INTERNATIONAL STANDARD

ISO 17267

First edition
2009-11-15

Intelligent transport systems — Navigation systems — Application programming interface (API)

Systèmes intelligents de transport — Systèmes de navigation — Interface de programmation (API)

Reference number
ISO 17267:2009(E)

Contents

Page

Foreword

ISO (the International Organization for Standardization) is a worldwide federation of national standards bodies (ISO member bodies). The work of preparing International Standards is normally carried out through ISO technical committees. Each member body interested in a subject for which a technical committee has been established has the right to be represented on that committee. International organizations, governmental and non-governmental, in liaison with ISO, also take part in the work. ISO collaborates closely with the International Electrotechnical Commission (IEC) on all matters of electrotechnical standardization.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 2.

The main task of technical committees is to prepare International Standards. Draft International Standards adopted by the technical committees are circulated to the member bodies for voting. Publication as an International Standard requires approval by at least 75 % of the member bodies casting a vote.

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO shall not be held responsible for identifying any or all such patent rights.

ISO 17267 was prepared by Technical Committee ISO/TC 204, *Intelligent transport systems*.

Introduction

The impetus for this International Standard was the recognition by the intelligent transport systems (ITS) industry of the need for standardization with respect to data access for map databases used by navigation applications. As the vehicle navigation industry has grown, so has incompatibility between navigation systems and map databases. Both a standardized physical storage format (PSF) and a standardized navigation application programming interface (API) can facilitate the interoperability between navigation systems and map databases.

The purpose of this International Standard is to define and structure the model for data access for Vehicle Navigation and Traveller Information Systems. This International Standard is not restricted to physical media and will be independent of any underlying physical storage format. While this API is primarily targeted at self-contained in-vehicle systems, it is expected to be usable by other applications that use map data results in essentially the same way. For example, it may be usable by client/server or distributed navigation systems and location-based services without further specialization.

This International Standard is the Application programming interface (API) specification. It represents the comprehensive specification of the API standard for navigation applications. This International Standard builds upon, and is consistent with, the other International Standards developed by ISO/TC 204/WG 3:

— ISO 14825, *Intelligent transport systems — Geographic Data Files (GDF) — Overall data specification*;

— ISO 17572 (all parts), *Intelligent transport systems (ITS) — Location referencing for geographic databases*.

Intelligent transport systems — Navigation systems — Application programming interface (API)

1 Scope

This International Standard specifies an application programming interface (API) for navigation systems. It specifies the data that may be retrieved from the map database and defines the interface for access. This International Standard specifies a set of function calls. It also specifies the design of the API and gives examples of its intended use. Furthermore, it gives the criteria to determine whether a data access library is in accordance with this International Standard.

This International Standard is applicable to the following functional categories of navigation applications:

— positioning;

— route planning;

— route guidance;

— map display;

— address location;

— services and point of interest (POI) information access.

2 Terms and definitions

For the purposes of this document, the following terms and definitions apply.

2.1
address location
application category that deals with the task of expressing a real world position in terms of the data representation

NOTE Address location is one of the six application categories supported by the API.

2.2
address type
attribute of road section entity that specifies the type of house number ranges

EXAMPLE Distinction between base address, county address, commercial address, etc., or no address.

2.3
application category
basic sub-function within the set of functionality for vehicle navigation and traveller information system applications

NOTE This International Standard identifies six application categories: positioning; route planning; route guidance; map display; address location; services and POI information access.

2.4
application programming interface
API
standard interface and set of function calls between application software and data access libraries of vehicle navigation systems, in accordance with this International Standard

2.5
base map
all transportation elements and all services, including their relationships to transportation elements

2.6
branded third-party data
BTPD
information about services which is supplied by third-party data providers (e.g. tourist or motoring organizations), who may impose proprietary restrictions on the use and presentation of the data

NOTE 1 Access is subject to authorization and licensing.

NOTE 2 BTPD is a subset of third party data (TPD); see 2.54.

2.7
cartographic feature
data model entity that represents geometrical information for display purposes

NOTE A cartographic feature has non-explicit topology; it has zero-, one- and two-dimensional types, i.e. Display Point, Polyline, and Polygon.

2.8
cartographic text
data model entity that stores name text associated with all or part of a cartographic feature

NOTE Cartographic text is language dependent and may contain a suggested display location, orientation, language code, priority (or importance), suggested scale range, and bounding box.

2.9
condition
information related to link(s) composed of condition type, condition modifiers, and condition scope

2.10
crossroad
data model entity that represents the single instance of the crossing of two named navigable features

NOTE Crossroad relates to the set of links and nodes which comprise the crossing, and to the crossing of the navigable features to a place.

2.11
destination node
node at the end of the link toward which travel takes place

NOTE See also origin node (2.25), "from" node (2.14), "to" node (2.55), source node (2.50), and target node (2.53). When a link is travelled in the direction of topological orientation, the destination node is the "to" node. When it is travelled in the direction opposite topological orientation, the destination node is the "from" node.

2.12
display point
zero-dimensional type of cartographic feature

2.13
dummy point
optional entity that represents a position along a link where the link crosses a parcel boundary and does not necessarily coincide with a shape point or node

2.14
“from” node
node at the end of a link away from which the link is topologically oriented

NOTE See also “to” node (2.55), origin node (2.25), destination node (2.11), source node (2.50), and target node (2.54). When a link is travelled in the direction of topological orientation, the “from” node is the origin node. When it is travelled in the direction opposite topological orientation, the “from” node is the destination node.

2.15
geocoding
determination of a link or node based on address information describing and/or naming a location

2.16
intersection
geographic data file (GDF) level 2 representation of a crossing which bounds a road or a ferry as a complex feature composed of one or more GDF level 1 junctions, road elements and enclosed traffic areas

2.17
junction
data model entity that represents a navigable feature which is either a named GDF junction or named GDF intersection, and that relates a named navigable feature to a set of links and nodes and a place

2.18
landmark
point, line, or area feature, possibly associated with a node or link, that can be used to clarify the directions generated to describe a route

NOTE A landmark may not be in the Services, Administrative Areas, or Public Transportation feature themes of a GDF; a facility in which a service is located may be a landmark.

2.19
layer
subset of map data resulting from a subdivision of data of the same coverage area based on contents and which is typically related to one or only a few of the application categories

NOTE This is similar to an ISO-GDF layer.

EXAMPLE Route guidance data may be considered as one layer.

2.20
level
subset of map data resulting from a classification of data of the same semantic content based on the level of detail or density, related to the concept of different map scales

NOTE Level 0 is considered the lowest level (greatest detail); higher levels are numbered level 1, level 2, etc.

EXAMPLE Map display data may be organised into 6 levels representing different zoom scales.

2.21
link
directed topological connection between two nodes, composed of an ordered sequence of one or more segments and represented by an ordered sequence of zero or more shape points (2.48)

2.22
map display
application category that deals with graphical information presentation

NOTE Map display is one of the six application categories supported by the API.

2.23
multilink
ordered aggregation of links which are at the same level, are connected in sequence, and share the same functional classification, form of way, direction of travel, and perhaps additional characteristics

EXAMPLE Each link is contained in exactly one multilink.

2.24
navigable feature name
data model entity that represents the name for the transportation element, including GDF road element, GDF ferry connection, GDF junction, GDF intersection

NOTE Navigable feature name is related to places, crossroads, junctions, and road sections.

2.25
node
data model entity for a topological junction of two or more links or for end-bounding a link

NOTE A node stores the coordinate value of the corresponding GDF junction.

2.26
origin node
node at the end of a link from which travel takes place

NOTE See also destination node (2.11), "from" node (2.14), "to" node (2.55), source node (2.50), and target node (2.54). When a link is travelled in the direction of topological orientation, the origin node is the "from" node. When it is travelled in the direction opposite topological orientation, the origin node is the "to" node.

2.27
parcel
database partitioning unit corresponding to a certain coverage area, associated with one level and containing data of one or more layers

NOTE A parcel contains (at least) all nodes with positions enclosed by or located on the outline of its coverage area plus (parts of) all links attached to these nodes; it can be partitioned so that the amount of data of a parcel may be nearly the same as that of another.

2.28
place
named area which can be used as part of the address location

2.29
place class
attribute of place entity, classifying into highest administrative or geographic division, administrative subdivision, postal, or colloquial (e.g. regions or neighbourhoods)

NOTE Place class can be partially ordered as "place class A is below place class B". This does not imply strict or complete containment.

2.30
place level
level associated with places of place classification "administrative subdivision"

NOTE Higher/lower level situations are constituted by the occurrence of a parent/child place relationship between places.

2.31
place relationship
bivalent relationship between place entities, constituting the place tree linking parent and child places

EXAMPLE Place A is in place B.

NOTE Place relationship does not imply strict or complete containment. It is attributed as: address significant, official, postal or useful for reverse geocoding.

2.32
point of interest
POI
destination and/or site of interest to travellers, usually non-commercial by nature

2.33
polygon
two-dimensional type of cartographic feature

2.34
polyline
one-dimensional type of cartographic feature

2.35
positioning
application category that deals with the determination of vehicle location and map matching

NOTE Positioning is one of the six application categories supported by the API.

2.36
postal code
data model entity for a government-designated code used to specify regions for addressing

NOTE Postal code is related to link (2.21), navigable feature name (2.23), place (2.27), and POI (2.31).

2.37
rectangle
unit of geographic space defined by two parallels of min./max. latitude and by two meridians of min./max. longitude and that represents the coverage area of the map data enclosed by or located on its outline

2.38
reverse geocoding
determination of the address description of a link or node, i.e. determination of an upwards path across the place tree

2.39
road
GDF level 2 feature composed of one, many or no road elements and joining two intersections, serving as the smallest independent unit of a road network at GDF level 2

2.40
road element side
RES
basic component of the road section entity that represents left or right side of a link and corresponds to one or more unique combinations of a navigable feature and a house number range

2.41
road section
data model entity that represents the house number ranges of both sides of a street and that carries a navigable feature name

NOTE Road section corresponds to a link (ID).

2.42
route guidance
application category that deals with the generation of graphical, textual, and/or audio instructions for following a planned route

NOTE Route guidance is one of the six application categories supported by the API.

2.43
route planning
application category that deals with the determination of routes between specified points

NOTE Route planning is one of the six application categories supported by the API.

2.44
segment
straight section of a link connecting two successive shape points, or a shape point and a node, or two nodes where a link does not contain shape points

2.45
service
data model entity for a commercial activity of interest to travellers as a destination and/or orientation that is associated with road element(s) by which it can be accessed and further described by attributes including (at least) name and type

NOTE A service may be associated with other services by parent/child relationships (many to many). Service is used synonymously with POI within the logical data model.

2.46
service attribute
item of descriptive information relating to a service

2.47
services and POI information access
application category that deals with the provision of POI information to the navigation application

NOTE Services and POI information access is one of the six application categories supported by the API.

2.48
shape point
position along a link used to more accurately represent its geometric course, bounded by exactly two segments

2.49
signpost
data model entity for a directional sign that represents a logical relationship between signpost information and two associated links

NOTE The first link (mandatory) represents the road element along which the signpost is located. The second link (optional) is the first road element which directs exclusively to the destination indicated on the signpost. The position of the signpost along the link and the link direction the signpost is facing is also stored.

2.50
source node
node at the end of a link from which exploration takes place for route calculation

NOTE See also target node (2.53), origin node (2.25), destination node (2.11), "from" node (2.14), and "to" node (2.55). When forward exploration is taking place from the origin of the route, the source node of a link is its origin node. When reverse exploration is taking place from the destination of the route, the source node of a link is its destination node.

2.51
super link
aggregation of linearly connected regular links present in the lowest level as a simplified representation of the road network in higher levels

2.52
symbol
data model entity that represents an icon associated with a cartographic feature

2.53
target node
node at the end of a link towards which exploration takes place for route calculation

NOTE See also source node (2.50), origin node (2.25), destination node (2.11), “from” node (2.14), and “to” node (2.56). When forward exploration is taking place from the origin of the route, the target node of a link is its destination node. When reverse exploration is taking place from the destination of the route, the target node of a link is its origin node.

2.54
third party data
TPD
information about services, which is supplied by third party data providers (e.g. tourist or motoring organizations), typically with a rich content of descriptive data

2.55
“to” node
node at the end of a link towards which the link is topologically oriented

NOTE See also “from” node (2.14), origin node (2.25), destination node (2.11), source node (2.50), and target node (2.54). When a link is travelled in the direction of topological orientation, the “to” node is the destination node. When it is travelled in the direction opposite topological orientation, the “to” node is the origin node.

2.56
traffic location
data model entity that contains an external reference (e.g. VICS or RDS-TMC) and is linked to either place or transportation entities

2.57
transportation element
feature from the Roads and Ferries feature theme of a GDF

3 Abbreviated terms

ANSI	American National Standards Institute
CPU	Central Processing Unit
DAL	Data Access Library
DBID	Database ID
DST	Daylight Savings Time
EEPROM	Electrically Erasable and Programmable Read-Only Memory
GDF	Geographic Data File
GMT	Greenwich Mean Time

HOV	High Occupancy Vehicle
HTML	HyperText Markup Language
IDL	Interface Definition Language
MIME	Multipurpose Internet Mail Extensions
OMG	Open Management Group
OS	Operating System
PSF	Physical Storage Format
RDS-TMC	Radio Data System-Traffic Message Channel
VICS	Vehicle Information and Communication System

4 Architecture of the API

4.1 General

Subclauses 4.2 to 4.13 specify the architecture requirements for the design of the ISO-API. Implementation details are not specified in this International Standard.

4.2 Paradigm

The ISO-API shall be specified in an object-oriented way.

4.3 Minimum low level platform interface

It is not necessary to define a minimum low level platform interface as long as a system-independent data access library is not feasible.

4.4 Forward compatibility

The ISO-API shall support forward compatibility in such a way that

— earlier versions of application software can use DALs corresponding to later ISO-API versions, and

— earlier versions of DAL can use data in later PSFs.

The mechanism for forward compatibility shall be entirely hidden from the API.

4.5 Error handling

The application software shall be responsible for handling errors; more specifically, the DAL gives notification of errors while the application software reacts on them.

The ISO-API shall specify a list of error conditions for each function call. The system designer can decide which error mechanism to use for the implementation.

4.6 Memory allocation

The lifetime of the objects and structures shall be controlled by the application software. Memory for objects and structures used internally by the DAL is managed by the DAL.

4.7 Prioritization and cancellation

The ISO-API shall support prioritization and cancellation to control input/output (I/O) operations, perform intelligent caching, etc. This functionality shall be supported by the ISO-API in the following way: Each class (object-orientation!) shall have two member functions "setPriority()" and "getPriority()". Due to this, the application software is able to assign, query and change the priority for an instance of this class. If the application software does not want to use this mechanism, it can assign the same value for each instance. There is no mandated behaviour of the DAL based on the application-specified priorities. The behaviour of the DAL is defined by the DAL supplier.

4.8 Byte ordering

Depending on the hardware (and operating system) of a navigation system, the data values will be physically used in one of two ways.

— Little endian: The least significant byte (LSB) of a value will be stored physically first, e.g. in systems with Intel CPUs.

— Big endian: The most significant byte (MSB) of a value will be stored physically first, e.g. in systems with Motorola 68k CPUs.

Some CPUs are bi-endian, such as R3000 ff, PowerPC. In such a case, the byte order is defined by the operating system.

The following data values are concerned:

— integer values (16, 32, 64 bit);

— floating point values (32, 64, 128 bit);

— characters (16 bit, multibyte).

An ISO-API compliant DAL shall return the data values in the byte order being used by the CPU/OS of the navigation system where the DAL is running. This shall happen regardless of whether a PSF is stored in little endian or big endian byte order. Therefore, a conversion inside the DAL may be necessary. The performance loss shall be minimized by doing the conversion on the lowest level.

4.9 Generic data types

Because the target systems are different it is necessary to define generic system-independent data types to be used for the ISO-API. They are translated by the DAL into the data types of the target system. The generic data types listed in Table 1 shall be defined for the ISO-API.

Table 1 — Generic data types

Meta type	IDL type	Size/Format	Description
MapByte[a]	octet	8-bit two's complement	Byte-length integer
MapShort[a]	short	16-bit two's complement	Short integer
MapLong[a]	long	32-bit two's complement	Integer
MapLongLong[b]	longlong	64-bit two's complement	Long integer
MapFloat[b]	float	32-bit IEEE 754	Single-precision floating point
MapDouble[b]	double	64-bit IEEE 754	Double-precision floating point
MapChar[c]	wchar char	16-bit or less character 8-bit character	Single character Single character
MapBoolean	Boolean	8-bit; 0x00 is false, any other is true	Boolean value: true or false
MapString	string	16-bit or less characters; max. length 65 535	Zero-terminated string

[a] Unsigned and signed.

[b] These types may not be supported natively on some systems. Their inclusion here does not necessarily mean that they have to be used.

[c] The size of the characters to be used depends on the definitions in the PSF. Variable length character sets are not necessarily considered yet.

The column entitled “IDL type” in Table 1 establishes the reference between the well-defined and stable meta type and the implementation in IDL.

4.10 Handling of large result sets

4.10.1 Background

For many API calls, the amount of data that will be returned will not be easy to predict in advance. For example, a call to retrieve all road elements within a bounding box may return a very large amount of data if the bounding box falls in a dense area.

On the other hand, DALs will be implemented on a wide variety of platforms. If memory for results is allocated dynamically, returning very large results may pose problems for memory allocation on some platforms.

A flexible solution is required, so that the result can be returned in pieces if necessary. The proposal below specifies such a solution. The general approach is as follows: One function is called to prepare for the return of a possibly large result set. A second function is called as many times as necessary to return the entire result set. A third function is called, perhaps before the entire result set has been returned, to "close" the operation.

The first and second functions return an indication of the size of the result set to be expected, so that applications which are capable of allocating large amounts of memory dynamically can allocate enough memory to receive the whole result set at once, thereby improving efficiency. However, there are cases in which it is computationally expensive to compute the exact size of the result set in advance; if the caller never asks for the whole result set, such computation is wasted. Therefore, the proposal allows for an overestimate, rather than an exact size, to be returned.

The following architectural requirement in 4.10.2 is therefore specified.

4.10.2 Requirements

Calling sequences for functions that may return results of unpredictable size shall be structured as follows:

a) Functions to return such result sets shall come in sets of three:

   1) The first function (the "opening" function) describes the request, initializing a data structure (herein called a "handle"), which is used to store the state information for the request.

   2) The second function (the "fetching" function) is called to return successive parts of the result. However, this function need not be called enough times to return the entire result; indeed, it need not be called at all.

   3) The third function (the "closing" function) is called to notify the library that the application is finished with the request. The use of handles allows more than one request to be in process at the same time. The library may place a reasonable upper limit on the number of handles that are active at one time. (The question of whether the memory for the handles is allocated by the library or by the application must be decided but is not addressed here.)

b) The opening function shall return a value indicating the number of bytes expected to be necessary to hold the result. Also, each call to the fetching function shall return a value indicating the number of bytes expected to be necessary to hold the remainder of the result. In order to allow more efficient implementation, these values may be overestimates, rather than exactly correct values. A special value is also reserved to indicate that no good (over)estimate can be provided. (Using the maximum positive integer as this special value is recommended but not required for conformance with this International Standard.)

c) One argument to the fetching function is a pointer to the buffer in which the result is to be returned. Another argument is the size of the buffer in bytes. This buffer (if necessary) shall be allocated and de-allocated by the calling application, not by the functions described here. Also, the address and size of the buffer need not be the same from call to call.

d) The fetching function shall return the number of objects returned with each call. In order to accommodate objects of variable size, the function shall also return the number of bytes returned.

e) The opening function shall also return an indication of whether the result will be non-empty, and the fetching function shall also return an indication of whether it has returned the entire result. (This can be derived from whether the estimates of the number of bytes required for the result or for the remainder of the result, respectively, is zero. However, that approach may be considered needlessly obscure, in which case a separate return value, for example the function's return value, should be used.) However, for purposes of efficiency, these indications must be allowed to give false positives, that is, to indicate that return values exist when in fact none do; this allows the determination of whether the values actually exist to be postponed to the next call to the fetching function.

f) The functions may defer disk reads, record unpacking, and similar operations until they are needed. Since it is expected that, in many instances, the fetching function will be called to return only part of the full result, this may result in substantial gains in efficiency.

4.10.3 Object-oriented example

The following example describes an object-oriented implementation of the same functionality. A class instead of a handle is used to store the state information for the query, and the constructor and destructor methods for the class serve as the "opening" and "closing" functions, respectively. The user-defined type "object_t" has been replaced with the class MapObject. Exceptions are used instead of negative return values to indicate failures. All other features of the design, including the allocation of memory by the application, are similar to the procedural example above.

```
//
// query interface
//
class MapQuery {
public:
    MapQuery([parameters specifying objects to return]) throw
   (open_failed);
    ~MapQuery(void) throw (close_failed);
    bool fetch(void) throw (fetch_failed);
    void setBuf(MapObject* bp);
    void setBufSize(size_t s);
    size_t getResultSize(void);
    size_t getResultObjects(void);
    size_t getRemainderSize(void);
};

//
// sample application
//

void main()
{
    MapObject* buf;
    size_t numObjects;

try
{
        // query map
        MapQuery query([parameters specifying objects to return]);

        // obtain prospective number of matching objects
        numObjects = query.getResultObjects();

        // application allocates memory to hold query results
        buf = new MapObject[numObjects];

        // application tells object location of allocated memory
        query.setBuf(buf);
        query.setBufSize(size of(MapObject) * numObjects);

        // process query
        while (query.fetch())
        {
            for (int i=0; i<query.getResultObjects(); ++i)
            {
                buf[i].handleObject();
            }
        }
    }

    // handle failures in MapQuery methods
    catch (MapException e)
    {
        cerr << e.what() << endl;
    }
    // application deallocates memory
    delete [] buf;
}
```

4.11 Multimedia issues

In addition to the map data, the database may contain multimedia objects, e.g. an image of a POI. It shall be possible to retrieve such information via the API.

The following types of multimedia object shall be supported:

— text files;

— HTML files;

— sound;

— still images;

— motion images.

The set of multimedia types that are supported by the API is not fixed, i.e. it shall be extendible.

The API shall be able to identify the type of a multimedia object so that the application software is able to associate it with the appropriate decoder/presentation subsystem. Therefore, a content type specifier shall be supported in the API for each multimedia object. The content type specifier shall be compliant with the MIME (Multipurpose Internet Mail Extensions).

Furthermore, the API shall allow the use of stream I/O for multimedia objects on platforms which support it.

Multimedia decoders and presentation subsystems are outside of the scope of this International Standard.

4.12 Location of application software, DAL and data

4.12.1 Application software

The application software implements

— the complete user interface of the navigation software (i.e. map display, user interaction elements), and

— the system-dependent parts (i.e. O/S level memory management, input/output).

There are four possibilities for the location of the application software within a certain navigation system:

1) Stored permanently in the system (i.e. EEPROM):

    — it can be easily accessed;

    — application updates are non-trivial, especially for the large number of systems on the market.

2) Stored together with the data on removable media (generally on CD-ROM):

    — requires that the application be pre-loaded into the system before execution;

    — allows easy updating of the application software (and data);

    — not suitable for PSF-compatible systems.

3) Stored separately on removable media (i.e. flash EEPROM card):

— requires that the application be pre-loaded into the system before execution;

— applications can be updated independently of data.

4) Any combination of 1) to 3):

— combines the disadvantages of the used possibilities.

4.12.2 Data access library

The data access library consists of all functions (presently 6 application categories) that are accessing data. For the DAL location there are the following cases:

1) Stored permanently in the system:

— can be easily accessed;

— DAL updates are complicated.

2) Stored together with the data on removable media:

— requires that the DAL be pre-loaded into the system;

— DAL and PSF are always compatible;

— DAL updates at the same time as the data.

3) Stored separately on removable media:

— requires that the DAL be pre-loaded into the system;

— DAL updates can be done independently of data.

4) Any combination of 1) to 3):

— combinations of 2) and 3) will raise the flexibility of the whole system.

4.12.3 Data

Data in the context of this International Standard is the summary of all geographical data that are required for certain functional categories. There are the following possible locations for the data:

1) Stored permanently in the system:

— data updates are non-trivial due to the high update rate of the data and the large amount of data.

2) Stored on removable data media:

— data updates are much easier;

— the approach generally used for navigation systems.

3) Combination of 1) and 2):

— combines the disadvantages of 1 with 2.

4) Accessible over the Internet or another network, either wired or wireless.

4.12.4 Conclusions

This International Standard does not place restrictions on the locations of any component. A more flexible solution allows for competition between the systems.

Figures 1 to 3 display examples showing several possibilities for the location of components.

Figure 1 a) shows how two different systems (hardware A with application software A, and hardware B with application software B) can be serviced by a single data medium. It is assumed that each navigation system can interface to

— any industry-wide OPEN PSF (by having a proper DAL built into its application software),

— the ISO-API (by having the ISO-API functions defined in its application software), and

— its own proprietary API-A or its proprietary PSF-A, both of which are outside the scope of this International Standard.

Hardware A (or system A) refers to both the operating system and the computing hardware. Should either change, it would no longer be system A.

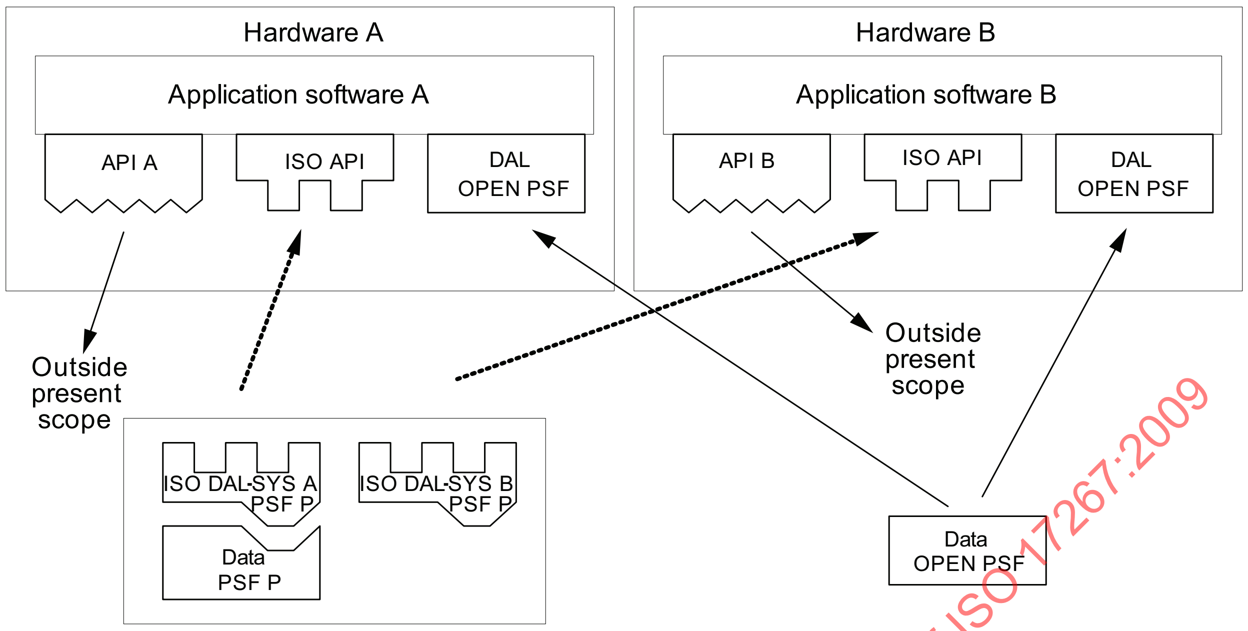
The medium on the right-hand side consists of data only, and it is in an OPEN PSF format. Both navigation systems can read an OPEN PSF via their built-in OPEN PSF DALs.

The medium on the left-hand side delivers its data in some proprietary PSF-P. The same medium has a DAL, which reads the proprietary PSF-P and returns that data in a format in accordance with this International Standard. The DAL is either in Java or in compiled binary code for hardware A or hardware B. In this scenario, the DALs are supplied on the medium along side the data. In Figures 2 and 3, the DALs are supplied independently of the data media. Given that the DAL software is much smaller in size than the actual data, there can be as many versions of it as there are known navigation systems. While this method is not forward compatible, neither is the quality and content of the data. Should new navigation systems come into existence, compiled versions of the DAL for PSF-P can be delivered the next time the data is updated and a new set of media is released.

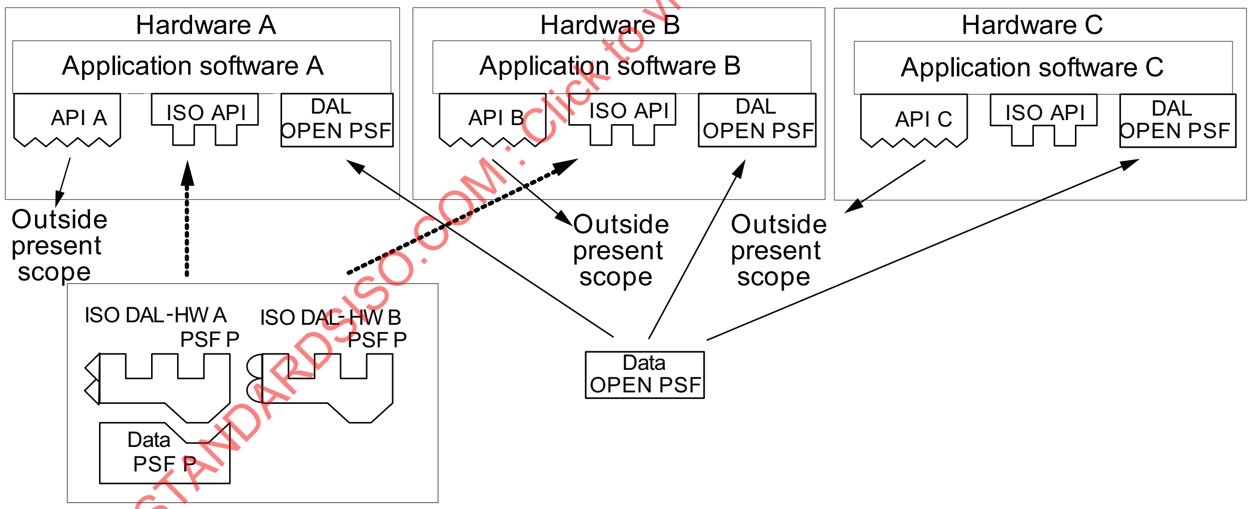
In Figures 1 b) and 1 c) the specific situation of forward compatibility is explored. Upon release of the medium with the set of DALs for all known systems at that time, the medium is fully interoperable. However, if the DAL is not a Java DAL (or a new system C shows up which cannot support Java), then the only means of interoperability with the future system C is via an OPEN PSF.

Figure 1 b) shows the situation of forward compatibility. A new system (system C) is introduced into the market. The medium does not contain the ISO-API compiled for system C. There are only two ways, given that medium, to be able to operate on system C: either the Java DAL is on the medium (and system C can make use of it), or an OPEN PSF is on the medium (which would have to be the case in order to be able to claim that medium compliant with a particular OPEN PSF). In this particular situation the ISO-API would not be (forward) compatible with the new system C. Figure 1 c) shows one possible remedy.

In Figure 1 c) a remedy to the failure of total forward compatibility of the ISO-API is proposed. Via collaboration between data supplier P (presumably the owner or licenser of PSF format P), the ISO-DAL creator (data supplier P or related to data supplier P), and the new system maker C, a supplemental DAL is made available, either with or without the data published in PSF P. If map data is supplied, potentially it is a newer and better version than that supplied originally.

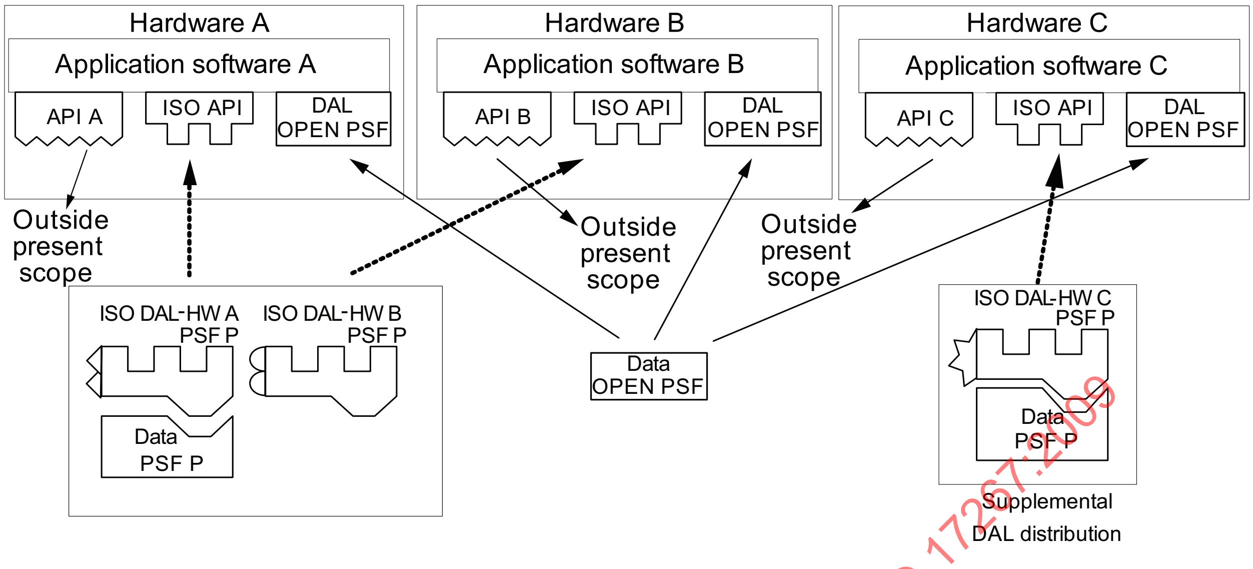


a) As supplied



b) A new system C is introduced into the market

Figure 1 — (DAL on the medium)



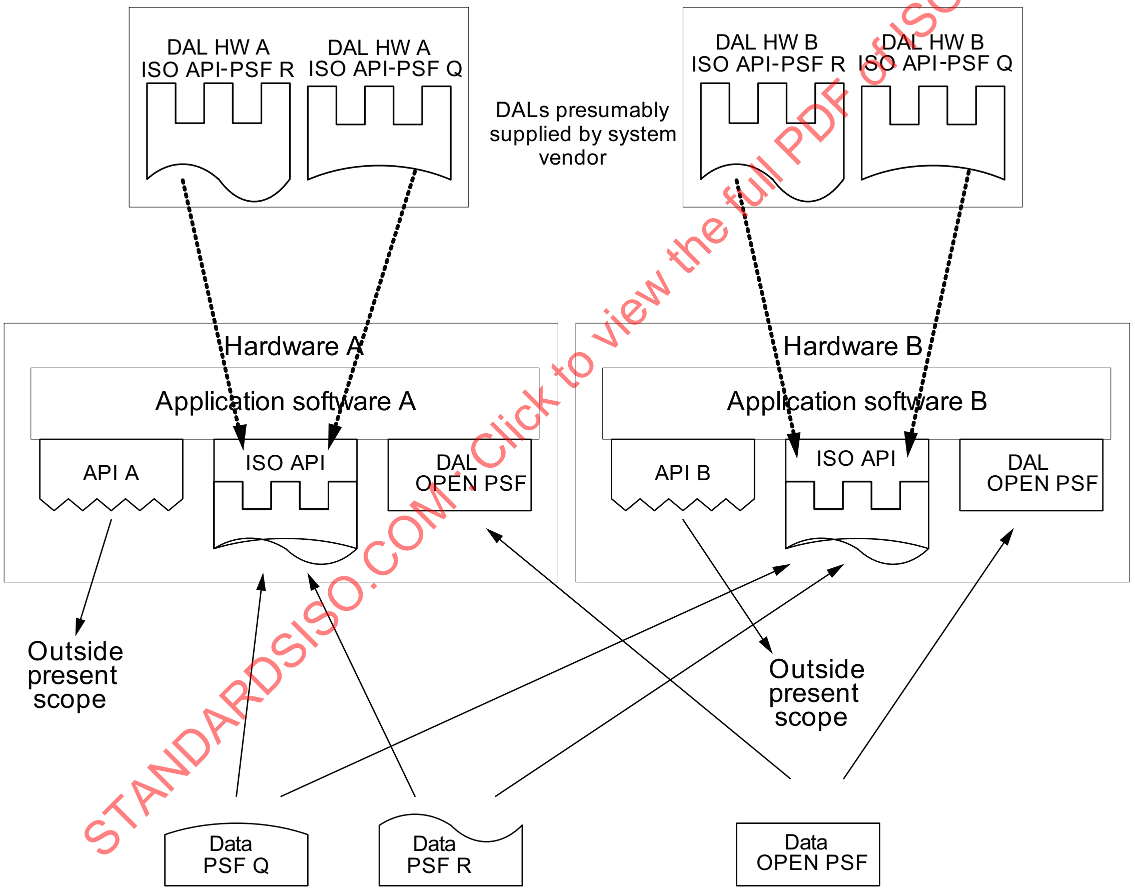
c) A supplement goes out to address the new system

Figure 1 (*continued*)

In Figure 2 a) the DALs are supplied independently of the data. Each system vendor supplies the DALs for the PSFs that it chooses to support using an access mechanism in accordance with this International Standard. The DALs are pre-loaded and therefore part of the application software by the time the data arrives. By having the data delivered in an ISO-API access mechanism, the application software can access the data in a uniform way, regardless of the delivery mechanism.

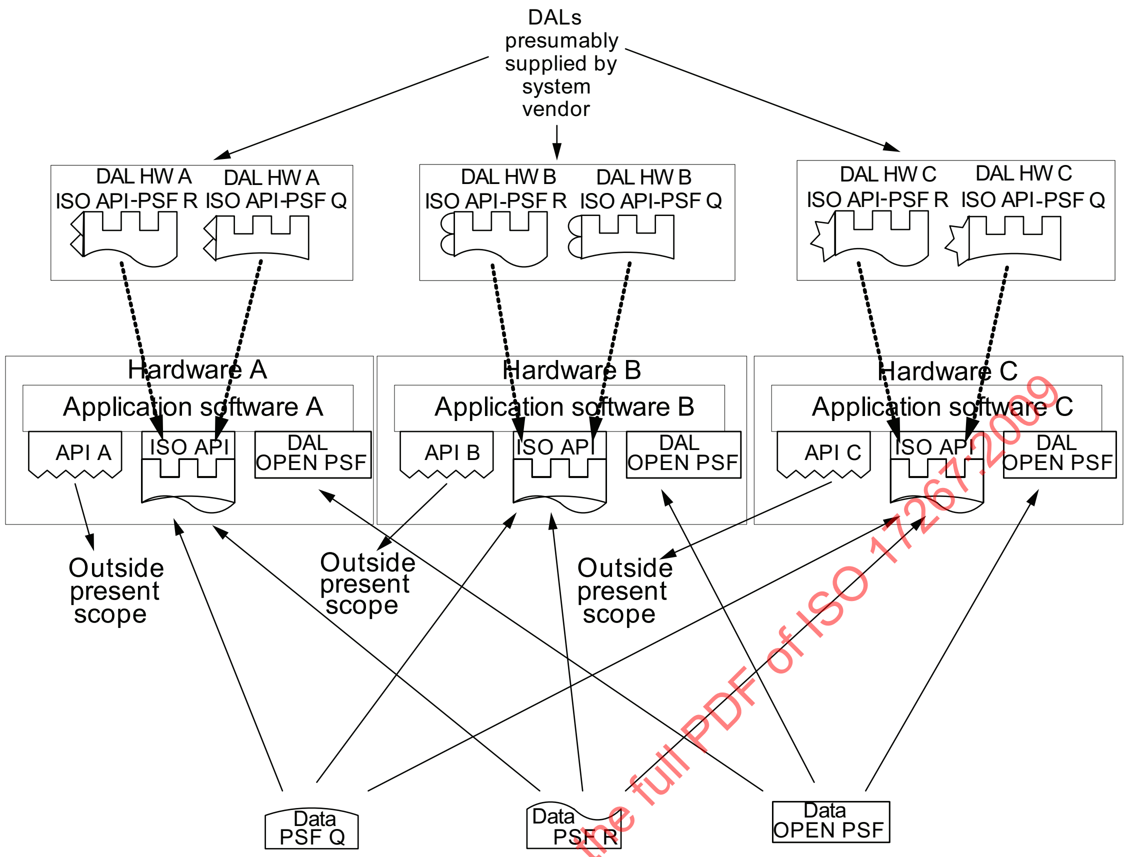
Similarly to the case of the DAL on medium, Figure 2 b) explores the situation of forward compatibility upon introduction of a new system C into the market. In this case forward compatibility is sustained.

In this scenario, the data on the medium is independent of the DAL which reads it, so, similarly to system providers A and B, it is the responsibility of the new system provider C to provide DALs for its own system which can read PSFs Q and R, and any other ones which are to be served under the ISO-API. The advantage of this scenario is that there is no market lag between the introduction of new system C and the availability of DALs for ISO-APIs that can read all the data media available in the market place. The disadvantage of this scenario is that it is necessary to expose the PSFs Q and R (and any other ones) to system vendor C.



a) As supplied

Figure 2 — DAL not on medium, supplied by system maker

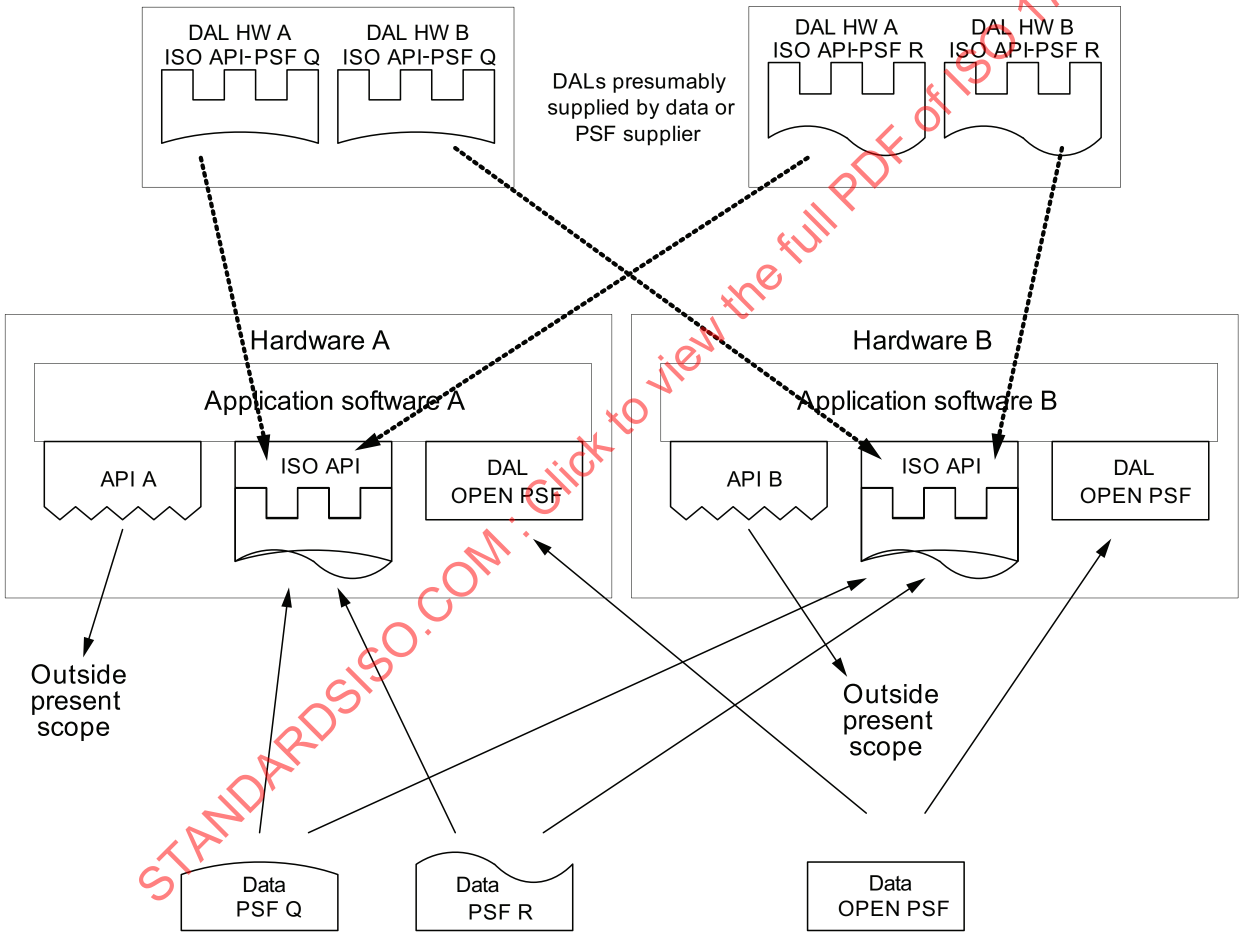


b) A new system C is introduced into the market

Figure 2 (*continued*)

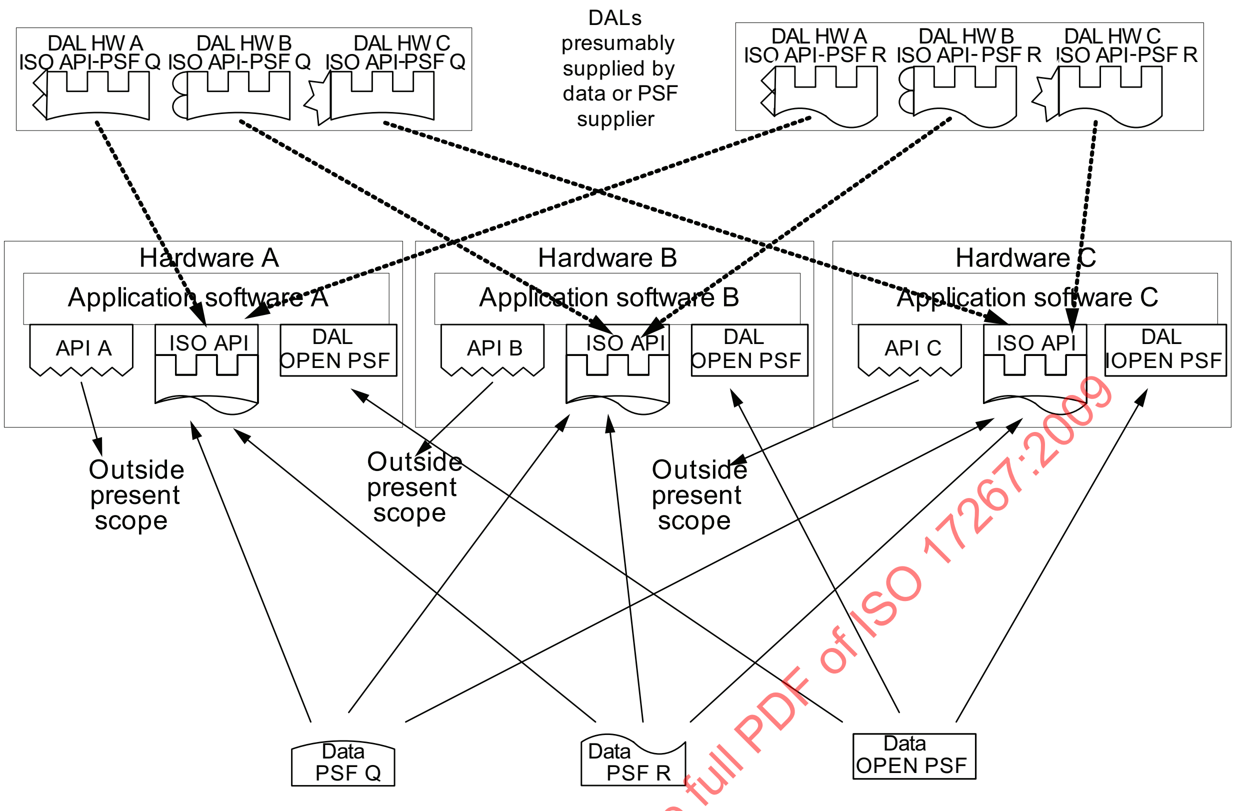
In Figure 3 a) the situation is similar to the situation of Figure 2 a), except for one aspect: The DALs are supplied by a data supplier of a proprietary PSF, rather than by the system vendors themselves. Each PSF supplier implements the DAL for any navigation system they wish to support. Figure 3 b) shows what happens when a new system C is introduced.

In this scenario, the data on the medium is, also, independent of the DAL which reads it, except here it is the responsibility of the data (or PSF) providers to provide for system C, as they did for systems A and B, DALs for their own PSF which would deploy also on system C in a manner that does not conflict with with this International Standard. The disadvantage of this scenario is that there is potentially a market lag between the introduction of new system C and the availability of DALs for APIs that comply with this International Standard which can read all the data media available in the market place. Even if some data providers release quick updates for system C, others might lag and leave system C in a somewhat vulnerable position, as its interoperability to all data on the market would be initially only partial. The advantage of this scenario is that no PSFs need be exposed to system maker C.



a) As supplied

Figure 3 — DAL not on medium, supplied by PSF provider



b) **A new system C is introduced into the market**

Figure 3 (*continued*)

4.13 Base and extended APIs

4.13.1 Terms

A Base API contains all *required* functions, and no *extending* functions. Functions are classified as *required* or *extending* according to Clause 5.

An Extended API contains all Base API functions plus all *extending* functions. Only functions explicitly tagged as *extending* in Clause 5 are considered *extending*.

4.13.2 Description

The application shall be able easily to determine whether a DAL supports a base or extended API.

The underlying rule for *extending* functions is that every parcel exposing function is an *extending* function. A function that takes either a parcel or a parcel ID as input, or gives a parcel or a parcel ID as output, is considered a "parcel exposing function".

5 Functional specification of the API

5.1 Introduction and level of API

5.1.1 General

The purpose of this subclause is to establish a framework and an understanding upon which to base the API level in this International Standard.

It was determined that ISO-API level should be defined within the combination layer and the instruction layer levels. To define the API level more clearly, the upper and lower API levels should also be defined as follows. Definition of API levels is shown in Figure 4 below. In this definition, API-5 and API-6 are internal API levels of the application software.

The decision to recommend the ISO-API layer at mostly the combination layer is based on the following criteria:

a) A higher level API lends itself to a smaller number of more complex functions and therefore to more rapid development.

b) A sufficiently high API allows the location of the actual map database to be independent of the application's ability to access the content, be it on-board or off-board.

c) Where necessary, lower level extensions are provisioned for small granularity access (MapDisplay and RoutePlanning).

The problem with the placement of the ISO-API at too low a level (API level 2 and below) is that there is no benefit to an extremely low-level common interface. At the very least, the ISO-API should isolate the application software from the PSF and media, and in addition, should not depend upon a given hardware configuration.

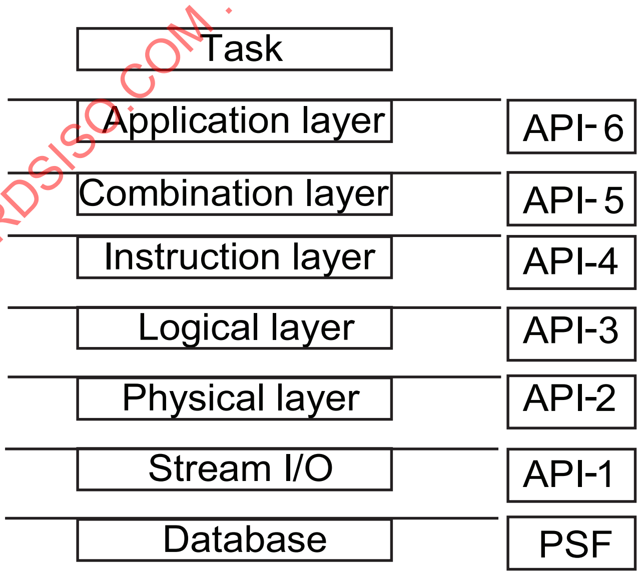


Figure 4 — API levels

5.1.2 Functional definition of the API level

Name of layer	Function of layer
API-6 Application layer	This layer is responsible for high-level system functionality. Examples: Route Planning Vehicle Positioning Route Guidance Map Display
API-5 Combination layer	API-6 layer functions are insulated from lower layers by the API-5 layer. API-5 shades the origin and method of data retrieval. The data returned from the functions in this layer consist of merged lower-level data, and they are returned transparently to the calling function. Examples: Combine data of different types. Get the position of the vehicle on a given link. Return landmark icon codes around a given intersection node. Project map display data for a geodetic coordinate system into the display coordinate system.
API-4 Instruction layer (High logical layer)	API-4 hides data division and data layout from higher layers. Examples: Process data of similar types with multiple access to the physical storage. Process large (memory intensive) data sets. Process data that require multiple searches on the physical media. In such instances, the functions of this particular layer will call the lower level functions multiple times.
API-3 Logical layer (Low logical layer)	API-3 insulates higher layers from the PSF, such as the physical layout of the data structures, like bit fields or word alignment. Functions from this layer convert API-2 PSF into logical structures. Complex operations are done at higher layers. Examples: Return the bounding box of the entire data set. Convert PSF-bytes to structures, including possible byte reordering. Optional data decompression. Convert positions of points from internal representation to latitude/longitude-based values.
API-2 Physical layer	API-2 insulates higher layers from absolute sector addresses. Examples: Read data blocks starting from the specified sector address, up to a given data length. Differentiate according to the file system used.
API-1 Stream I/O layer	Open, read, seek, close. Example: ANSI C file operating functions

5.1.3 ISO-API level policy

The ISO-API functions can be found either at level 5 or at level 4 (see Figure 5).

There are three categories of functions at level 4:

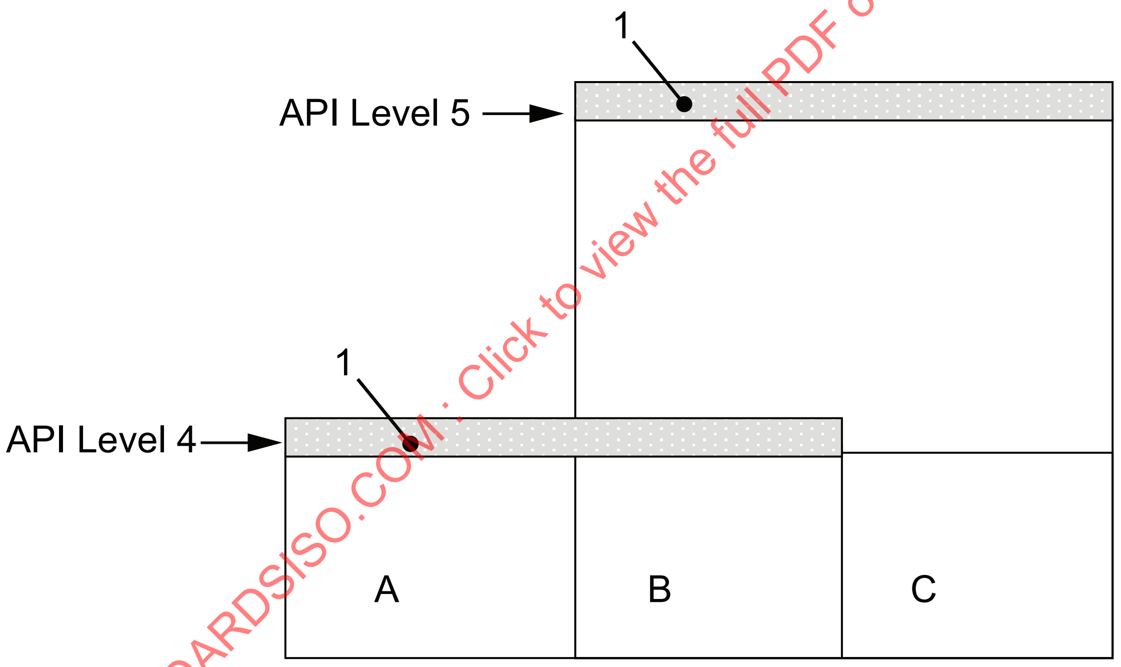
— functions which are visible through the ISO-API but are not used by any ISO-API level 5 functions (category A);

— functions which are visible through the ISO-API and are used by ISO-API level 5 functions (category B);

— functions which are used by ISO-API level 5 functions but are not visible through the ISO-API (category C).

Each function will be placed at the level and category deemed most appropriate.

Any function which reveals any physical detail of the PSF shall not be visible through the API interface. That is, it may only be of category (C).



Key

1 API interface

A level 4 functions which are not used by any level 5 functions

B visible level 4 functions which are used by level 5 functions

C hidden level 4 functions which are used by level 5 functions

Figure 5 — ISO-API level policy

5.2 Specification convention

5.2.1 General

The functional specification of the API is described in the specification for IDL, ISO/IEC 14750. IDL is a system- and implementation-independent description language for software interfaces. Constructs described in this language can be easily implemented by several programming languages, e.g. Java, C++, C.

NOTE This International Standard defines functions and supporting structures, error codes and constants. The structures have been ordered to achieve IDL conformance. The order is therefore not alphabetical.

5.2.2 Naming conventions

5.2.2.1 Additional conventions for the use of IDL to describe the API are specified in 5.2.2.2 to 5.2.2.14.

5.2.2.2 The names of

— classes,

— data types,

— exceptions,

— constants

shall have the prefix "Map".

5.2.2.3 The format of a name is dependent on the type of language element and can be described as follows:

— <ISO-API class name> ::= Map<any class name>

— <ISO-API data type name> ::= Map<any data type name>Type

— <ISO-API exception name> ::= Map<any exception name>Exception

— <ISO-API constant name> ::= Map<any constant name>Const

EXAMPLE class MapRoadElement{...}

5.2.2.4 The name of a member function of a class shall have a prefix and shall start with a capitalized character, e.g. MapGetNode(...).

5.2.2.5 A class member (instance variable) shall start with a capitalized character.

5.2.2.6 Global variables shall be avoided. If really needed they shall be named in the same manner as instances.

5.2.2.7 The single enumerators of an enumeration type shall have a three-digit prefix identifying the enumeration type they belong to.

EXAMPLE

```
// IDL

typedef enum {AngStraight, AngRight, ... } MapAngleType;
```

5.2.2.8 In names which consist of more than one word, the words shall be written together and each word that follows the prefix shall begin with an uppercase letter.

5.2.2.9 Whenever possible, entire words or syllables shall be preferred instead of abbreviations.

5.2.2.10 For data elements representing any kind of collection, plurals shall be used rather than inventing new names.

5.2.2.11 Member function names shall be created by using the order Verb/Object, e.g. GetNodeList().

5.2.2.12 The maximum length of a name shall be less than or equal to 32 characters.

5.2.2.13 IDL templates may not be used for the ISO-API because they cannot easily be ported to programming languages such as Java.

5.2.2.14 Multiple inheritance may not be used for the ISO-API because it is not supported by most programming languages.

5.2.3 Hungarian notation convention

Each function argument in this International Standard begins with a prefix which indicates the argument's type. (The practice of using such prefixes is called "Hungarian notation" after the nationality of its inventor, Charles Simonyi.) Implementers and users of this International Standard are encouraged to use these prefixes as well.

Table 2 lists the Hungarian prefixes of the basic data types.

Table 2 — Hungarian prefixes of basic data types

Data type	Prefix
`Any`	`a`
`Boolean`	`b`
`Char`	`c`
`Long`	`l`
`Octet`	`o`
`Short`	`s`
`Unsigned long`	`ul`
`Unsigned short`	`us`

Table 3 lists, alphabetically by data type, the Hungarian prefixes of the data types defined in this International Standard.

Table 3 — Hungarian prefixes of data types defined in this International Standard

Data type	Prefix
MapAbsoluteDateType	dta
MapAccessVehicleType	av
AddressFieldStatusEnum	afse
MapConditionAndDirectionType	cod
MapConditionAttrType	ca
MapConditionCategoryType	cc
MapConditionLinkAndNodeType	cln
MapConditionModifierEnum	cme
MapConditionModifierType	cm
MapConditionType	cnd
MapCursorInfoType	ci
MapCursorOriginEnum	coe
MapCursorReturnTypeEnum	crte
MapCursorType	cu
MapDateAttrType	da
MapDateTimeType	dtt
MapDateType	dt
MapDateTypeEnum	dte
MapDayOfMonthOfYearType	dmy
MapDayOfWeekEnum	dwe
MapDayOfWeekOfMonthOfYearType	dwmy
MapDayOfWeekOfMonthType	dwm
MapDayOfWeekOfYearType	dwy
MapDBIDType	dbid
MapEntityEnum	ee
MapEntityIDType	eid
FeatureTypeEnum	fte
MapFetchDirectionEnum	fde
MapLaneCategoryType	lc
MapLinkAttrType	la
MapLinkCharacteristicsType	lch
MapLinkSubattrType	ls
MapLinkType	lk
MapNodeAttrType	na
MapNodeType	nd
MapOrientationDirectionEnum	ode

Table 3 (*continued*)

Data type	Prefix
MapParcelIDType	pid
MapPosition2Dtype	p2
MapPosition3Dtype	p3
MapPriorityType	pr
MapRectangleType	rect
MapReturnType	ret
MapRouteControlType	rc
MapRouteCostModelType	rcm
MapRouteUsageEnum	rue
MapRouteLinkAndCostType	rlc
MapRouteMinimizeOptionEnum	rmoe
MapRoutePointType	rp
MapRoutePointSequenceType	rps
MapRouteDynamicTrafficUsageEnum	rtue
MapSectionIDType	sid
MapSpeedCategoryType	sc
MapSuccessorLinkAndNodeType	sln
MapSuccNodeCostEnum	snce
MapTimeZoneType	tz
MapTollType	tol

Table 4 lists, alphabetically by prefix, the data types corresponding to all Hungarian prefixes used in this International Standard.

Table 4 — Hungarian prefixes used in this International Standard

Prefix	Data type
a	any
afse	AddressFieldStatusEnum
av	MapAccessVehicleType
b	boolean
c	char
ca	MapConditionAttrType
cc	MapConditionCategoryType
ci	MapCursorInfoType
cln	MapConditionLinkAndNodeType
cm	MapConditionModifierType

Table 4 (*continued*)

Prefix	Data type
cme	MapConditionModifierEnum
cnd	MapConditionType
cod	MapConditionAndDirectionType
coe	MapCursorOriginEnum
crte	MapCursorReturnTypeEnum
cu	MapCursorType
da	MapDateAttrType
dbid	MapDBIDType
dmy	MapDayOfMonthOfYearType
dt	MapDateType
dta	MapAbsoluteDateType
dte	MapDateTypeEnum
dtt	MapDateTimeType
dwe	MapDayOfWeekEnum
dwm	MapDayOfWeekOfMonthType
dwmy	MapDayOfWeekOfMonthOfYearType
dwy	MapDayOfWeekOfYearType
ee	MapEntityEnum
eid	MapEntityIDType
fde	MapFetchDirectionEnum
fte	FeatureTypeEnum
l	long
la	MapLinkAttrType
lc	MapLaneCategoryType
lch	MapLinkCharacteristicsType
lk	MapLinkType
ls	MapLinkSubattrType
na	MapNodeAttrType
nd	MapNodeType
o	octet
ode	MapOrientationDirectionEnum
p2	MapPosition2Dtype
p3	MapPosition3Dtype
pid	MapParcelIDType
pr	MapPriorityType
rc	MapRouteControlType

Table 4 (*continued*)

Prefix	Data type
rcm	MapRouteCostModelType
rect	MapRectangleType
ret	MapReturnType
rue	MapRouteUsageEnum
rlc	MapRouteLinkAndCostType
rmoe	MapRouteMinimizeOptionEnum
rp	MapRoutePointType
rps	MapRoutePointSequenceType
rtue	MapRouteDynamicTrafficUsageEnum
s	short
sc	MapSpeedCategoryType
sid	MapSectionIDType
sln	MapSuccessorLinkAndNodeType
snce	MapSuccNodeCostEnum
tol	MapTollType
tz	MapTimeZoneType
ul	unsigned long
us	unsigned short

The following principles were followed in defining the Hungarian prefixes listed above:

— All (and only) basic data types not containing the word “unsigned” have single-letter prefixes.

— All (and only) basic data types containing the word “unsigned” have prefixes beginning with the letter “u”. The prefixes of these types consist of the letter “u” followed by the prefix of the corresponding signed type.

— All (and only) enumerated types (“enums”) have prefixes ending with the letter “e”.

— Prefixes have been kept as short as possible, consistent with reasonable clarity. No prefix is more than four characters long.

5.3 Application categories

5.3.1 General

The global structure of the API will be specified by using the IDL entities *module* and *interface*.

A *module* is used to group a related set of definitions, e.g. the module "MapRoutePlanning" contains all *interfaces* of the application area Route Planning.

An *interface* describes the set of access functions for accessing a logical data model entity and its relationships to other entities. For example, the interface MapLink contains all access functions that provide the information on a given Link as specified in the requirements drafts, e.g. the requirements starting with "For a given Link ...".

5.3.2 Global module specification

The sub-modules of the API are specified below. Initially, for each application area, a sub-module is specified. In a later stage some modules may be merged depending on the definition of conceptual data sets in the Logical Data Model.

```
module MapAPI {

    module MapRoutePlanning;

    module MapRouteGuidance;

    module MapPositioning;

    module MapDisplay;

    module MapAddressLocation;

    module MapService;

    module MapGeneral;

}
```

5.3.3 Definitions common to all functional categories

5.3.3.1 Constants

5.3.3.1.1 Vehicle access type constants

The following constants are to be used as components of a 32-bit long mask indicating which vehicles are being addressed. The left-most bit is the "tone" designator. If it is set, then the tone of the mask is "all but some vehicle"; if it is not set, then the tone is "some vehicle". When the "all but" tone is set, the vehicles whose bits are "0" are the ones that are referred to. The second bit from the left is the "all" bit. If it is set, then there is no need to look at individual vehicular bits, it simply states that all entities are selected (or on). Bits 12 through 15 are reserved for additional vehicle types and bits 26 through 29 are reserved for additional control uses in future versions of this International Standard. These 8 bits shall not be used in conjunction with the current version of this International Standard.

```
typedef unsigned long LONG_BITMASK;

typedef LONG_BITMASK V_BITMASK;

                                      // 1000 0000 0000 0000 0000 0000 0000 0000
                                      // (0x1 << 31)
const V_BITMASK ALL_BUT_MASK          = 0x80000000;

                                      // 0100 0000 0000 0000 0000 0000 0000 0000
                                      // (0x1 << 30)
const V_BITMASK ALL_ENTITIES_MASK     = 0x40000000;

                                      // 0000 0000 0000 0000 0000 0000 0000 0001
                                      // (0x1 << 0)
const V_BITMASK AUTOMOBILES           = 0x00000001;

                                      // 1011 1111 1111 1111 1111 1111 1111 1110
                                      // (~ALL_ENTITIES_MASK) & (~AUTOMOBILES)
const V_BITMASK ALL_BUT_AUTOMOBILES   = 0xbffffffe;
```

```
                                        // 0000 0000 0000 0000 0000 0000 0000 0010
                                        // (0x1 << 1)
const V_BITMASK BUSES                   = 0x00000002;

                                        // 1011 1111 1111 1111 1111 1111 1111 1101
                                        // (~ALL_ENTITIES_MASK) & (~BUSES)
const V_BITMASK ALL_BUT_BUSES           = 0xbffffffd;


                                        // 0000 0000 0000 0000 0000 0000 0000 0100
                                        // (0x1 << 2)
const V_BITMASK TAXIS                   = 0x00000004;

                                        // 1011 1111 1111 1111 1111 1111 1111 1011
                                        // (~ALL_ENTITIES_MASK) & (~TAXIS)
const V_BITMASK ALL_BUT_TAXIS           = 0xbffffffb;


                                        // 0000 0000 0000 0000 0000 0000 0000 1000
                                        // (0x1 << 3)
const V_BITMASK HOVS                    = 0x00000008;   // Carpools

                                        // 1011 1111 1111 1111 1111 1111 1111 0111
                                        // (~ALL_ENTITIES_MASK) & (~HOVS)
const V_BITMASK ALL_BUT_HOVS            = 0xbffffff7;


                                        // 0000 0000 0000 0000 0000 0000 0001 0000
                                        // (0x1 << 4)
const V_BITMASK PEDESTRIANS             = 0x00000010;

                                        // 1011 1111 1111 1111 1111 1111 1110 1111
                                        // (~ALL_ENTITIES_MASK) & (~PEDESTRIANS)
const V_BITMASK ALL_BUT_PEDESTRIANS     = 0xbfffffef;


                                        // 0000 0000 0000 0000 0000 0000 0010 0000
                                        // (0x1 << 5)
const V_BITMASK BICYCLES                = 0x00000020;

                                        // 1011 1111 1111 1111 1111 1111 1101 1111
                                        // (~ALL_ENTITIES_MASK) & (~BICYCLES)
const V_BITMASK ALL_BUT_BICYCLES        = 0xbfffffdf;


                                        // 0000 0000 0000 0000 0000 0000 0100 0000
                                        // (0x1 << 6)
const V_BITMASK TRUCKS                  = 0x00000040;

                                        // 1011 1111 1111 1111 1111 1111 1011 1111
                                        // (~ALL_ENTITIES_MASK) & (~TRUCKS)
const V_BITMASK ALL_BUT_TRUCKS          = 0xbfffffbf;

                                        // 0000 0000 0000 0000 0000 0000 1000 0000
                                        // (0x1 << 7)
const V_BITMASK RESIDENT_AND_GUESTS     = 0x00000080;

                                        // 1011 1111 1111 1111 1111 1111 0111 1111
                                // (~ALL_ENTITIES_MASK) & (~RESIDENT_AND_GUESTS);
const V_BITMASK ALL_BUT_RESIDENT_AND_GUESTS
                                        = 0xbfffff7f;
```

```
                                          // 0000 0000 0000 0000 0000 0001 0000 0000
                                          // (0x1 << 8)
const V_BITMASK DELIVERIES                = 0x00000100;

                                          // 1011 1111 1111 1111 1111 1110 1111 1111
                                          // (~ALL_ENTITIES_MASK) & (~DELIVERIES)
const V_BITMASK ALL_BUT_DELIVERIES        = 0xbffffeff;


                                          // 0000 0000 0000 0000 0000 0010 0000 0000
                                          // (0x1 << 9)
const V_BITMASK SCHOOL_BUSES              = 0x00000200;

                                          // 1011 1111 1111 1111 1111 1101 1111 1111
                                          // (~ALL_ENTITIES_MASK) & (~SCHOOL_BUSES)
const V_BITMASK ALL_BUT_SCHOOL_BUSES      = 0xbffffdff;
                                          // 0000 0000 0000 0000 0000 0100 0000 0000
                                          // (0x1 << 10)
const V_BITMASK MOTORCYCLES               = 0x00000400;

                                          // 1011 1111 1111 1111 1111 1011 1111 1111
                                          // (~ALL_ENTITIES_MASK) & (~MOTORCYCLES)
const V_BITMASK ALL_BUT_MOTORCYCLES       = 0xbffffbff;

                                          // 0000 0000 0000 0000 0000 1000 0000 0000
                                          // (0x1 << 11)
const V_BITMASK AUTHORIZED_VEHICLES       = 0x00000800;

                                          // 1011 1111 1111 1111 1111 0111 1111 1111
                                    // (~ALL_ENTITIES_MASK) & (~AUTHORIZED_VEHICLES)
const V_BITMASK ALL_BUT_AUTHORIZED_VEHICLES
                                          = 0xbffff7ff;


                                          // 0000 0000 0000 0001 0000 0000 0000 0000
                                          // (0x1 << 16)
const V_BITMASK Reserved_VEHICLES_01      = 0x00010000;

                                          // 1011 1111 1111 1110 1111 1111 1111 1111
                                    // (~ALL_ENTITIES_MASK) & (~RESERVED_VEHICLES)
const V_BITMASK ALL_BUT_ Reserved_VEHICLES_01
                                          = 0xbffeffff;

                                          // 0000 0000 0000 0010 0000 0000 0000 0000
                                          // (0x1 << 17)
const V_BITMASK Reserved_VEHICLES_02      = 0x00020000;

                                          // 1011 1111 1111 1101 1111 1111 1111 1111
                                    // (~ALL_ENTITIES_MASK) & (~RESERVED_VEHICLES)
const V_BITMASK ALL_BUT_ Reserved_VEHICLES_02
                                          = 0xbfffdffff;


                                          // 0000 0000 0000 0100 0000 0000 0000 0000
                                          // (0x1 << 18)
const V_BITMASK Reserved_VEHICLES_03      = 0x00040000;

                                          // 1011 1111 1111 1011 1111 1111 1111 1111
                                    // (~ALL_ENTITIES_MASK) & (~RESERVED_VEHICLES)
const V_BITMASK ALL_BUT_ Reserved_VEHICLES_03
                                          = 0xbffbffff;
```

```
                                      // 0000 0000 0000 1000 0000 0000 0000 0000
                                      // (0x1 << 19)
const V_BITMASK Reserved_VEHICLES_04   = 0x00080000;

                                      // 1011 1111 1111 0111 1111 1111 1111 1111
                                 // (~ALL_ENTITIES_MASK) & (~RESERVED_VEHICLES)
const V_BITMASK ALL_BUT_ Reserved_VEHICLES_04
                                       = 0xbff7ffff;


                                      // 0000 0000 0001 0000 0000 0000 0000 0000
                                      // (0x1 << 20)
const V_BITMASK Reserved_VEHICLES_05   = 0x00100000;

                                      // 1011 1111 1110 1111 1111 1111 1111 1111
                                 // (~ALL_ENTITIES_MASK) & (~RESERVED_VEHICLES)
const V_BITMASK ALL_BUT_ Reserved_VEHICLES_05
                                       = 0xbfefffff;


                                      // 0000 0000 0010 0000 0000 0000 0000 0000
                                      // (0x1 << 21)
const V_BITMASK Reserved_VEHICLES_06   = 0x00200000;

                                      // 1011 1111 1101 1111 1111 1111 1111 1111
                                 // (~ALL_ENTITIES_MASK) & (~RESERVED_VEHICLES)
const V_BITMASK ALL_BUT_ Reserved_VEHICLES_06
                                       = 0xbfdfffff;


                                      // 0000 0000 0100 0000 0000 0000 0000 0000
                                      // (0x1 << 22)
const V_BITMASK Reserved_VEHICLES_07   = 0x00400000;

                                      // 1011 1111 1011 1111 1111 1111 1111 1111
                                 // (~ALL_ENTITIES_MASK) & (~OPTION_VEHICLES)
const V_BITMASK ALL_BUT_ Reserved_VEHICLES_07
                                       = 0xbfbfffff;


                                      // 0000 0000 1000 0000 0000 0000 0000 0000
                                      // (0x1 << 23)
const V_BITMASK Reserved_VEHICLES_08   = 0x00800000;

                                      // 1011 1111 0111 1111 1111 1111 1111 1111
                                 // (~ALL_ENTITIES_MASK) & (~OPTION_VEHICLES)
const V_BITMASK ALL_BUT_ Reserved_VEHICLES_08
                                       = 0xbf7fffff;


                                      // 0000 0001 0000 0000 0000 0000 0000 0000
                                      // (0x1 << 24)
const V_BITMASK Reserved_VEHICLES_09   = 0x01000000;

                                      // 1011 1110 1111 1111 1111 1111 1111 1111
                                 // (~ALL_ENTITIES_MASK) & (~OPTION_VEHICLES)
const V_BITMASK ALL_BUT_ Reserved_VEHICLES_09
                                       = 0xbeffffff;
```

```
                                         // 0000 0010 0000 0000 0000 0000 0000 0000
                                         // (0x1 << 25)
const V_BITMASK Reserved_VEHICLES_10     = 0x02000000;

                                         // 1011 1101 1111 1111 1111 1111 1111 1111
                                    // (~ALL_ENTITIES_MASK) & (~OPTION_VEHICLES)
const V_BITMASK ALL_BUT_ Reserved_VEHICLES_10
                                         = 0xbdffffff;

                                         // 0011 1111 1111 1111 1111 1111 1110 1111
const V_BITMASK ALL_MOTOR_VEHICLES       = 0x3fffffef;    // Without pedestrians

                                         // 1000 0000 0000 0000 0000 0000 0001 0000
const V_BITMASK ALL_BUT_ALL_MOTOR_VEHICLES // Same as pedestrians (except tone)
                                         = 0x80000010;

                                         // 0111 1111 1111 1111 1111 1111 1111 1111
const V_BITMASK ALL_VEHICLE_TYPES        = 0x7fffffff;    // Includes pedestrians!

                                         // 1100 0000 0000 0000 0000 0000 0000 0000
const V_BITMASK ALL_BUT_ALL_VEHICLE_TYPES      // Same as none (except for tone)
                                         = 0xc0000000;


typedef V_BITMASK MapAccessVehicleType;        // (Hungarian av)
```

5.3.3.1.2 Condition type constants

```
typedef LONG_BITMASK C_BITMASK;

// Prohibited/Allowed single link access or prohibited/mandatory multiple links
// maneuver.
const C_BITMASK TRAFFIC_COND                        = 0x00000001;

// Information as to whether road is open or closed during construction.
const C_BITMASK CONSTRUCTION_STATUS_COND            = 0x00000002;

// Describe monetary cost requirement for travel.
const C_BITMASK TOLL_COND                           = 0x00000004;

// Travel blocked by physical gate.
const C_BITMASK GATE_COND                           = 0x00000008;

// Resident and guest regulation.
const C_BITMASK RES_AND_GUEST_COND                  = 0x00000010;

// This value is defined to be used by the MapGetLinkConditions() function for
// requesting conditions of any type.
const C_BITMASK ANY_COND                            = 0xFFFFFFFF;

typedef C_BITMASK MapConditionCategoryType;      // (Hungarian cc)
```

5.3.3.2 Data structures

5.3.3.2.1 LocusType (Hungarian: loc)

This structure defines a position along a link (it can also be just the link).

```
typedef struct locusType_s {

    MapEntityIDType    link;

    // normalized [0.0, 1.0] fraction
    // "< 0" means "the entire link" or "unknown",
    // whichever means something in the context.
    FractionType       positionAlongLink;

    // [LHS, RHS, both/either, unspecified]
    SideTypeEnum       side;
} LocusType;
```

5.3.3.2.2 LocusList

This structure defines a list of loci for use as an origin, destination, or crisp waypoint. The order of elements in this list is not significant.

```
typedef struct locusList_s {

    int                nLoci;

    // list of loci (or set of pointers to loci)
    LocusType    <sequence>       loci;
} LocusList;
```

5.3.3.2.3 bbox3PtType

This structure defines a bounding box using 3 points. A bounding box is used to frame requests of data. It is assumed to be a rectangle in some orientation and is described as 3 points (topLeft, bottomLeft, bottomRight). If the 3 points are not corners of the rectangle, the result will be the smallest rectangle such that one edge of the rectangle contains the "bottomLeft" point and the "bottomRight" point and the rectangle contains the "topLeft" point.

```
typedef struct bbox3PtType_s {

    MapPosition2DType     lowerLeftCorner;
    MapPosition2DType     upperLeftCorner;
    MapPosition2DType     lowerRightCorner;
} bbox3PtType;
```

5.3.3.2.4 GeometryTypeEnum

This structure defines geometry types.

```
typedef enum GeometryTypeEnum_e {

    Shape   = -1,
    Point   =  0,
    Line    =  1,
    Area    =  2,
    Complex =  3
} GeometryTypeEnum;
```

5.3.3.2.5 FeatureNameType

This structure defines a single name.

```
typedef struct FeatureNameType_s {
    char[3]         charset;  // charset of the name Marc code

    string          name;     // the name itself

    short           symbol;   // a symbol associated with the name
                              // like a highway shield
} FeatureNameType;
```

5.3.3.2.6 PointDimensionEnum

If shape points have 2 or 3 dimensions of data.

```
typedef enum pointDimensionEnum_e {

   D2,
   D3
} pointDimensionEnum;
```

5.3.3.2.7 MapPositionType

If shape points have 2 or 3 dimensions of data.

```
typedef struct mapPositionType_s {


   union dimensionType switch (pointDimensionEnum) {
     case D2:       MapPosition2Dtype   point;
     case D3:       MapPosition3Dtype point;
          };
} mapPositionType;
```

5.3.3.2.8 ComponentBoundaryType

This structure defines a single component boundary.

```
typedef struct ComponentBoundaryType_s {

    short       nPts;     // how many points in the boundaries

    MapPositionType <sequence>    point; // the actual list of points



} ComponentBoundaryType;
```

5.3.3.2.9 FeatureComponentType

This structure defines a single feature component.

```
typedef struct FeatureComponentType_s {

    short              nBnds;     // how many boundaries
    ComponentBoundaryType <sequence>
                                 boundary;      // the boundaries

    GeometryTypeEnum      componentDimension;
                                                    // dimension of the
                                                  component
} FeatureComponentType;
```

5.3.3.2.10 MapFeatureType

This structure defines a single mapFeature.

```
typedef struct MapFeatureType_s {

    short              nCmps;     // how many components

    FeatureComponentType  <sequence>
                                 component;     // the components

    GeometryTypeEnum      dimension;       // feature geometrical type

    featureTypeEnum    featureType;

    short              featureClass;

    unsigned long      flags;    // various encodings (more later)
                                                    // (like 1-way)

    FeatureNameType    name;          // name structure

    MapIDtype          mapID;     // which map is the feature from?

} MapFeatureType;
```

5.3.3.2.11 FeatureTypeEnum (Hungarian: fte)

Structure containing feature information. This type is used to reverse-geocode non-street and non-intersection map features such as landmarks, POIs, parks, lakes, etc.

```
typedef enum featureTypeEnum_e {
```

SupraNationalArea	= 1110,
Country	= 1111,
Order-1Area	= 1112,
Order-2Area	= 1113,
Order-3Area	= 1114,
Order-4Area	= 1115,

```
Order-5Area                          = 1116,
Order-6Area                          = 1117,
Order-7Area                          = 1118,
Order-8Area                          = 1119,
Order-9Area                          = 1120,
AdministrativePlaceA                 = 1165,
AdministrativePlaceB                 = 1166,
AdministrativePlaceC                 = 1167,
AdministrativePlaceD                 = 1168,
AdministrativePlaceE                 = 1169,
AdministrativePlaceF                 = 1170,
AdministrativePlaceG                 = 1171,
AdministrativePlaceH                 = 1172,
AdministrativePlaceI                 = 1173,
AdministrativePlaceJ                 = 1174,
AdministrativePlaceK                 = 1175,
AdministrativePlaceL                 = 1176,
AdministrativePlaceM                 = 1177,
AdministrativePlaceN                 = 1178,
AdministrativePlaceO                 = 1179,
AdministrativePlaceP                 = 1180,
AdministrativePlaceQ                 = 1181,
AdministrativePlaceR                 = 1182,
AdministrativePlaceS                 = 1183,
AdministrativePlaceT                 = 1184,
AdministrativePlaceU                 = 1185,
AdministrativePlaceV                 = 1186,
AdministrativePlaceW                 = 1187,
AdministrativePlaceX                 = 1188,
AdministrativePlaceY                 = 1189,
AdministrativePlaceZ                 = 1190,
AdministrativeBoundaryJunction       = 1198,
AdministrativeBoundaryElement        = 1199,
BuiltUpArea                          = 3110,
NamedArea                            = 3120,
PoliceDistrict                       = 3131,
EmergencyMedicalDispatchDistrict     = 3132,
SchoolDistrict                       = 3133,
CensusDistrict                       = 3134,
FireDispatchDistrict                 = 3135,
PostalDistrict                    = 3136,
PhoneDistrict                     = 3137,
ElectiveDistrict                  = 3138,
BoundaryJunction                  = 3198,
BoundaryElement                   = 3199,
RoadElement                       = 4110,
Junction                          = 4120,
FerryConnection                   = 4130,
EnclosedTrafficArea               = 4135,
Road                              = 4140,
Intersection                      = 4145,
Ferry                             = 4150,
AddressArea                       = 4160,
AddressAreaBoundaryElement        = 4165,
AggregatedWay                     = 4170,
Interchange                       = 4180,
Roundabout                        = 4190,
RailwayElement                    = 4210,
RailwayElementJunction            = 4220,
WaterBody                         = 4310,
WaterBoundaryElement              = 4330,
WaterBoundaryJunction             = 4335,
InlandWater                       = 4350,
Reservoir                         = 4351,
Lake                              = 4352,
WaterCourse                       = 4353,
Canal                             = 4354,
River                             = 4355,
MarineWater                       = 4370,
CoastalLagoon                     = 4371,
Estuary                           = 4372,
SeaandOcean                       = 4373,
ChainageReferencingSection        = 4910,
ReferencePoint                    = 4920,
RouteLink                         = 5010,
PublicTransportJunction           = 5015,
StopPoint                         = 5020,
PublicTransportPoint              = 5025,
StopArea                          = 5030,
Route                             = 5040,
Line                                      = 5050,
Building                                  = 7110,
ArtificialSurface                         = 7111,
UrbanFabric                               = 7112,
IndustrialCommercialandTransportUnit      = 7113,
MineDumpandConstructionSite               = 7114,
ArtificialNonAgriculturalVegetationArea   = 7115,
ContinuousUrbanFabric                     = 7116,
DiscontinuousUrbanFabric                  = 7117,
IndustrialorCommercialUnit                = 7118,
RoadAndRailNetAndAssociatedLand           = 7119,
PortArea                                  = 7130,
Airport                                   = 7131,
MineralExtractionSite                     = 7132,
DumpSite                                  = 7133,
ConstructionSite                          = 7134,
GreenUrbanArea                            = 7135,
SportAndLeisureFacility                   = 7136,
AgriculturalArea                          = 7137,
ArableLand                                = 7138,
PermanentCrop                             = 7139,
Pasture                                   = 7140,
HeterogeneousAgriculturalArea             = 7142,
NonIrrigatedArableLand                    = 7143,
PermanentlyIrrigatedLand                  = 7144,
RiceField                                 = 7145,
Vineyard                                  = 7146,
FruitTreeAndBerryPlantation               = 7147,
OliveGrove                                = 7148,
AnnualCropAssociatedWithPermCrop          = 7149,
ComplexCultivationPattern                 = 7150,
LandPrincipallyAgWithSignifNaturalVeg     = 7152,
AgroForestryArea                          = 7153,
ForestAndSemiNaturalArea                  = 7154,
Forest                                    = 7155,
ScrubAndOrHerbaceousVegetation            = 7156,
OpenSpaceWithLittleOrNoVegetation         = 7157,
BroadLeavedForest                         = 7158,
ConiferousForest                          = 7159,
MixedForest                               = 7160,
            NaturalGrassland                    = 7161,
            MoorAndHeathland                    = 7162,
            SclerophyllousVegetation            = 7163,
            TransitionalWoodlandScrub           = 7164,
            BeachDuneAndSandPlain               = 7165,
            BareRock                            = 7166,
            SparselyVegetatedArea               = 7167,
            BurntArea                           = 7168,
            GlaciersAndPerpetualSnow            = 7169,
            ParkGarden                          = 7170,
            Island                              = 7180,
            Signpost                            = 7210,
            TrafficSign                         = 7220,
            TrafficLight                        = 7230,
            PedestrianCrossing                  = 7240,
            EnvironmentalEquipment              = 7251,
            Lighting                            = 7252,
            MeasurementDevice                   = 7253,
            RoadMarkings                        = 7254,
            SafetyEquipment                     = 7255,
            EntryPointofService                 = 7300,
            Structure                           = 7500,
            CentrePointOfFeature                = 8000,
            TrafficLocation                     = 8001,
            UserDefined_000                     = 9000,
            UserDefined_001                     = 9001,
            // fill in the blanks in between…
            UserDefined_999                     = 9999
    } FeatureTypeEnum;
```

5.3.3.2.12 FractionType

This type is used to denote a point a fraction of the way down a link. Because some platforms provide much faster computation with integers than with floating-point values, a long integer is used, with the value n denoting $n/2^{24}$. For example, the value 0 denotes $0/2^{24}$ of the way down the link (the beginning of the link); the value 0x1 000 000 = 2^{24} denotes $2^{24}/2^{24}$ of the way down the link (the end of the link); and the value 0x400 000 = 2^{22} denotes $2^{22}/2^{24}$ = 1/4 of the way down the link.

Twenty-four of the available 32 bits are used because using 24 bits allows enough accuracy for any purpose, since links are divided into 2^{24} = 16 777 216 parts (if the link is 40 000 km long, the circumference of the earth, $1/2^{24}$ of the link is still only 2,38 m). Allowing eight bits of room at the top makes it easy to construct integer computations without concern for overflow.

```
    // interpreted in units of 1/2^24
    typedef long int FractionType;
```

5.3.3.2.13 featureFilterType

This structure specifies a complex list of features and/or attribute values to be included in (+) or excluded from (-) a search.

Filters shall support a wide variety of selections of feature types, classes, and attributes. Rather than defining a complex language for specifying such selections, this International Standard specifies simple data structures which can be combined in a variety of complex ways.

The data structures used to define filters consist of “operands” and “operators”. An operand can be a basic specification, such as one meaning “feature type = 4110” — this kind of atomic specification is called a “leaf”, because all leaves of the tree specifying a filter are of this type — or it can be the result of an operation. Three operations are supported to combine or negate operands, namely AND, OR, and NOT.

Specifically, an operandType is a union which contains as a subordinate structure either a leafType or an operatorType. A leafType describes a basic filtering operation. An operatorType contains an operator, which is either AND, OR, or NOT, and pointers to one or two operands (two for AND and OR, one for NOT). The semantics of these operators are exactly as one would expect, as shown in Table 5.

Table 5 — operatorType

If operatorType = …	**… then the feature passes the test if and only if …**
AND	… it passes the tests of both leftOperand and rightOperand.
OR	… it passes the test of either leftOperand or rightOperand (or both).
NOT	… it does not pass the test of rightOperand. (The field leftOperand is not used.)

A leafType, which describes a basic filtering operation, has several fields. One of the fields, valueTypeFromFeature, determines with which property of the feature each feature is tested, as shown in Table 6.

Table 6 — valueTypeFromFeature

If valueTypeFromFeature = …	**… then the following is compared:**
FT	feature type
FC	feature class
AV	attribute value specified by the attributeType field

Another field is valueTypeForComparison. This determines what kind of value the property of the feature is to be compared to, and which field(s) of the leafType are used for the comparison, as shown in Table 7.

Table 7 — valueTypeForComparison

If valueTypeForComparison = …	… which stands for …	… then the comparison is to …
NUM	numeric	num1
STR	string	stringValue
NUM_INT	numeric interval	interval from num1 through num2 inclusive
NUM_SET	numeric set	intSet (which contains setCount elements)
STR_SET	string set	stringSet (which contains setCount elements)
NO	not used	no comparison value (valid only when testType = EX, "exists" — see below)

Another field is testType. This field specifies what kind of comparison is to be performed between the property of the feature, f, and the value type for comparison, v, as shown in Table 8.

Table 8 — testType

If testType = …	… which stands for …	… then the feature passes the filter if and only if …	Valid types for v
EQ	equal to	$f = v$	NUM, STR
NE	not equal to	$f \neq v$	NUM, STR
LT	less than	$f < v$	NUM, STR
LE	less than or equal to	$f \leq v$	NUM, STR
GE	greater than or equal to	$f \geq v$	NUM, STR
GT	greater than	$f > v$	NUM, STR
CS	contains substring	f contains v as a substring	STR
SS	starts with substring	f begins with v	STR
EO	element of	f is an element of v	NUM_SET, STR_SET
NO	not element of	f is not an element of v	NUM_SET, STR_SET
EX	exists	f (which must be an attribute) exists	NO

This is sufficiently complex that it is useful to show some examples. Because the leaf data structures require a lot of information, the tree structures are shown schematically in the figures and the leaf data structures are shown in the text. Fields not used in a given leaf data structure are omitted.

EXAMPLE 1 All features with feature type 4110. See Figure 6. Leaf L1 is as follows:

L1	
Field	**Value**
valueTypeFromFeature	FT
valueTypeForComparison	NUM
num1	4110
testType	EQ

EXAMPLE 2 All features with feature type 4110 or 4120. See Figure 6. Leaf L1 is as follows:

L1	
Field	**Value**
valueTypeFromFeature	FT
valueTypeForComparison	NUM_SET
setCount	2
intSet	{4110, 4120}
testType	EQ

EXAMPLE 3 All features except those with feature type 4110. See Figure 6. Leaf L1 is as follows:

L1	
Field	**Value**
valueTypeFromFeature	FT
valueTypeForComparison	NUM
num1	4110
testType	NE

EXAMPLE 4 All features except those with feature type 4110 or 4120. See Figure 6. Leaf L1 is as follows:

L1	
Field	**Value**
valueTypeFromFeature	FT
valueTypeForComparison	NUM_SET
setCount	2
intSet	{4110, 4120}
testType	NO

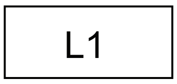


Figure 6 — All features with feature type 4110

EXAMPLE 5 All features with feature type 4110 and feature class 3. See Figure 7. Operator O1 is AND. Leaves L1 and L2 are as follows:

L1	
Field	**Value**
valueTypeFromFeature	FT
valueTypeForComparison	NUM
num1	4110
testType	EQ

L2	
Field	**Value**
valueTypeFromFeature	FC
valueTypeForComparison	NUM
num1	3
testType	EQ

EXAMPLE 6 All features with feature type 4110 and feature class 3 or 4. See Figure 7. Operator O1 is AND. Leaves L1 and L2 are as follows:

L1	
Field	**Value**
valueTypeFromFeature	FT
valueTypeForComparison	NUM
num1	4110
testType	EQ

L2	
Field	**Value**
valueTypeFromFeature	FC
valueTypeForComparison	NUM_SET
setCount	2
intSet	{3, 4}
testType	EO

EXAMPLE 7 All features with feature type 4110 and feature class 3 through 6. See Figure 7. Operator O1 is AND. Leaves L1 and L2 are as follows:

L1	
Field	**Value**
valueTypeFromFeature	FT
valueTypeForComparison	NUM
num1	4110
testType	EQ

L2	
Field	**Value**
valueTypeFromFeature	FC
valueTypeForComparison	NUM_INT
num1	3
num2	6
testType	EO

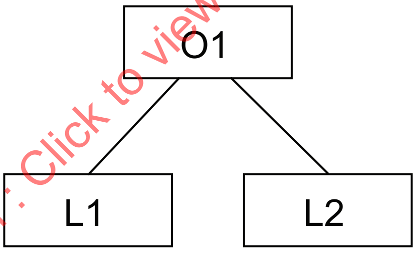


Figure 7 — All features with feature type 4110 and feature class 3

EXAMPLE 8 All features with feature type 4110 and feature class 1 through 3 or 5 through 7. See Figure 8. Operand O1 is AND; operator O2 is OR. Leaves L1, L2, and L3 are as follows:

L1	
Field	**Value**
valueTypeFromFeature	FT
valueTypeForComparison	NUM
num1	4110
testType	EQ

L2	
Field	**Value**
valueTypeFromFeature	FC
valueTypeForComparison	NUM_INT
num1	1
num2	3
testType	EO

L3	
Field	**Value**
valueTypeFromFeature	FC
valueTypeForComparison	NUM_INT
num1	5
num2	7
testType	EO

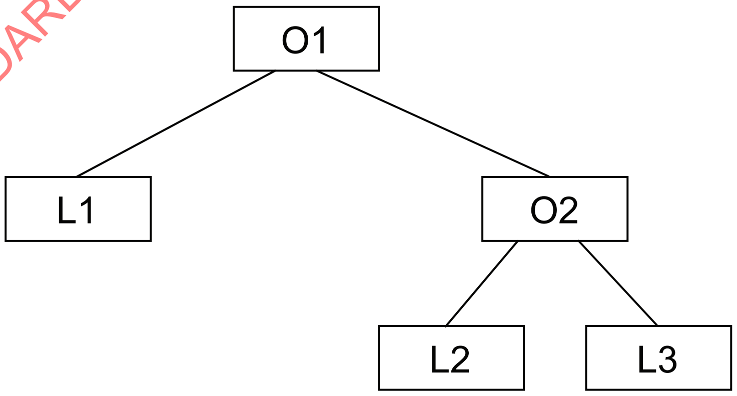


Figure 8 — All features with feature type 4110 and feature class 1 through 3, or 5 through 7

EXAMPLE 9 All features with feature type 4110 and feature class other than 3. See Figure 7. Operator O1 is AND. Leaves L1 and L2 are as follows:

L1	
Field	**Value**
valueTypeFromFeature	FT
valueTypeForComparison	NUM
num1	4110
testType	EQ

L2	
Field	**Value**
valueTypeFromFeature	FC
valueTypeForComparison	NUM
num1	3
testType	NE

EXAMPLE 10 All features with feature type 4110 and feature class other than 1 through 3 or 5 through 7. See Figure 8. Operators O1 and O2 are both AND. Leaves L1, L2, and L3 are as follows:

L1	
Field	**Value**
valueTypeFromFeature	FT
valueTypeForComparison	NUM
num1	4110
testType	EQ

L2	
Field	**Value**
valueTypeFromFeature	FC
valueTypeForComparison	NUM_INT
num1	1
num2	3
testType	NO

L3	
Field	**Value**
valueTypeFromFeature	FC
valueTypeForComparison	NUM_INT
num1	5
num2	7
testType	NO

EXAMPLE 11 All features with either feature type 4110 (road) and feature class 3 or feature type 4210 (rail) and feature class 5. See Figure 9. Operator O1 is OR; operators O2 and O3 are both AND. Leaves L1, L2, L3, and L4 are as follows:

L1	
Field	**Value**
valueTypeFromFeature	FT
valueTypeForComparison	NUM
num1	4110
testType	EQ

L2	
Field	**Value**
valueTypeFromFeature	FC
valueTypeForComparison	NUM
num1	3
testType	EQ

L3	
Field	**Value**
valueTypeFromFeature	FT
valueTypeForComparison	NUM
num1	4210
testType	EQ

L4	
Field	**Value**
valueTypeFromFeature	FC
valueTypeForComparison	NUM
num1	5
testType	EQ

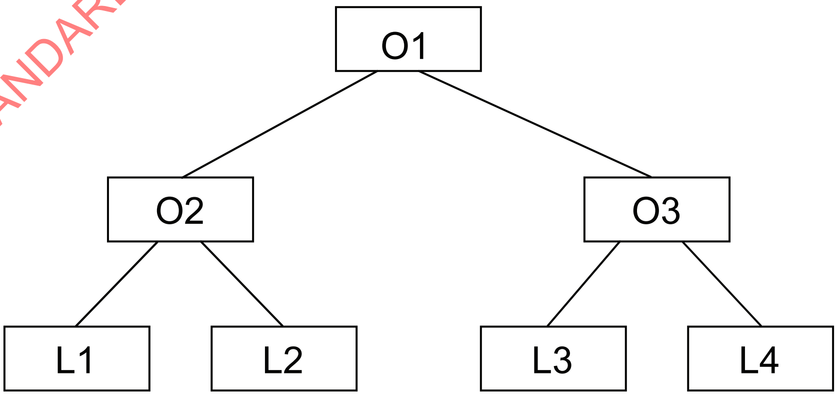


Figure 9 — All features with either feature type 4110 and feature class 3 or feature type 4210 and feature class 5

EXAMPLE 12 All features with feature type 4110 and attribute “SP” (speed category), having attribute values 25 through 70. See Figure7. Operand O1 is AND. Leaves L1 and L2 are as follows:

L1	
Field	**Value**
valueTypeFromFeature	FT
valueTypeForComparison	NUM
num1	4110
testType	EQ

L2	
Field	**Value**
valueTypeFromFeature	AV
attributeType	“SP”
valueTypeForComparison	NUM_INT
num1	25
num2	70
testType	EQ

EXAMPLE 13 All features with feature type 4110 and attribute “DF” (direction of traffic flow) present. See Figure 7. Operand O1 is AND. Leaves L1 and L2 are as follows:

L1	
Field	**Value**
valueTypeFromFeature	FT
valueTypeForComparison	NUM
num1	4110
testType	EQ

L2	
Field	**Value**
valueTypeFromFeature	AV
attributeType	“DF”
valueTypeForComparison	NO
testType	EX

5.3.3.2.14 featureFilterLeafType

Structure of a leaf.

```
typedef struct featureFilterLeafType_s {

   featureFilterValueTypeFromFeatureEnum      valueTypeFromFeature;
   char[2]                          attributeType;
           // Values for attributeType are defined in the tables 8.2

   valueTypeForComparisonEnum       valueTypeForComparison;
   int                              num1;
   int                          num2;
   string                           stringValue;
   int                          setCount;
   int<sequence>                        intSet;
   string <sequence>                        stringSet;
   featureFilterTestTypeEnum                testType;
} featureFilterLeafType;
```

5.3.3.2.15 featureFilterOperandType

An item on which an operation, such as AND, OR, or NOT, is performed. It can be either an operator or a leaf. This structure is used both for the root of the tree specifying feature filtering and for operand nodes lower in the tree.

```
typedef struct featureFilterOperandType_s {


   union operType switch (operandTypeEnum) {
     case LEAF:     featureFilterLeafType      leaf;
     case OPERATOR: featureFilterOperatorType  operator;
          };
} featureFilterOperandType;
```

5.3.3.2.16 featureFilterOperandTypeEnum

Type of operand (leaf/left side or operator/right side) for a filter comparison.

```
typedef enum featureFilterOperandTypeEnum_e {

   LEAF,
   OPERATOR
} featureFilterOperandTypeEnum;
```

5.3.3.2.17 featureFilterOperatorType

Structure of an operator for a filter comparison. For negation the operator is NOT and leftOperand is set to NULL.

```
typedef struct featureFilterOperatorType_s {

   featureFilterOperatorTypeEnum    operatorType;
   featureFilterOperandType   node;
   featureFilterOperandType      leftOperand;
   featureFilterOperandType      rightOperand;

} featureFilterOperatorType;
```

5.3.3.2.18 featureFilterOperatorTypeEnum

Type of operator for a filter comparison.

```
typedef enum featureFilterOperatorTypeEnum_e {

   AND,
   OR,
   NOT
} featureFilterOperatorTypeEnum;
```

5.3.3.2.19 featureFilterTestTypeEnum

Type of test to be applied in a filter operation.

```
typedef enum featureFilterTestTypeEnum_e {

   EQ, // Equal
   NE, // Not Equal
   LT, // Less Than
   LE, // Less than or Equal
   GT, // Greater Than
   GO, // Greater than or Equal
   CS, // Contains Substring
   SS, // Starts with Substring
   EO, // Element Of
   NO, // Not element Of
   EX  // EXists
} featureFilterTestTypeEnum;
```

5.3.3.2.20 featureFilterValueTypeForComparisonEnum

Type of value for comparison. Right-hand side of a filter test.

```
typedef enum featureFilterValueTypeForComparisonEnum_e {

    NUM,        // Number
    STR,        // String
    NUM_INT,    // Number of Intervals
    NUM_SET,    // Number of Sets
    STR_SET,    // String Set
    NO          // Not Used
} featureFilterValueTypeForComparisonEnum;
```

5.3.3.2.21 featureFilterValueTypeFromFeatureEnum

Type of value derived from a feature. Left-hand side of a filter test.

```
typedef enum featureFilterValueTypeFromFeatureEnum_e {

    FT, // Feature Type
    FC, // Feature Class
    AV  // Attribute Value
} featureFilterValueTypeFromFeatureEnum;
```

5.3.3.2.22 featureSortOrderEnum

This enumeration defines the ordering of the selected service(s) or POI(s) returned. Not every case of these enumerations applies when using this enumeration.

```
typedef enum featureSortOrderEnum_e {

    UNORDERED,
    ALPHA,
    EUCLID_DIST,
    DRIVE_DIST,
    DRIVE_TIME,
    LIKELIHOOD_BASED      // in order of likelihood (where appropriate)
} featureSortOrderEnum;
```

5.3.3.2.23 namedAreaType

Structure defining an input named area. It includes a name string and a feature code to help disambiguate it. Disambiguating context is supported, for example, to distinguish Chuo-ku in Tokyo from Chuo-ku in Sapporo, or to distinguish Springfield, Massachusetts from Springfield, Illinois. More than one such area context may be needed; consequently, a recursive structure can be formed.

This namedAreaType is of the same kind as addressType, rather than of the same kind as FeatureNameType.

```
typedef struct namedAreaType_s {
     int  AddressStructureID; // which address structure is this

      int LevelInStructure;      // numbering starts at bottom as "1" and
   grows

      FeatureTypeEnum  featureType;

      string    AreaName; // for simplicity, use string even for
                                // numeric value;
     namedAreaType  parentArea;   // set NULL for no parent
} namedAreaType;
```

5.3.3.2.24 radiusDistanceEnum

Type of radius/distance desired for the search.

```
typedef enum radiusDistanceEnum_e {

    EUCLID_DIST,
    DRIVE_DIST,
    DRIVE_TIME
} radiusDistanceEnum;
```

5.3.3.2.25 sideTypeEnum

Side of road (left hand, right hand, both, or unspecified).

```
typedef enum sideTypeEnum_e {

    SIDE_TYPE_LEFT,
    SIDE_TYPE_RIGHT,
    SIDE_TYPE_BOTH,
    SIDE_TYPE_UNSPECIFIED
} sideTypeEnum;
```

5.3.3.2.26 MapConditionAttrType (Hungarian: ca)

This structure modifies a condition type.

```
typedef struct MapConditionAttrType_s {

    boolean  bTimeDependent;  // Date and/or time information exists

    boolean  bMultipleLinks;  // Condition applies to more than one
                                   // link
} MapConditionAttrType;
```

5.3.3.2.27 MapConditionModifierEnum (Hungarian: cme)

This enumeration defines a condition modifier.

```
typedef enum MapConditionModifierEnum_e {

    //
    // Used for conditions which do not require modifier.
    //
    NOT_APPLICABLE,

    //
    // Additional information for TRAFFIC_COND.
    //
    POSITIVE_CONDITION,        // Mandatory multiple-link maneuver (such as
                               // all traffic must turn right or taxis must
                               // turn left followed by another left) or
                               // single-link access permission

    NEGATIVE_CONDITION,        // Prohibited multiple-link maneuver (such as
                               // no right turn or vehicles may not turn left
                               // followed by another left on weekdays) or a
                               // single-link access prohibition

    //
    // Additional information for TOLL_COND.
    //
    SIMPLE_TOLL,               // Toll is dependent only on the link
    COMPLEX_TOLL,              // Toll is dependent on destination link
    COMPLEX_TOLL_INSTANCE,     // Toll is one element of a complex toll
                               // structure
```

```
        //
        // Additional information for CONSTRUCTION_STATUS_COND.
        //
        OPEN_TO_TRAFFIC,            // Open to traffic during the construction
                                    // period
        CLOSED_TO_TRAFFIC,          // Closed to traffic during the construction
                                    // period
    } MapConditionModifierEnum;
```

5.3.3.2.28 MapTollType (Hungarian: tol)

This structure contains toll cost information.

```
    typedef struct MapTollType_s {

        //
        // Examples:
        //      1) 2.25 Euros is represented as usTollAmount = 225,
        //         oTollExponent = -2, and strTollCurrency = "EUR".
        //
        //      2) 700 yen can be represented as usTollAmount = 700,
        //         oTollExponent = 0, and strTollCurrency = "JPY"
        //         or as usTollAmount = 7, oTollExponent = 2, and
        //         strTollCurrency = "JPY".
        //
        //      3) 75000 Italian Lire is represented as usTollAmount = 75,
        //         oTollExponent = 3, and strTollCurrency = "ITL".
        //
        unsigned short     usTollAmount;
        octet              oTollExponent;
        char               cTollCurrency[4];    // ISO 4217 currency code

    } MapTollType;
```

5.3.3.2.29 MapConditionModifierType (Hungarian: cm)

This structure contains additional information for the condition type.

```
    typedef struct MapConditionModifierType_s {

        //
        // NOTE: Implement a union switch on MapConditionEnum when there is
        more
        //       than one data field in this structure.
        //

        MapTollType        tolTollCost;  // Toll cost information

        // Resident and guest area code.
        unsigned long      ulResidentAndGuestCode;

        // When a condition applies to HOVs, the following field specifies the
        // minimum number of passengers for an HOV:
        octet              oHOVMin;

    } MapConditionModifierType;
```

5.3.3.2.30 MapCursorInfoType (Hungarian: ci)

This structure contains information about a large result set cursor.

```
typedef struct MapCursorInfoType_s {

// Number of the gap before the set of records in the DAL’s buffer,
// ready to be returned.
long  lCursorBufferStart;

// Number of the gap after the set of records in the DAL’s buffer,
// ready to be returned.
long  lCursorBufferEnd;

// Indication of whether the last result record is ready to be
// returned.  This field is true if and only if lCursorBufferEnd
// describes the gap after the last record.
boolean  bResultSetComplete;

// Current cursor position, as a gap number.  Reading forward will
// start after this gap; reading backward will start before this gap.
Long  lCurrentCursorPosition;
} MapCursorInfoType;
```

5.3.3.2.31 MapCursorOriginEnum (Hungarian: coe)

This enumeration is used for location-relative positioning.

```
typedef enum MapCursorOriginEnum_e {

    CURRENT_POSITION,  // From the current cursor position.
    RESULT_SET_START,  // Relative to the first gap, which is number 0.
    RESULT_SET_END     // Relative to the last gap,
                       // whose number is the total number of results.
} MapCursorOriginEnum;
```

5.3.3.2.32 MapCursorReturnTypeEnum (Hungarian: crte)

This enumeration lists all the possible cursor return types – nothing else is allowed.

Table 9 associates the enum member with the data type:

Table 9 — MapCursorReturnTypeEnum

enum entry	data type	calling function(s)
MAP_CONDITION	MapConditionType	MapGetComplexToll()
MAP_CONDITION_AND_DIRECTION	MapConditionAndDirectionType	MapGetLinkConditions()
MAP_CONDITION_LINK_AND_NODE	MapConditionLinkAndNodeType	MapGetConditionLinksAndNodes()
MAP_DATE_TIME	MapDateTimeType	MapGetConditionDateTime()
MAP_FEATURE	MapFeatureType	GetNearbyMapFeatures() GetFeatures() GetFeaturesFiltered()
MAP_LINK	MapLinkType	MapGetIntersectionLinks() MapSearchLinks() MapSearchNearestLinks() MapGetTollDestinations()
MAP_MANEUVER_DESCRIPTION	MapManeuverDescriptionType	MapGuidePath()
FEATURE_NAME	FeatureNameType	ScrollerHelper()
MAP_NODE	MapNodeType	MapSearchNodes()
POI	POIType	GetPOIsByBBox() GetPOIsByPointRad() GetPOIsFromPath() GetPOIsFromNamedArea()
MAP_ROUTE_LINK_AND_COST	MapRouteLinkAndCostType	checkMapGetLinksAndCostsFromPath() and therefore indirectly via a pathHandle MapComputePath() MapComputePathWithFuzzyWaypoints() MapComputePathWithWaypointSets() MapComputePathMultipleRoutes() MapComputePathsMultipleDestinations()
MAP_SUCCESS_LINK_AND_NODE	MapSuccessLinkAndNodeType	MapGetSuccessors()

```
typedef enum MapCursorReturnTypeEnum_e {

    MAP_CONDITION,              // returned by MapGetComplexToll()
    MAP_CONDITION_AND_DIRECTION, // returned by MapGetLinkConditions()
    MAP_CONDITION_LINK_AND_NODE, // returned by MapGetConditionLinksAndNodes()
    MAP_DATE_TIME,              // returned by MapGetConditionDateTime()
    MAP_FEATURE,                // returned by GetNearbyMapFeatures() ,
                                //    GetFeatures() , and GetFeaturesFiltered()
    MAP_LINK,                   // returned by MapGetIntersectionLinks(),
                                //    MapSearchLinks(),
                                //    MapSearchNearestLinks(), and
                                //    MapGetTollDestinations()
    MAP_MANEUVER_DESCRIPTION, // returned by MapGuidePath()
    FEATURE_NAME,               // returned by ScrollerHelper()
    MAP_NODE,                   // returned by MapSearchNodes()
    POI,                        // returned by GetPOIsByBBox(),
                                //    GetPOIsByPointRad(),
                                //    GetPOIsFromPath(), and
                                //    GetPOIsFromNamedArea()
    MAP_ROUTE_LINK_AND_COST,      // returned by
```

```
                                          //    checkMapGetLinksAndCostsFromPath()
                                          //    and therefore indirectly via a pathHandle
                                          //    MapComputePath() ,
                                          //    MapComputePathWithFuzzyWaypoints() ,
                                          //    MapComputePathWithWaypointSets(),
                                          //    MapComputePathMultipleRoutes() , and
                                          //    MapComputePathsMultipleDestinations()
    MAP_SUCCESS_LINK_AND_NODE // returned by MapGetSuccessors()
} MapCursorReturnTypeEnum;
```

5.3.3.2.33 MapCursorType (Hungarian: cu)

This structure defines a unique handle to a cursor. A cursor maintains an arbitrary-sized query result set obtained through a database search operation. This handle identifies a cursor for subsequent manipulation, allowing the application to specify how much of the result set to fetch in a single operation.

```
typedef unsigned long MapCursorType;
```

5.3.3.2.34 MapFetchDirectionEnum (Hungarian: fde)

This enumeration defines the result-set retrieval direction.

This enumeration is not to be confused with MapOrientationDirectionEnum, which is used to specify the direction(s) relative to the topological orientation of a link for which conditions are to be returned.

```
typedef enum MapFetchDirectionEnum_e {

    MapFetchForward,   // Next set of data records
    MapFetchBackward   // Previous set of data records

} MapFetchDirectionEnum;
```

5.3.3.2.35 MapDateAttrType (Hungarian: da)

This structure is used to modify a date. It is possible to use multiple Booleans to denote a composite date, for instance an exclusive date from dawn to dusk during the spring and summer would require four such Booleans.

```
typedef struct MapDateAttrType_s {

    boolean   bSpring;
    boolean   bSummer;
    boolean   bFall;
    boolean   bWinter;
    boolean   bHoliday;
    boolean   bExcludeDate; // Exclude the date range specified by
    StartDate
                            // and EndDate
    boolean   bDawnToDusk;  // Time period is from dawn to dusk
    boolean   bDuskToDawn;  // Time period is from dusk to dawn

} MapDateAttrType;
```

5.3.3.2.36 MapDateTypeEnum (Hungarian: dte)

This enumeration specifies date formats.

```
        typedef enum MapDateTypeEnum_e {

            MAP_DAY_OF_WEEK,
            MAP_ABSOLUTE_DATE,
            MAP_DAY_OF_YEAR,
            MAP_DAY_OF_MONTH,
            MAP_DAY_OF_WEEK_OF_MONTH,
            MAP_DAY_OF_WEEK_OF_YEAR,
            MAP_WEEK_OF_MONTH,
            MAP_WEEK_OF_YEAR,
            MAP_MONTH_OF_YEAR,
            MAP_DAY_OF_MONTH_OF_YEAR,
            MAP_DAY_OF_WEEK_OF_MONTH_OF_YEAR

} MapDateTypeEnum;
```

5.3.3.2.37 MapDayOfWeekEnum (Hungarian: dwe)

This enumeration specifies days of the week.

```
        typedef enum MapDayOfWeekEnum_e {

           MAP_DAY_OF_WEEK_SUNDAY,
           MAP_DAY_OF_WEEK_MONDAY,
           MAP_DAY_OF_WEEK_TUESDAY
           MAP_DAY_OF_WEEK_WEDNESDAY,
           MAP_DAY_OF_WEEK_THURSDAY,
           MAP_DAY_OF_WEEK_FRIDAY,
           MAP_DAY_OF_WEEK_SATURDAY

        } MapDayOfWeekEnum;
```

5.3.3.2.38 MapDateType (Hungarian: dt)

This structure can specify a date in one of many formats. Some auxiliary structures are pre-defined for utilization within the MapDateType structure.

```
        typedef struct MapAbsoluteDateType_s   {

            unsigned short     usMonthOfYear;
            unsigned short     usDayOfMonth;
            unsigned short     usYear;

        } MapAbsoluteDateType; // (Hungarian dta)


        typedef struct MapDayOfWeekOfMonthType_s   {

            MapDayOfWeekEnum   dweDayOfWeek;
            unsigned short  usWeekOfMonth;

        } MapDayOfWeekOfMonthType;// (Hungarian dwm)
```

```
typedef struct MapDayOfWeekOfYearType_s  {
    MapDayOfWeekEnum   dweDayOfWeek;
    unsigned short  usWeekOfYear;

} MapDayOfWeekOfYearType; // (Hungarian dwy)


typedef struct MapDayOfMonthOfYearType_s  {

    unsigned short     usDayOfMonth;
    unsigned short     usMonthOfYear;

} MapDayOfMonthOfYearType;// (Hungarian dmy)


typedef struct MapDayOfWeekOfMonthOfYearType_s  {

    MapDayOfWeekEnum   dweDayOfWeek;
    unsigned short  usWeekOfMonth;
    unsigned short  usMonthOfYear;

} MapDayOfWeekOfMonthOfYearType; // (Hungarian dwmy)


typedef struct MapDateType_s {

    union MapDateDataType switch (MapDateTypeEnum)  {
       case MAP_DAY_OF_WEEK:    MapDayOfWeekEnum   dweDayOfWeek;

       case MAP_ABSOLUTE_DATE: MapAbsoluteDateType   dtaAbsoluteDate;

       case MAP_DAY_OF_YEAR:   unsigned short     usDayOfYear;

       case MAP_DAY_OF_MONTH:  unsigned short     usMonth;

       case MAP_DAY_OF_WEEK_OF_MONTH:
                           MapDayOfWeekOfMonthType
                                dwmDayOfWeekOfMonth;

       case MAP_DAY_OF_WEEK_OF_YEAR:
                           MapDayOfWeekOfYearType
                                dwyDayOfWeekOfYear;

       case MAP_WEEK_OF_MONTH:  unsigned short     usWeekOfMonth;

       case MAP_WEEK_OF_YEAR:   unsigned short     usWeekOfYear;

       case MAP_MONTH_OF_YEAR:  unsigned short     usMonthOfYear;

       case MAP_DAY_OF_MONTH_OF_YEAR:
                           MapDayOfMonthOfYearType
                                dmyDayOfMonthOfYear;

       case MAP_DAY_OF_WEEK_OF_MONTH_OF_YEAR:
                           MapDayOfWeekOfMonthOfYearType
                                dwmyDayOfWeekOfMonthOfYear;
   } x;

   } MapDateType;
```

5.3.3.2.39 MapTimeZoneType (Hungarian: tz)

This structure represents a time zone.

```
typedef struct MapTimeZoneType_s {

    long      lCode;                /* Reference data code */
    short     sMinutesFromGMT;      /* Offset from GMT in minutes */
    char      cDescription[4];      /* Time zone description */

} MapTimeZoneType;
```

5.3.3.2.40 MapDateTimeType (Hungarian: dtt)

This structure contains date and time information used to specify the operating hours for POIs or the time period during which a condition is in effect, or the starting or ending time of a route computation. For the latter purpose, the time may be unspecified.

```
typedef struct MapDateTimeType_s {

    MapDateType          dtStartDate;
    MapDateType          dtEndDate;
    unsigned short       usStartTime;  // elapsed seconds from midnight
    unsigned short       usEndTime; // elapsed seconds from midnight
    MapTimeZoneType      tzTimeZone;
    octet                oDaylightSavingsTimeCode;
    MapDateAttrType      daDateAttributes;
    Boolean              bUnspecified;

} MapDateTimeType;
```

5.3.3.2.41 MapOrientationDirectionEnum (Hungarian: ode)

This enumeration is used to select the direction(s), relative to the topological orientation of a link, for which it is wished to retrieve conditions. It is possible to request conditions for the forward direction of a link, for the reverse direction, or for both directions.

This enumeration is not to be confused with MapFetchDirectionEnum, which is used to specify the direction relative to a cursor in which parts of a large result set are to be returned.

```
typedef enum MapOrientationDirectionEnum_e {

    MAP_DIRECTION_FORWARD,// Direction of topological orientation
    MAP_DIRECTION_REVERSE,// Direction against topological orientation
    MAP_DIRECTION_BOTH    // Both directions

} MapOrientationDirectionEnum;
```

5.3.3.2.42 MapSectionIDType (Hungarian: sid)

This is a section’s ID. A section is a collection of parcels.

```
typedef struct MapSectionIDType_s {

    unsigned long   ulSectionID;

} MapSectionIDType;
```

5.3.3.2.43 MapParcelIDType (Hungarian: pid)

This is a parcel's ID.

```
typedef struct MapParcelIDType_s {

    unsigned long    ulParcelID;

} MapParcelIDType;
```

5.3.3.2.44 MapEntityIDType (Hungarian: eid)

This is an entity's ID. It might not be unique across all open databases.

```
typedef struct MapEntityIDType_s {

    unsigned long   ulPartOne;
    unsigned long   ulPartTwo;
    unsigned long   ulPartThree;
    unsigned long   ulPartFour;
    unsigned long   ulPartFive;

} MapEntityIDType;
```

5.3.3.2.45 MapDBIDType (Hungarian: dbid)

This is an entity's full ID. This type must provide a unique ID across all databases accessible by a DAL during any open session.

```
typedef struct MapDBIDType_s {

    MapSectionIDType   sidSectionID;
    MapParcelIDType pidParcelID;
    MapEntityIDType eidEntityID;

} MapDBIDType;
```

5.3.3.2.46 MapEntityEnum (Hungarian: ee)

This enumeration lists all entity types.

```
typedef enum MapEntityEnum_e {

    MapConditionEntity,
    MapNodeEntity,
    MapLinkEntity

} MapEntityEnum;
```

5.3.3.2.47 MapLaneCategoryType (Hungarian: lc)

This structure classifies a road's lane category. If `oMinimumLaneCount` and `oMaximumLaneCount` both equal zero, the lane information is either unknown or not applicable.

```
typedef struct MapLaneCategoryType_s {

    octet oMinimumLaneCount;   // Lower lane count bound.
    octet oMaximumLaneCount;   // Upper lane count bound.

} MapLaneCategoryType;
```

5.3.3.2.48 MapLinkCharacteristicsType (Hungarian: lch)

This structure contains link characteristics.

```
typedef MapLinkCharacteristicsType_s {

    // The following fields' values are as defined in GDF:
    octet   oRoadClass;
    octet   oNationalRoadClass;
    octet   oFormOfWay;

} MapLinkCharacteristicsType;
```

5.3.3.2.49 MapSpeedCategoryType (Hungarian: sc)

This structure classifies a road's speed category. If `oMinimumSpeed` and `oMaximumSpeed` both equal zero, the speed information is either unknown or not applicable.

```
typedef struct MapSpeedCategoryType_s {

    octet oMinimumSpeed;   // Minimum speed in km/hour
    octet oMaximumSpeed;   // Maximum speed in km/hour

} MapSpeedCategoryType;
```

5.3.3.2.50 MapLinkAttrType (Hungarian: la)

This contains various real-world link attributes.

```
typedef struct MapLinkAttrType_s {

    MapAccessVehicleType
                      avForwardVehicleAccessTypes;

    MapAccessVehicleType
                      avReverseVehicleAccessTypes;

    unsigned long     ulLinkLength;       // meters

    // Link traversal time, in tenths of seconds
    unsigned long     ulEstLinkTraversalTime;

    // Maximum legal speed limit in km/hour where 0 means data not
    available
```

```
        // and 255 means limit does not apply
        octet           oLegalSpeedLimit;

        // Observed speed in km/hour where 0 means data not available
        MapSpeedCategoryType
                        scActualSpeedCategory;

        octet           oToBearing;   // clockwise angle from north at “to”
        node
        octet           oFromBearing; // clockwise angle from north at “from”
                                      // node

        // Link classification based on available lanes in a single direction
        MapLaneCategoryType
                        lcLaneCategory;

        MapLinkCharacteristicsType
                        lchLinkCharacteristics;

        boolean         bLinkHasCondition;

    } MapLinkAttrType;
```

5.3.3.2.51 MapLinkType (Hungarian: lk)

This structure contains information related to a link. The `oEndToEndHiliteLevel` provides the lowest level at which the link can be drawn with just the end points.

```
    typedef struct MapLinkType_s {

        MapDBIDType     dbidLinkID;
        MapDBIDType     dbidToNodeID;
        MapDBIDType     dbidFromNodeID;
        MapLinkAttrType laLinkAttrs;

        // Highest RP level at which the link appears
        octet       oLinkLevel;


        // Number of navigable feature names associated with this link
        octet       oAssocNameCount;

        // Entity ID of the first two navigable feature names associated with
        // this link
        sequence <MapEntityIDType, 2>
                          eidAssocNames;

        octet       oEndToEndHiliteLevel;

    } MapLinkType;
```

5.3.3.2.52 MapLinkSubattrType (Hungarian: ls)

This structure contains link attributes that are only rarely used.

```
typedef struct MapLinkSubattrType_s {

    boolean             bLinkHasMedian;          // Link has a median strip
    MapTimeZoneType     tzTimeZone;              // Time zone of link
    octet               oDSTCode;                // Daylight  savings time
    code
    octet               oHOVMin;                 // Minimum number of
                                                 // passengers
    unsigned long       ulResidentAndGuestCode;  // Code identifying
    resident
                                                 // and guest area

} MapLinkSubattrType;
```

5.3.3.2.53 MapPosition2DType (Hungarian: p2)

This structure specifies a longitude/latitude position.

```
typedef struct MapPosition2DType_s {

    long  lLongitude;
    long  lLatitude;

} MapPosition2DType;
```

5.3.3.2.54 MapPosition3DType (Hungarian: p3)

This structure specifies a longitude/latitude position and an elevation.

```
typedef struct MapPosition3DType_s {

    MapPosition2DType  p2LatLong;
    short        sElevation;      // in meters

} MapPosition3DType;
```

5.3.3.2.55 MapNodeAttrType (Hungarian: na)

This structure contains information about the attributes for a Node type.

```
typedef struct MapNodeAttrType_s {

      // Is the node aggregated into an intersection?
    boolean  bIsNodeAggregatedIntoIntersection;

      // Does the node correspond to an intersection?
    boolean  bDoesNodeRepresentIntersection;

    boolean  bIsNodeNamed;    // Is the node named?

} MapNodeAttrType;
```

5.3.3.2.56 MapNodeType (Hungarian: nd)

This structure contains information related to a node.

```
typedef struct MapNodeType_s {

    MapDBIDType         dbidNodeID;
    MapPosition3DType   p3Position3D;
    MapNodeAttrType     naNodeAttrs;
    octet               oNodeLevel;    // Greatest level of attached link

    // Highest level of unaggregated node
    octet               oNodeMaxSignificantLevel;

} MapNodeType;
```

5.3.3.2.57 MapSuccessorLinkAndNodeType (Hungarian: sln)

This structure identifies a link connected to a node, the link's "other" node, and the cost of travelling through the "current" node to the link. This structure is returned by `MapGetSuccessors()`.

```
typedef struct MapSuccessorLinkAndNodeType_s {

    MapLinkType       lkLink;
    MapNodeType       ndTargetNode;

    // This is the time, in tenths of seconds, to travel through a
    // node from the "entry" link (specified as a parameter to
    // MapGetSuccessors()), to the exit link (identified in this
    // structure)
    unsigned short    usEstNodeTraversalTime;

} MapSuccessorLinkAndNodeType;
```

5.3.3.2.58 MapPriorityType (Hungarian: pr)

This type is used to tell the DAL what the priority of the executable is.

```
typedef unsigned long MapPriorityType;
```

5.3.3.2.59 MapRectangleType (Hungarian: rect)

This structure defines a bounding box.

```
typedef struct MapRectangleType_s {

    MapPosition2DType     p2LowerLeftCorner;
    MapPosition2DType     p2UpperRightCorner;

} MapRectangleType;
```

5.3.3.2.60 MapReturnType (Hungarian: ret)

This type is an API's return type.

```
typedef long MapReturnType;
```

5.3.3.2.61 MapSuccNodeCostEnum (Hungarian: snce)

This enumeration specifies the types of node cost information that can be provided.

```
typedef enum MapSuccNodeCostEnum_e {

    NO_NODE_COST = 0,             // Do not return node cost information.

    NODE_COST_TO_SUCCESSOR,        // Return node cost travelling from the
                                   input
                                  // link to the successor.

    NODE_COST_FROM_SUCCESSOR      // Return node cost travelling from the
                                  // successor to the input link.
} MapSuccNodeCostEnum;
```

5.3.3.2.62 MapConditionType (Hungarian: cnd)

This structure contains information about conditions associated with a link.

A condition may be associated with multiple links. For example, a “no left turn” condition is associated with the links just before and just after the prohibited turn. However, a single condition shall not be associated with multiple unrelated links. For example, unrelated links that are under construction shall not be associated with the same condition identifier.

```
typedef struct MapConditionType_s {

MapDBIDType  dbidConditionID;

// e.g. Traffic condition
MapConditionCategoryType  ccConditionCategory;

// Attributes of condition (e.g. time dependency)
MapConditionAttrType   caConditionAttrs;

// e.g. Toll cost
MapConditionModifierType  cmConditionModifier;

// Vehicles to which condition applies
MapAccessVehicleType   avVehicleScope;

    // The vehicle scope above is expressed as a set of prohibited
    // vehicles (i.e., negative flavour).  This field is true if the
    // road sign listed a set of allowed vehicles, which was complemented
    // to get the above list.  It is important to note, that the correct
    // interpretation of the above vehicle access mask occurs regardless
    // of this Boolean.  This Boolean is for informational purposes only
    // (mostly to allow the application to know what the actual sign said).
    boolean                     bVehicleExpressionReversed;

} MapConditionType;
```

A condition caused by a “no left turn” sign is encoded as follows: ConditionCategory = TRAFFIC_COND and ConditionModifier = NEGATIVE_CONDITION indicates a negative restriction; ConditionAttrs.bMultipleLinks = TRUE indicates a condition that applies to the starting and ending link of the “no left turn”; avVehicleScope describes all vehicles.

A sign indicating that all vehicles must either go straight or turn right is encoded by two conditions, both with ConditionCategory = TRAFFIC_COND and ConditionModifier = POSITIVE_CONDITION. The semantics of positive restrictions is defined such that, when multiple positive restrictions start with the same link, it is sufficient to satisfy one of them. This means that one positive restriction for going straight and another for turning right, taken together, mean *not* that one must both go straight and turn right, but rather that one must *either* go straight *or* turn right. Aside from ConditionModifier, all fields of the ConditionType data structure are the same as in the previous example.

When a link is closed to traffic due to construction, that fact is represented as follows: ConditionCategory = CONSTRUCTION_STATUS_COND; ConditionModifier = CLOSED_TO_TRAFFIC; ConditionAttrs.bMultipleLinks = FALSE; and avVehicleScope describes all vehicles.

When there is a toll of 2,25 US dollars for all vehicles traversing a link, the condition is encoded as follows: ConditionCategory = TOLL_COND; ConditionAttrs.bMultipleLinks = FALSE; and avVehicleScope describes all vehicles. In addition, the fields of ConditionModifier.TollCost are set as follows: usTollAmount = 225; oTollExponent = –2; and strTollCurrency = “USD”. This indicates that the toll is 225×10^{-2} and is measured in USD, i.e., US dollars.

By comparison, suppose the toll is zero for high-occupancy vehicles, USD 2,25 for all vehicles except trucks, and USD 4,00 for trucks. Then, there are two toll conditions (not three — the zero toll for HOVs does not require a condition at all). For both, ConditionCategory = TOLL_COND and ConditionAttrs.bMultipleLinks = FALSE. For one condition, avVehicleScope lists all vehicles except HOV and trucks, and the fields of ConditionModifier.TollCost are as in the example above. For the other condition, avVehicleScope lists only trucks, ConditionModifier.strTollCurrency = “USD”, and the other fields of ConditionModifier.TollCost may be set with usTollAmount = 400 and oTollExponent = –2, or with usTollAmount = 4 and oTollExponent = 0 (or, less usefully, with usTollAmount = 40 and oTollExponent = –1).

5.3.3.2.63 MapConditionAndDirectionType (Hungarian: cod)

This structure is returned by the function MapGetLinkConditions() to describe each condition associated with a link, along with the direction (forward or reverse, with respect to topological orientation) in which the condition applies.

```
typedef struct MapConditionAndDirectionType_s {

MapConditionTypecndCondition; // condition description
boolean   bForward; // applies to forward direction of link
boolean   bReverse; // applies to reverse direction of link
} MapConditionAndDirectionType;
```

5.3.3.2.64 MapConditionLinkAndNodeType (Hungarian: cln)

This structure contains the maneuver link ID and the link’s destination node ID. A cursor for retrieving these structures is returned by MapGetConditionLinksAndNodes().

```
typedef struct MapConditionLinkAndNodeType_s {

   MapDBIDType  dbidLinkID;
   MapDBIDType  dbidDestinationNodeID;

} MapConditionLinkAndNodeType;
```

5.3.3.2.65 PointAndRadiusType

This structure specifies a point and a radius from the point; this is one form of geographic context.

```
typedef struct PointAndRadiusType_s {

   // specified long/lat for a radius search
   MapPosition2DType       referencePoint;

   // distance from addressed point (meters)
   //    Negative radius means no filtering by distance
   int            distance;

} PointAndRadiusType;
```

5.3.3.3 Error codes

Error codes are returned as the return values of functions. All error codes are negative integers. Likewise, any negative function return value is an error code. To facilitate the distinction of an empty result set from more serious errors, the error code MapNoRecordsFound has the value –1. Other error codes are consecutive negative integers starting at –2.

This definition of error codes notwithstanding, implementations may raise exceptions on platforms on which exception handling is supported.

Error code	Description
`MapCallbackFailed`	User-defined callback function failed.
`MapCursorNotAvailable`	No more cursors can be allocated.
`MapDataFormatError`	Unexpected media data format error.
`MapInvalidArgument`	Invalid argument passed to API.
`MapInvalidRequest`	Attempt to request something nonsensical.
`MapNoMemory`	DAL has insufficient memory to complete the request.
`MapNoRecordsFound`	Search yielded no results.
`MapRouteNotFound`	Failed to compute a route.
`MapSearchOutsideRegion`	Search area does not intersect the coverage area.

5.3.4 Route planning

5.3.4.1 General

This subclause specifies the route planning module that shall be provided by the API. Route computation is the process of determining one or more routes to be suggested to a traveller for travelling from a starting point (origin) to an ending point (destination), perhaps by way of one or more intermediate points (waypoints). In various instances of route computation, the traveller may specify various desired characteristics for the route or routes to be suggested. For example, the traveller may specify a criterion to be optimized, such as travel time or travel distance. Also, the traveller may specify a set of one or more vehicle types by which the route should be traversable. (For this purpose, pedestrian travel is considered a type of vehicle.) Further, the traveller may also specify types of road to be preferred, avoided, or disallowed. If there are to be waypoints, the traveller may specify the order in which the waypoints are to be visited or may leave the order unspecified.

Route computation does *not* include the process of presenting the suggested route to the traveller, whether pictorially, verbally, or in any other way. The presentation of the route is part of the application area of route guidance.

5.3.4.2 Constants

5.3.4.3 Data structures

5.3.4.3.1 WaypointCandidate

One choice for a crisp waypoint, for example the sets of loci representing entrances and exits of a restaurant that is one of the set of alternative waypoints.

```
typedef struct waypointCandidate_s {
    union waypointCandidateDataType switch (distBetweenOrigAndDestP)  {
       case distBetweenOrigAndDest:
          LocusList destinationLoci, originLoci;

       // some implementations may not be able to handle this level of
       // specificity.
       case distNotBetweenOrigAndDest:
          LocusList destinationOriginLoci;
    }
} WaypointCandidate;
```

5.3.4.3.2 WaypointCandidateList

This data structure is used in two different ways in different function calls. In some cases it is used as a set of alternatives for a (crisp) waypoint in which order is not significant. For example, if the user is saying "take me to a bank near here, but I don't care which one", there would be one *waypointCandidate* for each bank and one *waypointCandidateList* representing the whole set. In other function calls, it is used as a sequence of successive individual waypoints in which order is significant. For example, it may be used to say "take me to this specific POI, then this other specific POI, then this third specific POI." The function being called determines which meaning is used.

```
typedef struct waypointCandidateList_s {

    int               nWaypointCandidates

    WaypointCandidate <sequence> waypointCandidates;
} WaypointCandidateList;
```

5.3.4.3.3 WaypointSetList

List of waypoints (or waypoint sets) to visit. For example, if the user is saying "take me to a bank from this list (but I don't care which one) and to a restaurant from this list (but I don't care which one), there would be one *WaypointCandidateList* for the banks and another for the restaurants. The set would be encapsulated in this data structure.

```
typedef struct waypointSetList_s {

    int                   nWaypoints

    WaypointCandidateList <sequence>waypoints;
} WaypointSetList;
```

5.3.4.3.4 WaypointChoice

Specifies a choice of waypoint and a choice of candidate for the waypoint.

```
typedef struct waypointChoice_s {

    int          waypointNo, candidateNo;
} WaypointChoice;
```

5.3.4.3.5 WaypointChoiceList

List of choices made. With three waypoint sets, in the case of a request to optimize waypoint order, there might be the result {{1, 2}, {2, 2}, {0, 1}}. This would mean “the path visits candidate number 2 of waypoint set number 1, then candidate number 2 of waypoint set number 2, then candidate number 1 of waypoint set number 0.” Note that waypoint sets and candidates are numbered from 0, so that the first waypoint set is number 0, the second waypoint set is number 1, and so on.

```
typedef struct waypointChoiceList_s {

    int                  nWaypoints;

    WaypointChoice <sequence> waypointChoice;
} WaypointChoiceList;
```

5.3.4.3.6 FuzzyWaypoint

A fuzzy waypoint is represented by a disc, i.e., by a point and a radius.

```
typedef struct fuzzyWaypoint_s {
    MapPosition2DType  point;

    // radius around the point, in meters
    int          radius;
} FuzzyWaypoint;
```

5.3.4.3.7 FuzzyWaypointList

This represents a list of fuzzy waypoints.

```
typedef struct fuzzyWaypointList_s {

    int               nFuzzyWaypoints;

    FuzzyWaypoint <sequence>  waypoints;
} FuzzyWaypointList;
```

5.3.4.3.8 MapPathHandle

A MapPathHandle is an opaque handle that is used to retrieve a MapRouteLinkAndCostType sequence (i.e., a path structure).

```
typedef struct mapPathHandle_s {

    unsigned long  ulPartOne;
    unsigned long  ulPartTwo;
    unsigned long  ulPartThree;
    unsigned long  ulPartFour;
} MapPathHandle;
```

5.3.4.3.9 MapRouteUsageEnum (Hungarian: rue)

This enumeration is used for highway usage preference.

```
typedef enum MapRouteUsageEnum_e {

   // (e.g., motorway in UK, freeway or turnpike in US, and
   // expressway in Japan and Korea).
   PROHIBIT,     // Never use even if it means
                 // returning a failure
   AVOID,        // Avoid when possible (the degree of
                 // avoidance is left to the implementation)
   PREFER,          // Use when possible (the degree of
                 // preference is left to the implementation)
   NO_PREFERENCE// No preference as to usage

}  MapRouteUsageEnum;
```

5.3.4.3.10 MapRouteDynamicTrafficUsageEnum (Hungarian: rtue)

This enumeration is used for dynamic traffic usage preference.

```
typedef enum MapRouteDynamicTrafficUsageEnum_e {

   TRAFFIC_DYNAMIC,    // No historical usage
   TRAFFIC_HISTORICAL, // Only historical usage
   TRAFFIC_BOTH,
   TRAFFIC_NONE        // [default state]

}  MapRouteDynamicTrafficUsageEnum;
```

5.3.4.3.11 MapRouteMinimizeOptionEnum (Hungarian: rmoe)

This enumeration is used for route minimization options.

```
typedef enum MapRouteMinimizeOptionEnum_e {

    SHORTEST_DISTANCE,      // shortest road distance
    SHORTEST_TIME           // shortest road travel time
    SIMPLE_ROUTE,           // Short road travel time, but biased
                            //   toward few maneuvers

 } MapRouteMinimizeOptionEnum;
```

5.3.4.3.12 MapRouteCostModelType (Hungarian: rcm)

This structure contains information about the cost criteria. It contains information about how to assign “exchange rate” cost to:

— road distance,

— road travel time,

— highway usage preference,

— toll road usage preference,

— dirt road usage preference,

— U-turn maneuver preference,

— scenic route preference, and

— dynamic traffic usage. This indicates whether static cost values or dynamic cost values based on traffic are to be used. If dynamic values are to be used, it indicates whether historical values and/or actual current values are to be used.

It is not required that every implementation support every combination of these parameters. For example, an implementation might not have access to historical traffic information and therefore may not support routing based on it. Similarly, an implementation might not support *preferring* dirt roads or U-turns. Such an application shall accept the statement of the preference but need not heed it.

```
typedef struct MapRouteCostModelType_s {

     MapRouteMinimizeOptionEnum     rmoeMinimizerGoal;
     MapRouteUsageEnum              rueHighwayAffinity;
     MapRouteUsageEnum              rueTollroadAffinity;
     MapRouteUsageEnum                   rueDirtRoadAffinity;
     MapRouteUsageEnum                   rueUTurnAffinity;
     MapRouteUsageEnum                   rueScenicRouteAffinity;
     MapRouteDynamicTrafficUsageEnum     rtueDynamicTrafficUsage;

}  MapRouteCostModelType;
```

5.3.4.3.13 MapRoutePointType (Hungarian: rp)

This structure contains a set of links associated with a route point.

```
typedef struct MapRoutePointType_s {

     unsigned short                usNumLinks;
     sequence <MapDBIDType>        dbidLinkIDs;

}  MapRoutePointType;
```

5.3.4.3.14 MapRoutePointSequenceType (Hungarian: rps)

This structure contains the origin, intermediate waypoints (should there be any), and the destination for the `MapComputePath` function. These points are traversed in order; that is, they are a true sequence.

```
typedef struct MapRoutePointSequenceType_s {

    // There must be at least two points, one for the origin and one
    // for the destination. They are the first and the last point in
    // this sequence.  Intermediate waypoints, if there are any, are
    // defined in the middle of the sequence.
    unsigned short                  usNumPoints;
    sequence <MapRoutePointType>     rpRoutepoints;

}  MapRoutePointSequenceType;
```

5.3.4.3.15 MapRouteLinkAndCostType (Hungarian: rlc)

This structure contains the link and the travel node cost information returned by the `MapComputePath` function. The node traversal cost value for the first MapComputePath result record should be zero.

```
typedef struct MapRouteLinkAndCostType_s {

    MapDBIDType     dbidComputePathLinkID;

    // Link traversal time, in tenths of seconds
    unsigned long   ulEstLinkTravelTime;

    // Estimated node travel cost to the link above, units of
    // measurement are in tenths of seconds.
    unsigned short usNodeTravelTime;

}  MapRouteLinkAndCostType;
```

5.3.4.3.16 MapRouteControlType (Hungarian: rc)

This structure contains the criteria to be used by the `MapComputePath` function for computing a path.

The cost criteria include the following information:

— Should travel time or travel distance be minimized?

— Should highways be used?

— Should toll roads be used?

— Should a scenic route be preferred?

— For which vehicle type(s)?

— For what starting and/or end times?

```
typedef struct MapRouteControlType_s {

    // Cost criteria
    MapRouteCostModelType       rcmCostCriteria;
```

```
        // Vehicle type(s)
        MapAccessVehicleType       avVehicleTypes;

        // Planned departure date and time
        MapDateTimeType            dttRouteStartDateTime;
        // Planned arrival date and time
        MapDateTimeType            dttRouteEndDateTime;
        // Note:  If both departure and arrival dates and times are specified,
        // the computed route should be navigable for the entire specified
        // interval.
    }  MapRouteControlType;
```

5.3.4.4 Error codes

Error code	Description
`MapCallbackFailed`	User-defined callback function failed.
`MapCursorNotAvailable`	No more cursors can be allocated.
`MapDataFormatError`	Unexpected media data format error.
`MapInvalidArgument`	Invalid argument passed to API.
`MapInvalidRequest`	Attempt to request something nonsensical.
`MapNoMemory`	DAL has insufficient memory to complete the request.
`MapNoRecordsFound`	Search yielded no results.
`MapRouteNotFound`	Failed to compute a route.
`MapSearchOutsideRegion`	Search area does not intersect the coverage area.

5.3.4.5 Route planning functions

5.3.4.5.1 MapComputePath

This function returns a handle to an ordered list of links and node travel costs for the computed path. The computed path is the path for the specified compute control values.

The bRelaxConstraint input parameter allows the caller to specify whether all the specified compute control values must be met, or whether the DAL has permission to relax one or more. If the specified compute control values would result in no route being found because highways or toll roads are prohibited then, if bRelaxConstraint is TRUE, the function may relax one or more control values so that a route can be found. If the specified compute control values do allow a result to be found, then bRelaxConstraint has no effect. The bConstraintRelaxed output parameter shows whether one or more compute control values has been relaxed. The cursor (which is obtainable form the pathHandle output parameter) contains `MapRouteLinkAndCostType` elements.

Although the application uses rcComputeControl to specify the type of route requested, the exact choice of route is up to the DAL, and is outside the scope of this International Standard.

```
    MapReturnType MapComputePath (
       in    LocusList                originLoci,
       in    LocusList                destinationLoci,

       in    MapRouteControlType      rcComputeControl,
      in    boolean                 bRelaxConstraint,

      in    MapPriorityType         prPriority,

      out   boolean                 bConstraintRelaxed,
      // pathHandle is a data structure containing information about the
      // computed route, which can be used as input to other functions.  The
      // actual large result set containing a list of path elements can be
      // obtained by calling MapGetCursorInfo, which returns a cursor
      // containing MapRouteLinkAndCostType elements.

      out   MapPathHandle           pathHandle )
  raises(MapInvalidArgument,
      MapNoRecordsFound,
      MapCursorNotAvailable,
      MapInvalidRequest,
      MapNoMemory,
     MapRouteNotFound);
```

The following examples clarify the use of compute control values and bRelaxConstraint. In the examples, for clarity we will assume that rcComputeControl.rcmCostCriteria.rmoeMinimizerGoal is set to SHORTEST_DISTANCE, i.e., that distance is being minimized.

Consider case 1 in Figure 10. If rcComputeControl.rcmCostCriteria.rtueTollroadAffinity is set to NO_PREFERENCE, route A will be chosen. If it is set to AVOID_TOLLROADS, route B will be chosen. If it is set to PROHIBIT_TOLLROADS, route C will be chosen. In all of these cases, bConstraintsRelaxed will be set to FALSE because no constraint has been relaxed.

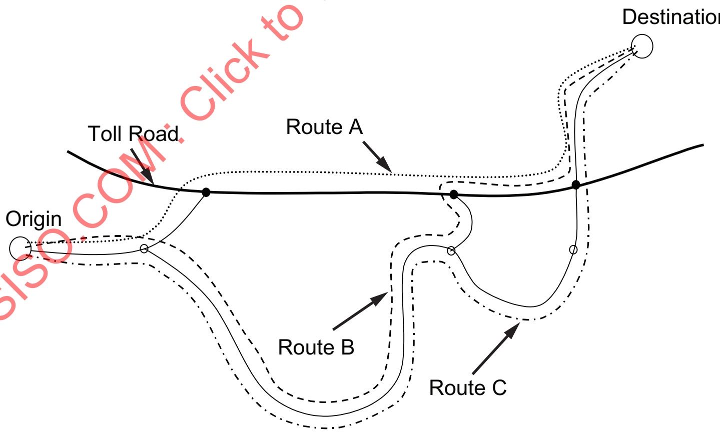


Figure 10 — Case 1

Next consider case 2 (Figure 11), in which the destination is actually on a toll road. If rcComputeControl.rcmCostCriteria.rtueTollroadAffinity is set to NO_PREFERENCE, route A will be chosen. If it is set to AVOID_TOLLROADS, route B will be chosen. If it is set to PROHIBIT_TOLLROADS, then if bRelaxConstraints is FALSE, the computation will fail and the function will return MapRouteNotFound; but if bRelaxConstraints is TRUE, then the function will relax PROHIBIT_TOLLROADS to AVOID_TOLLROADS, route B will be chosen, and bConstraintsRelaxed will be set to TRUE.

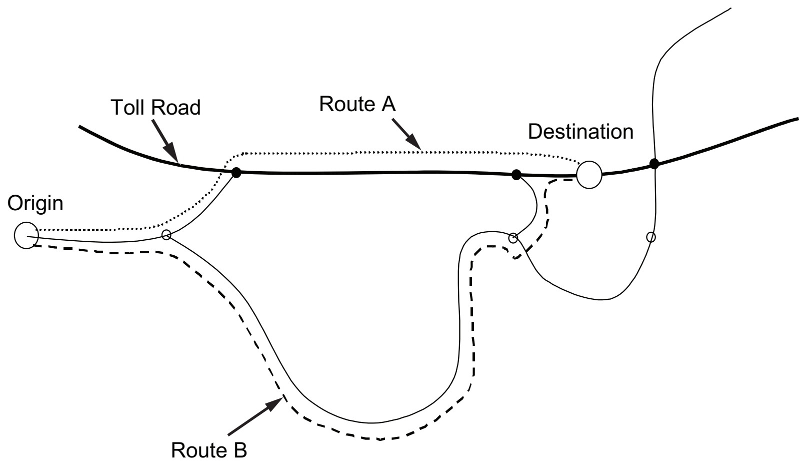


Figure 11 — Case 2

5.3.4.5.2 MapComputePathWithFuzzyWaypoints

This function differs from MapComputePath() in the preceding section only by adding a list of "fuzzy waypoints" as an input parameter. A fuzzy waypoint is not a specific point at which the driver wishes to stop; rather, it is a general area through which the driver wishes to go. It is anticipated that this function will be used primarily to allow a driver to modify a route by pointing at a general area through which to pass. For example, there might be two highways, Highway 1 and Highway 2, between an origin city and a destination city. The first route computed for the driver might be along Highway 1, but the driver knows that she likes Highway 2 better. If the user interface allows it, the driver could tap her finger on the map somewhere near Highway 2 and the application could then use the finger-tap to specify a fuzzy waypoint for a second computation.

```
MapReturnType MapComputePathWithFuzzyWaypoints(
   in    LocusList               originLoci,
   in    FuzzyWaypointList       waypoints,
   in    LocusList               destinationLoci,

   in    MapRouteControlType     rcComputeControl,

   in    boolean                 bRelaxConstraint,

   in    MapPriorityType         prPriority,

   out   boolean                 bConstraintRelaxed,

   // one per step, i.e., if there are n waypoints there are (n + 1)
   // MapPathHandles
   out   MapPathHandle          pathHandle )
raises(MapInvalidArgument,
   MapNoRecordsFound,
   MapCursorNotAvailable,
   MapInvalidRequest,
   MapNoMemory,
  MapRouteNotFound);
```

5.3.4.5.3 MapComputePathWithWaypointSets

This function differs from MapComputePath() by adding as inputs a set of waypoints and a Boolean option regarding whether the order of waypoints is to be optimized. There is also an additional output which returns the set of waypoints actually chosen.

Each waypoint is specified not as a single waypoint but rather as a set of candidates. The set of candidates can contain a single element when a single, unique waypoint is specified. However, the set can contain multiple candidates. In that case, the function chooses one of the candidates as a waypoint and ignores the rest. This can be used when there are multiple possible stops all of which are equally acceptable to the driver. For example, a driver who wants to stop at an automatic teller machine may not care which of his bank's machines he visits. In that case it is natural to use multiple candidates.

Each waypoint candidate, in turn, is specified with sets of destination and origin loci, which represent entry points to, and exit points from, the waypoint stop. In most cases, there will be only one destination locus and one origin locus per waypoint, and they will be the same locus. Still, the function supports multiple loci because some waypoints will be accessible from loci on multiple links. The function also supports destination loci distinct from origin loci because, for some waypoints, the entry point from a road and the exit point to a road may be substantially different.

The Boolean parameter optimizeWaypointOrder specifies whether the order in which the waypoints are visited is to be optimized by the function or is to be the same as the order in which the waypoints are listed in the function call. If the order is to be optimized by the function, this amounts to solving the "travelling salesman problem" (TSP), which is known to be computationally difficult for large numbers of waypoints. There is therefore no guarantee that an implementation will produce a result quickly when more than a few waypoints are specified; conversely, there is no guarantee that an implementation will produce an optimal result when speed is chosen as more important than optimality. As a result, the order of the WaypointCandidateLists in the WaypointSetList is significant if optimizeWaypointOrder is false and insignificant if optimizeWaypointOrder is true. However, the order of the waypoints in each WaypointCandidateList is not significant in either case.

This function also has two output parameters that differ from those of MapComputePath(). The parameter waypointChoices specifies which waypoint from each set was chosen and the order in which the waypoints were chosen. Also, the output parameter pathHandle returns a set of path handles, one per step of the route (origin to waypoint, waypoint to waypoint, or waypoint to destination), in order.

```
MapReturnType MapComputePathWithWaypointSets(
   in    LocusList               originLoci,
   in    WaypointSetList         waypoints,
   in    LocusList               destinationLoci,

   // true => optimize order of waypoints; false => use stated order
   in    boolean                 optimizeWaypointOrder,

   in    MapRouteControlType     rcComputeControl,

   in    boolean                 bRelaxConstraint,

   in    MapPriorityType         prPriority,

   out   boolean                 bConstraintRelaxed,

   out   waypointChoiceList      waypointChoices,

   // one per step, i.e., if there are n waypoints there are (n + 1)
   // MapPathHandles
   out   MapPathHandle <sequence>   pathHandle )
raises(MapInvalidArgument,
   MapNoRecordsFound,
   MapCursorNotAvailable,
   MapInvalidRequest,
   MapNoMemory,
   MapRouteNotFound);
```

5.3.4.5.4 MapComputePathMultipleRoutes

This function differs from MapComputePath() by allowing a single call to return multiple alternative routes. To allow this, there is an additional parameter nRoutesDesired, which specifies the number of alternative routes desired. The input parameter rcComputeControl is a set of MapRouteControlTypes, one per desired route, to support, for example, simultaneous requests for shortest, fastest, and pedestrian routes, which require different control parameters.

In this function, the output parameter pathHandle is a sequence of n sequences of w + 1 MapPathHandles, where n is the number of routes desired and w is the number of waypoints supplied. Each of the n sequences is for one of the multiple routes computed; each route is returned as w + 1 steps, one per step of the route (origin to waypoint, waypoint to waypoint, or waypoint to destination), in order.

The order of waypoints in the WaypointCandidateList is significant. The computed routes will visit the waypoints in the order in which they are supplied.

```
MapReturnType MapComputePathMultipleRoutes(
   in    LocusList              originLoci,
   in    WaypointCandidateList  waypoints,
   in    LocusList              destinationLoci,

   in    int                    nRoutesDesired,   // how many routes?
   in    MapRouteControlType <sequence>
                                rcComputeControl, // nRoutesDesired many.
   // This is a set of nRoutesDesired MapRouteControlTypes.
   // This way the caller could ask for, e.g., two different short-time
   // routes and one short-distance route, or for one short-time route,
   // one short-distance route, and one scenic route, or for three
   // different short-time routes.

   in    boolean                bRelaxConstraint,

   in    MapPriorityType        prPriority,

   out   boolean                bConstraintRelaxed,

   // one per route, i.e., a total of nRoutesDesired MapPathHandles are
   // returned
   out   MapPathHandle <sequence> of <sequence>
                                pathHandle )

raises(MapInvalidArgument,
   MapNoRecordsFound,
   MapCursorNotAvailable,
   MapInvalidRequest,
   MapNoMemory,
   MapRouteNotFound);
```

5.3.4.5.5 MapComputePathsMultipleDestinations

This function differs from MapComputePath() by allowing a single call to compute routes from a single origin to multiple destinations at once, without waypoints. To allow this, there is an additional input parameter nDestinations, and the destinations are specified by a sequence of LocusLists, rather than a single LocusList.

In addition, the output parameter pathHandle is a sequence of MapPathHandles, one per destination, in the order in which the destinations were specified.

```
MapReturnType MapComputePathsMultipleDestinations(
   in    LocusList                originLoci,
   // Note:  No waypoints in this parameterization.
   in    int                      nDestinations,
   in    LocusList <sequence>     destinationLoci,

   in    MapRouteControlType      rcComputeControl,

   in    boolean                  bRelaxConstraint,

   in    MapPriorityType          prPriority,

   out   boolean                  bConstraintRelaxed,

   // one per destination, i.e., a total of nDestinations MapPathHandles
   // are returned
   out   MapPathHandle <sequence>   pathHandle )
raises(MapInvalidArgument,
   MapNoRecordsFound,
   MapCursorNotAvailable,
   MapInvalidRequest,
   MapNoMemory,
  MapRouteNotFound);
```

5.3.4.5.6 MapGetLinksAndCostsFromPath

A MapPathHandle is an opaque handle that is used to retrieve a MapRouteLinkAndCostType sequence (i.e., a path structure).

```
MapReturnType MapGetLinksAndCostsFromPath (
   in    MapPathHandle            pathHandle,
   out   MapCursorReturnTypeEnum crteCursorType = MAP_ROUTE_LINK_AND_COST,
   out   MapCursorType            cuMapRouteLinkAndCost)
raises(MapSearchOutsideRegion,
  MapNoMemory,
  MapInvalidRequest,
  MapInvalidArgument,
  MapCursorNotAvailable,
  MapNoRecordsFound);
```

5.3.4.6 Route planning optional extended low level interface

5.3.4.6.1 MapGetComplexToll

This function returns a cursor containing MapConditionType data structures representing the possible tolls from one link (the “origin link”) to another (the “destination link”) for a certain vehicle type. The argument avVehicle is used to specify the vehicle or set of vehicles for which tolls are desired.

The MapConditionType data structures contained by the cursor have condition category TOLL_CONDITION and condition modifier COMPLEX_TOLL_INSTANCE.

When there is a single toll from the origin link to the destination link, the cursor will contain a single MapConditionType data structure. When there are multiple toll values from the origin link to the destination link — whether for different vehicle types, different time periods, different routes, different currencies (e.g., US dollars versus Canadian dollars), or some combination of these — one MapConditionType data structure will be returned for each toll value.

As for other uses of MapConditionType, a time dependency is denoted by the Boolean bTimeDependent, and the times for which it applies are retrieved with the function MapGetConditionDateTime().

When a toll depends on the route taken, the route to which the MapConditionType structure applies is conveyed to the caller as follows: The Boolean bMultipleLinks is set to indicate the presence of a route to which the toll applies. The function MapGetConditionLinksAndNodes() is used to retrieve a sequence of links and associated (destination) nodes. The sequence of links is *not* required to be consecutive. The first is the origin link for the toll price; the last is the destination link for the toll price; and the intermediate links are links such that, if the route traverses them in order, the toll price in the MapConditionType data structure is correct. To allow for cases in which the toll is bi-directional, the DAL is allowed to return the list of links in reverse order (in which case the “destination” nodes will be destination nodes in the order in which the list is returned, and origin nodes when the list is traversed in the order originally described by the caller).

```
MapReturnType MapGetComplexToll(
   in    MapDBIDType             dbidOriginLinkID,
   in    MapDBIDType             dbidDestinationLinkID,
   in    MapAccessVehicleType    avVehicle,
   in    MapPriorityType         prPriority,
   out   MapCursorType           cuTollInstances )
raises(MapSearchOutsideRegion,
   MapNoMemory,
   MapInvalidRequest,
   MapInvalidArgument,
   MapCursorNotAvailable,
   MapNoRecordsFound);
```

5.3.4.6.2 MapGetCondition

This function retrieves the specified condition record.

```
MapReturnType MapGetCondition(
   in    MapDBIDType             dbidConditionID,
   in    MapPriorityType         prPriority,
   out   MapConditionType        cndRetrievedCondition )
raises(MapSearchOutsideRegion,
   MapNoMemory,
   MapInvalidRequest,
   MapInvalidArgument,
   MapCursorNotAvailable,
   MapNoRecordsFound);
```

5.3.4.6.3 MapGetConditionDateTime

This function creates a result set containing all of the date/time information for the specified condition. The cursor contains `MapDateTimeType` elements.

```
MapReturnType MapGetConditionDateTime(
   in    MapDBIDType                dbidConditionID,
   in    MapPriorityType            prPriority,
   out   MapCursorReturnTypeEnum crteCursorType = MAP_DATE_TIME,
   out   MapCursorType              cuMapDateTime )
raises( MapInvalidArgument,
        MapNoRecordsFound,
        MapCursorNotAvailable,
        MapInvalidRequest,
        MapNoMemory );
```

5.3.4.6.4 MapGetConditionLinksAndNodes

This function creates a result set containing a list of the link IDs and the intermediate node IDs associated with the specified condition in the order in which the condition applies. The cursor contains `MapConditionLinkAndNodeType` elements. Note that, despite the fact that the function's name refers to links and nodes in the plural, the result set contains only one link-and-node pair if the condition applies only to a single link.

```
MapReturnType MapGetConditionLinksAndNodes(
   in    MapDBIDType                dbidConditionID,
   in    MapPriorityType            prPriority,
   out   MapCursorReturnTypeEnum crteCursorType = MAP_CONDITION_LINK_AND_NODE,
   out   MapCursorType              cuMapConditionLinkAndNode )
raises( MapInvalidArgument,
        MapNoRecordsFound,
        MapCursorNotAvailable,
        MapInvalidRequest,
        MapNoMemory );
```

5.3.4.6.5 MapGetContainingIntersection

This function retrieves the DBID of the intersection containing a particular node or link.

```
MapReturnType MapGetContainingIntersection(
   in    MapDBIDType             dbidNodeORLinkID,
   in    MapEntityEnum           eeEntityType,
   in    MapPriorityType         prPriority,
   out   MapDBIDType             dbidIntersectionID )
raises( MapInvalidArgument,
        MapInvalidRequest );
```

5.3.4.6.6 MapGetIntersectionLinks

This function creates a result set containing all of the links (that is, both internal links and links that are attached to the intersection) associated with the specified intersection. The cursor contains `MapLinkType` elements.

```
MapReturnType MapGetIntersectionLinks(
   in    MapDBIDType                dbidIntersectionID,
   in    MapPriorityType            prPriority,
   out   MapCursorReturnTypeEnum crteCursorType = MAP_LINK,
   out   MapCursorType           cuMapLink )
raises( MapInvalidArgument,
        MapNoRecordsFound,
        MapCursorNotAvailable,
        MapInvalidRequest,
        MapNoMemory );
```

5.3.4.6.7 MapGetLink

This function retrieves the specified link.

```
MapReturnType MapGetLink(
   in    MapDBIDType             dbidLinkID,
   in    MapPriorityType         prPriority,
   out   MapLinkType             lkRetrievedLink )
raises( MapInvalidArgument );
```

5.3.4.6.8 MapGetLinkSubattr

This function gets the rarely used link attribute information for a link.

```
MapReturnType MapGetLinkSubattr(
  in MapDBIDType dbidLinkID,
  in   MapPriorityType  prPriority,
  out  MapLinkSubattrType  lsRetrievedLinkSubattr )
raises( MapInvalidArgument, MapInvalidRequest );
```

5.3.4.6.9 MapGetLinkConditions

This function creates a result set containing any condition associated with the specified link. The cursor contains `MapConditionAndDirectionType` elements.

Note that this function can return a “no record found” result for a link to which conditions apply, if all the conditions apply to one direction and the request is limited to the other direction.

A condition applicable to multiple links is retrievable from any of its participating individual links.

```
MapReturnType MapGetLinkConditions(
   in     MapDBIDType         dbidLinkID,   // link for which conditions
                                                 // are desired
   in     MapConditionCategoryType
                              ccConditionTypes; // type(s) of condition
                                                 // desired
   in     MapOrientationDirectionEnum
                              odeDirection,      // direction(s) for which
                                                 // conditions are desired
   in     MapPriorityType     prPriority,
   out    MapCursorReturnTypeEnum crteCursorType = MAP_CONDITION_AND_DIRECTION,
   out    MapCursorType       cuMapConditionAndDirection )
raises( MapInvalidArgument,
        MapNoRecordsFound,
        MapCursorNotAvailable,
        MapInvalidRequest,
        MapNoMemory );
```

5.3.4.6.10 MapGetMultLinks

This function retrieves the requested link records. The links are specified by entity ID.

```
MapReturnType MapGetMultLinks(
   in     sequence <MapEntityIDType>   eidLinkIDs,
   in     long                         lNumOfEntityIDs,
   in     MapPriorityType              prPriority,
   out    sequence <MapLinkType>       lkRetrievedLinks )
raises( MapInvalidArgument );
```

5.3.4.6.11 MapGetNode

This function retrieves the specified node.

```
MapReturnType MapGetNode(
   in     MapDBIDType              dbidNodeID,
   in     MapPriorityType          prPriority,
   out    MapNodeType              ndRetrievedNode )
raises( MapInvalidArgument );
```

5.3.4.6.12 MapGetSuccessors

This function creates a result set containing the links connected to the specified node. The cursor contains `MapSuccessorLinkAndNodeType` elements, one for each successor link. `snceSuccNodeCostType` specifies whether node traversal costs between `dbidLinkID` and the successor link should be calculated in the forward or reverse direction of travel. It is possible to set `dbidLinkID` to NULL, but `dbidLinkID` shall not be NULL when node travel cost values are to be returned.

```
MapReturnType MapGetSuccessors(
   in    MapDBIDType              dbidNodeID,
   in    MapDBIDType              dbidLinkID,
   in    long                     lTargetLevel,
   in    boolean                  bIncludeInputLink,
   in    MapSuccNodeCostEnum      snceSuccNodeCostType,
   in    MapPriorityType          prPriority,
   out   MapCursorReturnTypeEnum crteCursorType = MAP_SUCCESS_LINK_AND_NODE,
   out   MapCursorType            cuMapSuccessLinkAndNode )
raises( MapInvalidArgument,
        MapInvalidRequest,
        MapNoRecordsFound,
        MapCursorNotAvailable,
        MapNoMemory );
```

5.3.4.6.13 MapGetTimeZone

This function retrieves the time zone information for a particular time zone code. Time zones required for the PSF are contained in metadata.

```
MapReturnType MapGetTimeZone(
   in    unsigned long          ulTimeZoneCode,     // From metadata
   in    MapPriorityType        prPriority,
   out   MapTimeZoneType       tzRetrievedTimeZone )
raises( MapInvalidArgument,
        MapNoRecordsFound );
```

5.3.4.6.14 MapGetTollDestinations

For a given link, this function returns a cursor which returns a list of IDs of other links to or from which there is toll information. If the Boolean bIsOrigin is TRUE, the cursor returns the list of links *to* which tolls are known. If it is FALSE, the cursor returns the list of links *from* which tolls are known.

```
MapReturnType MapGetTollDestinations(
   in    MapDBIDType              dbidLinkID,
   in    boolean                  bIsOrigin,
   in    MapPriorityType          prPriority,
   out   MapCursorReturnTypeEnum crteCursorType = MAP_LINK,
   out   MapCursorType            cuLinkIDs )
raises( MapInvalidArgument,
        MapInvalidRequest );
```

5.3.4.6.15 MapSearchLinks

This function creates a result set containing the link records satisfying the specified criteria. The cursor contains `MapLinkType` elements.

This function searches for links *only* at level 0 and returns all links whose geometry lies completely or partially within the specified set of rectangles.

```
MapReturnType    MapSearchLinks(
    in     unsigned short                usNumOfRectangles,
    in     sequence <MapRectangleType>   rectSearchAreas,
    in     MapReturnType(any, any)       retUserDefineFilterFunc,
    in     any                           aFilterFuncParam,
    in     unsigned long                 ulFilterFuncParamSize,
    in     MapPriorityType               prPriority,
    out    MapCursorReturnTypeEnum       crteCursorType = MAP_LINK,
    out    MapCursorType                 cuMapLink)
raises( MapInvalidArgument,
        MapSearchOutsideRegion,
        MapNoRecordsFound,
        MapCallbackFailed,
        MapCursorNotAvailable,
        MapNoMemory);
```

5.3.4.6.16 MapSearchNearestLinks

This function creates a result set containing the link records whose geometry is nearest to the search points which satisfy the specified criteria. The cursor contains `MapLinkType` elements.

This function searches for links *only* at level 0.

This function's search area is considered to be the disk centred at the search point with the specified radius. The error `MapSearchOutsideRegion` is raised only if there is no intersection between the database coverage area and the search area.

```
MapReturnType    MapSearchNearestLinks(
    in     MapPosition2DType             p2SearchPoint,
    in     long                          lMaxDistance,      // meters
    in     MapReturnType(any, any)       retUserDefineFilterFunc,
    in     any                           aFilterFuncParam,
    in     unsigned long                 ulFilterFuncParamSize,
    in     MapPriorityType               prPriority,
    out    MapCursorReturnTypeEnum       crteCursorType = MAP_LINK,
    out    MapCursorType                 cuMapLink)
raises( MapInvalidArgument,
        MapSearchOutsideRegion,
        MapNoRecordsFound,
        MapCallbackFailed,
        MapCursorNotAvailable,
        MapNoMemory);
```

5.3.4.6.17 MapSearchNodes

This function creates a result set containing the node records satisfying the specified criteria. The cursor contains `MapNodeType` elements.

```
MapReturnType MapSearchNodes(
        in    unsigned short                  usSearchLevel,
        in    unsigned short                  usNumOfRectangles,
        in    sequence <MapRectangleType>     rectSearchAreas,
        in    MapReturnType(any, any)         retUserDefineFilterFunc,
        in    any                             aFilterFuncParam,
        in    unsigned long                   ulFilterFuncParamSize,
        in    MapPriorityType                 prPriority,
        out   MapCursorReturnTypeEnum         crteCursorType = MAP_NODE,
        out   MapCursorType                   cuMapNode )
    raises( MapInvalidArgument,
            MapSearchOutsideRegion,
            MapNoRecordsFound,
            MapCallbackFailed,
            MapCursorNotAvailable,
            MapNoMemory );
```

5.3.5 Route guidance

5.3.5.1 Data structures

5.3.5.1.1 MapManeuverDescription

This is the structure in the cursor. There is one per reported maneuveror, one per component of an aggregated maneuver.

```
    typedef struct mapManeuverDescription_s {

          // Code indicating the maneuver type.  There must be a whole
       catalogue,
          //  to be collected from the group.  For example:
          //       0 = continue straight on <name>
          //       1 = turn sharp right onto <name>
          //       2 = turn right onto <name>
          //       3 = turn slight right onto <name>
          //       4 = turn slight left onto <name>
          //       5 = turn left onto <name>
          //       6 = turn sharp left onto <name>
          //       7 = bear right onto <name>
          //       8 = bear left onto <name>
          //       9 = make U turn onto <name>
          //      10 = name becomes <name>
          //      11 = exit on right onto <name>
          //      12 = exit on left onto <name>
          //      13 = merge on right onto <name>
          //      14 = merge on left onto <name>
          //      15 = follow signs to <name>
          //      16 = cross <name>
          // ...
          //      17 = street/road curves
          //      18 = enter ramp
          //      19 = exit ramp
          //      20 = enter roundabout
          //      21 = exit  roundabout at <Nth> exit
          //           note that <Nth> is to be treated as <name>
          //           above, it is passed by the same argument
          //      22 =
          //

          int        maneuverCode;
```

```
        // name of street being joined, or destination on signage
        string    targetName;

        boolean   aggregatedToNext;     // TRUE if aggregated to next
    maneuver

        // distance from previous maneuver, in meters
        int       distance;

       // Estimated driving time from previous maneuver, in seconds.
       // How time is estimated is up to the implementation.
       int     time;

    }  MapManeuverDescription;
```

5.3.5.1.2 MapRouteGuidanceControl

This structure contains the criteria to be used by the MapGuidePath function for guiding a path.

Each of the data members of the structure is set to indicate whether a maneuver is to be reported based on the conditions described.

```
    typedef struct MapRouteGuidanceControl_s {

    // Should an instruction for name change be issued when the road
    //    changes name and there is no opportunity for maneuver?
    boolean   nameChangeNoManeuver;

    // 0 => Just crossing a road does not constitute a reportable maneuver.
    // 1 => Crossing a more important road constitutes a reportable
    //        maneuver.
    // 2 => Crossing a more important or equally important road
    //    constitutes a reportable maneuver.
    // 3 => Crossing any road constitutes a reportable maneuver.
    int       continueAcrossRoad;

    // The following two Booleans cover the case when the driver can either
    // continue straight (but on a road with a different name) or continue
    // on the same road name (but by turning).

    // Should an instruction be reported when the route does not turn,
    // but the road name changes, and it would be possible to keep the same
    // road name by turning?
    boolean   continueButNameTurns;

    // Should an instruction be reported when the route stays on the same
    // name, and the road turns, but another road continues straight?
    boolean   turnWithName;

    // Should an instruction be reported when the road turns but a
    //    lower-priority road goes straight.  This is separate form the
    //    case where a same- or higher-priority road goes straight.
    boolean   turnWithNameLowerPriorityContinues;

    // 0 => There are no administrative boundaries for which crossings
    //     constitute reportable instructions.
    // 1 => Crossing a national boundary constitutes a reportable
    //     instruction.
    // 2 => Crossing the boundary of a GDF level 1 administrative area,
//     i.e., the boundary of a prefecture, province, state, or the
//     equivalent, or a more important boundary, constitutes a
//     reportable instruction.
// 3 => Crossing the boundary of a county or the equivalent, or a
//     more important boundary, constitutes a reportable instruction.
// 4 => Crossing the boundary of a city or the equivalent, or a
//     more important boundary, constitutes a reportable instruction.
// 5 => Crossing any administrative boundary constitutes a reportable
//     instruction.
int       crossAdminBoundary;
      // Crossing a (non-road) feature constitutes a reportable maneuver.
      boolean   bCrossFeature;
      int    lowestLevel;

     // Features below this level don't get announced. Range of [0..10]
      // where 10 is highest priority and 0 is lowest.  So, Mississippi
     // River might be a 10, while the smallest of tiny creeks would be
     // 0.  If the data isn't stored in a 10 level system, the proper
     // "translation" from such a value system is made by the DAL.
      // This parameter is relevant only if the previous boolean is
   'true'.

      boolean   formOfWayChange; // The form of way changes.

      // Should a maneuver be reported at the start/end of the trip?
      boolean   startOfTrip;
      boolean   endOfTrip;

      // Preferences for name use in descending order of preference.
      // If there are fewer than 16 preferences--the normal case--
      // trailing values should be set to negative numbers.  Possible
      // values are:
      //     0 => Top-level highway number
      //        (e.g., E-roads in EU outside UK, Interstates in US,
      //           M-roads in UK)
      //     1 => Second-level highway numbers
      //        (e.g., US routes in US, A-roads in UK)
      //     2 => Third-level highway numbers
      //        (e.g., state routes in US, B-roads in UK)
      //     3 => Fourth-level highway numbers
      //        (e.g., county routes in US)
      //
      //     8 => Text names
      //     9 => Name with longest extent along the computed route
      //
     // For example, if the user prefers text names, then US highway
     // numbers, then Interstate highway numbers, then state highway
     // numbers, the first four elements in the array would be 8, 1, 0, 2.
     //
     // namePreferenceHighway[] applies to true highways, not to ordinary
     // roads that also have highway numbers.  namePreferenceHighway[]
     // applies to highways in both urban and rural areas.
     // namePreferenceUrban[] applies to roads in urban areas other than
     // highways.  namePreferenceRural[] applies to roads in rural areas
     // other than highways.  The definitions of exactly what constitutes
     // "true highways", urban areas, and rural areas is left to the DAL
     // implementation.
      int       namePreferenceHighway[16],
                 namePreferenceUrban[16],
                 namePreferenceRural[16];
```

```
        //  When we don’t know whether we are in an urban area or a rural
       // area, we use the following default:  If ‘true’ then we assume
       // Urban when we don’t know where we are.
        Boolean   bUrbanDefault;

        // When should name changes be announced?
        //     0 => Never.
        //     1 => Whenever the preferred name changes.
        //     2 => Whenever the set of names splits at a fork,
      //           or the preferred name changes.
        //     3 => Whenever the set of names changes.
  //
 int       nameChangeAnnounce;

  //  As to what name(s) is (are) returned for maneuvers: When
  // nameChangeAnnounce is 0 or 1, only the preferred name of a road
  // (as determined by namePreferenceXxx[]) is returned.  When
  // nameChangeAnnounce is 2 or 3, all names of a road are returned, in
  // the order of preference specified by namePreferenceXxx[].  The
  // difference between nameChangeAnnounce = 2 and nameChangeAnnounce
  // = 3 is as follows:  nameChangeAnnounce = 3 triggers a maneuver
  // description whenever any change to the set of names occurs.
  // On the other hand, nameChangeAnnounce = 2 triggers a maneuver
  // description only at a point at which either (i) the preferred
  // name changes or else (ii) multiple different maneuvers are
  // possible and at least one name (not necessarily the preferred
  // name) proceeds along the road taken as part of the route and at
  // least one other name (again, not necessarily the preferred name)
  // proceeds along a road not taken as part of the route.


 // If this value is n, aggregate maneuvers when the estimated time
 // between them is less than n seconds.  If this value is 0, never
 // aggregate maneuvers based on estimated driving time between the
 // maneuvers.
 int       maneuverAggregateMinimumTime;


 // If this value is n, aggregate maneuvers when the distance
 // between them is less than n meters.  If this value is 0, never
 // aggregate maneuvers based on distance.  If this value is -1, aggregate
 // at the intersection and interchange level, if this value is -2,
 // aggregate at the interchange level only.
 int       maneuverAggregateMinimumDistance;

 // Note: At most one of maneuverAggregateMinimumDistance and
 // maneuverAggregateMinimumTime should be specified (i.e., nonzero).
 // If both are nonzero, the result is determined by the DAL.


 // Replace aggregated maneuvers with "follow signs to <destination>"
 // when it is possible to do so.  Do not aggregate if we are not within
 // the number of meters listed in this parameter.  If this value
 // is zero (or negative) then we never aggregate.  Use case for this
 // parameter is as follows:  For example even if the caller wants to
 // aggregate maneuvers through an interchange, the caller may not want
 // to say “follow signs to Phoenix” and then nothing else for hundreds
 // of kilometres.
 int   maneuverAggregateToFollowSignage;
```

```
// When a maneuver leads into an unnamed road, if the next maneuver is
// sufficiently close, but otherwise would not aggregate, then
// aggregate anyway.  For example, for a series of ramps, rather than
// saying separately "turn right [onto unnamed road]", "turn left
// [onto unnamed road]", "turn right onto Main Street", one would
// aggregate the directions into a single instruction "turn right,
// followed by a left, then turn right again onto Main Street".
// As above, this is done if within the number of meters specified
// in the parameter, otherwise no aggregation.  If value is zero
// or negative, then no aggregation is ever made.
int    maneuverAggregateUnnamed;


// Aggregate two left turns (or two right turns) into a U turn.
// The decision as to when this is done is left to the implementer of
// the DAL.  That is, it is up to the DAL implementer's discretion
// whether to aggregate two consecutive left or right turns that would
// otherwise make up a U turn if the component turns are too far apart
// in either distance or driving time, or if the starting and ending
// carriageways have two different names.  This Boolean merely
// authorizes the DAL implementer to perform the aggregation.
boolean   maneuverAggregateUTurn;

// Should names be returned at all?  If the following Boolean is true, the
// targetName field in the returned MapManeuverDescription will always be
// returned as the empty string, whether the road has a name or not.
// (This is of use for applications that will not display or announce road
// names in any case.)
boolean   suppressRoadName;

}  MapRouteGuidanceControl;
```

5.3.5.2 Error codes

Error code	Description
`MapCallbackFailed`	User-defined callback function failed.
`MapCursorNotAvailable`	No more cursors can be allocated.
`MapDataFormatError`	Unexpected media data format error.
`MapInvalidArgument`	Invalid argument passed to API.
`MapInvalidRequest`	Attempt to request something nonsensical.
`MapNoMemory`	DAL has insufficient memory to complete the request.
`MapNoRecordsFound`	Search yielded no results.
`MapRouteNotFound`	Failed to compute a route.
`MapSearchOutsideRegion`	Search area does not intersect the coverage area.

5.3.5.3 Route guidance functions — MapGuidePath

```
MapReturnType MapGuidePath (
   in    MapPathHandle            pathHandle,

   in    MapRouteGuidanceControl rcComputeControl,

   in    MapPriorityType          prPriority,

   // Cursor returns MapManeuverDescriptions.
   out   MapCursorReturnTypeEnum crteCursorType = MAP_MANEUVER_DESCRIPTION,
   out   MapCursorType            cuMapRouteGuidanceSteps )
raises(MapSearchOutsideRegion,
   MapNoMemory,
   MapInvalidRequest,
   MapInvalidArgument,
   MapCursorNotAvailable,
   MapNoRecordsFound);
```

5.3.6 Positioning

5.3.6.1 Error codes

Error code	Description
MapCallbackFailed	User-defined callback function failed.
MapCursorNotAvailable	No more cursors can be allocated.
MapDataFormatError	Unexpected media data format error.
MapInvalidArgument	Invalid argument passed to API.
MapInvalidRequest	Attempt to request something nonsensical.
MapNoMemory	DAL has insufficient memory to complete the request.
MapNoRecordsFound	Search yielded no results.
MapRouteNotFound	Failed to compute a route.
MapSearchOutsideRegion	Search area does not intersect the coverage area.

5.3.6.2 Positioning functions

5.3.6.2.1 GetPosition

Given a measured position, returns a corrected matched result.

```
MapReturnType GetPosition(
   // should some form of map matching based on history should be done?
   in    boolean                useHistory,

   // a measured position which is independent of the map, measured by an
   //    external position measuring device
```

```
        in    MapPosition2DType      measuredCurrentPosition,

        // matched map position
        out   locusType              locus )
    raises(MapSearchOutsideRegion,
       MapNoMemory,
       MapInvalidRequest,
       MapInvalidArgument,
       MapNoRecordsFound);
```

5.3.6.2.2 GetNearbyMapFeatures

Get a set of nearby map features.

```
    MapReturnType GetNearbyMapFeatures(
        // should some form of map matching based on history should be done?
        in    boolean                     useHistory,

        // a measured position which is independent of the map, measured by an
        //    external position measuring device
        in    lat-longType                measuredCurrentPosition,

        // the features found should be no further away then this distance
        //    measured in centimetres?
        in    long                        delta,

        // nearby features returned in a cursor of type MapFeatureType
        out   MapCursorReturnTypeEnum crteCursorType = MAP_FEATURE,
        out   MapCursorType               cuFeatures )
    raises(MapSearchOutsideRegion,
       MapNoMemory,
       MapInvalidRequest,
       MapInvalidArgument,
       MapCursorNotAvailable,
       MapNoRecordsFound);
```

5.3.7 Map display

5.3.7.1 Data structures

5.3.7.1.1 mdDAL

typedef void * mdDAL;

5.3.7.1.2 mdParcelList

typedef void * mdParcelList;

5.3.7.1.3 mdFeatureList

typedef void * mdFeatureList;

5.3.7.1.4 mdNameList

typedef void * mdNameList;

5.3.7.1.5 mdPointList

typedef void * mdPointList;

5.3.7.1.6 mdItemID

typedef void * mdItemID;

5.3.7.1.7 mdTypeList

typedef void * mdTypeList;

5.3.7.1.8 mdPoint

```
typedef struct mdPoint_t {
 long  x, y;
} mdPoint;
```

5.3.7.2 Error codes

Error code	Description
`MapCallbackFailed`	User-defined callback function failed.
`MapCursorNotAvailable`	No more cursors can be allocated.
`MapDataFormatError`	Unexpected media data format error.
`MapInvalidArgument`	Invalid argument passed to API.
`MapInvalidRequest`	Attempt to request something nonsensical.
`MapNoMemory`	DAL has insufficient memory to complete the request.
`MapNoRecordsFound`	Search yielded no results.
`MapRouteNotFound`	Failed to compute a route.
`MapSearchOutsideRegion`	Search area does not intersect the coverage area.

5.3.7.3 Map display functions

5.3.7.3.1 GetFeatures

Gets a set of map features.

```
MapReturnType GetFeatures(
   // a handle to a set of maps from which to fetch.  See mapHandleCreate()
   in    MapHandleType          mapHandle,

   // a bounding box to frame the request.  Assumed to be a rectangle in
   // some orientation.  Described as 3 points (topLeft, bottomLeft,
   // bottomRight).  If the 3 points are not corners of a rectangle the
```

```
        // result will be the smallest rectangle such that one edge of the
        // rectangle contains the "bottomLeft" point and the "bottomRight"
        point
        // and the rectangle contains the "topLeft" point
        in    bbox3PtType            bbox,

        // an integer within the range of "published" set of levels available
        // when not used should be < 0
        in    short                  genericGenLevel,
        // an alternative to the previous parameter.  Expressed in units of
        // centimetres as an integer.  When not used should be < 1.
        // this and the previous parameter are alternative, really.  If both
        // are specified the DAL can decide how to "best" accommodate the
        // request.
        in    long                   granularityExtent,

        // language of the text strings (when appropriate) expressed as 3
        // letter MARC Code (ISO 639-2, also available in a GDF annex).
        // When not used should be an empty string (or NULL)
        in    char[3]                language,

        // priority of this function - relative to other calls
        in    MapPriorityType        prPriority,

        // a large result set of type MapFeatureType.  Returns all features as
        // described in the request
        out   MapCursorReturnTypeEnum crteCursorType = MAP_FEATURE,
        out   MapCursorType          cuFeatures )
    raises(MapSearchOutsideRegion,
       MapNoMemory,
       MapInvalidRequest,
       MapInvalidArgument,
       MapCursorNotAvailable,
       MapNoRecordsFound );
```

5.3.7.3.2 GetFeaturesFiltered

Gets a set of map features. This version is with added filtering capabilities (beyond the regular getFeatures() function).

```
    MapReturnType GetFeaturesFiltered(
        // a handle to a set of maps from which to fetch.  See mapHandleCreate()
        in    MapHandleType          mapHandle,

        // feature and/or attribute values to which the search
        // applies + = include, - = exclude
        in    featureFilterType      filter,

        // a bounding box to frame the request.  Assumed to be a rectangle in
        // some orientation.  Described as 3 points (topLeft, bottomLeft,
        // bottomRight).  If the 3 points are not corners of a rectangle the
        // result will be the smallest rectangle such that one edge of the
        // rectangle contains the "bottomLeft" point and the "bottomRight"
        point
        // and the rectangle contains the "topLeft" point
        in    bbox3PtType            bbox,

        // an integer within the range of "published" set of levels available
        // when not used should be < 0
        in    short                  genericGenLevel,
      // an alternative to the previous parameter.  Expressed in units of
      // centimetres as an integer.  When not used should be < 1.
      // this and the previous parameter are alternative, really.  If both
      // are specified the DAL can decide how to "best" accommodate the
      // request.
      in    long                  granularityExtent,

      // language of the text strings (when appropriate) expressed as 3
      // letter MARC Code (ISO 639-2, also available in a GDF annex).
      // When not used should be an empty string (or NULL)
      in    char[3]               language,

      // priority of this function - relative to other calls
      in    MapPriorityType       prPriority,

      // a large result set of type MapFeatureType.  Returns all features as
      // described in the request
      out   MapCursorReturnTypeEnum crteCursorType = MAP_FEATURE,
      out   MapCursorType            cuFeatures )
   raises(MapSearchOutsideRegion,
      MapNoMemory,
      MapInvalidRequest,
      MapInvalidArgument,
      MapCursorNotAvailable,
      MapNoRecordsFound );
```

5.3.7.3.3 Map handle functions

5.3.7.3.3.1 General

This section contains functions used to manage map handles.

5.3.7.3.3.2 mapHandleCreate

This creates a map handle from a list of maps.

(Schematically: M1 + M2 + M3 + … -> H)

```
   MapReturnType mapHandleCreate(
      in    short                    nMaps,
      in    <sequence> MapIDtype     mapID,
      out   MapHandleType            mapHandle )
   raises(MapNoMemory,
      MapInvalidRequest,
      MapInvalidArgument );
```

5.3.7.3.3.3 mapHandleCombine

Creates a map handle by combining a number of existing map handles.

(Schematically: H1 $\bigcup$ H2 $\bigcup$ H3 $\bigcup$ … -> H)

```
   MapReturnType mapHandleCombine(
      in    short                    nHandles,
      in    <sequence> MapHandleType    mapHandle,
      out   MapHandleType            untionMapHandle )
```

```
        raises(MapNoMemory,
           MapInvalidRequest,
           MapInvalidArgument );
```

5.3.7.3.3.4 mapHandleAdd

Add extra maps to an existing map handle. If they are already there, continue as before.

(Schematically: H = M1 + M2 + M3)

```
        MapReturnType mapHandleAdd(
           in    MapHandleType              mapHandle,
           in    short                      nMaps,
           in    <sequence> MapIDtype       mapID,
           out   MapHandleType              mapHandle )
        raises(MapNoMemory,
           MapInvalidRequest,
           MapInvalidArgument );
```

5.3.7.3.3.5 mapHandleRelease

Releases a map handle that is no longer necessary.

```
        MapReturnType mapHandleRelease(
           in    MapHandleType              mapHandle )
        raises(MapNoMemory,
           MapInvalidRequest,
           MapInvalidArgument );
```

5.3.7.4 Map display extended low level interface

5.3.7.4.1 Informational functions

```
        // tells us the map's coordinate system
        //     example of precision: 6 digits (i.e. -6 power)
        //     example of precision: 7 digits (i.e. -7 power)
        int  GetCoordinateSystem (
          in       MapIDtype            mapID,
          out  int                  decimalPrecision,
          out      int                  coordType);


        // tells us what is the geographic extent in the map
        int  GetRootBBox (
          in       MapIDtype            mapID,
          out  long                 xmin,
          out  long                 xmax,
          out  long                 ymin,
          out  long                 ymax);


        // get the "highest" gen-level in this map
        int  GetMaxLevel (
          in    MapIDtype            mapID);
```

```
// get the "lowest" gen-level in this map
int  GetMinLevel (
  in    MapIDtype            mapID);


// Returns number of features
// input is bbox (in form of 4 longs)
// the features are then returned in a feature list
int  GetFeatureListBBox (
  in      MapIDtype    mapID,
  in      long              xmin,
  in      long              xmax,
  in      long              ymin,
  in      long              ymax,
  out FeatureList   featureList,
  out int              featureCount);
```

5.3.7.4.2 Feature list functions

```
int  FreeFeatureList (
  in      FeatureList      featureList,
  in      MapIDtype    mapID);


// use featureIndex to "index" into the feature list
// get the feature type
int  GetFeatureType(
  in      MapIDtype    mapID,
  in      FeatureList      featureList,
  in      int                 featureIndex,
  out int                featureType);


// use featureIndex to "index" into the feature list
// get the feature class
int  GetFeatureClass (
  in      MapIDtype    mapID,
  in      FeatureList      featureList,
  in      int                 featureIndex,
  out int                featureClass);


// use featureIndex to "index" into the feature list
// get the feature dimension
int  GetDimension (
  in      MapIDtype    mapID,
  in      FeatureList      featureList,
  in      int                 featureIndex,
  out GeometryTypeEnum  featureDimension);


// use component to "index" into the point list
// get the component dimension
int   GetComponentDim (
  in      MapIDtype    mapID,
  in      PointList       pointList,
  in      int                 component,
  out  GeometryTypeEnum   componentDim);


 // use featureIndex to "index" into the feature list
 // Returns number of point components in list
 int    GetPointList (
  in      MapIDtype     mapID,
  in      FeatureList       featureList,
  in      int                 featureIndex,
  out  PointList         pointList,
  out  int                  numOfComponents);


 // use featureIndex to "index" into the feature list
 // Returns number of names in list
 int   GetNameList (
  in      MapIDtype     mapID,
  in      FeatureList       featureList,
  in      int                 index,
  out  NameList          nameList,
  out  int                  numOfNames);


 // use featureIndex to "index" into the feature list
 // Returns attribute mask
 int   GetAttributeMask (
  in      MapIDtype     mapID,
  in      FeatureList       featureList,
  in      int                 featureIndex,     // Index in featureList
  out  int                  attributeValue);
```

5.3.7.4.3 Point list functions

```
 int   FreePointList (
  in      PointList         pointList,
  in      MapIDtype     mapID);


 // how many boundaries does the nth component of the point list have?
 // Returns number of boundaries in component
 int   GetNumBnds (
  in      MapIDtype     mapID,
  in      PointList         pointList,
  in      int                 component,
  out  int                  numOfBnds);


 // how many points does the Nth component, Mth boundary of the point list
 // have?
 // Returns number of points in boundary
 int   GetNumPoints (
  in      MapIDtype     mapID,
  in      PointList         pointList,
  in      int                 component,
  in      int                 bnd,
  out  int                  numOfPoints);
```

```
// get the points of the Nth component, Mth boundary of the point list
// Returns number of points placed in buffer
int  GetPointArray (
  in      MapIDtype    mapID,
  in      PointList        pointList,
  in      int                component,
  in      int                bnd,
  out  PointBuffer       points,     // put points here
  in      int                maxPts,     // don't want more than
                               //    this number
  out  int               actualNum);     // how many actual returned
```

5.3.7.4.4 Name list functions

```
int  FreeNameList (
  in      NameList         nameList,
  in      MapIDtype    mapID);
```

```
// use index to "index" into the name list, i.e. Nth name
// stuff the string of the Nth name into textBuffer
// Returns length of text string (not buffer), text buffer is NULL
// terminated and truncated if necessary
int  GetNameText (
  in      MapIDtype    mapID,
  in      NameList         nameList,
  in      int                index,
  out  char[3]           charSet,       // what charSet is it?
  out  char              textBuffer,     // put text here
  in      int                textBufferSize,    // no bigger than this
  out  int               symbol,        // could also add symb
  out  int       actualStringLength);    // how long was
                                              // string?
```

5.3.8 Address location

5.3.8.1 Data structures

5.3.8.1.1 AddressOutRequestType

Input specifying types of address output(s) from reverse geocoding.

Subsets may occur.

```
typedef struct AddressOutRequestType_s {

    boolean   bStreetAddressUnparsed;   // return un-parsed street address
    boolean   bStreetAddressParsed;  // return parsed street address
    boolean   bAreaUnparsed;         // return un-parsed area-based
    boolean   bAreaParsed;        // return parsed area-based
    boolean   bCrossStreet;          // return nearest cross
                              //   street in each direction
    boolean   bDistance;          // return distance to nearest
                              //   cross street in each
                              //   direction
} AddressOutRequestType;
```

5.3.8.1.2 AddressOutResponseType

This is the output from reverse geocoding. The elements are listed sequentially, rather than being made members of a union, because more than one of them, or even all of them, may be requested.

```
typedef struct AddressOutResponseType_s {
   AddressType      parsedStreetBasedAddress;
   string        unparsedStreetBasedAddress;
   AddressType      parsedAreaBasedAddress;
   string        unparsedAreaBasedAddress;
   AddressType      forwardCrossStreet;
   AddressType      reverseCrossStreet;
   int           forwardDistance;
     // distance along road to next intersection (meters)
   int           reverseDistance;
      // distance along road from previous intersection (meters)
   // Landmark name to be handled in a later release.

} AddressOutResponseType;
```

5.3.8.1.3 AddressLevelType

This structure is a generic definition that is used for street address context (in the West) and is the address in many usages (in the East). It is the definition of which area entities are used and their respective generic names. For instance, in the US, one most typically uses “city” and “state”. A typical usage in Japan would be “prefecture”, “city”, “ward”, “chome”, “banchi”, “go”.

This data structure is used to describe the structure of an address submitted or returned using an AddressType.

Names for corresponding address levels are to be chosen from a standardized list, so that applications can know the semantics of the terms. Terms which are for internal use rather than to be shown to users are to be in English, in cases where there is an English word, and otherwise in the language of origin.

The list so far includes the following: **city** (same as “Stadt” in Germany, same as “si” from Korea, also “town”, “township” and other equivalents), **county** (same as “parish” from Louisiana and “borough” from Alaska), **ward** (named in English instead of the local “ku”), **province** (same as “state” from the US and Australia, “estado” from Mexico and Brazil, and “prefecture” from Japan), **country**, **myan** (from Korea), **chome** (from Japan, same as “dong” from Korea), **banchi** (from Japan, same as bunji from Korea), **go** (from Japan, same as “ho” from Korea), **tang** (from Korea), **ban** (from Korea).

```
typedef struct AddressLevelType_s {

   // number of address levels
   int                iNumAddrLevels;

   string <sequence>  sAddrLevelName;  // name for corresponding address level

} AddressLevelType;
```

5.3.8.1.4 AddressType

This structure contains the parts of an address returned by reverse geocoding the functions getAddressFromPoint(), getAddressFromLocus(), and getAddressFromFeatureID(). Depending on location, some fields may be missing. This structure is also used as an input for the GetLocsFromParsedAddress() function which geocodes from an address (either fully specified or somewhat partially specified).

```
typedef struct AddressType_s {

      string    sDirectionalPrefix; // e.g., "N"
      string    sStreetTypePrefix;  // e.g., "Rue"
      string    sStreetNameBody;    // e.g., "Main"
      string    sStreetTypeSuffix;  // e.g., "Street", "Strasse"
      string    sDirectionalSuffix; // e.g., "SW"
      boolean      bBodyConcatenated;
         // indicates whether the street name body is concatenated to the
         // street type prefix/suffix (e.g., "Hauptstrasse") or not
         // (e.g., "Main Street")
      string    sHouseNumber;
         // A string is used because not all house numbers are strictly
         // numeric (e.g., 123-45, 47A, 29 1/2)
      string    sMinHouseNumber;
      string    sMaxHouseNumber;
         // In reverse geocoding, the result may include an approximate
         // house number and a house number range in which the address is
         // known to be ("I think you're around number 25, but I know
         // you're between 1 and 100")


      int       AddressStructureID;  // which address structure is this

      string <sequence>   AddrLevelValue;  // for simplicity, use string even for
                                   // numeric values
     string     PostalCode;      // of the local country
     // In some uses, this data structure is used to specify
     // a partial address. It is therefore necessary to
     // distinguish between the case in which a field is
     // specified as not present and the case in which a
     // field is unspecified.  The following indicate the
     // status of each field.
      AddressFieldStatusEnum       DirectionalPrefixState;
      AddressFieldStatusEnum       StreetTypePrefixState;
      AddressFieldStatusEnum       StreetNameBodyState;
      AddressFieldStatusEnum       StreetTypeSuffixState;
      AddressFieldStatusEnum       DirectionalSuffixState;
      AddressFieldStatusEnum       HouseNumberState;
      // Note: BodyConcatenated, MinHouseNumber, and MaxHouseNumber are
   not
      // specified in partial addresses and so have no state indicators

     AddressFieldStatusEnum <sequence>  AddrLevelStatus;

     AddressFieldStatusEnum          PostalCodeState;

} AddressType;
```

5.3.8.1.5 AddressTypeEnum

Type of address for an address input, whether parsed or unparsed.

```
typedef enum AddressTypeEnum_e {

    ADDRESS_TYPE_STREET_ADDRESS,
    ADDRESS_TYPE_INTERSECTION,
    ADDRESS_TYPE_AREA_BASED,
    ADDRESS_TYPE_NAMED_ADDRESS,  // e.g., landmark
    POSTAL_CODE
} AddressTypeEnum;
```

5.3.8.1.6 CandidateLocType

Structure for return of one candidate location.

```
typedef struct CandidateLocType_s {

    LocationType        location;

    int             posError;
    // estimate of positional accuracy in meters

    // A complex structure returning parsed
    // address fields, with associated status fields
    AddressType         fields;

} CandidateLocType;
```

5.3.8.1.7 GeoContextEnum

Type of geographic context for a geocoding/reverse geocoding request.

```
typedef enum GeoContextEnum_e {

    GEO_CONTEXT_BBOX,
    GEO_CONTEXT_POINT_AND_RADIUS,
    GEO_CONTEXT_NAMED_AREA,
    GEO_CONTEXT_NONE_EMPTY
} GeoContextEnum;
```

5.3.8.1.8 GeoContextType

Structure defining the geographic context for a geocoding/reverse geocoding request.

```
typedef struct GeoContextType_s {

    union GeoContext switch (GeoContextEnum)  {
       case GEO_CONTEXT_BBOX:              bbox3PtType    bBox;
       case GEO_CONTEXT_POINT_AND_RADIUS:  pointAndRadiusType pointAndRadius;
       case GEO_CONTEXT_NAMED_AREA:        namedAreaType  namedArea;
       case GEO_CONTEXT_NONE_EMPTY:        None_Empty_Type    none_empty;
    }
} GeoContextType;
```

5.3.8.1.9 LocationType

Locations returned from geocoding, or input to reverse geocoding, consisting of longitude/latitude coordinates and/or sets of loci.

```
typedef struct LocationType_s {

   MapPosition2DType   coord;
   LocusList     loci;
} LocationType;
```

5.3.8.1.10 PartialAddressCompletionType

Input specifying which parts of the address structure are either to be returned or being supplied from/to the reverse geocoding functions.

More than one part can be set to “1”.

```
typedef struct PartialAddressCompletionType_s {

    boolean   bDirectional;      // return “N” and such
    boolean   bStreetType;    // return “Rue” and such
    boolean   bStreetName;    // return street name
    boolean   bHouseNumber;      // return
    boolean   bHouseNumberRange; // return
    boolean   bLandmarkName;     // return house name and such

    int       AddressLevelID; // use this structure
    boolean   <sequence> bAddrLevelValue; // return the ones which are “1”

    boolean   bPostalCode;    // return

} PartialAddressCompletionType;
```

5.3.8.1.11 SpellerOutType

Structure for return of chars and counts from a speller.

```
typedef struct SpellerOutType_s {

    string     <sequence> firstChars;
    int             count;

} SpellerOutType;
```

5.3.8.1.12 AddressFieldStatusEnum (Hungarian: afs)

In the address structures (either as input or output parameters), fields may be of the following statuses:

```
typedef enum AddressFieldStatusEnum_e {

    ADDRESS_FIELD_SPECIFIED,
    ADDRESS_FIELD_SPECIFIED_EMPTY,   // known NULL value
    ADDRESS_FIELD_UNKNOWN,    // unknown value
    ADDRESS_FIELD_NEAR_VALUE, // approximate value as input
    ADDRESS_FIELD_NEAR_MATCH, // actual nearest value(s) as output
} AddressFieldStatusEnum;
```

5.3.8.1.13 FeatureNameType

Structure for the return of feature names and distances from the ScrollerHelper function.

```
typedef struct FeatureNameType_s {

    AddressType     <sequence>   featureNames;
    int                distance;     // in meters

} FeatureNameType;
```

5.3.8.2 Error codes

Error code	Description
`MapCallbackFailed`	User-defined callback function failed.
`MapCursorNotAvailable`	No more cursors can be allocated.
`MapDataFormatError`	Unexpected media data format error.
`MapInvalidArgument`	Invalid argument passed to API.
`MapInvalidRequest`	Attempt to request something nonsensical.
`MapNoMemory`	DAL has insufficient memory to complete the request.
`MapNoRecordsFound`	Search yielded no results.
`MapRouteNotFound`	Failed to compute a route.
`MapSearchOutsideRegion`	Search area does not intersect the coverage area.

5.3.8.3 Address location functions

5.3.8.3.1 GetLocsFromUnparsedAddress

Given an unparsed address, and a geographic context, gets geocoded locations.

```
MapReturnType GetLocsFromUnparsedAddress(


   // Type of address for an address input
   in    AddressTypeEnum        addressForm,

   // Address string
   in    string                 address,
   // This is an unparsed, human-understandable string.  It can be an
   //    address, an intersection, a junction name, a landmark name, etc.

   // Geographic Context
   in    GeoContextEnum         geoContextSelect,
   in    GeoContextType         geoContextInstance,

   // ordering of candidates returned:
   //    It must be noted that alpha numeric sorting
   //    may depend on the language that is used.  For
   //    example in Spanish "ch" comes right after "c"
   in    ReturnOrderEnum        returnOrder,

   // a set of type CandidateLocsType, ordered per returnOrder
   out   CandidateLocType       <sequence> candidateLocs )
raises(MapSearchOutsideRegion,
   MapInvalidArgument,
   MapNoRecordsFound,
   MapInvalidRequest,
   MapNoMemory,
   MapRouteNotFound );
```

5.3.8.3.2 GetLocsFromParsedAddress

Given a parsed address, and a geographic context, gets geocoded locations.

```
MapReturnType GetLocsFromParsedAddress(
   // Type of address for an address input
   in    AddressTypeEnum        addressForm,

   // Address structure
   in    AddressType            parsedAddress,

   // Geographic Context
   in    GeoContextEnum         geoContextSelect,

   // ordering of candidates returned:
   //    It must be noted that alpha numeric sorting
   //    may depend on the language that is used.  For
   //    example in Spanish "ch" comes right after "c"
   in    ReturnOrderEnum        returnOrder,


   // a set of type CandidateLocsType, ordered per returnOrder
   out   CandidateLocType       <sequence> candidateLocs )
raises(MapSearchOutsideRegion,
   MapInvalidArgument,
   MapNoRecordsFound,
   MapInvalidRequest,
   MapNoMemory,
   MapRouteNotFound );
```

5.3.8.3.3 GetAddressFromPoint

Given a point (longitude/latitude), gets reverse geocoded address.

```
MapReturnType GetAddressFromPoint(

   // position to be reverse geocoded
   in    MapPosition2DType      point,

   // type(s) of address to be returned
   in    AddressOutRequestType  addressOut,

   // return address structure
   out   AddressOutResponseType candidateLocs )
raises(MapSearchOutsideRegion,
   MapInvalidArgument,
   MapNoRecordsFound,
   MapInvalidRequest,
   MapNoMemory,
   MapRouteNotFound );
```

5.3.8.3.4 GetAddressFromLocus

Given a locus, gets reverse geocoded address.

```
MapReturnType GetAddressFromLocus(

   // position to be reverse geocoded
   in    LocusType              locus,

   // type(s) of address to be returned
   in    AddressOutRequestType  addressOut,

   // return address structure
   out   AddressOutResponseType candidateLocs )
raises(MapSearchOutsideRegion,
   MapInvalidArgument,
   MapNoRecordsFound,
   MapInvalidRequest,
   MapNoMemory,
   MapRouteNotFound );
```

5.3.8.3.5 GetAddressFromFeatureID

Given a GDF feature ID for a non-street and non-intersection map feature such as a landmark, POI, park, lake, etc., gets reverse-geocoded address(es).

```
MapReturnType GetAddressFromFeatureID(

   // feature type to be reverse geocoded
   in    FeatureTypeEnum        featureType,

   // type(s) of address to be returned
   in    AddressOutRequestType  addressOut,

   // return address structure
   out   AddressOutResponseType  <sequence>      candidateLocs )
   // A sequence of answers is provided here because a large object such
   as a park may front on multiple streets and have multiple addresses.

raises(MapSearchOutsideRegion,
   MapInvalidArgument,
   MapNoRecordsFound,
   MapInvalidRequest,
   MapNoMemory,
   MapRouteNotFound );
```

5.3.8.3.6 ScrollerHelper

Given an initial substring (possibly null) for a name, returns a list of candidates for that name.

Another use of this function is to get the list of all candidates for one part of a parsed address given specifications of some or all of the other parts. For example, to get the list of all cities in a state containing a specified street name, one would fill in the state and the street name and ask for all completions of the city name starting with the empty string.

```
MapReturnType ScrollerHelper(
   // partial address string
   in    string                 address,
        // Specify which part of the address is to be completed.
        in PartialAddressCompletionType  addressCategoriesRequested,

        // Specify which part of the address is provided to the function.
        //    when specified values are "null" that is the actual value.
        in PartialAddressCompletionType  addressCategoriesSupplied,

        // reference point and radius for a distance context
        // Either NULL or radius < 0 means no filtering by distance
        in    PointAndRadiusType     radius,

        // large result set of names structures and distances
        out   MapCursorReturnTypeEnum crteCursorType = FEATURE_NAME,
        out   MapCursorType          cuNames )
    raises(MapSearchOutsideRegion,
       MapInvalidArgument,
       MapNoRecordsFound,
       MapInvalidRequest,
       MapCursorNotAvailable,
       MapNoMemory,
       MapRouteNotFound);
```

5.3.8.3.7 SpellerHelper

Given an initial substring (possibly null) for a name (of a city, state, street, etc.), returns the set of possible values for the next character and a count (perhaps approximate) of possible names which complete the string.

```
    MapReturnType SpellerHelper(

        // initial name substring
        in    string                 initSubstring,

        // Specify which part of the address is to be completed.
        in PartialAddressCompletionType  addressCategoriesRequested,

        // Specify which part of the address is provided to the function.
        //    when specified values are "null" that is the actual value.
        in PartialAddressCompletionType  addressCategoriesSupplied,

        // reference point and radius for a distance context
        // Either NULL or radius < 0 means no filtering by distance
        in    PointAndRadiusType     radius,

        // list of next characters (one each) for a name, and a count
        // of possible names which complete the string
        out   SpellerOutType         spellerOut )
    raises(MapSearchOutsideRegion,
       MapInvalidArgument,
       MapNoRecordsFound,
       MapInvalidRequest,
       MapNoMemory,
       MapRouteNotFound );
```

5.3.8.3.8 AddressLevelStructSet

Sets an address level structure with its values. The caller can specify an ID for it. If the value provided for that ID is less than zero, the DAL will assign one on its own. If a requested ID is already in use (including pre assignment by the DAL), an error condition will be raised.

```
        int AddressLevelStructSet(

           // The Structure with values provided
           in    AddressLevelType       addressLevelStruct,

           // An ID for that structure, if < 0, DAL will assign
           in int                       addressLevelStructID,

           // Address Level Struct ID
           out   int                    addressLevelStructIDassigned )
        raises(MapSearchOutsideRegion,
           MapInvalidArgument,
           MapNoRecordsFound,
           MapInvalidRequest,
           MapNoMemory,
           MapRouteNotFound );
```

5.3.8.3.9 AddressLevelStructGet

Gets the address level structure corresponding to a specified ID.

```
        int AddressLevelStructGet(

           // An ID for the requested structure
           in int                       addressLevelStructID,

           // The Structure with values provided
           out   AddressLevelType       addressLevelStruct)
        raises(MapSearchOutsideRegion,
           MapInvalidArgument,
           MapNoRecordsFound,
           MapInvalidRequest,
           MapNoMemory,
           MapRouteNotFound );
```

5.3.9 Services/POIs

5.3.9.1 Data structures

5.3.9.1.1 POIRequestType

This structure contains the type(s) of information requested for returned POIs.

```
        typedef struct POIRequestType_s {

            boolean   bPOIdescript;       // get POIs
            boolean   bExtendedAttributes;   // get extended POI attrs
            boolean   bLatLong;        // get lat/longs
            boolean   bLoci;           // get loci
            boolean   bFeatureID;      // get Feature IDs
        } POIRequestType;
```

5.3.9.1.2 POIAttrType (Hungarian: pat)

This structure contains the common/basic attributes of the POI or service.

```
typedef struct POIAttrType_s {
      FeatureNameType  POIName;      // name of feature

     FeatureTypeEnum   POIFeatureType;  // type of POI

     short          sPOIFeatureClass;   // class of POI

} POIAttrType;
```

5.3.9.1.3 POISubAttrType (Hungarian: psut)

This structure contains POI or service attributes.

```
that are only rarely used.

typedef struct POISubAttrType_s {
      MapDateTimeType  dttPOIOpeningHours;

} POISubAttrType;
```

5.3.9.1.4 POIType (Hungarian pt)

This structure defines POIs and services.

This structure is closely related to the address location category.

```
typedef struct POIType_s {
      MapDBIDType      dbidPOI_ID;      // instance ID number

      FeatureTypeEnum  POIFeatureType;  // feature type of Service or POI

      POIAttrType      paPOIAttrs;      // basic attributes

      MapPosition2DType   p2POILatLongLoc;  lat/long location of feature

      LocusType     POILocusLoc;  // locus location of feature

      int        orderOffset      // when used in sorted order
                           //    this value is the offset
                           //    from the locus of the search
                           //    units of seconds or meters
                           //    (depending on the ordering)

} POIType;
```