "human face" for latitude/longitude coordinates: a way to represent a location on Earth by a short, easy to recognize, easy to remember code, sufficiently precise to specify a location on the human scale (e.g. as the destination for a trip, or as identification of a landmark).

Note that a mapcode (like a coordinate) specifies *where* a location is, while an address at best only *names* a certain location, which can only be found with knowledge of the arbitrary names given to cities and streets within a territory. Furthermore, for many locations no house number, street name (or even city name) is available, requiring even more indirect descriptions, usually relative to *other* named locations, to help find a location. Finally, even when street name and house number *are* available, the address may give no indication where the entrance, or the parking garage, of the building is located.

The mapcode system was explicitly designed to be

- included as part of addresses, e.g. on business cards, similar to how zipcodes and post codes are included, and thereby
  - help *locate* the address
  - disambiguate an ambiguous address (this includes ambiguous or faulty spelling of address components)
  - enhance the precision of an address
- offer a simple way to identify locations that HAVE no (complete, known) formal address
- make automatic sorting and processing of addresses easier; be used AS post codes in countries that do not have a sufficiently sophisticated system as yet;
- be supported by navigation systems, GIS systems and map systems, as a way (one of many) to enter a destination or identify a location

The system is explicitly NOT designed:

- for high precision (although the mapcode system *is* capable of arbitrary precision in its representation of coordinates, it is *designed* to be optimal for use when an accuracy of a few meters is sufficient);
- for 3-dimensional use (mapcodes represent locations on the Earth's surface, just like the latitude/longitude which form their basis);
- to replace addresses (on the one hand, an address may specify floor, apartment, recipient, company, and other details that can not be represented by a mapcode; on the other hand, mapcodes can specify any location, not just those that have or can have a building).

iii. Normative references

to be written…
ISO 3166 standard for territory abbreviations
PRC GB/T 2260 standard **for territory abbreviations in China**
WGS84 standard for representing Earth surface locations by latitude/longitude
…

iv. Symbols and abbreviations

to be written…

Coordinate	A WGS84 latitude and longitude
GIS	abbreviation for Geographic Information System
GPS	Global Positioning System
IEC	? International European Committee ?
SAC	Standardization Administration of China
ISO	International Standard Organisation
Prefix	the characters before the dot of a proper mapcode
Postfix	the characters after the dot of a proper mapcode
Proper mapcode	the part of a mapcode that excludes the territory code or any high-precision extensions.
WGS84	name of a standard latitude/longitude coordinate system that can be used to specify a particular location on the surface of the earth

…

1. The Format of mapcodes

1.1. Mapcode components

A full mapcode consists of an optional “territory code” and a “proper mapcode”, optionally followed by a hyphen and a “high precision **extension**”.

FullMapcode ::== [**TerritoryCode** space] **ProperMapcode** [hyphen **Extension**]

The proper mapcode consists of two groups of letters and digits separated by a dot, called the “prefix” and the “postfix”.

ProperMapcode ::== **Prefix** dot **Postfix**

The prefix is between 2 and 5 characters, the postfix between 2 and 4 characters, and the extension between 0 and 2 characters.

The territory code identifies the particular set of grids on the Earth’s surface that can be used to decode the proper mapcode back into a coordinate.

When the proper mapcode is “international” (i.e. consists of a 5-character prefix and a 4-character postfix), the territory code should be left out. However, systems will also often have to cope with non-international mapcodes for which the territory is abbreviated or left out because it is assumed to be “obvious”. See Chapter 1.4.1 on how to cope with the resulting ambiguities.

1.2. Displaying mapcodes

When displaying a mapcode, it is <u>recommended</u> to use only uppercase characters for all its components (territory code, prefix, postfix and extension), and to use a single space to separate the territory code from the proper mapcode.

By default, a mapcode has no extension. A high precision extension should *only* be generated when explicitly requested.

If the proper mapcode is in international format (i.e. consists of a 5-character prefix and a 4-character postfix), you should *not* display the territory code.

See Appendix A on how to display mapcodes in alphabets other than the Latin alphabet.

See Appendix B (and specifically Appendix B 3) for details about how to *generate* a (Latin-alphabet) mapcode based on a WGS84 coordinate.

1.3. Handling mapcode input

If characters from another alphabet than the Latin alphabet are encountered in an input that is supposed to represent a mapcode, first convert the input to the Latin alphabet (see Appendix A for details).

Once in the latin alphabet, interpret lowercase input characters as their uppercase equivalents. Leading and trailing whitespace should be removed. Whitespace around hyphens should be removed. After this, at most one whitespace sequence should remain (to separate territory code from proper mapcode), and can be replaced by a space. Otherwise, the input can not represent a mapcode.

See Appendix B (and specifically Appendix B 2) for details about how to decode a mapcode into a WGS84 coordinate.

1.4 Format of a territory code.

Appendix C lists all valid territory codes. In terms of *format*, a valid territory code is either 3 letters (following the ISO 3166-1 alpha 3 standard), or two letters followed by a hyphen followed by 2 or 3 letters or digits (following the ISO 3166-2:XX standards). Territory codes should be displayed in uppercase, although lowercase input should be accepted as valid.

It is <u>recommended</u> to *also* accept territory codes that have been changed in one of the following ways:

3-letter country codes for subdivisions

> People may enter a territory code like "US-TX" as "USA-TX" – an easy to make mistake given that USA *is* the proper *three*-letter territory code for the country. It is <u>recommended</u> to accept such territory codes as valid, since there is never any ambiguity.

2-letter country codes

> People may also abbreviate the territory code "USA" to "US", again an easy mistike to make since "US" is the proper code when used in combination with a subdivision (an american state like California, "US-CA").
>
> There are eight countries that have subdivisions in the mapcode system. In four cases (US, AU, RU, CN) there is no danger of ambiguity, so it is <u>recommended</u> to accept at least these four abbreviations as valid alternatives for entering USA, AUS, RUS or CHN.
>
> In the other four cases, there *is* an ambiguity:
> - CA as an abbreviation for CAN (Canada) may be confused with CA as an abbreviation for US-CA (California)
> - BR as an abbreviation for BRA (Brazil) may be confused with BR as an abbreviation for IN-BR (Bihar)
> - IN as an abbreviation for IND (India) may be confused with IN as an abbreviation for US-IN (Indiana)
> - MX as an abbreviation for MEX (Mexico) may be confused with MX as an abbreviation for MX-MEX (Mexico Federal District)
>
> Such ambiguity can still be solved in the same way disambiguation is done when NO territory code is specified, see Chapter 1.4.1)

1.4.1. Disambiguation of partial or missing territory codes

Systems will often have to cope with incomplete or abbreviated (and therefore ambiguous) input, because people will often consider their country context obvious, and will leave out the territory code when they communicate a mapcode.

For the same reason, they will often *abbreviate* a subdivision territory code to just the part *after* the hyphen (i.e. leaving out the country code, e.g. abbreviate "US-TX" to just "TX"). In some cases, this *does* not cause ambiguity, in the sense that there is only one valid territory code with those same letters after the hyphen. "TX" can only be the abbreviation of "US-TX", for example, and it is certainly <u>recommended</u> to accept abbreviations in all cases where there is no ambiguity. Bit there are also ambiguous abbreviations. For example, AR could be the abbreviation for US-AR (Arkansas, USA) *or* for IN-AR (Arunachal Pradesh, India).

Systems will therefore often have to cope with incomplete or abbreviated (and therefore ambiguous) input. There are four possible approaches:

- refuse the input (preferably explaining the ambiguity)
- make an assumption, but let the user *verify* the assumption
- determine *all* possible interpretations, and let the user choose between them (not recommended when more than 3 interpretations are possible)
- make an assumption and proceed as if it is correct (and assume the user will notice and can easily correct if the assumption leads to problems)

Only the fourth approach allows misinterpretation, but it will still be the right approach for many situations, e.g. for a search box on a map website.

The right approach depends on the situation, weighing the importance of interpreting a mapcode input exactly right, the chance of an assumption being wrong, and the extra effort required of the user.
Some examples of how to make an assumption:

- Assume the territory intended is the same territory that the user entered the previous time
- Choose between possible territories based on the location, language, and/or other information known about the user
- Choose between possible territories based on the location, language, and/or intended audience of the system
- Assume the territory intended based on the current context (e.g. if a screen is showing a map of France and a search box, assume France as the intended territory when a context-less mapcode is entered into the search box).

1.5. Format of the high precision extension

A high-precision extension is an *optional* part of the mapcode. It consists of a hyphen followed by one or two characters, and is appended at the end of the proper mapcode. The two characters can be digits or letters, but <u>never</u> one of the letters A,E,I,O,U or Z. (If the letters I or O are encountered, however, it is <u>recommended</u> they are interpreted as the digits 1 and 0; the alternative is to mark the input as invalid).

An extended mapcode represents locations more accurately. Each additional letter reduces the area covered by the mapcode by a factor of 30.

An average mapcode without extension covers an area of roughly 10 by 10 meters, and thus could be off by as much as 5 meters both in longitude and in latitude (or 7,1 meters combined). With a 2-letter extension, an average mapcode covers roughly a square foot (0,11 m^2), and will never be off by more than 24 centimeters.

Note: in future, the mapcode format might be extended to allow more than 2 characters after the hyphen (e.g. to specify coordinates even more accurately).

1.6. Format of a proper mapcode

A proper mapcode consists of two groups of letters and digits, separated by a dot. The part before the dot is called the **prefix**, the part after the dot is called the **postfix**. The vowels I and O can never occur in a mapcode. When encountered, it is <u>recommended</u> to assume they were intended as the digits 1 and 0, and to interpret them as such (the alternative is to not recognize the input as a valid mapcode).

The vowels A, E and U can appear only in the last two characters of the postfix, and only if all the preceding characters (of both prefix and postfix) are digits.
In some alphabets (Greek) the letter **A** (alpha) may also occur in the first position. Since the alpha is indistinguishable from the Latin letter A, you should *allow* the first letter of a proper mapcode to be an A as well.

The prefix can be 2, 3, 4 or 5 characters. The postfix can be 2, 3 or 4 characters.
If the prefix is 5 characters, the postfix *must* be 4 characters, and the mapcode is ***international***.

1.6.1. Summary: possible formats

Type	**Format**	**Description**
International mapcode	#####.####-## (note: a territory code *is* allowed!)	An international mapcode consists of exactly 5 characters before the dot and 4 characters after the dot (plus a possible extension). No territory code is required, and it is <u>**recommended**</u> to never display it. Note 1: the territory code AAA identifies the world-wide grids used to decode an internatonal mapcode. Note 2: Whether a territory code is provided or not, and whatever territory code is provided, an international mapcode must always be decoded using the AAA grids. Note 3: if a territory code *is* provided for an international mapcode, and the resulting coordinate is outside of the territory's "encompassing rectangle", it is <u>recommended</u> to refuse the mapcode as invalid.

National mapcode	**CCC** ###.###-##	A "national mapcode" consists of a 3-letter territory code CCC (see Appendix C 1 and C 3.3), a prefix of 2-4 characters, a postfix of 2-4 characters, and an optional extension. Note 1: CCC is one of the territory codes listed in appendix C 1 or Appendix C 3.3, which are the ISO 3166 alpha 3 country codes, extended with a few extra codes Note 2: Sometimes, you may see a 2-letter territory code like US instead of USA, an easy mistake to make since the US is used for local mapcodes (see below). It is recommended to at least accept US, AU, RU and CN as valid abbreviations since they are unambiguous even if not officially valid. See @@@ for more information.
Local Mapcode	CC-SS ###.###-## CC-SSS ###.###-##	A "local mapcode" consists of a 2-letter territory code CC, a 2- or 3-letter subdivision code SS, a prefix of 2-4 characters, a postfix of 2-4 characters, and an optional extension. CC is one of US, CA, MX, BR, IN, AU, RU, or CN, i.e. the ISO 3166-1 alpha 2 code for the USA, Canada, Mexico, Brazil, India, Australia, Russia, or China. Note 1: it is recommended to also accept, in this context, the THREE-letter ISO 3166-1 alpha 3 codes for the USA, Canada, Mexico, Brazil, India, Australia, Russia, or China. For example, to accept USA-TX as US-TX. See @@@ for more information. Note 2: CC-SS is one of the codes listed in Appendix C 2, and matches **ISO 3166-2:CC** codes, but *also* some alternatives for the sake of legacy or clarity. For example, **PRC GB/T 2260** codes are available as alternatives for the **ISO 3166-2:CN** codes.
Abbreviated local mapcode	SS ###.###-## SSS ###.###-##	A local mapcode where the territory code CC-SS (or CC-SSS) is *abbreviated* to SS (or SSS). Note 1. Ambiguity can derive from the fact that there are several countries in which SS or SSS is a state. For example, AL can abbreviate US-AL (Alabama), BR-AL (Alagoas) or RU-AL (Altai Republic). See Chapter 1.4.1 on how to cope with ambiguity.

		Note 2. Ambiguity can *also* derive from a 3-letter SSS being itself a valid territory code. For example, abbreviating Belgorod (RU-BEL) to BEL would be ambiguous with Belgium. Unless you are in a purely local setting, it is highly recommended to always interpret it as a national mapcode in that case and not as an abbreviation (otherwise it will be impossible to indicate mapcodes in Belgium).
Implied mapcode	###.###-##	If the prefix is less than 5 characters and no territory is spedicied, the territory is implied (e.g. considered obvious). You should refrain from ever *generating or displaying* implied mapcode. However, people will often leave out the territory code when memorizing or communicating. Implied mapcodes are always ambiguous. See Chapter 1.4.1 on how to cope with ambiguity.

Appendix A. Handling non-latin alphabets and vowels

A 1. Non-Latin alphabets

In order to support mapcodes in other alphabets, the mapcode standard includes a simple substitution table that specifies which foreign character is equivalent to which Latin character.

Encoding yields proper mapcodes using 24 letters of the Latin alphabet (forbidding the use of the O and I) and the 10 digits.
The characters of a proper mapcode and of an extension can be replaced (letter by letter and digit by digit) by "equivalent" letters from other alphabets.

To *decode* a proper mapcode or extension written in a foreign alphabet, simply substitue the Latin equivalents for each individual letter or digit.

The substitution tables that specify the equivalences between the 24 letters and 10 digits of the Latin alphabet and characters from a foreign alphabet were designed based on the following ambitions:

- if possible, do not pick any vowels; if vowels must be picked, make them the equivalent of the latin vowels A, E, U, Y (and in that order).
- if many choices exist, prefer foreign letters that
  - can be pronounced easily
  - are easy to write (e.g. have few strokes, have no accents, are similar in size to the other choices, have the same baseline…)
  - can be typed easily (e.g. without using key combinations on a ketboard)
  - are easy to recognize (e.g. are as different as possible from all other choices)
- if a letters is has (almost) the same shape as a latin letter, make it the equivalent of that letter (a good example is the latin H, the greek *eta* H and the Cyrillic *en,* H)

A 1.1. Alphabets with less than 24 symbols

As yet, the only alphabet for which we have failed to define a simple substitution is the Greek alphabet, which does not have sufficient characters to provide a substitute for all 24 Latin characters. There were enough to define 22 characters, however, which is why Latin mapcodes with the vowels E or U are pre-processed (see @@@ A 2) before converting into the Greek alphabet, and mapcodes that (after conversion into all-Latin characters) start with the vowel A are pre-processed are pre-processed before decoding.

A 2. Vowels versus all-digit mapcodes

To prevent all-digit mapcodes, the results of the encoding process (see Appendix B 3) are *post-processed* when they are all-digit, by replacing the last 2 digits of an all-digit proper mapcode with an A, E or U, followed by another character (which may also be an A, E or U). This also means that any mapcode that has an A, E or U in the one-but-last position needs to be *pre-processed* (see Appendix A 2.1) before it is further decoded (as described in Appendix B 2).

An alternative technique was added later to support the Greek alphabet (which only has 22 characters when I and O are excluded). Using this technique, the *first* character of the mapcode is replaced by the letter A, and the last two characters are replaced by a combination of two letters (both if which may also be an A, but never E or U).

A 2.1. removing vowels

The algorithm to post-process a proper mapcode **mc** that is in uppercase Latin alphabet and potentially has vowels (before further decoding into a coordinate) is as follows:

```
function removeVowels(mc)
{
    character array encodes[] = "0123456789BCDFGHJKLMNPQRSTVWXYZAEU"
    character c1 = mc[len-2]
    character c2 = mc[len-1]
    // handle mapcodes with a leading A
    if ( mc[0]=='A' )
    {
        integer len = mc.length
        integer v = encodes.IndexOf(c1) + (32 * encodes.IndexOf(c2))

        mc[    0] = encodes[v div 100]
        mc[len-2] = encodes[(v div 10) mod 10]
        mc[len-1] = encodes[v mod 10]
    }
    // handle mapcodes with a vowel in the one-but-last position
    else if ( c1=='A' || c1=='E' || c1=='U' )
    {
        integer v = ( 34*(encodes.IndexOf(c1)-31) ) + encodes.IndexOf(c2)
        mc[len-2] = encodes[v div 10]
        mc[len-1] = encodes[v mod 10]
    }
    Return mc
}
```

Note that this pre-processing is necessary no matter where the mapcode came from. For example, the greek mapcode A0.23 looks like a Latin mapcode, and may be entered in your system as if it is a Latin mapcode, since is is virtually impossible to distinguish the greek capital *alpha* (**A**) from the Latin capital *a* (**A**).

A 2.2. adding vowels to prevent all-digit mapcodes

The algorithm to post-process an all-digit proper mapcode **mc** is as follows:

```
function string packAlldigitCode( mc, useGreekSystem )
{
    character array encodes[] = "0123456789BCDFGHJKLMNPQRSTVWXYZAEU"
    integer len = mc.length
    integer v = ( 10 * encodes.IndexOf(mc[len-2]) ) + encodes.IndexOf(mc[len-1])

    if ( useGreekSystem )
    {
        v = v + ( 100 * encodes.IndexOf(mc[0]) )
        mc[0]     = 'A'
        mc[len-2] = encodes[ v div 32 ]
        mc[len-1] = encodes[ v mod 32 ]
    }
    else
    {
        mc[len-2] = encodes[ 31 + (v div 34) ]
        mc[len-1] = encodes[ v mod 34 ]
    }
    return mc;
}
```

Note that this algorithm should be used with **useGreekSystem** set to false.

A 2.3. adding vowels for the Greek alphabet

The algorithm in the previous section should normally be used with **useGreekSystem** set to false. *Only* when **mc** contains an E or U, and *only* just before conversion of **mc** into Greek (or another alphabet that has no substitute letter defined for the character E), is this parameter ever set to **true**, as in:

```
// repack latin mapcode mc just before conversion into Greek alphabet:
if ( mc.indexOf('E')>=0 || mc.indexOf('U')>0 )
{
    packAlldigitCode( removeVowels(mc), true )
}
```

Appendix B Encoding and Decoding Mapcodes

B 1. Basic routines

B 1.1. Basic routines for territories

The mapcode system defines a database for over 500 territories. Each territory is identified by its territory code (see appendix C). For example, the Netherlands has territory code "NLD". A territory can be a subdivision of another territory (which is called the "parent" territory). For example, California ("US-CA") is a subdivision of the United States of America ("USA"). The following routines are used to navigate this simple structure:

isSubdivision(tc)	returns true if **tc** is a subdivision (e.g. a state)
ParentTerritoryOf(tc)	returns the parent territory of a subdivision **tc**

The algorithms to encode and decode mapcodes depend heavily on a large data table, specifying population-density-based grids that cover the territories. For every territory, one or more **territory rectangles** are available. The last of these rectangles is called the **encompassing rectangle**. The mapcode algorithm has access to these rectangles through

firstRectangle(tc)	returns the first territory rectangle of **tc**
lastRectangle(tc)	returns the last territory rectangle of **tc** (also called the *encompassing rectangle*)

To access information about any territory rectangle **i**, the following routines are available:

minx(i)	returns the minimum longitude (inclusive) in millionths of degrees
maxx(i)	returns the maximum longitude (exclusive) in millionths of degrees
miny(i)	returns the minimum latitude (inclusive) in millionths of degrees
maxy(i)	returns the maximum latitude (exclusive) in millionths of degrees
prefixLength(i)	returns the prefix length defined for this territory rectangle
postfixLength(i)	returns the postfix length defined for this territory rectangle
coDex(i)	returns prefixLength(i) * 10 + postfixLength(i)
coDexLen(i)	returns prefixLength(i) + postfixLength(i)
recType(i)	returns the rectangle type (0,1,2 or 3)
recLetter(i)	returns a mapcode character *(only if recType(i)==1)*
smartDiv(i)	returns the "divider value" for the territory rectangle i
isNameless(i)	returns true if the "nameless" algorithm is required
isNonEncoding(i)	returns true if the validity depends on other territory rectangles @@@
isSpecialShape(i)	returns true if the territory rectangle has a "special shape" @@@

B 1.2. Basic data tables

There are several small arrays of integers required by the algorithms:

```
integer array nc = [   1, 31, 961, 29791, 923521, 28629151, 887503681 ]

integer array xside = [ 0, 5,  31,  168,    961,    5208]

integer array yside = [ 0, 6,  31,  176,    961,    5456]

integer array xdivider19 = [
          360,  360,  360,  360,  360,  360,  361,  361,  361,  361,
          362,  362,  362,  363,  363,  363,  364,  364,  365,  366,
          366,  367,  367,  368,  369,  370,  370,  371,  372,  373,
          374,  375,  376,  377,  378,  379,  380,  382,  383,  384,
          386,  387,  388,  390,  391,  393,  394,  396,  398,  399,
          401,  403,  405,  407,  409,  411,  413,  415,  417,  420,
          422,  424,  427,  429,  432,  435,  437,  440,  443,  446,
          449,  452,  455,  459,  462,  465,  469,  473,  476,  480,
          484,  488,  492,  496,  501,  505,  510,  515,  520,  525,
          530,  535,  540,  546,  552,  558,  564,  570,  577,  583,
          590,  598,  605,  612,  620,  628,  637,  645,  654,  664,
          673,  683,  693,  704,  715,  726,  738,  751,  763,  777,
          791,  805,  820,  836,  852,  869,  887,  906,  925,  946,
          968,  990, 1014, 1039, 1066, 1094, 1123, 1154, 1187, 1223,
         1260, 1300, 1343, 1389, 1438, 1490, 1547, 1609, 1676, 1749,
         1828, 1916, 2012, 2118, 2237, 2370, 2521, 2691, 2887, 3114,
         3380, 3696, 4077, 4547, 5139, 5910, 6952, 8443,10747,14784,
         23681,59485 ]
```

In fact, the algorithms only make use of xdivider19 through the following access routine:

```
// Get divider for a latitude range
// (where miny and maxy specified in millionths of degrees)
// note: (d>>19) is equal to (d div 524288)
function xDivider(integer miny, integer maxy)
{
    if (miny>=0)
        return xdivider19[ miny>>19 ]
    if (maxy>=0)
        return xdivider19[0]
    return xdivider19[ (-maxy)>>19 ]
}
```

B 1.3. Required low-level routines

The algorithms below are described in pseudo-code. We will assume simple string manipulations:

s.length	returns length of string s
s[x]	returns x-th character of string s (as a one-letter string)
s.charCodeAt(x)	returns the ascii code of the x-th character of string s (as an integer)
s.indexOf(c)	return index in s of character c (or negative)
s.substr(x,n)	returns n characters of string s starting as of the x-th character
s.substr(x)	returns all characters of string s starting as of the x-th character

We also assume the usual arithmetic operators, including **div** and **mod** operators:

A div B	returning the integer result (or mathematical "floor") of dividing integer A by integer B
A mod B	returning the integer remainder of a division of integer A by integer B

B 1.4. Basic routines to see if a coordinate is inside a territory rectangle

To check if a coordinate (of which the **x** and **y** are expressed as integers in millionths of degrees) is inside territory rectangle **i**:

```
function fitsInside( coordinate, i )
{
    if ( isInRangeY( coordinate.y, miny(i), maxy(i) ) )
        return( isInRangeX( coordinate.x, minx(i), maxx(i) ) )
    return false
}
```

However, for several checks performed in the algorithms, we need to use the following:

```
function fitsWithRoom( coordinate, i )
{
    if ( isInRangeY( coordinate.y, miny(i)-45, maxy(i)+45 ) )
    {
        degrees xdiv8 = xDivider(miny(i),maxy(i))/4
        return( isInRangeX( coordinate.x, minx(i)-xdiv8, maxx(i)+xdiv8 ) )
    }
    return false
}
```

where

```
function isInRangeY(y, miny, maxy)
{
    return (miny<=y && y<maxy)
}
```

and

```
function isInRangeX(x, minx, maxx)
{
    if ( minx<=x && x<maxx )
        return true
    if (x<minx)
        x+=360000000
    else
        x-=360000000
    return ( minx<=x && x<maxx )
}
```

B 1.5. Basic arrays to encode and decode Latin characters

```
// decode_chars[c] is negative when ASCII character c can not be decoded
integer decode_chars = [
     -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
     -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
     -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
      0,  1,  2,  3,  4,  5,  6,  7,  8,  9, -1, -1, -1, -1, -1, -1,
     -1, -2, 10, 11, 12, -3, 13, 14, 15,  1, 16, 17, 18, 19, 20,  0,
     21, 22, 23, 24, 25, -4, 26, 27, 28, 29, 30, -1, -1, -1, -1, -1,
     -1, -2, 10, 11, 12, -3, 13, 14, 15,  1, 16, 17, 18, 19, 20,  0,
     21, 22, 23, 24, 25, -4, 26, 27, 28, 29, 30, -1, -1, -1, -1, -1,
     -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
     -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
     -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
     -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
     -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
     -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
     -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
     -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1 ]

// encode_chars[x] returns a normal mapcode character for 0<=x<=30
// encode_chars[x] returns a mapcode vowel for 31<=x<=33
integer encode_chars = [
    '0','1','2','3','4','5','6','7','8','9',
    'B','C','D','F','G','H','J','K','L','M',
    'N','P','Q','R','S','T','V','W','X','Y','Z',
    'A','E','U']
```

B 2. Decoding a mapcode

B 2.1. The decode algorithm

Decoding a mapcode into a WGS84 coordinate is done in several steps. One needs to separate it into a **territory code**, a **proper mapcode**, and the optional **high-precision extension**. After that, the following steps must be taken:

Step 1: disambiguation of the territory (if necessary)

First, check and if necessary disambiguate the territory code (see Chapter 1.4.1 on how to handle ambiguity, or assume a territory if none is given).

Step 2: conversion into Latin alphabet (if necessary)

The proper mapcode *and* the optional extension may be using letters from non-Latin alphabets, in which case the foreign characters need first be replaced by their Latin equivalents. See A 1 on details @@@.

Step 3: pre-processing vowels

When it was encoded, a mapcode may have been post-processed to prevent it from consisting only of digits. In that case (and only in that case) the proper mapcode may contain vowels. To be precise: either the first letter of the proper mapcode is an A, or the one-but-last letter is an A, E or U.

The **removeVowels** algorithm, for pre-processing such vowels away and allowing normal decoding in the fourth and final step, is described in A 2@@@.

Step 4: decoding

The *principle* behind the decoding algorithm is as follows:

- The mapcode system defines a database for over 500 territories. Each territory is covered by a set of one or more **territory rectangles**. The territory code identifies such a set.
- The length of the mapcode prefix and the length of the mapcode postfix together identify a particular territory rectangle.
- The characters in the prefix, postfix, and high-precision extension together identify a particular rectangular sub-area within that territory rectangle (a particular “cell” in the “grid” defined by that territory rectangle).
- The coordinate in the centre of that sub-area is returned as the result (i.e. it represents the coordinate “equivalent” of the mapcode).

The following algorithm decodes a proper mapcode string **mapcode** which has already been pre-processed (i.e. all characters are in Latin alphabet, the vowels were pre-processed away, the extension (of 0, 1 or 2 letters) is in a separate string **extension**, and a valid (possibly disambiguated) territory code is in **territory**. The algorithm returns a coordinate in **result**.

Note that the algorithms are based on integer arithmetic, so that x and y are returned as integers representing millionths of degrees.

```
// determine length of prefix and postfix
```

```
integer prelen  = mapcode.indexOf(".")
integer postlen = mapcode.length - 1 - prelen

// refuse mapcodes of invalid length
if ( prelen<2 || prelen>5 || postlen<2 || postlen>4 )
    return ERROR

// international mapcodes must be interpreted in territory AAA
if (prelen+postlen==9)
    territory="AAA"

// long mapcodes must be interpreted in the parent of a subdivision
if (prelen+postlen==7 && ParentTerritoryOf(territory)=="IND" )
    territory = "IND"
if (prelen+postlen==7 && ParentTerritoryOf(territory)=="MEX" )
    territory = "MEX"
else if (prelen+postlen==8 && isSubdivision(territory) )
    territory = ParentTerritoryOf(territory)

// to decode, try all territory rectangles
for ( i = firstRectangle(territory); i <= lastRectangle(territory); i++ )
{
    if ( recType(i)==0 && isNameless(i)==false
            && prefixLength(i)==prelen && postfixLength(i)==postlen)
    {
        result = decode_grid( mapcode, i, extension )

        if (isNonEncoding(i) )
        {
            // results MUST be inside some rectangle that is marked “Encoding”
            boolean fitsSomewhere=false
            for (j=lastRectangle(territory)-1; j>=firstRectangle(territory); j--)
            {
                if ( isNonEncoding(j)==false && fitsWithRoom( result, j ) )
                {
                    fitsSomewhere =true
                    break
                }
            }
            if (fitsSomewhere ==false)
                return ERROR
        }
        break
    }
    else if ( recType(i)==1 && prefixLength(i)+1==prelen
            && postfixLength(i)==postlen && recLetter(i)==mapcode[0] )
    {
        result=decode_grid( mapcode.substr(0,1), i, extension )
        break
    }
    else if (isNameless(i) &&
     (  (prefixLength(i)==2 && postfixLength(i)==1 && prelen==2 && postlen==2)
     || (prefixLength(i)==2 && postfixLength(i)==2 && prelen==3 && postlen==2)
     || (prefixLength(i)==1 && postfixLength(i)==3 && prelen==2 && postlen==3) ) )
    {
        result = decode_nameless( mapcode, i, extension )
        break
    }
    else if ( recType(i)>=2 && postlen==3
        && prefixLength(i)+postfixLength(i)==prelen+2 )
    {
        result = decode_starpipe( mapcode, i, extension )
        break
    }
}

// normalise and check if really in territory
if (result==ERROR)
    return ERROR

// normalise the result
if ( result.x>180000000)
    result.x-=360000000
else if ( result.x<-180000000 )
    result.x+=360000000

// make sure result fits the country
```

```
if ( territory != "AAA" )
    if ( fitsWithRoom(result, lastRectangle(territory))==false )
        return ERROR
```

This routine uses one of three subroutines to decode a mapcode: decode_grid, decode_nameless, and decode_starpipe. The algorithms for these are as follows:

B 2.1. the decode_grid algorithm

The following routine takes a proper mapcode **mc** and extension **extension** and decodes it (if possible) for the territory rectangle **m**.

```
function decode_grid( mc, m, extension )
{
    // copy information about territory rectangle m
    integer minx = minx(m)
    integer miny = miny(m)
    integer maxx = maxx(m)
    integer maxy = maxy(m)
    integer divy = smartDiv(m)

    // determine the length of the mapcode prefix and postfix
    integer prelen  = mc.indexOf(".")
    integer postlen = mc.length - 1 - prelen

    // rewrite an 1.3 mapcode into a 2.2 mapcode
    if (prelen==1 && postlen==3)
    {
        prelen++
        postlen--
        mc = mc[0] + mc[2] + mc[1] + mc[3] + mc[4]
    }

    // determine the way the rectangle will be divided horizontally and vertically
    if (divy==1)
    {
        divx = xside[prelen]
        divy = yside[prelen]
    }
    else
    {
        divx = ( nc[prelen] div divy )
    }

    // for 961x961 grids, swap the 2nd and 3d prefix letters
    if ( prelen==4 && divx==961 && divy==961 )
    {
        mc = mc[0] + mc[2) + mc[1] + mc.substr(3)
    }

    // decode the prefix into an integer
    integer v = fast_decode(mc)

    coordinate rel
    if ( divx!=divy && prelen>=3 )
    {
        // large non-square grids use “type 6” grid cell naming
        rel = decode6(v,divx,divy)
    }
    else
    {
        rel.x=       (v div divy)
        rel.y=divy-1-(v mod divy)
    }

    integer ygridsize = ((maxy-miny+divy-1) div divy)
    integer xgridsize = ((maxx-minx+divx-1) div divx)

    rel.y = miny + (rel.y*ygridsize)
    rel.x = minx + (rel.x*xgridsize)

    integer dividery =  (ygridsize + yside[postlen] - 1) div yside[postlen]
```

```
    integer dividerx =  (xgridsize + xside[postlen] - 1) div xside[postlen]

    // get the postfix
    var rest = mc.substr(prelen+1)

    // decode postfix versus rel
    coordinate dif
    if ( postlen==3 )
    {
        // decode 3-letter postfix
        dif = decode_triple(rest)
    }
    else
    {
        // swap 2nd and 3d characxter of 4-letter postfix
        if ( postlen==4 )
            rest = rest[0] + rest[2] + rest[1] + rest[3]

        // decode 2- or 4-letter postfix
        integer v = fast_decode(rest)
        dif.x = ( v div yside[postlen] )
        dif.y = ( v mod yside[postlen] )
    }

    dif.y = yside[postlen]-1-dif.y

    // return result including extension
    coord corner
    corner.y = rel.y + (dif.y * dividery)
    corner.x = rel.x + (dif.x * dividerx)
    return decode_extension( corner, 4*dividerx, dividery, 1, extension )
}

// lowest level encode/decode routines
function fast_decode(code)
{
  integer value = 0
  for ( i=0; i<code.length; i++ )
  {
    integer c = code.charCodeAt(i)
    if ( c==46 ) // dot?
      break
    if ( decode_chars[c]<0 )
      return ERROR
    value = value*31 + decode_chars[c]
  }
  return value
}

// adjust point with extension
function decode_extension(point,dividerx4,dividery,dy,extension)
{
    if (extension.length==0)
    {
        point.x += (dividerx4 div 8)
        point.y += ((dividery div 2)*ydirection)
        return point
    }

    integer c1 = decode_chars[extension.charCodeAt(0)]
    if (c1<0) c1=0 else if (c1>29) c1=29
    integer y1 = (c1 div 5)
    integer x1 = (c1 mod 5)
    integer c2 = 15
    if (extension.length>1)
    {
        c2 = decode_chars[extension.charCodeAt(1)]
        if (c2<0) c2=0 else if (c2>29) c2=29
    }
    point.x += ( ( ( (x1*12 + 2*(c2 mod 6) + 1)*dividerx4 + 120) div 240) )
    point.y += ( ( ( (y1*10 + 2*(c2 div 6) + 1)*dividery  +  30) div  60) * dy)
    return point
}

function decode_triple(str)
{
  integer x = fast_decode( str.substr(1) )
```

```
  integer c = decode_chars[ str.charCodeAt(0) ]
  coordinate triple
  if ( c<24 )
  {
    triple.x = (c mod 6) * 28 + (x div 34)
    triple.y = (c div 6) * 34 + (x mod 34)
  }
  else
  {
    triple.y = (x mod 40) + 136
    triple.x = (x div 40) + 24*(c-24)
  }
  return triple
}

function decode6(v,width,height)
{
    integer D=6
    integer col = (v div (height*6))
    integer maxcol = ((width-4) div 6)
    if ( col>=maxcol )
    {
        col=maxcol
        D = width-maxcol*6
    }
    integer w = v - (col * height * 6 )

    coordinate r
    r.x = col*6    + (w mod D)
    r.y = height-1 - (w div D)
    return r
}
```

B 2.2. the decode_nameless algorithm

The following routine takes a proper mapcode mapcode and extension extension decodes it (if possible) for the territory rectangle firstrec.

```
function decode_nameless( mapcode, firstrec, extension )
{
    // remove the dot
    integer prelen  = mc.indexOf(".")
    string mc = mapcode.substr(0,prelen) + mapcode.substr(prelen+1)

    // determine nr of nameless records available
    integer codex = coDex(firstrec)
    integer A = count_city_coordinates_for_country(firstrec,firstrec)

    integer p = (31 div A)
    integer r = (31 mod A)
    boolean swapletters=false
    integer v

    integer X
    if ( codex!=21 && A<=31 )
    {
        integer offset = decode_chars[ mc.charCodeAt(0) ]

        if ( offset < r*(p+1) )
        {
            X = ( offset div (p+1) )
        }
        else
        {
             if (codex==22 && p==1)
                swapletters = true
            X = r + ( (offset-(r*(p+1))) div p )
        }
    }
    else if ( codex!=21 && A<62 )
    {
        X = decode_chars[ mc.charCodeAt(0) ]
        if ( X < (62-A) )
        {
```

```
        if (codex==22)
            swapletters = true
    }
    else
    {
        X = X+(X-(62-A))
    }
}
else // codex==21 || A>=62
{
    integer BASEPOWERA = (((codex==21) ? 961*961 : 961*961*31) div A)
    if (A==62)
        BASEPOWERA++
    else
        BASEPOWERA = 961*(BASEPOWERA div 961)

    // decode and determine X
    v = fast_decode(mc)
    X  = (v div BASEPOWERA)
    v  = (v mod BASEPOWERA)
}

if (swapletters && isSpecialShape(firstrec+X)==false )
{
    mc = mc[0] + mc[1] + mc[3] + mc[2] + mc[4]
}

if ( codex!=21 && A<=31 )
{
    v = fast_decode(mc)
    if (X>0)
    {
        v -= ( (X*p + (X<r ? X : r)) * (961*961) )
    }
}
else if ( codex!=21 && A<62 )
{
    v = fast_decode(mc.substr(1))
    if ( X >= (62-A) )
        if ( v >= (16*961*31) )
        {
            v -= (16*961*31)
            X++
        }
}

if (X>A)
    return ERROR

integer m = firstrec+X
integer minx = minx(m)
integer miny = miny(m)
integer maxx = maxx(m)
integer maxy = maxy(m)
boolean specialShape = isSpecialShape(m)
integer SIDE = smartDiv(m)

integer xSIDE=SIDE
if ( specialShape )
{
    xSIDE *= SIDE
    SIDE = 1 + ((maxy-miny) div 90)
    xSIDE = (xSIDE div SIDE)
}

coordinate d
if ( specialShape )
{
    d = decode6(v,xSIDE,SIDE)
    d.y = SIDE-1-d.y
}
else
{
    d.y = (v mod SIDE)
    d.x = (v div SIDE)
}
```

```
    if ( d.x >= xSIDE ) // out of range
        return ERROR

    integer dividerx4 = xDivider(miny,maxy)
    integer dividery  = 90

    coordinate corner
    corner.x = minx + ((d.x*dividerx4) div 4)
    corner.y = maxy - (d.y*dividery)
    return decode_extension(corner,dividerx4,dividery,-1,extension)
}

function firstNamelessRecordForCountry(index,firstrec)
{
    integer i=index
    while ( i>=firstrec && coDex(i)==coDex(index) && isNameless(i) ) i--
    return (i+1)
}

function countCityCoordinatesForCountry(index,firstrec)
{
    integer e = index
    while ( coDex(e)==coDex(index) ) e++
    return (e-1) - firstNamelessRecordForCountry(index,firstrec)
}
```

B 2.3. the decode_starpipe algorithm

The following routine takes a proper mapcode mc and extension extension decodes it (if possible) for the territory rectangle firstindex.

```
function decode_starpipe(mc,firstindex,extension)
{
    // decode prefix
    integer value  = fast_decode(mc)*31*31*31

    // decode postfix (always 3 characters!)
    integer triple = decode_triple( mc.substr( mc.length - 3 ) )

    integer STORAGE_START=0
    for( i=firstindex; CoDexLen(i)==CoDexLen(firstindex); i++)
    {
        // copy information about territory rectangle i
        integer minx = minx(i)
        integer miny = miny(i)
        integer maxx = maxx(i)
        integer maxy = maxy(i)
        integer rt = recType(i)

        integer H = ((maxy-miny+89) div 90)
        integer xdiv = xDivider(miny,maxy)
        integer W = ( ( (maxx-minx)*4 + (xdiv-1) ) div xdiv )

        H = 176*( (H+176-1) div 176 )
        W = 168*( (W+168-1) div 168 )

        integer product = (W div 168) * (H div 176) *31*31*31

        if ( rt==2 )
        {
            integer GOODROUNDER = codex>=23 ? (31*31*31*31) : (31*31*31)
            product = (((STORAGE_START+product+GOODROUNDER-1)
                div GOODROUNDER) * GOODROUNDER) - STORAGE_START
        }

        if ( value >= STORAGE_START && value < STORAGE_START + product )
        {
            // code belongs in THIS territory rectangle!
            integer dividerx = ((maxx-minx+W-1) div W)
            integer dividery = ((maxy-miny+H-1) div H)

            value = ( (value-STORAGE_START) div (31*31) )

            integer vx = (value div (H div 176)) * 168 + triple.x
```

```
            integer vy = (value mod (H div 176)) * 176 + triple.y

            coordinate corner
            corner.y = maxy - vy * dividery
            corner.x = minx + vx * dividerx

            corner = decode_extension(corner,dividerx*4,dividery,-1,extension)
            if (corner.x<minx || corner.x>=maxx || corner.y<miny || corner.y>maxy)
                return ERROR
            return corner
        }

        // try the next territory rectangle...
        STORAGE_START += product
    }
    Return ERROR
}
```

B 3. Encoding a coordinate

Given a coordinate, and optionally the territory code to encode it in, one can generate one or more mapcodes, each "representing" the coordinate.

B 3.1. the principle behind encoding

The *principle* behind the encoding algorithm is as follows:

- The mapcode system defines a database for over 500 territories. Each territory is identified by a unique territory code (see appendix C).
- Each territory has a set of one or more **territory rectangles**. The last of those rectangles is called the **encompassing rectangle.** Only *some* of the territory rectangles are marked as "encoding".
- If a coordinate lies within the encompassing rectangle of a territory **T** *and also* within the boundaries of an "encoding" territory rectangle **R**, then the coordinate can be "encoded in rectangle R": the algorithm below will yield a **proper mapcode** and optionally a 1- or 2-character **extension**, which combined with the **territory code** of territory **T** yields a **full mapcode**.
- Basically, each territory rectangle is divided into a grid of small rectangular, numbered sub-areas. The number of the cell in which the coordinate is located is returned. The number doesn't just use digits, but a combination of digits and letters.
- The cells are usually 10 by 10 meters, which means a mapcode represents the coordinate imprecisely: the coordinate known to be *somewhere* within the cell. When decoding a mapcode, the center of the cell will be returned as coordinate, which can thus be up to 5 meters distant from the original both in latitude and in longitude (or 7,1 meters, worst-case, diagonally).

B 3.1.1. one coordinate, multiple mapcodes in a territory

The territory rectangles within a single territory **T** may overlap, so that a single location (i.e. a single coordinate) may have more than one mapcode.

Since any single mapcode is sufficient to represent the location, it is <u>recommended</u> to only offer the *first* code generated in that territory (i.e. for the first territory rectangle that can encode the coordinate). This will also always yield the *shortest* possible mapcode in territory **T**).

An alternative is to offer <u>all</u> options and leave the choice to the user. Picking a *single* mapcode that is *not* the shortest code is never recommended.

> **Note:** given that recommendation to always offer the shortest mapcode in a territory, one may ask why the mapcode system is not simply designed to *only* produce the shortest code within a particular territory. The reason was twofold. First of all, it was deemed useful to allow a user a choice in case the default was somehow not to his or her liking (e.g. because the number 13 occurs in it). Secondly, we could imagine circumstances in which it would be beneficial to standardize on a particular length. For example, *every* dwelling in The Netherlands has a 6-character mapcode with a 3-character prefix. Although we expect people in the capital to prefer their 4-character alternatives, and we thus do <u>not</u> recommend ever defaulting to anything but the shortest mapcode, *some* systematic or bureaucratic benefit might ensue from using only the 6-character codes. The mapcode system *as such* therefore considers all possible mapcodes equally valid.

B 3.1.2. one coordinate, mapcodes in multiple territories

The encompassing rectangles of different territories may overlap, so that the encoding algorithm may yield mapcodes in more than one territory.

In fact, any on-land coordinates is virtually certain to have mapcodes in more than one territory since the “international” territory encompasses the whole world and overlaps *every* other territory.

All mapcodes are of course “valid” in the sense that they will correctly decode back to (approximately) the original coordinate. But not all of them may be valid “*politically*”, since territories are divided and encompassed by simple mapcode rectangles whereas the real world is not. It may be possible to develop a system (*outside* of the mapcode system) which can automatically decide which mapcodes are *politically* valid – but such a system would itself be political, given the many disagreements about precise boundaries that exist in the world today.

In general, it is recommended to stimulate the requestor of a mapcode as much as possible to provide the mapcode territory to encode a coordinate in *beforehand.* The *user* of a mapcode will usually have little problem in deciding the proper context (e.g. in what territory he lives). If no territory is defined beforehand, we recommend one of the following ways to choose between the mapcodes of different territories:

(1) the choice is **left to the user** (<u>**recommended**</u>). In that case, we further recommend to help the user with a “default” choice based on reasonable assumptions (e.g. if the previous request from a user had territory code FRA, assume FRA; on a Dutch website, always assume NLD, etc.);
(2) the choice is **based on a separate system**, capable of determining the political territory in which the coordinate lies (in which case it could just as well have been passed as part of the encoding *request*!)
(3) given a choice between more than one *national* territory, always **choose the international mapcode** (in other words, err on the safe side, since the international code is always politically and physically correct. The disadvantage is that the international alternative is, unfortunately, always the longest possible code, most awkward to remember and use)

These possibilities can be combined at will, of course, e.g. using (2) in politically uncontroversial locations and (3) when territorial borders are physically or politically unclear. Or using (3) as the default for (1).

B 3.1.3. the encode algorithm

This section describes how to convert (“encode”) a WGS84 coordinate into one or more mapcodes.

Step 1: disambiguation of the territory (if necessary)

The encoding process needs both a coordinate and a valid territory code (see Appendix A) of the territory to encode it in. If you *only* have a coordinate, you need to either try to encode it in *every* territory (which may succeed in more than one territory) or determine the “right” one to encode it in beforehand. See the previous section (Appendix B 3.1.2) for a further discussion about possible approaches

Step 2: production of a Latin-alphabet mapcode

Given a valid mapcode territory **t** and a coordinate **coord** (with integers coord.y and coord.x specifying a latitude and longitude in millionths of degrees), the algorithm **encode(coord,t,results,d)** below specified below appends (zero or more) mapcodes in territory **t** to the array **results** (which should be emptied before the call). Each mapcode will be generated with a high-precision extension of **extraDigits** characters (note that **extraDigits**=0 is always the recommended default).

Step 3: conversion into a foreign alphabet (if necessary)

The **encode** algorithm below generates results in the Latin alphabet. Results can be postprocessed into other alphabets using a simple character-substitution system described in @@@ A 1.

However, note that the **encode** algorithm uses **packAlldigitCode(r,false)** to prevent all-digit mapcodes. This may introduce the vowels E and U into some results. When converting a mapcode that contains an E or a U into a non-Latin alphabet, you may need to re-pack it first as described in @@@ A 2.3 to cope with alphabets that have no substitute for these vowels.

```
function encode(coord,territory,results_array,extraDigits)
{
    // make sure it belongs to the territory
    integer from = firstRectangle(territory)
    integer upto = lastRectangle(territory)
    if ( territory!= "AAA" )
        if ( fitsInside(coord, upto)==false )
            return results_array

    integer initial_length = results_array.length

    for ( i=from; i<=upto; i++ )
    {
        if ( coDex(i)<54 )
        {
            if ( fitsInside(coord, i) )
            {
                String r = ERROR
                if ( i==upto && isNonEncoding(i) && isSubdivision(territory) )
                {
                    // last record of a subdivision is nonEncoding:
                    // recursively add parent mapcodes
                    return master_encode( coord, ParentTerritoryOf(territory),
                        results_array)
                }
                else if (recType(i)==0 && isNameless(i)==0)
                {
```

```
                    if (isNonEncoding(i) && results_array.length==initial_length)
                    {
                        // ignore: nothing was yet found in this territory
                    }
                    else
                    {
                        r = encode_grid(coord,i,"",territory,extraDigits)
                    }
                }
                else if (recType(i)==1)
                {
                    r = encode_grid(coord,i,recLetter(i),territory,extraDigits)
                }
                else if (isNameless(i))
                {
                    r = encode_nameless(coord,i,from,extraDigits)
                }
                else // recType(i)>1
                {
                    r = encode_starpipe(coord,i,territory,extraDigits)
                }

                // add the result to the array
                if ( r!=ERROR )
                {
                    r = packAlldigitCode(r,false)
                    results_array.appendToArray(r)
                }
            }
        }
    }

    return results_array
}
```

B 3.2. the encode_grid algorithm

```
function string fast_encode(value,nrchars)
{
  string str = ""
  while ( nrchars-- > 0 )
  {
    str = encode_chars[ value mod 31 ] + str
    value = (value div 31)
  }
  return str
}

function string encode_triple(coord)
{
    if ( coord.y < 4*34 )
        return encode_chars[ ((coord.x div 28) + 6*(coord.y div 34)) ] +
fast_encode( (coord.x mod 28)*34 + (coord.y mod 34), 2 )
    else
        return encode_chars[  (coord.x div 24) + 24                  ] +
fast_encode( (coord.x mod 24)*40 + (coord.y - 136 ), 2 )
}


function integer encode6(x,y,width,height)
{
    integer col = (x div 6)
    integer maxcol = ((width-4) div 6)
    integer d=6
    if ( col>=maxcol )
    {
        col=maxcol
        d = width-maxcol*6
    }
    return (height*col*6) + (height - 1 - y)*d + (x - col*6)
}

function string encode_extension(extrax4,extray,dividerx4,dividery,extraDigits)
{
```

```
    if (extraDigits<=0)
        return ""

    integer gx = ((30*extrax4) div dividerx4)
    integer gy = ((30*extray ) div dividery )
    integer x1=(gx div 6)
    integer x2=(gx mod 6)
    integer y1=(gy div 5)
    integer y2=(gy mod 5)

    string extension=encode_chars[ y1*5+x1 ]
    if (extraDigits==2)
        extension = extension + encode_chars[ y2*6+x2 ]
    return extension
}

function encode_grid(coord,m,firstletter,territory,extraDigits)
{
    // copy information about territory rectangle m
    integer minx = minx(m)
    integer miny = miny(m)
    integer maxx = maxx(m)
    integer maxy = maxy(m)
    integer prelen  = prefixLength(m)
    integer postlen = postfixLength(i)
    integer orglen  = prelen
    if (prelen==1)
    {
        prelen++
        postlen--
    }

    // determine way to subdivide rectangle
    divy = smartDiv(m)
    if (divy==1)
    {
        divx = xside[prelen]
        divy = yside[prelen]
    }
    else
    {
        divx = ( nc[prelen] div divy )
    }


    integer ygridsize = ((maxy-miny+divy-1) div divy)
    integer rel.y = coord.y - miny
    rel.y = (rel.y div ygridsize)
    integer xgridsize = ((maxx-minx+divx-1) div divx)

    integer rel.x = coord.x - minx
    if (rel.x<0)
    {
        coord.x += 360000000
        rel.x += 360000000
    }
    else if (rel.x>=360000000)
    {
        coord.x -= 360000000
        rel.x -= 360000000
    }
    if (rel.x<0)
        return ERROR
    rel.x = ( rel.x div xgridsize)
    if (rel.x >= divx)
        return ERROR

    integer v
    if ( divx!=divy && prelen>=3 )
    {
        v = encode6(rel.x,rel.y,divx,divy)
    }
    else
    {
        v = rel.x * divy + (divy - 1 - rel.y)
    }
```

```
    string result = fast_encode( v, prelen )

    // swap 2nd and 3d letters of mapcodes for 961x961 grids
    if ( prelen==4 && divx==961 && divy==961 )
        result = result[0] + result[2] + result[1] + result[3]

    rel.y = miny + (rel.y * ygridsize)
    rel.x = minx + (rel.x * xgridsize)

    integer dividery = ( (((ygridsize))+yside[postlen]-1) div yside[postlen] )
    integer dividerx = ( (((xgridsize))+xside[postlen]-1) div xside[postlen] )


    // encode relative to rel
    integer  dif.x = coord.x - rel.x
    integer  dif.y = coord.y - rel.y
    integer  extrax = dif.x mod dividerx
    integer  extray = dif.y mod dividery
    dif.x = (dif.x div dividerx)
    dif.y = (dif.y div dividery)
    dif.y = yside[postlen] - 1 - dif.y

    if ( postlen==3 )
    {
        result = result + "." + encode_triple(dif)
    }
    else
    {
        string postfix = fast_encode( dif.x * yside[postlen] + dif.y, postlen )
        if ( postlen==4 )
        {
            postfix = postfix[0] + postfix[2] + postfix[1] + postfix[3]
        }
        result = result + "." + postfix
    }

    if (orglen==1)
    {
        result = result[0] + "." + result[1] + result.substring(3)
    }

    return firstletter + result +
encode_extension(extrax*4,extray,dividerx*4,dividery,extraDigits)
}
```

B 3.3. the encode_nameless algorithm

```
function encode_nameless(coord,m,firstrec,extraDigits)
{
    // copy information about territory rectangle m
    integer minx = minx(m)
    integer miny = miny(m)
    integer maxx = maxx(m)
    integer maxy = maxy(m)
    integer SIDE = smartDiv(m)
    boolean specialshape = isSpecialShape(m)
    integer codex = coDex(m)
    integer codexlen = coDexLen(m)

    // determine index of rectangle and number of rectangles
    integer A = countCityCoordinatesForCountry(m,firstrec)
    integer X = m - firstNamelessRecordForCountry(m,firstrec)
    if (A<2)
        return ERROR

    integer p = (31 div A)
    integer r = (31 mod A)

    // determine storage_offset
    integer storage_offset=0
    if ( codex!=21 && A<=31 )
    {
        storage_offset = (X*p + (X<r ? X : r)) * (31*31*31)
    }
```

```
    else if ( codex!=21 && A<62 )
    {
        if ( X < (62-A) )
        {
            storage_offset = X*(31*31*31*31)
        }
        else
        {
            storage_offset = (62-A + ((X-62+A) div 2) )*(31*31*31*31)
            if ( (X+A) & 1 )
            {
                // X+A is odd
                storage_offset += (16*31*31*31)
            }
        }
    }
    else
    {
        integer basepower = (((codex==21) ? 31*31*31*31 : 31*31*31*31*31) div A)
        if (A==62)
            basepower++
        else
            basepower = 961 * (basepower div 961)

        storage_offset = X * basepower
    }

    // determine core value v
    integer orgSIDE=SIDE
    integer xSIDE=SIDE
    if ( specialshape )
    {
        xSIDE *= SIDE
        SIDE = 1+((maxy-miny) div 90)
        xSIDE = (xSIDE div SIDE)
    }

    integer dividerx4 = xDivider(miny,maxy)
    integer dx = ( (4*(coord.x-minx)) div dividerx4 )
    integer extrax4 = (coord.x-minx)*4 - dx*dividerx4

    integer dividery = 90
    integer dy     = (maxy-coord.y) div dividery
    integer extray = (maxy-coord.y) mod dividery

    integer v = storage_offset
    if ( specialshape )
        v += encode6(dx,SIDE-1-dy,xSIDE,SIDE)
    else
        v+= (dx*SIDE + dy)

    // turn core value into mapcode and insert dot
    string result = fast_encode( v, codexlen+1 )
    if ( codexlen==3 )
    {
        result = result.substr(0,2) + "." + result.substr(2)
    }
    else if ( codexlen==4 )
    {
        if (codex==22 && A<62 && orgSIDE==961 && specialshape==false)
            result = result[0] + result[1] + result[3] + result[2] + result[4]

        if (codex==13)
            result = result.substr(0,2) + "." + result.substr(2)
        else
            result = result.substr(0,3) + "." + result.substr(3)
    }

    // return result with optional extra digits
    return result +
encode_extension(extrax4,extray,dividerx4,dividery,extraDigits)
}
```

B 3.4. the encode_starpipe algorithm

```
function encode_starpipe(coord,m,territory,extraDigits)
{
    integer STORAGE_START=0

    // search back to first record
    integer firstindex = m
    integer codexlen = CoDexLen(m)
    while ( recType(firstindex-1)>=2 && CoDexLen(firstindex-1)==codexlen )
        firstindex--

    for(i=firstindex; CoDexLen(i)==codexlen; i++)
    {
        // copy information about territory rectangle i
        integer minx = minx(i)
        integer miny = miny(i)
        integer maxx = maxx(i)
        integer maxy = maxy(i)

        integer H = ((maxy-miny+89) div 90)
        integer xdiv = xDivider(miny,maxy)
        integer W = ( ( (maxx-minx)*4 + (xdiv-1) ) div xdiv )

        H = 176*( (H+176-1) div 176 )
        W = 168*( (W+168-1) div 168 )

        integer product = (W div 168) * (H div 176) * 31*31*31

        if ( recType()==2 )
        {
            // recType 2 rounds upward to a multiple of 4 or 5 characters
            integer GOODROUNDER = codex>=23 ? (31*31*31*31*31) : (31*31*31*31)
            product = ((STORAGE_START+product+GOODROUNDER-1) div GOODROUNDER) *
                GOODROUNDER - STORAGE_START
        }

        if ( i==m && fitsInside(coord,i) )
        {
            integer dividerx = ((maxx-minx+W-1) div W)
            integer vx      = ((coord.x - minx) div dividerx)
            integer extrax = ((coord.x - minx) mod dividerx)

            integer dividery = ((maxy-miny+H-1) div H)
            integer vy      = ((maxy - coord.y) div dividery)
            integer extray = ((maxy - coord.y) mod dividery)

            coordinate sp;
            sp.x = vx mod 168
            sp.y = vy mod 176

            vx = (vx div 168)
            vy = (vy div 176)

            integer value = (STORAGE_START div (31*31*31)) + (vx*(H div 176) + vy

            return fast_encode( value, codexlen-2 )
                + "."
                + encode_triple(sp)
                + encode_extension(extrax*4,extray,dividerx*4,dividery,extraDigits)

        }
        STORAGE_START += product
    }
    return ERROR
}
```

Appendix C. Territory codes

This appendix lists all the unabbreviated territory codes supported by the mapcode system. See Chapter 1.4.1. about how to copy with abbreviated or missing territory codes.

C 1. Main territories

All 249 codes in the **ISO 3166-1 alpha 3** set are valid as a mapcode **territory code:**

Territory	ISO 3166-1
Aaland Islands	**ALA**
Afghanistan	**AFG**
Albania	**ALB**
Algeria	**DZA**
American Samoa	**ASM**
Andorra	**AND**
Angola	**AGO**
Anguilla	**AIA**
Antarctica	**ATA**
Antigua and Barbuda	**ATG**
Argentina	**ARG**
Armenia	**ARM**
Aruba	**ABW**
Australia	**AUS**
Austria	**AUT**
Azerbaijan	**AZE**
Bahamas	**BHS**
Bahrain	**BHR**
Bangladesh	**BGD**
Barbados	**BRB**
Belarus	**BLR**
Belgium	**BEL**
Belize	**BLZ**
Benin	**BEN**
Bermuda	**BMU**
Bhutan	**BTN**
Bolivia	**BOL**
Bonaire, St Eustasuis and Saba	**BES**
Bosnia and Herzegovina	**BIH**
Botswana	**BWA**
Bouvet Island	**BVT**
Brazil	**BRA**
British Indian Ocean Territory	**IOT**
British Virgin Islands	**VGB**
Brunei	**BRN**
Bulgaria	**BGR**
Burkina Faso	**BFA**
Burundi	**BDI**
Cambodia	**KHM**
Cameroon	**CMR**

Canada	**CAN**
Cape Verde	**CPV**
Cayman islands	**CYM**
Central African Republic	**CAF**
Chad	**TCD**
Chile	**CHL**
China	**CHN**
Christmas Island	**CXR**
Cocos Islands	**CCK**
Colombia	**COL**
Comoros	**COM**
Congo-Brazzaville	**COG**
Congo-Kinshasa	**COD**
Cook islands	**COK**
Costa Rica	**CRI**
Croatia	**HRV**
Cuba	**CUB**
Curacao	**CUW**
Cyprus	**CYP**
Czech Republic	**CZE**
Denmark	**DNK**
Djibouti	**DJI**
Dominica	**DMA**
Dominican Republic	**DOM**
East Timor	**TLS**
Ecuador	**ECU**
Egypt	**EGY**
El Salvador	**SLV**
Equatorial Guinea	**GNQ**
Eritrea	**ERI**
Estonia	**EST**
Ethiopia	**ETH**
Falkland Islands	**FLK**
Faroe Islands	**FRO**
Fiji Islands	**FJI**
Finland	**FIN**
France	**FRA**
French Guiana	**GUF**
French Polynesia	**PYF**
French Southern and Antarctic Lands	**ATF**
Gabon	**GAB**
Gambia	**GMB**
Georgia	**GEO**
Germany	**DEU**
Ghana	**GHA**
Gibraltar	**GIB**
Greece	**GRC**
Greenland	**GRL**
Grenada	**GRD**
Guadeloupe	**GLP**
Guam	**GUM**
Guatemala	**GTM**
Guernsey	**GGY**
Guinea	**GIN**

Guinea-Bissau	**GNB**
Guyana	**GUY**
Haiti	**HTI**
Heard Island and McDonald Islands	**HMD**
Honduras	**HND**
Hong Kong	**HKG**
Hungary	**HUN**
Iceland	**ISL**
India	**IND**
Indonesia	**IDN**
Iran	**IRN**
Iraq	**IRQ**
Ireland	**IRL**
Isle of Man	**IMN**
Israel	**ISR**
Italy	**ITA**
Ivory Coast	**CIV**
Jamaica	**JAM**
Japan	**JPN**
Jersey	**JEY**
Jordan	**JOR**
Kazakhstan	**KAZ**
Kenya	**KEN**
Kiribati	**KIR**
Kuwait	**KWT**
Kyrgyzstan	**KGZ**
Laos	**LAO**
Latvia	**LVA**
Lebanon	**LBN**
Lesotho	**LSO**
Liberia	**LBR**
Libya	**LBY**
Liechtenstein	**LIE**
Lithuania	**LTU**
Luxembourg	**LUX**
Macau	**MAC**
Macedonia	**MKD**
Madagascar	**MDG**
Malawi	**MWI**
Malaysia	**MYS**
Maldives	**MDV**
Mali	**MLI**
Malta	**MLT**
Marshall Islands	**MHL**
Martinique	**MTQ**
Mauritania	**MRT**
Mauritius	**MUS**
Mayotte	**MYT**
Mexico	**MEX**
Micronesia	**FSM**
Moldova	**MDA**
Monaco	**MCO**
Mongolia	**MNG**
Montenegro	**MNE**

Montserrat	**MSR**
Morocco	**MAR**
Mozambique	**MOZ**
Myanmar	**MMR**
Namibia	**NAM**
Nauru	**NRU**
Nepal	**NPL**
Netherlands	**NLD**
New Caledonia	**NCL**
New Zealand	**NZL**
Nicaragua	**NIC**
Niger	**NER**
Nigeria	**NGA**
Niue	**NIU**
Norfolk and Philip Island	**NFK**
North Korea	**PRK**
Northern Mariana Islands	**MNP**
Norway	**NOR**
Oman	**OMN**
Pakistan	**PAK**
Palau	**PLW**
Palestinian territory	**PSE**
Panama	**PAN**
Papua New Guinea	**PNG**
Paraguay	**PRY**
Peru	**PER**
Philippines	**PHL**
Pitcairn Islands	**PCN**
Poland	**POL**
Portugal	**PRT**
Puerto Rico	**PRI**
Qatar	**QAT**
Reunion	**REU**
Romania	**ROU**
Russia	**RUS**
Rwanda	**RWA**
Saint Helena, Ascension and Tristan da Cunha	**SHN**
Saint Kitts and Nevis	**KNA**
Saint Lucia	**LCA**
Saint Pierre and Miquelon	**SPM**
Saint Vincent and the Grenadines	**VCT**
Saint-Barthelemy	**BLM**
Saint-Martin	**MAF**
Samoa	**WSM**
San Marino	**SMR**
Sao Tome and Principe	**STP**
Saudi Arabia	**SAU**
Senegal	**SEN**
Serbia	**SRB**
Seychelles	**SYC**
Sierra Leone	**SLE**
Singapore	**SGP**
Sint Maarten	**SXM**
Slovakia	**SVK**

Slovenia	**SVN**
Solomon Islands	**SLB**
Somalia	**SOM**
South Africa	**ZAF**
South Georgia and the South Sandwich Islands	**SGS**
South Korea	**KOR**
South Sudan	**SSD**
Spain	**ESP**
Sri Lanka	**LKA**
Sudan	**SDN**
Suriname	**SUR**
Svalbard (Spitsbergen) and Jan Mayen	**SJM**
Swaziland	**SWZ**
Sweden	**SWE**
Switzerland	**CHE**
Syria	**SYR**
Taiwan	**TWN**
Tajikistan	**TJK**
Tanzania	**TZA**
Thailand	**THA**
Togo	**TGO**
Tokelau	**TKL**
Tonga	**TON**
Trinidad and Tobago	**TTO**
Tunisia	**TUN**
Turkey	**TUR**
Turkmenistan	**TKM**
Turks and Caicos Islands	**TCA**
Tuvalu	**TUV**
Uganda	**UGA**
Ukraine	**UKR**
United Arab Emirates	**ARE**
United Kingdom	**GBR**
United States Minor Outlying Islands	**UMI**
Uruguay	**URY**
US Virgin Islands	**VIR**
USA	**USA**
Uzbekistan	**UZB**
Vanuatu	**VUT**
Vatican City	**VAT**
Venezuela	**VEN**
Vietnam	**VNM**
Wallis and Futuna	**WLF**
Western Sahara	**ESH**
Yemen	**YEM**
Zambia	**ZMB**
Zimbabwe	**ZWE**

C 2. Subdivisions of territories

In some very large countries, an address has little meaning without knowing the state, province or oblast (just like elsewhere, an address has little meaning without knowing the country). For example, there are 27 cities called Washington in the USA. If you want to refer to a location in the capital city, you would always refer to "Washington DC".

For eight countries (The USA, Canada, Mexico, Brazil, India, Australia, Russia, and China), mapcode supports territory codes for specific subdivisions. Where possible, ISO 3166-2:XX codes are supported as territory codes, which consist of a two-letter country code, a hyphen, and a two- or three-letter state code. For example, the state of Florida in the United States has territory code **US-FL**.

C 2.1. Brazil

For this country, mapcode territory codes for its subdivisions (its states) are based on ISO 3166-2:**BR**

Territory	ISO 3166-2:BR
Acre	**BR-AC**
Alagoas	**BR-AL**
Amapá	**BR-AP**
Amazonas	**BR-AM**
Bahia	**BR-BA**
Ceará	**BR-CE**
Espírito Santo	**BR-ES**
Federal District	**BR-DF**
Goiás	**BR-GO**
Maranhão	**BR-MA**
Mato Grosso	**BR-MT**
Mato Grosso do Sul	**BR-MS**
Minas Gerais	**BR-MG**
Pará	**BR-PA**
Paraíba	**BR-PB**
Paraná	**BR-PR**
Pernambuco	**BR-PE**
Piauí	**BR-PI**
Rio de Janeiro	**BR-RJ**
Rio Grande do Norte	**BR-RN**
Rio Grande do Sul	**BR-RS**
Rondônia	**BR-RO**
Roraima	**BR-RR**
Santa Catarina	**BR-SC**
São Paulo	**BR-SP**
Sergipe	**BR-SE**
Tocantins	**BR-TO**

C 2.2. Canada

For this country, mapcode territory codes for its subdivisions (provinces and territories) are based on ISO 3166-2:**CA**

Territory	ISO 3166-2:CA
Alberta	**CA-AB**
British Columbia	**CA-BC**

Manitoba	**CA-MB**
New Brunswick	**CA-NB**
Newfoundland and Labrador	**CA-NL**
Nova Scotia	**CA-NS**
Ontario	**CA-ON**
Prince Edward Island	**CA-PE**
Quebec	**CA-QC**
Saskatchewan	**CA-SK**
Northwest Territories	**CA-NT**
Nunavut	**CA-NU**
Yukon	**CA-YT**

C 2.3. The United States of America

For this country, mapcode territory codes for its subdivisions (its states, and the Federal District of Columbia) are based on ISO 3166-2:**US**

Territory	**ISO 3166-2:US**
Alaska	**US-AK**
Alabama	**US-AL**
Arkansas	**US-AR**
Arizona	**US-AZ**
Californië	**US-CA**
Colorado	**US-CO**
Connecticut	**US-CT**
Washington D.C.	**US-DC**
Delaware	**US-DE**
Florida	**US-FL**
Georgia	**US-GA**
Hawaï	**US-HI**
Iowa	**US-IA**
Idaho	**US-ID**
Illinois	**US-IL**
Indiana	**US-IN**
Kansas	**US-KS**
Kentucky	**US-KY**
Louisiana	**US-LA**
Massachusetts	**US-MA**
Maryland	**US-MD**
Maine	**US-ME**
Michigan	**US-MI**
Minnesota	**US-MN**
Missouri	**US-MO**
Mississippi	**US-MS**
Montana	**US-MT**
North Carolina	**US-NC**
North Dakota	**US-ND**
Nebraska	**US-NE**
New Hampshire	**US-NH**
New Jersey	**US-NJ**

New Mexico	**US-NM**
Nevada	**US-NV**
New York	**US-NY**
Ohio	**US-OH**
Oklahoma	**US-OK**
Oregon	**US-OR**
Pennsylvania	**US-PA**
Rhode Island	**US-RI**
South Carolina	**US-SC**
South Dakota	**US-SD**
Tennessee	**US-TN**
Texas	**US-TX**
Utah	**US-UT**
Virginia	**US-VA**
Vermont	**US-VT**
Washington	**US-WA**
Wisconsin	**US-WI**
West Virginia	**US-WV**
Wyoming	**US-WY**

The mapcode system also accepts the following ISO 3166-2:US codes as valid territory codes for US oversees territories – although mapcodes are *generated* using their ISO 3166-1 alpha-3 code:

Territory	**ISO 3166-2:US** *Accepted but never generated*	***Normal Code*** *(From ISO 3166-1)*
American Samoa	**US-AS**	***ASM***
Guam	**US-GU**	***GUM***
Northern Mariana Islands	**US-MP**	***MNP***
Puerto Rico	**US-PR**	***PRI***
United States Minor Outlying Islands	**US-UM**	***UMI***
US Virgin Islands	**US-VI**	***VIR***

C 2.4. India

For this country, mapcode territory codes for its subdivisions (its states and unions) are based on ISO 3166-2:**IN**

Territory	**ISO 3166-2:IN**	**Mapcode Alternative** *Accepted but never generated*
Andaman and Nicobar Islands	**IN-AN**	
Andhra Pradesh	**IN-AP**	
Arunachal Pradesh	**IN-AR**	
Assam	**IN-AS**	
Bihar	**IN-BR**	
Chandigarh	**IN-CH**	
Chhattisgarh	**IN-CT**	**IN-CG**

Dadra and Nagar Haveli	**IN-DN**	
Daman and Diu	**IN-DD**	
Delhi	**IN-DL**	
Goa	**IN-GA**	
Gujarat	**IN-GJ**	
Haryana	**IN-HR**	
Himachal Pradesh	**IN-HP**	
Jammu and Kashmir	**IN-JK**	
Jharkhand	**IN-JH**	
Karnataka	**IN-KA**	
Kerala	**IN-KL**	
Lakshadweep	**IN-LD**	
Madhya Pradesh	**IN-MP**	
Maharashtra	**IN-MH**	
Manipur	**IN-MN**	
Meghalaya	**IN-ML**	
Mizoram	**IN-MZ**	
Nagaland	**IN-NL**	
Odisha (formerly known as Orissa)	**IN-OR**	**IN-OD**
Puducherry (Pondicherry)	**IN-PY**	
Punjab	**IN-PB**	
Rajasthan	**IN-RJ**	
Sikkim	**IN-SK**	
Tamil Nadu	**IN-TN**	
Telangana	**IN-TG**	
Tripura	**IN-TR**	
Uttarakhand	**IN-UT**	**IN-UK**
Uttar Pradesh	**IN-UP**	
West Bengal	**IN-WB**	

Three non-standard mapcode alternatives are accepted to cope with widely-used abbreviations (e.g. for vehicle registration).

C 2.5. China

For this country, mapcode territory codes for its subdivisions (provinces, municipalities, autonomous regions and special administrative regions) are based on **ISO 3166-2:CN**. For three of those (Taiwan, Hong Kong and Macao), an ISO 3166 3-letter territory code is also available. Since the **ISO 3166-2:CN** codes are numerical, the mapcode system *also* supports the **PRC GB/T 2260** 2-letter codes as alternative territory codes. This is a Chinese national standard, issued by the Standardization Administration of China (SAC), the Chinese National Committee of the ISO and IEC.

Territory	**ISO 3166-2:CN** *Accepted but never generated*	**PRC GB/T 2260**	*ISO 3166-1 equivalent*
Beijing	**CN-11**	**CN-BJ**	
Tianjin	**CN-12**	**CN-TJ**	
Hebei	**CN-13**	**CN-HE**	
Shanxi	**CN-14**	**CN-SX**	
Nei Mongol (mn)	**CN-15**	**CN-NM**	

Liaoning	CN-21	**CN-LN**	
Jilin	CN-22	**CN-JL**	
Heilongjiang	CN-23	**CN-HL**	
Shanghai	CN-31	**CN-SH**	
Jiangsu	CN-32	**CN-JS**	
Zhejiang	CN-33	**CN-ZJ**	
Anhui	CN-34	**CN-AH**	
Fujian	CN-35	**CN-FJ**	
Jiangxi	CN-36	**CN-JX**	
Shandong	CN-37	**CN-SD**	
Henan	CN-41	**CN-HA**	
Hubei	CN-42	**CN-HB**	
Hunan	CN-43	**CN-HN**	
Guangdong	CN-44	**CN-GD**	
Guangxi	CN-45	**CN-GX**	
Hainan	CN-46	**CN-HI**	
Chongqing	CN-50	**CN-CQ**	
Sichuan	CN-51	**CN-SC**	
Guizhou	CN-52	**CN-GZ**	
Yunnan	CN-53	**CN-YN**	
Xizang	CN-54	**CN-XZ**	
Shaanxi	CN-61	**CN-SN**	
Gansu	CN-62	**CN-GS**	
Qinghai	CN-63	**CN-QH**	
Ningxia	CN-64	**CN-NX**	
Xinjiang	CN-65	**CN-XJ**	
Taiwan	CN-71	CN-TW	**TWN**
Hong Kong (Xianggang)	CN-91	CN-HK	**HKG**
Macao (Aomen)	CN-92	CN-MC	**MAC**

C 2.6. Australia

For this country, mapcode territory codes for its subdivisions (states and union territories) are based on ISO 3166-2:**AU**

Territory	**ISO 3166-2:AU**
New South Wales	**AU-NSW**
Queensland	**AU-QLD**
South Australia	**AU-SA**
Tasmania	**AU-TAS**
Victoria	**AU-VIC**
Western Australia	**AU-WA**
Australian Capital Territory	**AU-ACT**
Northern Territory	**AU-NT**

There is no ISO 3166 code for the Jarvis Bay Territory, but mapcode defines its own code, **AU-JBT**:

Territory	**ISO 3166-2:AU**	**Mapcode Alternative**

Jervis Bay Territory	*none*	**AU-JBT**

The following external territories of Australia already have their own 3-letter (ISO 3166-1 alpha 3) "country" code (see *"The main territories of the world"*). Since they do *not* have a ISO 3166-2:AU code, but *do* have two-letter ISO 3166 country codes, mapcode accepts those as valid subdivision codes.

Territory	Normal code *(from ISO 3166-1)*	Mapcode Alternative *Accepted but never generated*
Christmas Island	***CXR***	AU-CX
Cocos (Keening) Island	***CCK***	AU-CC
Heard Island and McDonalds Islands	***HMD***	AU-HM
Norfolk Island	***NFK***	AU-NF

Note:

- Ashmore Reef and Cartier Island are included in **AU-WA** , Western Australia
- Coral Sea Islands is included in **AU-QLD**, Queensland, Australia
- Macquarie Island is included in **AUS**, i.e. Australia as whole

C 2.7. Mexico

For this country, mapcode territory codes for its subdivisions (states and federal district) are based on ISO 3166-2:**MX**. Mapcode also accepts self-defined 2-letter alternative codes for the subdivisions as well.

Territory	ISO 3166-2:MX	Mapcode alternative
Aguascalientes	**MX-AGU**	MX-AG
Baja California	**MX-BCN**	MX-BC
Baja California Sur	**MX-BCS**	MX-BS
Chiapas	**MX-CHP**	MX-CH
Chihuahua	**MX-CHH**	MX-CS
Campeche	**MX-CAM**	MX-CM
Coahuila	**MX-COA**	MX-CO
Colima	MX-COL	**MX-OL**
Distrito Federal	**MX-DIF**	MX-DF
Durango	**MX-DUR**	MX-DG
Guanajuato	**MX-GUA**	MX-GR
Guerrero	**MX-GRO**	MX-GT
Hidalgo	**MX-HID**	MX-HG
Jalisco	**MX-JAL**	MX-JA
Mexico (Federal District)	MX-MEX	**MX-MX** MX-ME
Michoacán	**MX-MIC**	MX-MI
Morelos	**MX-MOR**	MX-MO
Nayarit	**MX-NAY**	MX-NA
Nuevo León	**MX-NLE**	MX-NL
Oaxaca	**MX-OAX**	MX-OA
Puebla	**MX-PUE**	MX-PB

Querétaro	**MX-QUE**	MX-QE
Quintana Roo	**MX-ROO**	MX-QR
San Luis Potosí	**MX-SLP**	MX-SI
Sinaloa	**MX-SIN**	MX-SL
Sonora	**MX-SON**	MX-SO
Tabasco	**MX-TAB**	MX-TB
Tamaulipas	**MX-TAM**	MX-TM
Tlaxcala	**MX-TLA**	MX-TL
Veracruz	**MX-VER**	MX-VE
Yucatán	**MX-YUC**	MX-YU
Zacatecas	**MX-ZAC**	MX-ZA

Note: the 3-letter subdivision code MEX conflicts with the country code for Mexico as a whole. The subdivision code COL conflicts with 3-letter country codes for Columbia. See "Duplicate codes" for more about such conflicts.

C 2.8. Russia

For this country, mapcode territory codes for its subdivisions (identifies republics, territories, regions, districts and autonomous cities) are based on ISO 3166-2:**RU**. The republics have 2-letter codes, the rest has 3-letter codes. Mapcode defines a few 2-letter alternatives for those codes that precisely match 3-letter **country** codes, such as **BEL** (Belgium). See "Duplicate codes" for more about this.

Territory	ISO 3166-2:RU	Mapcode alternative
Adygeya, Respublika	**RU-AD**	
Altay, Respublika	**RU-AL**	
Bashkortostan, Respublika	**RU-BA**	
Buryatiya, Respublika	**RU-BU**	
Chechenskaya Respublika	**RU-CE**	
Chuvashskaya Respublika	**RU-CU**	
Dagestan, Respublika	**RU-DA**	
Ingushetiya, Respublika	**RU-IN**	
Kabardino-Balkarskaya Respublika	**RU-KB**	
Kalmykiya, Respublika	**RU-KL**	
Karachayevo-Cherkesskaya Respubl.	**RU-KC**	
Kareliya, Respublika	**RU-KR**	
Khakasiya, Respublika	**RU-KK**	
Komi, Respublika	**RU-KO**	
Mariy El, Respublika	**RU-ME**	
Mordoviya, Respublika	**RU-MO**	
Sakha, Respublika	**RU-SA**	
Severnaya Osetiya-Alaniya, Respubl.	**RU-SE**	
Tatarstan, Respublika	**RU-TA**	
Tyva, Respublika	**RU-TY**	
Udmurtskaya Respublika	**RU-UD**	
Altayskiy kray	**RU-ALT**	
Kamchatskiy kray	**RU-KAM**	
Khabarovskiy kray	**RU-KHA**	
Krasnodarskiy kray	**RU-KDA**	

Krasnoyarskiy kray	**RU-KYA**	
Permskiy kray	RU-PER	**RU-PM**
Primorskiy kray	RU-PRI	**RU-PO**
Stavropol'skiy kray	**RU-STA**	
Zabaykal'skiy kray	**RU-ZAB**	
Amurskaya oblast'	**RU-AMU**	
Arkhangel'skaya oblast'	**RU-ARK**	
Astrakhanskaya oblast'	**RU-AST**	
Belgorodskaya oblast'	RU-BEL	**RU-BE**
Bryanskaya oblast'	**RU-BRY**	
Chelyabinskaya oblast'	RU-CHE	**RU-CH**
Irkutskaya oblast'	**RU-IRK**	
Ivanovskaya oblast'	**RU-IVA**	
Kaliningradskaya oblast'	**RU-KGD**	
Kaluzhskaya oblast'	**RU-KLU**	
Kemerovskaya oblast'	**RU-KEM**	
Kirovskaya oblast'	RU-KIR	**RU-KI**
Kostromskaya oblast'	**RU-KOS**	
Kurganskaya oblast'	**RU-KGN**	
Kurskaya oblast'	**RU-KRS**	
Leningradskaya oblast'	**RU-LEN**	
Lipetskaya oblast'	**RU-LIP**	
Magadanskaya oblast'	**RU-MAG**	
Moskovskaya oblast'	**RU-MOS**	
Murmanskaya oblast'	**RU-MUR**	
Nizhegorodskaya oblast'	**RU-NIZ**	
Novgorodskaya oblast'	**RU-NGR**	
Novosibirskaya oblast'	**RU-NVS**	
Omskaya oblast'	**RU-OMS**	
Orenburgskaya oblast'	**RU-ORE**	
Orlovskaya oblast'	**RU-ORL**	
Penzenskaya oblast'	**RU-PNZ**	
Pskovskaya oblast'	**RU-PSK**	
Rostovskaya oblast'	**RU-ROS**	
Ryazanskaya oblast'	**RU-RYA**	
Sakhalinskaya oblast'	**RU-SAK**	
Samarskaya oblast'	**RU-SAM**	
Saratovskaya oblast'	**RU-SAR**	
Smolenskaya oblast'	**RU-SMO**	
Sverdlovskaya oblast'	**RU-SVE**	
Tambovskaya oblast'	RU-TAM	**RU-TT**
Tomskaya oblast'	**RU-TOM**	
Tul'skaya oblast'	**RU-TUL**	
Tverskaya oblast'	**RU-TVE**	
Tyumenskaya oblast'	**RU-TYU**	
Ul'yanovskaya oblast'	**RU-ULY**	
Vladimirskaya oblast'	**RU-VLA**	
Volgogradskaya oblast'	**RU-VGG**	
Vologodskaya oblast'	**RU-VLG**	
Voronezhskaya oblast'	**RU-VOR**	
Yaroslavskaya oblast'	**RU-YAR**	
Moskva (autonomous city)	**RU-MOW**	

Sankt-Peterburg (autonomous city)	**RU-SPE**	
Yevreyskaya **avtonomnaya oblast'**	**RU-YEV**	
Chukotskiy *avtonomnyy okrug*	**RU-CHU**	
Khanty-Mansiyskiy *avtonomnyy okrug*-Yugra	**RU-KHM**	**RY-KM**
Nenetskiy avtonomnyy okrug	**RU-NEN**	
Yamalo-Nenetskiy avtonomnyy okrug	**RU-YAN**	

C 2.9. Subdivisions of other countries

It should be noted that many countries, not just the eight mentioned above, have ISO 3166-2:XX codes. Even a small country like Belgium has ISO 3166-2:BE codes for 10 provinces. However, only the eight countries listed above were deemed to *merit* from their subdivision into states, provinces, regions etc. *as far as their mapcodes are concerned* (delivering shorter proper mapcodes at the cost of explicitly mentioning their territory code).

What deserves special mention is that six countries have “dependent overseas territories” that have *their own country code*. For China and the USA, the dependent territories also have subdivision codes, just like the states and provinces. For example, American Samoa has the US subdivision code **US-AS** as well as its own country code **ASM**. But the dependent territories of the other four countries (Finland, The Netherlands, France and Norway) can only be identified through their 3-letter **country** codes. For completeness’ sake, they are listed here:

Subdivision codes included in ISO 3166-1 alpha-2, but NOT valid as mapcode territory code	*ISO 3166-1 alpha 3 equivalent (valid in mapcodes)*
FI-01 Åland	*ALA*
FR-BL Saint Barthélemy	*NLM*
FR-GF French Guiana	*GUF*
FR-GP Guadeloupe	*GLP*
FR-MF Saint Martin	*MAF*
FR-MQ Martinique	*MTQ*
FR-NC New Caledonia	*NCL*
FR-PF French Polynesia	*PYF*
FR-PM Saint Pierre and Miquelon	*SPM*
FR-RE Réunion	*REU*
FR-TF French Southern Territories	*ATF*
FR-WF Wallis and Futuna	*WLF*
FR-YT Mayotte	*MYT*
NL-AW Aruba	*ABW*
NL-BQ1 Bonaire	*BES*
NL-BQ2 Saba	*BES*
NL-BQ3 Sint Eustatius	*BES*
NL-CW Curaçao	*CUW*
NL-SX Sint Maarten	*SXM*

NO-21 Svalbard	*SJM*
NO-22 Jan Mayen	*SJM*

Also see "*Legacy or reserved 3-letter codes*" about similar issues.

C 3. Special cases

C 3.1. The "international" territory

To cover the world as a whole, the special territory code AAA was introduced:

Territory	ISO 3166-1
World	**AAA**

Mapcodes never need to include this territory code explicitly, since there is no other territory context it can ever be confused with. World mapcodes are always 9 characters, and no other mapcode is ever 9 characters.

C 3.2. Two-letter country codes

All countries have a three-letter territory code (see Appendix C 1), but eight countries have territory codes for subdivisions that use a *two*-letter country codes in combination with a subdivision code. To make the use of mapcode easier for people, systems should be implemented such that

1. 3-letter country codes are allowed instead of 2-letter codes when specifying a state; for example, **USA-FL** is a valid alternative for **US-FL**.
2. In four cases, the two-letter country code is unambiguous in the context of the mapcode system, and should be allowed as a valid alternative for the official 3-letter code:

Official code	Mapcode alternative
USA	**US**
AUS	**AU**
RUS	**RU**
CHN	**CN**

This is not possible for the other four countries that have state codes:

MEX	***MX*** *would conflict with MX-MX*
CAN	***CA*** *would conflict with US-CA*
BRA	***BR*** *would conflict with IN-BR*
IND	***IN*** *would conflict with US-IN*

although it may still be possible to disambiguate based on the situation (see Chapter 1.4.1 for more information

C 3.3. Legacy or reserved 3-letter codes

The following 3-letter "legacy" or "reserved" ISO 3166 codes are accepted by mapcode (as *aliases*, i.e. TAA and ASC are *interpreted* as SHN):

Territory	**ISO 3166 exceptional reservation** *Accepted but never generated*	**ISO 3166 Legacy** *Accepted but never generated*	**Normal code** *(from ISO 3166-1)*
Tristan da Cunha (part of SHN)	**TAA**		***SHN***
Ascension (part of SHN)	**ASC**		***SHN***
Diego Garcia (part of IOT)	**DGA**		***IOT***
Wake Island (part of MHL)		**WAK**	***MHL***
Johnston Atoll (part of UMI)		**JTN**	***UMI***
Midway (part of Hawaii US-HI)		**MID**	***US-HI***

Clipperton Island also has a 3-letter *reserved* ISO 3166 code ("**CPT**") available, but unlike the above territories it has no *existing* country code. Mapcode therefore *defines* CPT (making it the only way to refer to Clipperton Island):

Territory	**ISO 3166 exceptional reservation**
Clipperton Island (part of France)	**CPT**