Conversion Style manual

The mkgmap team

Conversion Style manual

The mkgmap team

Publication date 19 February 2013

Table of Contents

List of Tables

List of Examples

Chapter 1. Introduction

This manual explains how to write a mkgmap style to convert between OSM tags and features on a Garmin GPS device.

A style is used to choose which OSM map features appear in the Garmin map and which Garmin symbols are used.

There are a few styles built into mkgmap, but as there are many different purposes a map may be used for, the default styles in mkgmap will not be ideal for everyone, so you can create and use styles external to mkgmap.

The term *style* could mean the actual way that the features appear on a GPS device, the colour, thickness of the line and so on. This manual does not cover such issues, and if that is what you are looking for, then you need the documentation for **TYP files**.

Few people will want to write their own style from scratch, most people will use the built in conversion style, or at most make a few changes to the default style to add or remove a small number of features. For general information about running and using mkgmap see the **Tutorial document**.

To be clear this is only needed for converting OSM tags, if you are starting with a Polish format file, there is no style involved as the garmin types are already fully specified in the input file.

For general information about the OpenStreetMap project see the OpenStreetMap wiki [http://wiki.openstreetmap.org].

Chapter 2. The structure of a style

A style consists of a number of files in a single directory. The best way is to start out with an existing style that is close to what you want and then work from there.

A style can be packed into a single file using the standard zip utilities that are available on every operating system, or it can be written as one large text file using the single file style format. These alternatives are explained in making a style package.

2.1. Files

These files are read in the order that they are listed here. In general, files that are read first take priority over files read later. The only one of these files that is actually required is the `version` file, as that is used to recognise the style. At least one of the `points`, `lines` or `polygons` files must be present or else the resulting maps will be empty.

2.1.1. Top level folder

Choose a short name for your style, it should be one word or a couple of words joined by an underscore or hyphen. This is how people will refer to the style when it is finished. Create a directory or folder with that name. Then you must create one or more files in this directory as detailed below. Only the `version` file is required.

2.2. The version file

This file *must* exist as it is used to recognise a valid style. It contains the version number of the style language itself, (not the version number of your style, which you can specify in the `info` file if you so wish). The current version number of the style language is 1. Make sure that there is a new line after the number, place an empty line afterwards to be sure.

2.3. The info file

This file contains information about your style. It is all optional information, and there is only really any point adding this information if you are going to distribute your style, or you have more than one style that you maintain.

The file consists of key=value pairs in the same syntax as the command line option file. To summarise you can use either an equal sign = or a colon *:* to separate the key from the value. You can also surround the value with curly braces *{ }* and this allows you to write the value over several lines.

version	The version number of your style.
summary	A short description of your style in one line.
description	A longer description of your style.
base-style	Do not use anymore. This was used to base a style on another one. However, it is bug prone and behaves in a way that is not intuitive without a good understanding of how things work. The preferred way to do this is to use the include mechanism.

Example 2.1. An example info file

Here is an example based on the `info` file from the default style. You can see it uses both equal and colon as separators, normally you would just pick one and use it consistently, but it doesn't make any difference which one you use. The description is written over several lines surrounded in curly brackets. Lines beginning with a hash symbol # are comments and are ignored.

```
#
# This file contains information about the style.
#

summary: The default style

version=1.0

description {
The default style.  This is a heavyweight style that is
designed for use when mapping and especially in lightly covered
areas.
}
```

2.4. The options file

This file contains a number of options that should be set for this style as if they were set on the command line. Only command line options that affect the style will have any effect. The current list is `name-tag-list`, `levels` and `extra-used-tags`.

It is advisable to set up the levels that you want, as the default is not suitable for all kinds of maps and may change in the future. Ideally, you should set the same levels as are used in your style files. For example, if your style files use levels 12,16,20,22,23,24 then it's a good idea to make sure your options style file declares these levels explicitly.

Example 2.2. An example options file

```
name-tag-list = name:en, int_name, name
levels = 0:24, 1:22, 2:20, 3:18, 4:16
extra-used-tags=
```

2.4.1. Non command line options

Most of the options are the same as the command line option of the same name and so you should see its description in the option help. There are however some options that can only be set in this file (just the currently).

extra-used-tags
: A list of tags used by the style. You do not normally need to set this, as mkgmap can work out which tags are used by a style automatically in most cases. It exists only to work around cases where this doesn't work properly.

2.5. The points file

This files contains a set of rules for converting OSM nodes to Garmin POIs (restaurants, bars, ATMs etc). It can also contain rules for some kind of OSM nodes that may affect routing behavior, for example barriers, traffic_calming, traffic_signals, etc.

If this file is not present or empty then there will be no POI's in the final map.

The syntax of the file is described in the style rules section. Like all other files, a hash symbol # introduces a comment.

2.6. The lines file

This file contains a set of rules for converting OSM ways to Garmin lines (roads, rivers, barriers, etc). The syntax of the file is described in the style rules section.

2.7. The polygons file

This file contains a set of rules for converting polygons to Garmin areas (fields, buildings, residential areas, etc). The syntax of the file is described in the style rules section.

2.8. The relations file

This file contains a set of rules to convert OSM relations. Unlike the `points`, `lines` and `polygons` files this file does not lead directly to a Garmin object. Instead it is used to modify the ways or nodes that are contained in the relation.

So for example, if the relation represents a route, then you might add one or more tags to all the ways that make up the route so that they can be processed in the `lines` file specially.

The syntax of the file is also described in the style rules section, but the rules can only have an action part, they must not have a type description part.

Chapter 3. Style rules

Rules allow you to take a map feature in the OSM format, which uses a set of tags to describe the feature into the format required by Garmin maps, where features are identified by a number.

The rules for converting points, lines and polygons are held in correspondingly named files, as described in the structure of a style.

Each file contains a number of rules where you test the values of the tags of an OSM node or way and select a specific Garmin type based on the result of those tests.

3.1. Introduction

Each rule starts off with an expression to test the value of one or more tags.

A rule is made up of two or three parts. The three possible parts are:

- The first part is **required**: this is a set of tests that are performed on the tags of the item to be converted.
- The second part is the action block that can be used to do things with the tags of objects that match the tests and is contained in curly brackets `{...}`.
- The third part is the element type definition and sets the Garmin type and sometimes other parameters that will be used if the tests match. This part is contained in square brackets `[...]`.

As a general point, space and newlines don't matter. There is no need to have rules all on the same line (although most of the examples here are shown in one line), and you can spread them out over several lines and add extra spaces wherever you like if it helps to make them easier to read. Here is an example of a rule containing all three sections:

```
natural=cliff { name '${name} cliff' | 'cliff' } [0x10501 resolution 22]
```

- The tests section is `natural=cliff`
- The action block is `{ name '${name} cliff' | 'cliff' }`
- The element type definition is `[0x10501 resolution 22]`

3.2. Tag tests

The most common test is that a particular OSM tag has a given value. So for example if we have

```
highway=motorway
```

This means that we look up the highway tag in the OSM input file and if it exists and has the value *motorway* then this test has matched.

You can also compare numeric quantities:

```
population > 10000
maxspeed >= 30
population < 10000000
```

Respectively, these mean: a population greater than ten thousand, a max speed greater than or equal to 30 and a population less than one million. You will be able to compare quantities that have units too, for example

```
max_speed > 30mph
```

If a different unit is given in the tag (say km/h) then it will be removed before comparison. Conversion of units is not implemented at the time of writing.

You may also use regular expressions:

```
ele ~ '\d*00'
```

This checks whether ele is a multiple of 100.

3.2.1. Allowed operations

The following table describes the operations that may be used.

Table 3.1. Full list of operations

Operation	description and examples
tag=value	This matches when a tag has the given value.
tag!=value	The tag exists and does not have the given value.
tag=*	Matches when the tag exists, regardless of its value.
tag!=*	Matches when the tag does *not* exist.
tag < value	Matches when the tag when converted as a number is less than the given value. If the value is not numeric then this is always false. A unit is removed before comparison however so `max_speed < 100` will work if `max_speed` is `80kmh` for example.
tag <= value, tag > value, tag >= value	As above, for less than or equal, greater than and greater than or equal.
tag ~ REGEX	This is true when the value of the tag matches the given regular expression. The Java regular expression [:http://docs.oracle.com/javase/1.4.2/docs/api/java/util/regex/Pattern.html] syntax is recognised. For example `name ~ '.*[Ll]ane'` would match every name that ended in *Lane* or *lane*.
! (expr)	The *not* operator (*!*) reverses the truth of the expression following. That expression must be in brackets.

3.2.2. Combining tag tests

Although it is possible to convert many OSM nodes and ways just using one tag, it is also often necessary to use more than one.

For example, say you want to take roads that are tagged both as `highway=unclassified` and `maxspeed>60` differently to roads that are just `highway=unclassified`. In this type of case, you might create two separate rules as follows:

```
highway=unclassified & maxspeed>60 [0x06]
highway=unclassified [0x05]
```

This means that roads that are unclassified and have a maxspeed of greater than 60 would use Garmin element type 0x06, whereas unclassified roads without a maxspeed tag, or where it is less than 60 would use type 0x05.

It is important to note that the order of the rules is important here. The rules are matched in the order that they occur in the style file and mkgmap stops trying to apply them after the first one that matches. If you had the rules above in the reverse order, then the `highway=unclassified` rule would match first to any OSM way with that tag/key pair, and the second rule would never get applied. Therefore, in general you want the most specific rules first and simpler, more general rules later on to catch the cases that are not caught by the more complex rules.

You can also combine alternatives into the one rule using a logical or, represented with a pipe (|) symbol. For example

```
highway=footway | highway=path [0x07]
```

This means if the road has either the **highway=footway** tag or the **highway=path** tags (or both), then the condition matches and mkgmap would use type 0x07 for the map. This works exactly the same as if you had written two separate rules - one for footway and one for path - and indeed is converted to two separate rules internally when mkgmap runs.

You are not limited to two tests for a given rule… you can combine and group tests in almost whatever way you like. So for a slightly forced example the following would be possible:

```
place=town & (population > 1000000 | capital=true) | place=city
```

This would match if there was a `place` tag which had the value `town` and either the population was over a million or it was tagged a capital, or there was a `place` tag with the value `city`.

There used to be some restrictions on the kind of expression you could use. Now the only restriction is you must have at least one test that depends on a tag existing. So you cannot match on everything, regardless of tags, or test for an object that does *not* have a tag.

3.2.3. Functions

Functions calculate a specific property of an OSM element.

Table 3.2. Style functions

Function	Node	Way	Relation	Description
length()		x	x	Calculates the length in m. For relations its the sum of all member length (including sub relations).
is_complete()		x		`true` if all nodes of a way are contained in the tile. `false` if some nodes of the way are missing in the tile.
is_closed()		x		`true` the way is closed. `false` the way is not closed and cannot be processed as polygon.

The following rule matches for all service ways longer than 50m.

```
highway=service & length()>50
```

3.3. Action block

An action block is enclosed in braces `{ … }` and contains one or more statements that can alter the element being displayed; multiple statements are separated by ‘;’ symbol. When there is an action block, the element type definition is optional, but if used it must come after the action block.

A list of all the command that can be used in the action block follows. In the examples you will see notation of the form `${name}`, this is how tag values can be substituted into strings, in a similar way to many computer languages. For full details see the section on variable substitution.

3.3.1. name

This sets the final name of the element, that is, the name that will be used in the Garmin map. It is distinct from any *name* tag on the element. You can give a list of alternatives separated by | pipe symbols. The first alternative that matches will be used. Once the name is set it cannot be overridden, so if more than one *name* command matches then only the first to set the name will take effect.

Example 3.1. Setting the name

```
{name '${name} (${ref})' | '${ref}' | '${name}'}
```

If both the `name` and `ref` tags are are set, then the first alternative would be completed and the resulting name might be *Main St (A1)*. If just `name` was set, then the first two alternatives can not be fully and so the final name might in that case be *Main St*.

For highway shields, you can use the notation `${tagname|highway-symbol:box}`. Valid symbols are `interstate`, `shield`, `round`, `hbox`, `box` and `oval`. The appropriate kind of highway shield will be added to the value of `tagname`. The exact result of the way it looks is dependant on where you view the map.

3.3.2. add

The add command adds a tag if it does not already exist. This is often used if you want to set the value of a tag as a default but do not want to overwrite any existing tag.

For example, motorways are one way by default so we need to add the `oneway=yes` tag in the style so that is treated as one way by the device. But there are some stretches of motorway that are one-way and these will be tagged as `oneway=no`. If we used `set` then that tagging would be lost, so we use `add`.

```
 highway=motorway { add oneway=yes }
```

The other use is in in relations with the *apply* command.

All the same you can set any tag you want, it might be useful so you can match on it elsewhere in the rules.

You can also use substitutions.

```
{set name='${ele}'; set name='${ref}';}
```

These two commands would set the *name* tag to the value of the *ele* tag if it exists, or to the value of the *ref* tag if that exists.

You can also give a list of alternative expressions separated with a vertical bar in the same way as on the name command. The first one that is fully defined will be used.

```
{set key123 = '${name:en}' | '${name}'; }
```

Will set *key123* to the value of the *name:en* tag if it exists and to the *name* tag if not.

3.3.3. set

The *set* command is just like the *add* command, except that it sets the tag, replacing any existing value it had.

3.3.4. delete

The delete command deletes a tag.

```
{ delete key123 }
```

3.3.5. apply

The "apply" action only makes sense in relations. Say you have a relation marking a bus route, but none of the ways that are in the relation have any special tags to indicate that they form part of that bus route, and you want to be able to tell from looking at the map which buses go where. You can write a rule in the **relations file** such as:

```
type=route & route=bus {
        apply {
                set route=bus;
                set route_ref='${route_ref}';
        }
}
```

Then in the **lines file** you will need to write a rule to match *route=bus*. All the relation rules are run before any others so that this works.

The substitution `${route_ref}` takes the value of the tag on the **relation** and applies it to each of the ways in the relation.

The substitution `$(route_ref)` (with parenthesis, rather than curly brackets) can be used for accessing the value of the tag on the actually processed **member** of the relation, e.g.

```
type=route & route=bus {
        apply {
                set route=bus;
                set name='$(name) ${route_ref}';
        }
}
```

3.3.6. apply_once

The apply_once action is like `apply`, but it will apply the action once per relation member. A round-trip route relation may include the same ways multiple times, unless all member ways have been defined as parallel one way streets.

3.4. Variables

You can substitute the value of tags within strings in an action. A dollar sign (`$`) introduces the substitution followed by the tag name surrounded by curly braces like so `${name}`.

The most obvious use for variables is in setting the name of the element. You are able to use any combination of tags to make the name from. Here we name a fuel station by its brand and the name in brackets following.

```
amenity=fuel { name '${brand} (${operator})' } [ 0x2f01 ]
```

If the operator tag was not set, then the name would not be set because **all** substitutions in a string must exist for the result to be valid. This is why the "name" command takes a list of possibilities, if operator was simply replaced with a blank, then you would have an empty pair of brackets. So you would fix the previous rule by adding another name option.

```
amenity=fuel
   { name '${brand} (${operator})' | '${brand}' }
```

```
    [ 0x2f01 ]
```

If only the brand tag exists, then the first option will be skipped and the second will be used.

3.4.1. Variable filters

The value of a variable can be modified by ‘filters’. The value of the tag can be transformed in various ways before being substituted.

A filter is added by adding a vertical bar symbol "|" after the tag name, followed by the filter name, then a colon ":" and an argument. If there is more than one argument required then they are usually separated by colons too, but that is not a rule.

```
${tagname|filter:arg1:arg2}
```

You can apply as many filter expressions to a substitution as you like.

```
${tagname|filter1:arg|filter2:arg}
```

Table 3.3. List of all substitution filters

Name	Arguments	Description
def	`default`	If the variable is not set, then use the argument as a default value. This means that the variable will never be ‘unset’ in places where that matters. `${oneway|def:no}`
conv	`factor`	Use for conversions between units. The only supported version is from meters to feet number. It is multiplied by the argument. `${height|conv:m=>ft}`
subst	`from=>to`	Substitutes all occurrences of the string `from` with the string `to` in the value of the tag. There isn’t a large number of uses for this, perhaps you can use it to correct mistakes. The `to` can be empty to remove the `from` string altogether and this is probably the most popular use. `${name|ref:A=>}`
highway-symbol	`symbol max-num max-alpha`	Prepares the value as a highway reference such as "A21" "I-80" and so on. A code is added to the front of the string so that a highway shield is displayed, spaces are removed and the text is truncated so as not to overflow the symbol. `${ref|highway-symbol:box:4:8}` See below for a list of the `highway-symbol` values. The first number is the maximum number of characters to allow for references that contain numbers and letters. The second is the maximum length of references that do not contain numbers. If there is just the one number then it is used in both cases.
height	`m=>ft`	This is the same as the `conv` filter, except that it prepends a special separation character before the value which is

Name	Arguments	Description
		intended for elevations. As with `conv` the only supported conversion currently is from meters to feet. `${ele|m=>ft}`

not-equal	`tag`	Used to check for duplicate tags. If the value of this tag is equal to the value of the tag named as the argument to `not-equal`, then value of this tag is set to undefined.
		`place=* {`
		`  name '${name} (${int_name|not-equal:name})'`
		`  | '${name}'`
		`  }`
		In that example, if the international name is different to the name then it will be placed in parenthesis after the name. Otherwise there will just be the name as given in the "name" tag.

substring	`start:end`	Extract part of the string. The start and end positions are counted starting from zero and the end position is not included. `${name|2:5}` If the "name" was "Dorset Lane", then the result is "rse". If there is just the one number, then the substring starts from that character until the end of the string.

3.4.2. Symbol codes

Here is a list of all the symbols that can be created with images to give an idea of where they should be used. The actual symbol will depend on the device that it is displayed on.

Table 3.4. Highway symbol codes

Shield name	Symbol	Description
interstate		US Interstate, digits only
shield		US Highway shield, digits only
round		US Highway round, digits only
hbox		Box for major roads
box		Box for medium roads
oval		Box for smaller roads

3.5. Useful tags for routing and address search

For general routing, using avoid options, and address search, the use of some special tags is necessary.

3.6. Element type definition

As noted above this is contained in square brackets and if used must be the **last part of the rule**.

The first and only mandatory part of this section is the Garmin type code which must always be written in hexadecimal. Following this the element type definition rule can contain a number of optional keywords and values.

3.6.1. level

This is the highest zoom level that this element should appear at (like EndLevel in the mp format). The lower the level the detailed the view. The most detailed, most zoomed in, level is level 0. A map will usually have between three and five levels. If the level for an object is not given then it defaults to 0 and so the specified feature will only appear at the most detailed level.

In the following example, we set highways to appear from zoom level 4 down to zoom level 0:

```
highway=motorway [0x01 level 4]
```

You can use `level` to place elements into the layers of the map that you want but you can't force the device to actually display them.

Some pieces of software (such as QLandkarteGT, I believe) will honour your selections, but actual GPS devices have their own ideas about which POI's can be shown at which resolutions.

Level ranges. You can also give a range (e.g. 1-3) and the map will then contain the object only between the specified levels.

```
highway=motorway [0x01 level 3-5]
```

In this example, motorways will appear at zoom level 5, which is most zoomed out, and continue to be visible until zoom level 3, which is moderately zoomed in, and then will not be shown in zoom levels 2, 1 and 0 (most zoomed-in).

Of course you are unlikely to want a feature to disappear as you zoom in, but this can be used for interesting effects where a different representation takes over at the lower zoom levels. For example a building may be a point at high levels and then become a polygon at lower levels.

3.6.2. resolution

This is an alternative way of specifying the zoom level at which an object appears. It is specified as a number from 1-24, which corresponds to one of the zoom levels that Garmin hardware recognises. You should not use resolution if you have used level as they achieve the same outcome.

In either case, the mapping between level and resolution is given in the options style file, where you will see something like this:

```
# The levels specification for this style
#
levels = 0:24, 1:23, 2:22, 3:20, 4:18, 5:16
```

This sets level zero equal to resolution 24, level 1 to resolution 23 and so on.

Although the default style uses `resolution` rather than `level` it is on the whole much easier to use `level` as it is immediately clear where the element will end up. If you use a `resolution` that is 'between' two levels for example it will only show up in the lower one.

Resolution ranges. Just as with levels, you can specify a range of resolutions at which an object should appear. Here is an example.

```
highway=residential [0x06 resolution 16-22 continue]
highway=residential [0x07 resolution 23-24]
```

This example creates roads of type 0x08 between resolutions 16 and 22, then roads of type 0x09 between resolutions 23 and 24. This example makes use of the continue statement, which is discussed in more detail below.

Since 24 is the default upper bound for a range, that second range could just have been written as the single number ‘23’.

3.6.3. default_name

If the element has not already had a name defined elsewhere in the rule, it will be given the name specified by `default_name`. This might be useful for things that usually don’t have names and don’t have a recognisable separate Garmin symbol. You could give a default name of ‘bus stop’ for example and all bus stops that didn’t have their own name would now be labelled as such.

Be careful to use this sparingly and not overwhelm the map or the search.

3.6.4. road_class

Setting this makes the line a "road" and it will be routable and can be part of an address search. It gives the class of the road where class 4 is used for major roads that connect different parts of the country, class 3 is used for roads that connect different regions, down to class 0 which is used for residential streets and other roads that you would only use for local travel.

It is important for routing to work well that most roads are class 0 and there are fewer and fewer roads in each of the higher classes.

Table 3.5. Road classes

Class	Used as
4	Major HW/Ramp
3	Principal HW
2	Arterial St / Other HW
1	Roundabout / Collector
0	Residential Street / Unpaved road / Trail

3.6.5. road_speed

This keyword is used along with `road_class` to indicate that the line is a "road" that can be used for routing and for address searches. It is an indication of how fast traffic on the road is. 0 is the slowest and 7 the fastest. This is **not** a speed limit and does not activate the maximum speed symbol on the newer Garmin car navigation systems. The speed limits that Garmin knows are shown in the following table:

Table 3.6. Road Speeds

road_speed	highest speed
7	No speed limit
6	70 mph / 110 kmh
5	60 mph / 90 kmh
4	50 mph / 80 kmh
3	35 mph / 60 kmh
2	25 mph / 40 kmh
1	15 mph / 20 kmh
0	3 mph / 5 kmh

3.6.6. continue

As discussed above, style rules are matched in the order that they occur in the style file. By default, for any given OSM object mkgmap will try each rule in turn until one rule wth a *element type definition* matches; it will then stop trying to match further rules against the current OSM object. If the rule only has an *action block* mkgmap will continue to find other matches.

However, if you add a *continue* statement to the definition block of a rule, mkgmap will not stop processing the object but will instead carry on trying to match subsequent rules until it either runs out of rules or finds a matching rule that does not include a *continue* statement.

This feature is used when you want more than one symbol to result from a single OSM element. This could be for clever effects created by stacking two lines on top of each other. For example if you want to mark a bridge in a distinctive way you could match on `bridge=yes`, you would then almost always use `continue` so that the `highway` tag could be matched later. If you failed to do this then there might be a break in the road for routing purposes.

Note that by default when using the *continue* statement the action block of the rule (if there is one) will only be applied *within this rule* and not during any following rule matches. Use the *continue with_actions* statement if you want to change this behaviour (see next section).

3.6.7. continue with_actions

The with_actions statement modifies the continue behaviour in such a way, that the action block of this rule is also applied, when this element is checked for additional conversions.

Example of a full element type definition.

```
[0x2 road_class=3 road_speed=5 level 2 default_name 'example street' continue with_actions
```

3.7. Some examples

The following are some examples of style rules, with explanations of what they do.

3.7.1. Points style file

Example 3.2. Internet cafes

```
amenity=cafe & internet_access=wlan {name '${name} (wifi)'} [0x2a14 resolution 23]
```

Checks to see if an OSM object has both the amenity=cafe and internet_access=wlan key/tag pairs. If name=Joe's Coffee Shop, then the Garmin object will be named *Joe's Coffee Shop (wifi)*. The Garmin object used will be 0x2a14 and the object will only appear at resolutions 23 and 24

Example 3.3. Guideposts

```
information=guidepost
    { name '${name} - ${operator} - ${description} '
      | '${name} - ${description}'
      | '${name}'
      | '${description}'
      | '${operator}'
      | '${ref}'
     }
    [0x4c02 resolution 23 default_name 'Infopost']
```

Checks to see if an OSM object has the information=guidepost key/tag pair. If so then the name will be set depending on the available `name`, `operator` and `description` tags as follows.

1. If for example we have the tags `name="Route 7"`, `operator="Kizomba National Parks"` and `description="Trail signpost"`, then the Garmin object will be named *Route 7 - Kizomba National Parks - Trail signpost*.

2. If the OSM object just has the `name` and `description` tags set, the Garmin object will be named *Route 7 - Trail signpost*

3. If just the `name` tag is available, the Garmin object will be named *Route 7*

4. If just the `description` tag is available, the Garmin object will be named *Trail signpost*;

5. and if just the `operator` tag is available, the Garmin object will be named *Kizomba National Parks*.

The Garmin object used will be 0x4c02 and will only appear at resolutions 23 and 24

Example 3.4. Car sales rooms

```
shop=car {name '${name} (${operator})' | '${name}' |'${operator}'} [0x2f07 resolution 23]
```

If name="Alice's Car Salesroom" and operator=Nissan, the Garmin object will be named *Alice's Car Salesroom (Nissan)*

Example 3.5. Opening hours in postcode field

This is a trick to get opening hours to show up in the postcode field of a POI. Tricks like this can enhance the map for certain uses, but of course may prevent the proper use of the postcode field.

```
opening_hours=* {set addr:postcode = '${addr:postcode} open ${opening_hours}'
   | 'open ${opening_hours}'}
```

For *any* OSM object which has the opening_hours key set to a value, this sets the postcode to include the opening hours. For example, if addr:postcode=90210, addr:street=Alya Street, addr:city=Lagos and addr:housenumber=7 and opening_hours=09.00-17.00, the address field of the Garmin POI will be *7, Alya Street, Lagos, 90210 open 09.00-17.00*.

3.8. Troubleshooting

For each node/way/relation, mkgmap goes through the tags exactly once in order from the top of the file downward. For each rule that matches, any action block will be run. As soon as a rule that ends with a type definition is found then processing stops and that is the Garmin symbol that is produced.

The only exception is if the Type Definition contains the `continue` statement. In that case *mkgmap* will continue looking for further matches.

- Where possible always have the same tag on the left. This will make things more predictable.
- Always set made-up tag names if you want to also match on them later, rather than setting tags that might be used already.

3.9. Including files

Its often convenient to split a file into smaller parts or to use the same rules in two different files. In these cases you can include one rule file within another.

```
include "inc/common";
```

Here some common rules have been included in a rule file from a directory called "inc" within the style. Note that the line ends in a semi-colon which is easy to forget.

The included files don’t have to be located within the style and can be anywhere else.

When you include a file, the effect is exactly as if you had replaced the include line with the contents of the file. An `include` directive can occur anywhere that a rule could start, and it is possible to include another file from with in the file that is included.

Including from another style. It is also possible to include a file from another style. To do this you simply add +from +*stylename* to the end of the include statement.

```
include "points" from default;
```

That will include the `points` file from the default style. This might be useful if you want to only change a few things about the default style.

3.10. Simple example

In the majority of cases everything is very simple. Say you want roads that are tagged as **highway=motorway** to have the Garmin type 0x01 ("motorway") and for it to appear up until the zoom level 3.

Then you would write the following rule.

```
highway=motorway [0x01 level 3]
```

Nodes that have an id and a subid are referenced by concatenating both ids.

```
amenity=bank [0x2f06 level 3]
```

This will be explained in more detail in the following sections along with how to use more than one tag to make the choice. However with that one form of rule, you can do everything that the old map-features file could do.

Chapter 4. Creating a style

4.1. Testing a style

You can test your style by calling mkgmap with the `--style-file=`*path-to-style* and the `--list-styles` option. If you see your style listed, then your style is recognized by mkgmap. Then you can test if your style is valid by using it when creating a map.

A style can be used just as it was created, but if you want to make it available to others it will be easier if you make a zip file out of it and then you just have the one file to distribute. You just can zip all files of the style. Several different styles can be placed into the same zip archive file.

To use a zipped style, you can use `--style-file=`*stylename*`.zip`. If there is more than one style in the zip file, then you can use `--style-file=`*zipname*`.zip --style=`*stylename*.

4.2. Making a style package

A style can be used just as it was created, but if you want to make it available to others it will be easier if you combine all the individual files into a single archive file.

4.2.1. Zip archive

The first way of doing this is to combine the files into a zip file and then you just have the one file to distribute.

To use a zipped style, you can use --style-file=stylename.zip

It does not matter if you include the directory holding the files or not in the archive. The style is found by searching for the `version` file.

You can have more than one style in the zip file, each in their own directory. In this case you must include the top level directories of the style (and you can include other parent directories as well if you like). If there is more than one style in the zip file, then you can use the `--style` option alongside the `--style-file` option. --style-file=zipname.zip --style=stylename.

Example 4.1. Style package layout

```
.
`-- mystyles
    |-- cycle
    |   |-- lines
    |   |-- points
    |   |-- polygons
    |   `-- version
    `-- hiking
        |-- lines
        |-- points
        |-- polygons
        `-- version
```

Here there are two styles named *cycle* and *hiking*. You can select the ‘hiking’ style with the options --style-file=mystyles.zip --style=hiking

4.2.2. Simple file archive

This is formed by appending all of the files of a style into a single file separated by lines that contain the file name in triple angled brackets.

Single file archive.

```
<<<version>>>
0

<<<points>>>
amenity=doctor [0x2a2a level 0]
# More point definitions here...

<<<lines>>>
# All the line definitions here...
```

The file must have a name ending in `.style` to be recognised.

This file can be easily created in its entirity in a text editor, but you can also convert between the files-in-a-directory format and the single-file format using the following command:

```
 # (to be typed all on one line)
 java -cp mkgmap.jar uk.me.parabola.mkgmap.osmstyle.CombinedStyleFile
      mystyle > mystyle.style
```

To convert back then supply the file as the argument, rather than the directory.

Chapter 5. About

5.1. Licence

This manual is released under the Creative Commons Attribution-ShareAlike 2.0 license [http://creativecommons.org/licenses/by-sa/2.0/]. It makes use of some material that was added to the OSM Wiki which is release under the same licence.

5.2. Authors and acknowledgments

This manual is created from material that originated from the mkgmap doc files and added to OSM wiki. While on the OSM wiki modifications were made by many people.

People who have contributed suggestions and corrections to this document are: Carlos Dávila, Geoff Sherlock

The list of nicknames of everyone that had modified the wiki pages at the time that this manual was created is as follows: Brogo, Christian Gawron, Csdf, De muur, Derstefan, DirkS, Extremecarver, Gernot, !i!, Jinx1971, Katpatuka, MarkS, Master, Mezzanine, Nakor, Nop, Richard, Skela, SomeoneElse, Tommybgoode, Ulfl, Walterschloegl, WanMil, Willem1, Yggdrasil