OGDI
Open Geospatial Datastore Interface

OGDI

Programmer Reference

Revision 1.0
Version 3.0

Prepared by:

OGDI
Research Institute

Contact: Mr. Paul Morin

Telephone: (613) 992-7666
Fax: (613) 996-3328
Internet: www.ogdi.org

Document No. OGDI-RI-98001

May 1998

© Copyright 1998 OGDI RI

DOCUMENT REVISION HISTORY

Revision	Reason for Change	Origin Date
1.0	Original document issued.	May 1998

© Copyright 1998 OGDI RI

Preface

Preface

The Open Geographic Datastore Interface (OGDI) is a simple C and Tcl/Tk programming language interface that facilitates connectivity with various geographic information data formats and/or products.

This manual addresses the following questions:

- What is the Open Geographic Datastore Interface?
- What features does OGDI offer?
- How do applications use the interface?
- How are new OGDI drivers created?

The following topics provide information about the organization of this manual, describe the knowledge necessary to use the OGDI interface effectively and specify the typographic conventions used.

Organization of this manual

This manual is divided into the following parts:

Chapter 1 Introduction to OGDI provides conceptual information about the OGDI interface;

Chapter 2 C language API reference contains syntax and semantic information for all OGDI functions;

Chapter 3 Tcl/Tk API reference contains syntax and semantic information for all OGDI functions;

Chapter 4 Utility library reference contains complete information about the various utility functions available to driver developers; and

Chapter 5 Driver Development Reference Provides instructions for developing a custom driver.

Audience

The OGDI software development kit is available for use with the C and Tcl/Tk programming languages. It runs on the Microsoft Windows/NT and Microsoft Windows/95 operating systems as well as certain UNIX operating systems. Use of the OGDI interface requires some knowledge of C and/or Tcl/Tk programming, in addition to a sound basic knowledge of Geographic Information Systems. For information about Tcl/Tk programming, please refer to John Ousterhout's Tcl/Tk manual [3]. The OGDI manual assumes you have:

- a working knowledge of the C programming language;
- a working knowledge of the Tcl/Tk programming language; and
- some basic knowledge about the theory of Geographic Information Systems (GIS).

Conventions

This manual uses the following typographic conventions.

type	style
MYMACRO	Uppercase letters indicate SQL statements, macro names and terms used at the operating-system command level.
ecs_GetURLList	The typewriter font is used for sample command lines and program code.
argument	Italicized words indicate information that the user or the application must provide, variable names that are described in a block of text, or simply word emphasis.
cln_CreateClient	Bold type indicates that syntax must be typed exactly as shown, including function names
[ ]	Brackets indicate optional items.
?option?	Question marks delimit optional parameters for Tcl procedures.
|	A vertical bar separates two mutually exclusive choices in a syntax line.
{}	Braces delimit a list of items. This can be a Tcl list, or a set of mutually-exclusive choices in a syntax line.
...	An ellipsis indicates that arguments can be repeated several times.
. . .	A column of dots indicates the continuation of previous lines.

Credits

The OGDI RI would like to thank L.A.S. Inc.,(Global Geomatics) for the bulk of the manual, SOCOMAR International for the Driver Development chapter and the rest of the OGDI Research Institute members for their contributions to this manual.

Copyright and License

Copyright 1996 Her Majesty the Queen in Right of Canada. Permission to use, copy, modify and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies, that both the copyright notice and this permission notice appear in supporting documentation, and that the name of Her Majesty the Queen in Right of Canada not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission. Her Majesty the Queen in Right of Canada makes no representations about the suitability of this software for any purpose. It is provided “as is” without express or implied warranty.

Copyright © 1998 OGDI Research Instituite
http:/www.ogdi.org
All rights reserved.
All other brand or product names are trademarks of their respective companies or organizations.

Contents

Chapter 1 Introduction

Introduction

One of the main problems with today's Geographic Information Systems (GIS) is converting and integrating geospatial data. Very often, GIS developers need to import geospatial data from different sources, which has proven to be both difficult and time consuming. Industry experts believe that 60% to 85% of the total cost of implementing a GIS can be attributed to data conversion. Geospatial data products are offered in a large variety of different and incompatible formats. For example, there are a variety of different coordinate systems and cartographic projections. Furthermore, each GIS software vendor integrates its geospatial data uniquely into its software and therefore, suppliers must typically develop versions of geospatial data products for several software packages.

The GIS industry cannot expect sustained growth until the problem of incompatible data is significantly reduced. Considering the scope and complexity of geospatial data management, the industry cannot expect that this problem will be easily resolved. Part of the problem is that both tools and data file sizes are very large compared to other information systems such as word processing programs or spreadsheets.

Geospatial data format standardization is one solution to this problem. Efforts have recently been undertaken to minimize the number of geospatial data formats in the marketplace. The Spatial Data Transfer Specifications (SDTS), the DIgital Geographic information Exchange STandard (DIGEST) and the ISO TC/211 committee on geographic information are examples of this trend. However, it is highly unlikely that the industry will move to a single standard. It is probable that there will be at least a half-dozen important standards in addition to all the proprietary commercial data products already gaining momentum in the marketplace. This means that standardization efforts alone won't solve the geospatial data conversion/integration problem.

OGDI offers a solution expected to boost and accelerate standardization efforts.

OGDI is an application programming interface (API) that uses a standardized access method to work in conjunction with GIS software packages (the application) and various geospatial data products. OGDI uses a client/server architecture to facilitate the dissemination of geospatial data products over any TCP/IP network, and a driver-oriented approach to facilitate access to several geospatial data products/formats.

OGDI provides a solution for some of the most difficult geospatial data integration problems. These solutions include:

- converting various formats into a uniform transient data structure;
- adjusting coordinate systems and cartographic projections;
- retrieving geometric and attribute data;
- accessing a growing number of geospatial data products and formats; and
- making use of the Internet as a medium to distribute geospatial data products.

OGDI was developed to offer maximal flexibility. GIS developers can use this API to make their application compatible with the large number of geospatial data formats and products available on the market. Similarly, data providers can easily build new "drivers" for their own formats so that applications using OGDI can access their datasets directly. Drivers are currently being developed for DIGEST Vector Relational Format (VRF/VPF) products, DIGEST Raster products (ADRG) and DIGEST Matrix products (DTED). Drivers will later be developed for CADRG products, GeoTIFF, STDS formats, ARC/INFO coverages, Autocad DWG and DXF, Intergraph DGN, USGS DLG-3, Mapinfo MID/MIF and others.

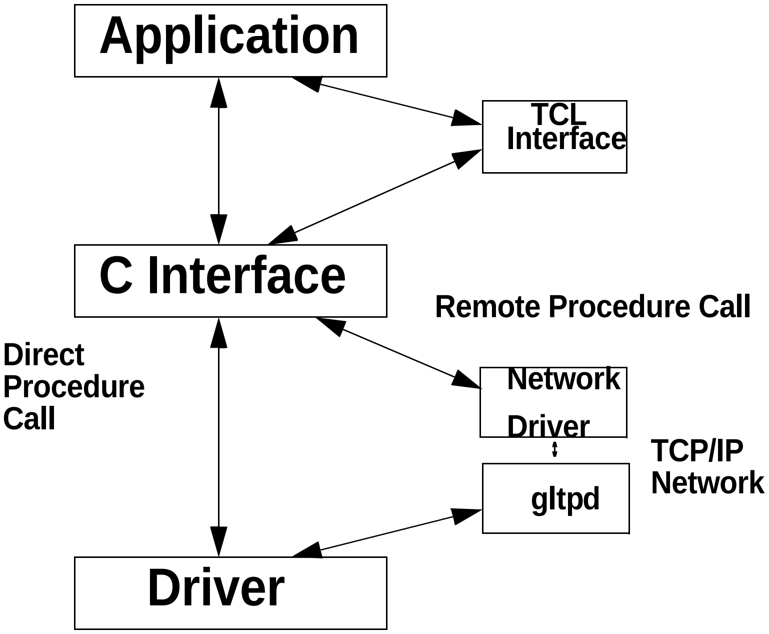
The APIs are available for UNIX operating systems (such as Solaris and Linux) and for Microsoft's Windows NT and Windows 95 operating systems. OGDI is offered as freeware, the only restrictions being those found in the copyright notice in the Preface of this guide.

Theory of Operation

The OGDI interface includes the following features:

- a library of functions that allow an application to connect to any geographic datastore (geospatial data product or format) and retrieve its contents regardless of its nature;
- a means to connect to a remote geographic datastore through the Internet or any TCP/IP network; and
- a uniform transient data structure to describe and retrieve geographic information.

The OGDI interface is open and highly flexible. The same object code can be used to access different geographic datastores (geographic information exchange formats or geographic products) without having to recompile using the "plug & play driver" concept. Applications using OGDI can ignore underlying data communication protocols between themselves and the datastore because data values are retrieved in a convenient and uniform transient data structure regardless of the source. Datastores can be accessed locally or remotely using a concept similar to that of the World Wide Web.



The interface provides the following three types of function calls:

1. **a low-level C language API;**
2. **a high-level Tcl/Tk scripting language API; and**
3. **a library of C language utility functions to facilitate driver development.**

Components

The OGDI architecture includes the following six components:

Application an application that performs processing, calls functions through the C API or Tcl/Tk API and retrieves results;

Tcl/Tk API a Tcl/Tk extension to access OGDI facilities;

C language API a dynamically-loadable C language library used to access OGDI facilities;

Drivers a dynamically-loadable library used to access various geospatial data formats;

Network Driver a special driver that allows remote access to external geospatial data drivers; and

gltpd a small application that links the remote driver to external geospatial data drivers through the Internet.

FIGURE 1. “OGDI's basic architecture” shows OGDI's basic architecture. The following sections describe each component in more detail.

FIGURE 1. OGDI’s basic architecture

Application

An application using the OGDI interface performs the following tasks:

- requests a connection with a geospatial data source;
- sends requests to the data source. These requests can be directed to specific geospatial data coverages and/or to specific geographic subregions;
- defines storage areas and data formats for the results of these requests;
- processes the results of the requests (performs spatial analysis or visualization);
- processes errors; and
- terminates the connection to the data source.

OGDI was developed mainly for GIS software vendors, but can be used in any application where GIS data retrieval is required.

C language API

The C language API is the heart of OGDI. It is a dynamically-loadable library that C programmers can use to access OGDI facilities. It is composed of 22 functions that perform the following actions:

- manage and load the geospatial data driver;
- provide an entry point to OGDI functions for each driver;
- allocate storage for geometric and attribute data;
- perform "garbage collection" of previously allocated storage;
- provide parameter validation and sequence validation for OGDI calls; and
- make all necessary coordinate and cartographic projection transformations.

See Chapter 2 C language API, on page 33, for a full description of the functions available using using this API.

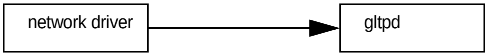
Tcl/Tk API

The OGDI API can also be accessed using the Tcl/Tk scripting language. The Tcl/Tk interface can be used for rapid prototyping of new ideas and for testing purposes. OGDI was conceived as a Tcl/Tk extension. All 22 functions of the C language API are mapped into Tcl commands. Each command is invoked using the data structures of Tcl lists and returns either a list or an associative array as a result.

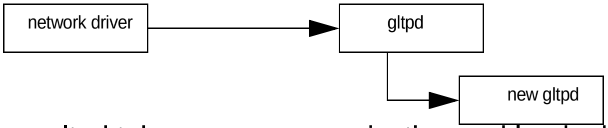
The Tcl API is a dynamically-loadable library that can easily be linked to a Tcl interpreter in the same way as other available Tcl extensions.

See John Ousterhout's Tcl/Tk Manual for more information about the Tcl/Tk scripting language. See Chapter 4 Tcl/Tk API, on page 71, for a full description of the functions available in this API.

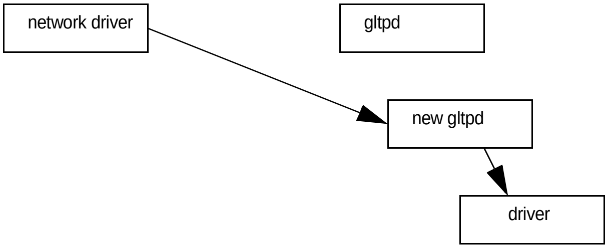
Step 1. Network Driver requests connection to gltpd



Step 2. gltpd creates a new thread (or fork) of itself



Step 3. new gltpd takes over communication and loads driver



Drivers

A driver is a dynamically-loadable library that processes C language API requests for a specific datastore. Once a driver is loaded, it receives requests, fetches information from the datastore, translates it into a uniform transient data structure and returns the results to the application.

Drivers are dynamically loaded at run time by the C language API. In this manual, the term "establishing a connection” is used to describe this process. Furthermore, the word "client” is used to describe each instance of a connection.

Network driver, gltpd, Clients and Servers

The gltpd is a small utility program that mimics the behavior of the C language API on a remote computer. The network driver is a special dynamically-loadable library that relays calls from the C API to a gltpd process running on a remote computer. The gltpd and the network driver are used together to link the application to a remote driver through a TCP/IP (Internet) network. The gltpd allows the application programmer to access remote drivers as if they were local drivers using a client/server paradigm.

When the gltpd receives its first request from an application, it creates a new thread (a fork). That new thread loads the requested driver type, takes control of the communication process with the network driver and serves all subsequent OGDI calls coming from the application. The combination of the gltpd and a specific driver becomes a server to the client (i.e.: the application's connection).

FIGURE 2. "How a network driver connects with the gltpd" shows how a network driver connects with the gltpd.

FIGURE 2. How a network driver connects with the gltpd

For a programmer using OGDI, there is no difference between a local and a remote driver. The gltpd and the network driver transparently handle the communication protocol and automatically provide data transformation between incompatible processor architectures. In the current implementation, the gltpd and the network driver are based on the ONC RPC 4.0 protocol.

To standardize vocabulary, the term client is used to describe a connection made by an application and the term server is used to describe an instance of one driver connected to one application.

Uniform Resource Locators

Each connection between the application (i.e. a client) and a driver (i.e. a server) is defined by an ASCII string similar to the World Wide Web's Uniform Resource Locators (URLs).

Each string is prefixed with the word gltp (analogous to URL prefixes like http or ftp). The prefix is followed by a hostname for remote driver access, a driver descriptor and then a file pathname that indicates the location of the datastore. The hostname is not used when accessing a local datastore.

```
gltp:[//<hostname>]/<name of driver>/<pathname>
```

The presence of the hostname string indicates that a connection to a remote driver using gltpd is being made.

The following are a few examples of connection strings:

```
gltp://copernic.las.com/grass/las3/gis/spearfish/PERMANENT
```

Describes a GRASS datastore named /las3/gis/spearfish/PERMANENT located on the host computer copernic.las.com.

```
gltp://jupiter.drev.dnd.ca/vrf/cdrom/dcw/noamer
```

Describes a DIGEST-VRF datastore named /cdrom/dcw/noamer located on the host computer jupiter.drev.dnd.ca.

```
gltp:/vrf/cdrom/dcw/noamer
```

Describes a DIGEST-VRF datastore named /cdrom/dcw/noamer located on a local host computer.

```
gltp:/grass/C:/spearfish/PERMANENT
```

Describes a GRASS datastore in the directory C:/spearfish/PERMANENT on a machine running Windows 95 or Windows NT.

Projection

The projection descriptor is a string that defines the server or client cartographic projection. If required, information retrieved from the server is transformed to the projection of the client. Arguments and parameters defined in the descriptor vary according to the projection selected and references should be made to specific projection documentation.

Options are processed from left to right, and re-entry of an option is ignored.

+proj=name is required for the selection of the cartographic transformation and the name is an acronym for the desired projection. Projection acronyms are defined in TABLE 1.“list of all valid projection acronyms” on page 222.

.

+R=R specifies that the projection should be computed as a spherical Earth with a radius corresponding to the numeric value you enter in this field.

+ellps=acronym selects standard, predefined ellipsoid figures. For spherical-only projections, the major axis is used as the radius. Valid ellipsoids are listed in TABLE 2.“list of valid ellipsoids” on page 224.

+a=a defines an elliptical Earth's major axis.

+es=e defines an elliptical Earth's squared eccentricity. Optionally, +b=b, +e=e, +rf=1/f or +f=f can be used whereb, e and f are minor axes, representing eccentricity and flattening, respectively.

+R_A must be used with elliptical Earth parameters. It specifies that spherical computations be used with the radius of a sphere that has a surface area equivalent to the selected ellipsoid.

+R_V can be used in a similar manner to calculate the sphere radius of an ellipse of equivalent volume.

+R_a must be used with elliptical Earth parameters. The spherical radius of the arithmetic mean of the major and the minor axes is used. +R_g and +R_h can be used for the equivalent geometric or harmonic mean of the major and minor axes.

+R_lat_a=o must be used with elliptical Earth parameters. The spherical radius of the arithmetic mean of the principal radii of the ellipsoid at latitude o is used.

+R_lat_g=o can be used for equivalent geometric means of the principle radii.

+x_0=x specifies false easting; the value entered is added to the x value of the Cartesian coordinate. This is used in grid systems to avoid negative grid coordinates.

+y_0=y specifies false northing; the value entered is added to the y value of the Cartesian coordinate. This is used in grid systems to avoid negative grid coordinates.

+lon_0=l specifies the central meridian. Along with +lat_0=l, it normally determines the geographic origin of the projection.

+lat_0=l specifies the central parallel. See +lon_0=l.

+units=name allows you to select the unit of measurement to which the Cartesian coordinates will be converted. Valid units are listed in TABLE 3.“list of valid units” on page 226.

+geoc when this option is selected, it specifies that geographic data coordinates are to be treated as geocentric.

+over inhibits the reduction of input longitude to a range between -180 degrees and +180 degrees of the central meridian.

+zone=n is used for UTM and MTM zone selection.

Data Model

The OGDI data model can currently handle two types of geographic data:

Vector Data which are composed of 4 subtypes of features (and divided into 3 subtypes of primitives which are not yet implemented):

1. **Line Features;**
2. **Area Features (each composed of one or more rings);**
3. **Point Features; and**
4. **Text Features.**

Matrix Data (Rasters) for information pertaining to points at regularly identified intervals. This data model is largely inspired by the DIGEST data model.

Each feature (and primitive) has a corresponding C data structure used by the C language API and by all servers. In addition, OGDI uses a number of supporting C data structures to describe geographic regions, attribute formats, raster meta-data and others. The following section describes the most important structures. All other structures are described in C Language API Commands, on page 50.

Line Feature

Line features are composed of two or more coordinates. Line features must be homogenous in direction.

```
struct ecs_Line {
    struct {
        u_int c_len;
        ecs_Coordinate *c_val;
    } c;
};
```

The c_len variable indicates the number of coordinates that describe this linear feature. Each coordinate c_val is defined by the following C sub-structure:

```
struct ecs_Coordinate {
    double x;
```

```
    double y;
};
typedef struct ecs_Coordinate ecs_Coordinate;
```

The current model can only support 2 dimensional vector representation.

Area Feature

Area features are composed of one or more rings. Rings are similar to line features except that the last coordinate is always equal to the first. Each area feature can be composed of several rings.

```
struct ecs_FeatureRing {
    ecs_Coordinate centroid;
    struct {
        u_int c_len;
        ecs_Coordinate *c_val;
    } c;
};
typedef struct ecs_FeatureRing ecs_FeatureRing;

struct ecs_Area {
    struct {
        u_int ring_len;
        ecs_FeatureRing *ring_val;
    } ring;
};
```

Point Feature

Point features are composed of a single instance of ecs_Coordinate.

```
struct ecs_Point {
    ecs_Coordinate c;
};
```

Text Feature

Text features are similar to Point features except for the fact that they also hold a text string.

```
struct ecs_Text {
```

```
    char *desc;
    ecs_Coordinate c;
};
```

Matrix Feature

Matrices (rasters) are accessed on a line-by-line basis. Each raster line is described as follows:

```
struct ecs_Matrix {
    struct {
        u_int x_len;
        u_int *x_val;
    } x;
};
```

Geographic Region

The following data structure is used to delimit a geographic region of interest:

```
struct ecs_Region {
    double north;
    double south;
    double east;
    double west;
    double ns_res;
    double ew_res;
};
```

The north, south, east, west parameters are used to geographically delimit the region. ns_res and ew_res are used to specify the target resolution for matrix coverages.

Basic application steps

To interact with a datastore, a simple application goes through the following steps:

- Establish a connection (i.e.: create a client).
- Select a geographic region.
- Select a layer (coverage).
- Extract objects sequentially or randomly.
- Process the results.
- Terminate the connection.

FIGURE 3. "Basic OGDI application steps" lists OGDI function calls that an application makes to connect to a client, select a layer (coverage), select a geographic region, retrieve objects and disconnect from the client.

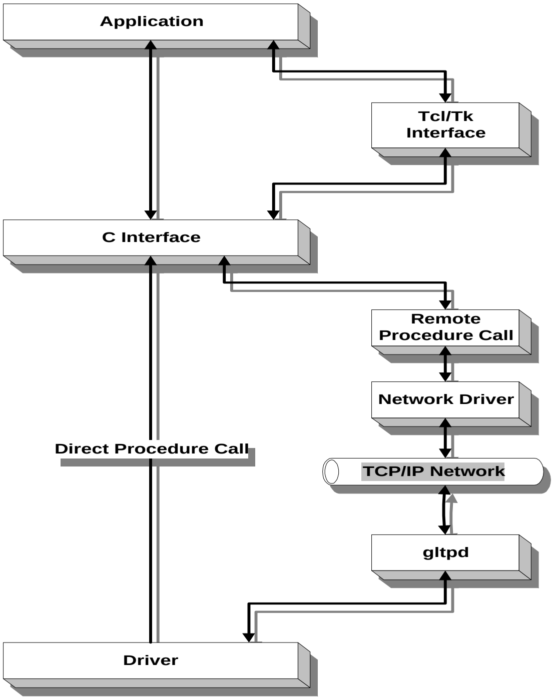


FIGURE 3. Basic OGDI application steps

Chapter 2 C language API

C Language API

This chapter explains how to use the OGDI API in real applications. The chapter is divided into two sections. The first section presents an example of how to use the OGDI library, and the second section describes all the available commands.

How can OGDI be used in an application?

The underlying philosophy of OGDI is to encapsulate all the tasks related to geographical database access in a simple and standard API. With OGDI, an application is shielded from the details of integrating a new kind of datastore. The task of navigating a datastore falls to the drivers themselves, and the C API provides a uniform way of retrieving information from these drivers regardless of the datastore format.

The following is an example of an application using OGDI to access geographical information:

```
#include "ecs.h"

char url[] = "gltp:/grass/c:/spearfish/PERMANENT";
char layer[] = "roads@PERMANENT";

main()
{
  int ClientID;
  ecs_Result *result;
  ecs_Region selectionRegion;
  ecs_LayerSelection selectionLayer;

  /* Create a client with ClientID as a reference */

  result = cln_CreateClient(&ClientID,url);

  /* The user must set a region value in the client geographic projection
*/

  selectionRegion.north = 4928000.0;
  selectionRegion.south = 4914000.0;
  selectionRegion.east  = 609000.0;
  selectionRegion.west  = 590000.0;
  selectionRegion.ns_res = 50.0;
  selectionRegion.ew_res = 50.0;
  result = cln_SelectRegion(ClientID,&selectionRegion);

  /* Define the layer to select */
```

```
  selectionLayer.Select = (char *) layer;
  selectionLayer.F = Line;
  result = cln_SelectLayer(ClientID,&selectionLayer);

  /* The application processes the result of cln_SelectLayer.*/

  result = cln_GetNextObject(ClientID);
  while (ECSSUCCESS(result)) {
    result = cln_GetNextObject(ClientID);
  }
  result = cln_ReleaseLayer(ClientID,&selectionLayer);
  result = cln_DestroyClient(ClientID);
  return 0;
}
```

This sample program is simple; it does not check for errors on the information returned by the API. A regular program would have to check whether ecs_Result contains information on whether the last command was successful or unsuccessful. (See Result and Error Handling: ecs_Result, on page 41 for more information on interpreting ecs_Result.) However, this example does provide a general overview of the various available commands.

To make a connection, call the cln_CreateClient function using a URL specified by the character string url. This creates a new client with the handle ClientId (see ClientId, on page 38). In the above example, the result is not used, but normally it should be parsed to determine whether an error has occurred.

The program then selects a coverage using the cln_SelectLayer command, gathers all the geographical objects in this coverage and terminates the session with this client. In this example only one client is open, but the application can open up to 32 clients simultaneously.

The client has an associated geographic projection and all data returned is in that projection. When the client is created, it is set by default to the same projection as that of the driver database. This example does not override the default projection, but it is possible to do this using the cln_SetClientProjection command.

The cln_SelectRegion command defines the boundaries of a geographical region within the datastore. This command allows the programmer to delimit an area with which subsequent commands will work. (See Coverage and Region Selection, on page 39).

All applications using OGDI must include the file ecs.h. This file contains a prototype of all the commands, structure definitions and macros of OGDI. To compile and link an application program with OGDI, only the ecs library is required (ecs.dll on Windows and libecs.so on UNIX). The ecsutil library, which contains some useful commands and macros, is optional.

ClientId

Each client is uniquely identified by a ClientId. This ID is the handle to a datastore that all other API commands use. It is an integer which is initially assigned during the call to the cln_CreateClient command. The ClientId can also be retrieved for an existing client by passing the client's URL as a char * to cln_GetClientIdFromURL. A ClientId is released when a client is deleted, and may then be re-used by another client.

Coverage and Region Selection

Many API commands work only on data which is within a "Selected Region” and/or "Selected Coverage”. Geographic objects are always located within the currently selected region. For example, the cln_GetNextObject command does not return any values that lie completely outside the region previously set by the cln_SelectRegion command. If no region was explicitly set, the default "Selected Region” is the global region occupied by the datastore.

"Selected Coverage” defines the layer upon which subsequent calls to coverage-oriented commands act. For example, the cln_GetNextObject command only retrieves objects within the layer that is specified by the cln_SelectLayer command. If no layer is selected, there is no default. A layer which is selected remains the current selection until the cln_SelectLayer command is used again with a new coverage, or until the cln_ReleaseLayer command has released it.

Caching

The purpose of the cache is to minimize the time required to access data. This is useful in situations where the transfer of data from a datastore is slow.

The cache is a locally held copy of a subset of a datastore. It has a geographical region which is set by the cln_SetRegionCaches command, and any number of layers may be cached within this region using the cln_LoadCache command. Any calls which retrieve data from this region first examine the cache to see whether the data is already stored there. If the required data is not found in the cache, it is retrieved from the datastore as usual.

Layers are released individually from the cache using the cln_ReleaseCache command.

Result and Error Handling: ecs_Result

To facilitate error and message handling, there is a large static structure called ecs_Result, which is used to store the results of calls to theC interface. The header is found in the hierarchy under ecs.h. (For an example of how ecs_Result can be processed, refer to the file ecs_tcl.c. This file includes code to parse ecs_Result and return it to the Tcl interpreter).

The top level of this result-reporting structure is the following:

```
enum ecs_ResultType {
    Object = 1,
    GeoRegion = 2,
    objAttributeFormat = 3,
    RasterInfo = 4,
    AText = 5
};
typedef enum ecs_ResultType ecs_ResultType;

struct ecs_ResultUnion {
    ecs_ResultType type;
    union {
        ecs_Object dob;
        ecs_Region gr;
        ecs_ObjAttributeFormat oaf;
        ecs_RasterInfo ri;
        char *s;
    } ecs_ResultUnion_u;
};
typedef struct ecs_ResultUnion ecs_ResultUnion;

struct ecs_Result {
    int error;
    char *message;
    ecs_ResultUnion res;
};
typedef struct ecs_Result ecs_Result;
```

Almost all calls to the C interface return a pointer to the *ecs_Result* structure. In all cases, where an error occurs, *ecs_Result->error* is equal to 0 (i.e. ECS_SUCCESS) or has a non-zero value. If there is an error, a human-readable message is returned in *ecs_Result->message*. Otherwise, *ecs_Result->ecs_ResultUnion* is set to contain the result from the command call.

ecs_Result->ecs_ResultUnion contains one of a number of different types of objects, depending on the type of value returned. *ecs_ResultUnion.type* indicates the type of result that is being returned, and includes another union containing the result itself. In many cases no extra information is returned, so the type is not among the enumerated type *ecs_ResultType*. The result may also be a simple character string, which is pointed to by *s*.

Since the driver is allowed to return an undefined code in *ecs_Result*, it is essential to ensure that any code which expects an **AText** *ecs_ResultType* is able to handle an undefined *ecs_ResultType*, and vice versa. For example, in the Tcl interface code, whenever an undefined *ecs_ResultType* is encountered, the string "OK" is returned by default. However, if an **AText** result message is encountered, the result-processing code is still able to return the proper character string.

ecs_Object

When *ecs_Result* returns an object, the object is contained in the following set of structures:

```
struct ecs_Geometry {
    ecs_Family family;
    union {
        ecs_Area area;
        ecs_Line line;
        ecs_Point point;
        ecs_Matrix matrix;
        ecs_Image image;
        ecs_Text text;
        ecs_Node node;
        ecs_Edge edge;
        ecs_AreaPrim ring;
    } ecs_Geometry_u;
};
typedef struct ecs_Geometry ecs_Geometry;

struct ecs_Object {
    char *Id;
    ecs_Geometry geom;
    char *attr;
    double xmin;
    double ymin;
    double xmax;
    double ymax;
};
typedef struct ecs_Object ecs_Object;
```

ecs_Object contains a character string Id, which uniquely identifies this object. ecs_Geometry describes the geometry specific to the type of object, and *attrreturns a pointer to a string describing the attributes of the object. The bounding rectangle is defined by the remaining ecs_Object parameters.

ecs_Geometry

To interpret an ecs_Geometry structure it is necessary to examine the contents of the enumerated type ecs_Geometry.family, and then interpret the corresponding type. For example, if thefamily is set to Area, you know that the ecs_Geometry.ecs_Geometry_u is an ecs_Area. For more information on these types, refer to section data-model.

The following is a list of types that can be returned within ecs_Geometry:

```
struct ecs_Coordinate {
    double x;
    double y;
};
typedef struct ecs_Coordinate ecs_Coordinate;

struct ecs_FeatureRing {
    ecs_Coordinate centroid;
    struct {
        u_int c_len;
        ecs_Coordinate *c_val;
    } c;
};
typedef struct ecs_FeatureRing ecs_FeatureRing;

struct ecs_Area {
    struct {
        u_int ring_len;
        ecs_FeatureRing *ring_val;
    } ring;
};
typedef struct ecs_Area ecs_Area;
```

An area is constructed of ring_len rings in an array ring_val. A ring has a centroid and an array of coordinates. For example:

```
struct ecs_Line {
    struct {
        u_int c_len;
        ecs_Coordinate *c_val;
```

```
} c;
};
typedef struct ecs_Line ecs_Line;

struct ecs_Point {
    ecs_Coordinate c;
};
typedef struct ecs_Point ecs_Point;
```

Points contain only a single coordinate, while lines contain an array of ecs_Coordinates of length c_val.

```
struct ecs_Matrix {
        struct {
            u_int x_len;
            u_int *x_val;
        } x;
    };
    typedef struct ecs_Matrix ecs_Matrix;

    struct ecs_Image {
        struct {
            u_int x_len;
            u_int *x_val;
        } x;
    };
    typedef struct ecs_Image ecs_Image;

    struct ecs_Text {
        char *desc;
        ecs_Coordinate c;
    };
    typedef struct ecs_Text ecs_Text;
```

Matrices and images each contain a list of data in the form of a one-dimensional array with length x_len. Text coverages include a coordinate plus a string.

ecs_Region

A region is described by its delimiting values plus an east-west and north-south resolution. For example:

```
struct ecs_Region {
    double north;
    double south;
    double east;
    double west;
    double ns_res;
    double ew_res;
};
typedef struct ecs_Region ecs_Region;
```

ecs_ObjAttributeFormat

```
enum ecs_AttributeFormat {
    Char = 1,
    Varchar = 2,
    Longvarchar = 3,
    Decimal = 4,
    Numeric = 5,
    Smallint = 6,
    Integer = 7,
    Real = 8,
    Float = 9,
    Double = 10
};
typedef enum ecs_AttributeFormat ecs_AttributeFormat;

struct ecs_ObjAttribute {
    char *name;
    ecs_AttributeFormat type;
    int length;
    int precision;
    int nullable;
};
typedef struct ecs_ObjAttribute ecs_ObjAttribute;
```

```
struct ecs_ObjAttributeFormat {
    struct {
        u_int oa_len;
        ecs_ObjAttribute *oa_val;
    } oa;
};
typedef struct ecs_ObjAttributeFormat ecs_ObjAttributeFormat;
```

Usually more than one attribute is returned in ecs_Result. Within this structure,ecs_ObjectAttributeFormat contains an array of attribute format descriptions in the form of ecs_ObjAttributeFormat structures. This array has the length oa_len. Each attribute has a type (e.g. a char), a field length and a precision (if applicable). This value may also be nullable.

ecs_RasterInfo

```
struct ecs_Category {
        long no_cat;
        u_int r;
        u_int g;
        u_int b;
        char *label;
        u_long qty;
    };
    typedef struct ecs_Category ecs_Category;

    struct ecs_RasterInfo {
        long mincat;
        long maxcat;
        int width;
        int height;
        struct {
            u_int cat_len;
            ecs_Category *cat_val;
        } cat;
    };
    typedef struct ecs_RasterInfo ecs_RasterInfo;
```

ecs_RasterInfo returns meta information related to the currently selected raster file. The categories range from mincat to maxcat, and the raster itself is a one-dimensional array which can fit into an area with width width and height height.

Each category has its own identifying number no_cat, as well as red, green and blue values. The category is also described by a label and a quantity.

Chapter 3 C Language API Commands

C Language API Commands

This chapter explains the functions available in the C API that can be used by developers at the programming level.

cln_CreateClient

NAME

cln_CreateClient creates a client (connects to a geographic datastore).

SYNOPSIS

```
ecs_Result *cln_CreateClient(ReturnedID,URL)[4]
int *ReturnedID;
char *URL;
```

ARGUMENTS

ReturnedID is the identifier number of a new client. This is the handle used by all other commands of the API.

URL this is the string used to create a new server.

DESCRIPTION

This command creates a client and loads the proper driver. The driver in turn connects to the geographic datastore identified by the URL. This command is always called before any data can be retrieved from a database.

In the case of a remote driver, the gltpd must already be running at the location pointed to by the URL.

By default, a newly-created client points to the server projection.

This command can also be used to ensure that an existing client is still valid. If you try to open a client that is already open, the previous client's number is returned as if it had been re-opened or an error message is displayed; however, the state of the connection is not affected.

cln_DestroyClient

NAME

cln_DestroyClient deletes a client and unloads the associated driver from memory. This terminates the communication with the geographic datastore.

SYNOPSIS

```
ecs_Result *cln_DestroyClient(ClientID)
int ClientID;
```

ARGUMENTS

ClientID is the client identifier.

DESCRIPTION

This command deletes a client and disconnects the interface to the geographic datastore. It also unloads the associated driver from memory.

If successful, this command returns an error code in ecs_Result->error; however, no message is returned in ecs_ResultUnion. The unsuccessful destruction of a client returns a non-zero value inecs_Result->error and a human-readable error message in ecs_Result->message.

SEE ALSO

cln_CreateClient, cln_CreateClient_OGDI

cln_GetAttributesFormat

NAME

cln_GetAttributesFormat specifies the attribute format of the currently selected layer.

SYNOPSIS

```
ecs_Result *GetAttributesFormat(ClientID)
    int ClientID;
```

ARGUMENTS

ClientID is the client identifier.

DESCRIPTION

This command returns a list that describes all the attributes of the currently selected coverage, based on the last selection made with theecs_SelectLayercommand.

If successful, an array of ecs_ObjAttributes is returned in ecs_Result. (See Appendix A, “” on page 205 for more information.)

Unlike the ecs_GetAttributesFormat command, if there is a Tcl callback procedure registered for this URL via OGDI’s Tcl interface, it is not executed. The C interface is completely independent from Tcl and does not take into account whether a Tcl callback procedure has been registered.

SEE ALSO

cln_SelectLayer, cln_SelectRegion

cln_GetDictionary

NAME

cln_GetDictionary retrieves an [incr Tcl] applet from the driver. The applet describes the contents of a geographic datastore.

SYNOPSIS

```
ecs_Result *cln_GetDictionary(ClientID)
     int ClientID;
```

ARGUMENTS

ClientID is the client identifier.

DESCRIPTION

This command returns a char * containing a Tcl list of two elements. The first is the declaration of the dictionary in the form itcl_class class_name. The second is an [incr Tcl] class definition (under itcl 1.5) that describes the contents of the datastore at the driver's end.

EXAMPLE

cln_GetDictionary

SEE ALSO

cln_SelectLayer, cln_UpdateDictionary, cln_GetDictionary

cln_GetGlobalBound

NAME

cln_GetGlobalBound specifies the driver’s global geographic region.

SYNOPSIS

```
ecs_Result *cln_GetGlobalBound(ClientID)
     int ClientID;
```

ARGUMENTS

ClientID is the client identifier.

DESCRIPTION

This command returns the server’s global geographic region. The returned value is an ecs_Region inside ecs_Result (refer to section ecs_region for more information). This command is used to return the global bounding rectangle of a datastore.

cln_GetNextObject

NAME

cln_GetNextObject specifies the next object in the currently-selected coverage.

SYNOPSIS

```
ecs_Result *cln_GetNextObject(ClientID)
     int ClientID;
```

ARGUMENTS

ClientID is the client identifier.

DESCRIPTION

This command returns the next geometric object that is either partially or totally contained within the current geographic region of the currently-selected coverage.

If successful, an ecs_Object structure is returned inecs_Result depending on the type of object that is selected. (see ecs_Object, on page 43 for details).

SEE ALSO

cln_SelectRegion, cln_SelectLayer

cln_GetObject

NAME

cln_GetObject specifies the attributes of the selected geometric object.

SYNOPSIS

```
ecs_Result *cln_GetObject(ClientID id)
     int ClientID;
     char *id;
```

ARGUMENTS

ClientID is the client identifier.

Id is the object identifier.

DESCRIPTION

This command returns the attributes of the geometric objectid. If successful, an ecs_Object of some sort is returned inecs_Result depending on the type of object that is selected. (See ecs_Object, on page 43 for details.)

If there is a Tcl callback procedure registered at the Tcl level of the OGDI interface, it is not called from this procedure.

SEE ALSO

cln_GetAttributesFormat, cln_SelectLayer

cln_GetObjectIdFromCoord

NAME

cln_GetObjectIdFromCoord retrieves the object in the currently-selected layer that is nearest to the set of specified coordinates.

SYNOPSIS

```
ecs_Result *cln_GetObjectIdFromCoord(ClientID, coord )
     int ClientID;
     ecs_Coordinate *coord;
```

ARGUMENTS

ClientID is the client identifier.

coord are the coordinates.

DESCRIPTION

This command returns the ID of the geometric object that is closest to the geographic location(x,y) in the currently selected coverage. This command only returns an Id if the (x,y) coordinate is entirely within a valid area. If a cache exists for this layer, the data is selected directly from the cache rather than from the datastore.

If successful, the Id is returned as an AText field within the ecs_Result (See Result and Error Handling: ecs_Result, on page 41).

SEE ALSO

cln_SelectLayer, cln_GetObject

cln_GetRasterInfo

NAME

cln_GetRasterInfo gathers information on the currently-selected raster coverage.

SYNOPSIS

```
ecs_Result *cln_GetRasterInfo(ClientID)
     int ClientID;
```

ARGUMENTS

ClientID is the client identifier.

DESCRIPTION

This command gathers raster information for the currently-selected layer. If the call is successful, an ecs_RasterInfo structure is returned inecs_Result. (See ecs_RasterInfo, on page 48 for details on interpreting ecs_Result).

SEE ALSO

cln_SelectLayer

cln_GetServerProjection

NAME

cln_GetServerProjection returns the server’s current projection.

SYNOPSIS

```
ecs_Result *cln_GetServerProjection(ClientID)
     int ClientID;
```

ARGUMENTS

ClientID is the client identifier.

DESCRIPTION

This command returns the cartographic projection of the server. If successful, the returned value is a AText field within a ecs_Result. The result is returned as a valid projection descriptor string (see Projection, on page 27 for details).

SEE ALSO

cln_SetServerProjection, cln_SetClientProjection

cln_LoadCache

NAME

cln_LoadCache loads data for the region set by the ecs_SetRegionCaches command.

SYNOPSIS

```
int cln_LoadCache(ClientID, ls, error_message)
     int ClientID;
     ecs_LayerSelection *ls;
     char **error_message;
```

ARGUMENTS

ClientID is the client identifier.

ls is the layer selection to load into the cache.

error_message is a pointer to a string with an error message.

DESCRIPTION

This command creates a new cache and loads object data into the cache to allow quicker recovery of objects. This data comes from the region set by thecln_SetRegionCaches command. Subsequent calls to thecln_GetObject command and the cln_GetNextObject command are routed to the cache to determine whether the data is already there.

The command takes several steps. First a check is done to determine whether the cache already exists. If it does not, a new cache is created and a coverage is allocated. All objects in the coverage are added to the cache and the new cache is added to the internal list of caches. If this is successful, the value TRUE is returned.

SEE ALSO

cln_SetRegionCaches, cln_ReleaseCache

cln_ReleaseCache

NAME

cln_ReleaseCache deletes the cache related to a coverage stored by the cln_LoadCache command.

SYNOPSIS

```
int cln_ReleaseCache(ClientID, ls, error_message )
     int ClientID;
     ecs_LayerSelection *ls;
     char **error_message;
```

ARGUMENTS

ClientID is the client identifier.

ls is the layer selection to release from the cache.

error_message is a pointer to a string with an error message.

DESCRIPTION

This command deletes the cached memory for a particular coverage. Subsequent calls to ecs_GetObject and ecs_GetNextObject for this particular coverage go to the original geographic datastore rather than to the cache. If it is successful, the value TRUE is returned, otherwise the result is FALSE and an error message is returned.

EXAMPLE

cln_ReleaseCache

SEE ALSO

cln_SetRegionCaches, cln_LoadCache

cln_ReleaseLayer

NAME

cln_ReleaseLayer - releases a layer.

SYNOPSIS

```
ecs_Result *cln_ReleaseLayer(ClientID, ls)
     int ClientID;
     ecs_LayerSelection *ls;
```

ARGUMENTS

ClientID is the client identifier.

ls is the layer information structure.

DESCRIPTION

This command releases the current layer. The geographic objects are released from the region previously selected using the cln_SelectRegion command. The cln_ReleaseLayer command deallocates the memory allocated by the cln_SelectLayercommand.

EXAMPLE

cln_ReleaseLayer

SEE ALSO

cln_GetNextObject, cln_GetDictionary, cln_UpdateDictionary, cln_SelectRegion

cln_SelectLayer

NAME

cln_SelectLayer - specifies the current coverage or layer.

SYNOPSIS

```
ecs_Result *cln_SelectLayer(ClientID, ls)
    int ClientID;
    ecs_LayerSelection *ls;
```

ARGUMENTS

ClientID is the client identifier.

ls is the layer information structure.

DESCRIPTION

This command defines the current coverage or layer. The selected layer is considered the current coverage by all other coverage-oriented command calls until this command is called again with a new value or the cln_ReleaseLayer command is called. When geographic objects are retrieved from this coverage, they are retrieved from the region previously selected by the cln_SelectRegion command. If the cln_SelectRegion command was not called, the default region is used.

If the layer is present in a local cache, data is retrieved from the cache rather than from the original datastore. (See Caching, on page 40 for details.)

SEE ALSO

cln_GetNextObject, cln_GetDictionary, cln_UpdateDictionary, cln_SelectRegion

cln_SelectRegion (OGDI)

NAME

cln_SelectRegion selects the current geographic region.

SYNOPSIS

```
ecs_Result *cln_SelectRegion(ClientID, gr)
     int ClientID;
     ecs_Region *gr;
```

ARGUMENTS

ClientID is the client identifier.

gr is the geographic region to be selected.

DESCRIPTION

This command specifies the current geographic region. Until the command is called again, all geographic objects retrieved are contained (partially or totally) within this region. The region is defined with the client’s projection. The result is returned in the standard ecs_Resultstructure.

SEE ALSO

cln_GetNextObject, cln_SelectLayer

cln_SetClientProjection

NAME

cln_SetClientProjection specifies the client’s projection.

SYNOPSIS

```
ecs_Result *cln_SetClientProjection(ClientID, projection)
     int ClientID;
     char *projection;
```

ARGUMENTS

ClientID is the client identifier.

projection is the projection descriptor string. (See Projection, on page 27 for details.)

DESCRIPTION

This command defines or changes the client projection. The string is a cartographic projection descriptor.

SEE ALSO

cln_SetServerProjection, cln_SetClientProjection cln_GetServerProjection

cln_SetRegionCaches

NAME

cln_SetRegionCaches specifies the geographic region of the data that will be kept in the caches.

SYNOPSIS

```
int cln_SetRegionCaches(ClientID, GR, error_message)
     int ClientID;
     ecs_Region *GR;
     char **error_message;
```

ARGUMENTS

ClientID is the client identifier.

GR is the geographic region.

error_message is a pointer to a string with an error message.

DESCRIPTION

This command tries to define the geographic region of the cache. If it is successful, the command returns the human-readable error message TRUE error_message; otherwise, it returns FALSE. (Note: This char * should not be de-allocated by the calling procedure.)

SEE ALSO

cln_LoadCache, cln_ReleaseCache

cln_SetServerLanguage

NAME

cln_SetServerLanguage specifies the language in which the server returns information.

SYNOPSIS

```
ecs_Result *cln_SetServerLanguage(ClientID, language)
     int ClientID;
     int language;
```

ARGUMENTS

ClientID is the client identifier.

language is the standard Microsoft country code corresponding to the selected language. For example, the code for English(US) is 001.

DESCRIPTION

This command specifies the language in which the server should return data. If cln_SetServerLanguage is not implemented in the server, an error message is returned. Many servers do not support this command.

cln_SetServerProjection

NAME

cln_SetServerProjection specifies the projection of the driver.

SYNOPSIS

```
ecs_Result *cln_SetServerProjection(ClientID, projection)
     int ClientID;
     char *projection;
```

ARGUMENTS

ClientID is the client identifier.

projection is the current server projection.

DESCRIPTION

This command defines or changes the driver projection. The projection parameter is a valid cartographic projection descriptor. (See Projection, on page 27 for details.)

EXAMPLE

cln_SetServerProjection

SEE ALSO

cln_SetClientProjection, cln_GetServerProjection

cln_UpdateDictionary

NAME

cln_UpdateDictionary returns an updated list that describes the contents of a datastore.

SYNOPSIS

```
ecs_Result *cln_UpdateDictionary(ClientID, info )
     int ClientID;
     char *info
```

ARGUMENTS

ClientID is the client identifier.

info is a string that can be used by some drivers to specify which part of the dictionary to return. Refer to the documentation about the specific driver for more information.

DESCRIPTION

This command returns a list of geographic coverages available at the driver end. This command is normally executed within a data dictionary object so that it can initialize itself and later refresh itself. The format of the returned value is specific to the driver and can usually only be correctly interpreted by a Data Dictionary object coming from the same source driver.

SEE ALSO

cln_GetDictionary

Chapter 4 Tcl/Tk API

Tcl/Tk API

This chapter explains the functions available in the Tcl/Tk API that can be used by developers at the programming level, as well as how to create a wish with the proper extension

Using the Extension with Tcl

The OGDI requires John Ousterhout’s official release version of Tcl 7.4 and Tk 4.0 under UNIX. Under Windows 95/NT, the OGDI extension for Tcl is only supported with Gordon Chaffee’s TkNT version of Tcl 7.4 and Tk 4.0. Until the official Windows 95/NT version is fully tested, is proven to be stable and provides the same level of functionality as TkNT, OGDI will only run under Mr. Chaffee’s version. OGDI will not run under Windows 3.x.

To add the extension to the interpreter under UNIX, you must add a call to ecs_Init into the Tcl_AppInit() procedure, and then ensure that the library is linked during compilation of the resulting wish. (A complete description of this process is found in John Ousterhout’s book on Tcl/Tk). The modification may look something like this:

```
int
Tcl_AppInit(interp)
    Tcl_Interp *interp;        /* Interpreter for application. */
{
    Tk_Window main;

    if (Tcl_Init(interp) == TCL_ERROR) {
        return TCL_ERROR;
    }
    if (Tk_Init(interp) == TCL_ERROR) {
        return TCL_ERROR;
    }
    if (ecs_Init(interp) == TCL_ERROR) {
        return TCL_ERROR;
    }

    .
    .
    .

}
```

To use the OGDI extension with Tcl under Windows, the interpreter itself does not require re-compilation. It is still possible to use the previous method with TkNT, but it is also possible to load a Tcl extension dynamically from within Tcl itself. This requires the name of the dynamically-loadable library, as well as the name of the function which initializes the commands so that the Tcl interpreter can use them. For the OGDI extension, this requires that the line:

```
extension ecs_tcl.dll ecs_Init
```

be called from a Tcl script. Often the most convenient place for this is within the init.tcl initialization script for your application.

Creating a Tcl Attribute-Callback Procedure

For some databases, a Tcl callback procedure must be executed during calls to ecs_GetObject, ecs_GetNextObject and ecs_GetAttributesFormat instead of going to the server for this information. The Tcl function is executed in the format:

```
procName clientID objectID TclVar
```

when ecs_GetObject or ecs_GetNextObject is called, and it is responsible for setting the TclVar according to the attributes. When ecs_GetAttributesFormat is executed, the TclProc called as

```
procName clientID {} {}
```

(i.e. with an empty list as the objectID). It returns the same value that you would expect from that function which shows the format of the attributes returned in the TclVar.

If there is no value specified for the Tcl procedure, it removes any existing Tcl procedure that has been registered for this URL. To remove an existing callback procedure, you must do this rather than passing a NULL list in place of the Tcl procedure. The basic structure of this Tcl callback should be:

```
proc myCallback {clientID objectID TclVar} {
    if {[string compare $objectID ""] == 0} {
        # we are being asked for the attributes format
        # so, return a list in the form
        # return { {TYPE <length> <precision> <nullable>} ... }
        # e.g. return {CHAR 5 0 0}

        # *** replace this line with your own code:
        return {CHAR 5 0 0}

    } else {
        global $TclVar

        if [info exists $TclVar] {
            unset $TclVar
        }
```

```
        set i 0
        # set attributes, one per line.  "attributeList" has
        # been set in advance by the programmer to be a list
        # of sublists.  Each sublist contains a list of the
        # attributes, as described by the format string
        # above.

        # *** add code here for creating the attributeList

        foreach attr $attributeList {
            set ${TclVar}($i) $attr
        incr i
        }

        # return a null list.
        return {}
    }
}
```

Chapter 5 Tcl/Tk API Commands

Tcl/Tk API Commands

This chapter explains the functions available in the Tcl/Tk API that can be used by developers at the programming level, as well as how to create a wish with the proper extension

ecs_AddAttributeFormat

NAME

ecs_AddAttributeFormat adds an attribute format to an objAttributeFormat attribute.

SYNOPSYS

```
int ecs_AddAttributeFormat (r,name,type,lenght,precision,nullable)
    ecs_Result *r;
    char *name;      ecs_AttributeFormat type;
    int length;
    int precision;
    int nullable;
```

ARGUMENTS

r is a pointer to a previously-defined structure.

name is the name of the attribute.

type is the enumerated type that describes the format of the attribute, for example, VARCHAR.

length is attribute format information.

precision is attribute format information.

nullable is attribute format information.

DESCRIPTION

This function adds an attribute format to the objAttributeFormat attribute.

SEE ALSO

ecs_SetObjAttributeFormat

ecs_AssignTclAttributeCallback

NAME

ecs_AssignTclFunction Assigns a Tcl callback procedure which is called during calls to ecs_GetObject, ecs_GetNextObject, and ecs_GetAttributesFormat.

SYNOPSIS

```
ecs_AssignTclFunction URL TclProc
```

ARGUMENTS

URL Uniform Resource Locator

tclProc A Tcl procedure that sets the attribute array or returns the format of that array.

DESCRIPTION

Sets a Tcl callback procedure to be executed during calls to ecs_GetObject, ecs_GetNextObject andecs_GetAttributesFormat instead of going to the server for this information. The Tcl function is executed in the format:

```
procName clientID objectID TclVar
```

when ecs_GetObject or ecs_GetNextObject is called, and it is responsible for setting TclVar according to the attributes. When ecs_GetAttributesFormat is executed, TclProc is called as

```
procName clientID {} {}
```

(i.e. with an empty list as the objectID). It returns the same value that you would expect from that function which shows the format of the attributes returned in TclVar.

If there is no value specified for the Tcl procedure, it removes any existing Tcl procedure that has been registered for this URL. To remove an existing callback procedure, you must do this rather than passing a NULL list in place of the Tcl procedure.

EXAMPLE

The programmer who wishes to use myProc as a callback procedure from the gltp:/GRASS/spearfish/USER1local URL would call the function in the following way:

```
ecs_AssignTclAttributeCallback gltp:/GRASS/spearfish/USER1 myProc
```

To clear this function as a callback, the syntax is:

```
ecs_AssignTclAttributeCallback gltp:/GRASS/spearfish/USER1
```

SEE ALSO

ecs_GetAttributesFormat, ecs_GetObject

ecs_BackSlash

NAME

ecs_BackSlash handles backslash sequences.

SYNOPSYS

```
char ecs_Backslash(src, readPtr)
    char *src;
    int *readPtr;
```

ARGUMENTS

src is a pointer. It points to the backslash character of a backslash sequence.

readptr is the number of characters read from src, unless NULL.

DESCRIPTION

This function extracts information from the URL and returns it in the form of arguments.

SEE ALSO

ecs_SplitURL, ecs_freeSplitURL, ecs_FindElement, ecs_CopyAndCollapse, ecs_SplitList

ecs_CreateClient

NAME

ecs_CreateClient Creates a client (connects to a geographic datastore).

SYNOPSIS

```
ecs_CreateClient URL
```

ARGUMENTS

URL Uniform Resource Locator (See Uniform Resource Locators, on page 26).

DESCRIPTION

This function creates a client and establishes a connection between this client and a geographic datastore identified by the URL. This command is always the first step before any data can be retrieved from the driver.

In the case of a remote driver, the gltpd must already be running at the location pointed by the URL.

A newly-created client is set to the projection of the server by default.

This command can also be used to ensure that an existing client is still valid. If you try to open a client that is already open, it will return the name of the client as if it had been re-opened or an error message, but it will not affect the state of the connection.

POSSIBLE ERROR MESSAGES

not able to understand this url The URL is not properly constructed.

not able to open the dynamic library The driver (.dll under Windows, .so file under UNIX) associated with this type of server could not be found. Ensure that it is in the directory that contain the executables.

EXAMPLE

The programmer who wishes to create a client that uses a local GRASS URL with a mapset located at C:\spearfish\USER1 would have to call the function in the following way:

```
ecs_CreateClient gltp:/GRASS/C:/spearfish/USER1
```

SEE ALSO

ecs_DestroyClient

ecs_DestroyClient

NAME

ecs_DestroyClient Destroys a client and unloads the associated driver from memory. This will disconnect the communication with the geographic datastore.

SYNOPSIS

```
ecs_DestroyClient URL
```

ARGUMENTS

URL Uniform Resource Locator

DESCRIPTION

This function destroys a client and disconnects the interface with the geographic datastore.

EXAMPLE

The programmer who wishes to destroy the client created for the local URL gltp:/GRASS/spearfish/USER1would have to call the function in the following way:

```
ecs_DestroyClient gltp:/GRASS/spearfish/USER1
```

SEE ALSO

ecs_CreateClient

ecs_GetAttributesFormat

NAME

ecs_GetAttributesFormat Get a list describing the attributes of the current coverage.

SYNOPSIS

```
ecs_GetAttributesFormat URL
```

ARGUMENTS

URL Uniform Resource Locator

DESCRIPTION

Returns a list that describes all attributes for the currently-selected coverage, based on the last selection made using the ecs_SelectLayercommand.

The returned value is a list composed of one sublist for each attribute. Each sublist describes the format for an object retrieved from ecs_GetObject and ecs_GetNextObject in the form of a character string followed by three integers. These values represent the attribute name (in quotation marks), the length of the attribute value field character string, the precision, and whether it is nullable or not.

If there is a tclCallback registered for this URL, it will execute this procedure instead. Please refer to ecs_AssignTclAttributeCallback, on page 80 for more details.

EXAMPLE

The programmer who wishes to get the list of attributes of the current coverage from the gltp:/GRASS/spearfish/USER1local URL would call the function in the following way:

```
ecs_GetAttributesFormat gltp:/GRASS/spearfish/USER1
```

ecs_GetDictionary

NAME

ecs_GetDictionary Retrieve an [incr tcl] applet describing the contents of a geographic datastore.

Synopsis

```
ecs_GetDictionary URL
```

ARGUMENTS

URL Uniform Resource Locator

DESCRIPTION

Returns a Tcl list of two elements. The first is the declaration of the dictionary in a sublist of the form itcl_class <class name>. The second is an [incr tcl] class definition (under itcl 1.5) that describes the content of the datastore at the driver's end. This object must be sourced, instantiated and integrated into the current session. The returned [incr tcl] object must be a syntactically and semantically valid itcl_class statement. It must also inherit from the tkg abstract class to be able to respond to drag & drop operations.

EXAMPLE

To get the dictionary from gltp:/GRASS/spearfish/USER1, the function would be called in the following way:

```
ecs_GetDictionary gltp:/GRASS/spearfish/USER1
```

SEE ALSO

ecs_UpdateDictionary

ecs_GetGlobalBound

NAME

ecs_GetGlobalBound Get the driver's global geographic region.

SYNOPSIS

```
ecs_GetGlobalBound URL
```

ARGUMENTS

URL Uniform Resource Locator

DESCRIPTION

This function returns the server's global geographic region. The returned value is a list of the form: {north south east west NS_resolution EW_resolution}. This is used to return the global bounding rectangle of a datastore.

EXAMPLE

The programmer that wishes to get the global bound of the current region from the gltp:/GRASS/spearfish/USER1local URL would call the function in the following way:

```
ecs_GetGlobalBound gltp:/GRASS/spearfish/USER1
```

ecs_GetNextObject

NAME

ecs_GetNextObject Get the next object in the currently-selected coverage.

SYNOPSIS

```
ecs_GetNextObject URL TclVar
```

ARGUMENTS

URL Uniform Resource Locator

var Tcl array variable that receives the attributes list of the next object.

DESCRIPTION

Returns the next geometric object that is either partially or totally contained within the current geographic region in the currently-selected coverage. The underlying function is intended to be used with the C Language API and is only provided in the Tcl interface for testing and debugging purposes.

This function returns attributes of geometric object Id. The returned value is a Tcl list composed of information for this coverage. The returned list is in the form:

```
{Family ObjectID {Object’s data} region}
```

The string supplied to TclVar should be a valid Tcl variable name. A call to ecs_GetNextObject erases whatever was in this variable, and creates an array in its place. This Tcl array (indexed by integers, starting from "0”) contains a list of attributes in the format returned by ecs_GetAttributesFormat.

If there is a Tcl callback procedure registered, this procedure sets the Tcl array and overwrites any values that were in this Tcl array before. However, the returned value for ecs_GetNextObject is still set by the client. Please refer to ecs_AssignTclAttributeCallback, on page 80 for more details.

When there are no more objects to be retrieved, the Tcl error "End of selection” is returned.

ecs_GetNextObject does not work in conjunction with ecs_GetObject. In other words, you cannot select an object randomly using the latter command and then continue retrieving objects from ecs_GetNextObject.

EXAMPLE

The programmer who wishes to get the next object from the current coverage and region from the local URL gltp:/GRASS/spearfish/USER1 would call the function in the following way:

```
ecs_GetNextObject gltp:/GRASS/spearfish/USER1
```

SEE ALSO

ecs_SelectRegion, ecs_SelectLayer, ecs_AssignTclAttributeCallback

ecs_GetObject

NAME

ecs_GetObject Get the attributes of the specified geometric object.

SYNOPSIS

```
ecs_GetObject URL Id TclVar
```

ARGUMENTS

URL Uniform Resource Locator

Id Object identification

TclVar A Tcl variable name which contains the attributes (if any) of the object.

DESCRIPTION

This function returns the attributes of the geometric object Id. The returned value is a Tcl list composed of information for this coverage. The returned list is in the form:

```
{Family ObjectID {list of object’s data} region}
```

The string supplied to Tclvar should be a valid Tcl variable name. A call to ecs_GetObject erases whatever was in this variable, and creates an array in its place. This Tcl array (indexed by integers, starting from "0”) contains a list of attributes in the format returned by ecs_GetAttributesFormat.

If there is a Tcl callback proc registered, it will set the Tcl array and override any values that were in this array before. However, the returned value is still set by the client. Please refer to ecs_AssignTclAttributeCallback, on page 80 for more details.

EXAMPLE

To retrieve the attributes for an object with ID "3” from the local URL gltp:/GRASS/spearfish/USER1 the prgrammer would call the function in the following way:

```
ecs_GetObject gltp:/GRASS/spearfish/USER1 3 x
```

The attributes will be placed in the Tcl variable x(0).

SEE ALSO

ecs_SelectLayer

ecs_GetObjectIdFromCoord

NAME

ecs_GetObjectIdFromCoord Get the object that is nearest to a set of coordinates in the current layer.

SYNOPSIS

```
ecs_GetObjectIdFromCoord URL x y
```

ARGUMENTS

URL Uniform Resource Locator

x Geographic coordinate (latitude) in client’s projection.

y Geographic coordinate (longitude) in client’s projection.

DESCRIPTION

Returns the Id of the geometric object that is the closest to the geographic location(x,y) in the currently-selected coverage. For an area, this function returns an Id if the (x,y) coordinate is fully within the valid area.

EXAMPLE

Given a point in latitude and longitude, an object at that point can be retrieved from gltp:/GRASS/spearfish/USER1 using:

```
ecs_GetObjectIdFromCoord gltp:/GRASS/spearfish/USER1 608525 4925130
```

SEE ALSO

ecs_SelectLayer, ecs_GetObject

ecs_GetRasterInfo

NAME

ecs_GetRasterInfo Get information on the currently-selected raster coverage.

SYNOPSIS

```
ecs_GetRasterInfo URL
```

ARGUMENTS

URL Uniform Resource Locator

DESCRIPTION

Get the information on the current raster coverage. This returns a Tcl list in the form:

```
{ {minimum category} {maximum category} width height } { {category-1}
{category-2} ... {category-n} }. Each element in the category list
contains a list describing a category:{ {category number} r g b {label}
{quantity} }
```

EXAMPLE

The programmer who wishes to get the information on the current raster coverage from the URL gltp:/GRASS/spearfish/USER1local URL would call the function in the following way:

```
ecs_GetRasterInfo gltp:/GRASS/spearfish/USER1
```

ecs_GetServerProjection

NAME

ecs_GetServerProjection Return the server's current projection.

SYNOPSIS

```
ecs_GetServerProjection URL
```

ARGUMENTS

URL Uniform Resource Locator

DESCRIPTION

Returns the cartographic projection of the server. The returned value is a projection descriptor string (see Uniform Resource Locators, on page 26 for more details).

EXAMPLE

The programmer who wishes to get the server's current projection from the local URL gltp:/GRASS/spearfish/USER1would call the function in the following way:

```
ecs_GetServerProjection gltp:/GRASS/spearfish/USER1
```

SEE ALSO

ecs_SetServerProjection, ecs_SetClientProjection

ecs_GetURLList

NAME

ecs_GetURLList Return the list of currently-established connections to geographic datastores.

SYNOPSIS

```
ecs_GetURLList
```

none This command takes no arguments.

DESCRIPTION

Returns a Tcl list of currently-established driver connections.

EXAMPLE

The programmer who wishes to get the list of currently-established server connections would call the function in the following way:

```
ecs_URLList
```

ecs_LoadCache

NAME

ecs_LoadCache Load data for the region set by ecs_SetCache into the cache.

SYNOPSIS

```
ecs_LoadCache URL Coverage
```

ARGUMENTS

URL Uniform Resource Locator

family the Family of the objects of the geographic data set. This must be one of Area, Line, Point, Matrix, Image, Text, Edge, Face, Node or Ring. Note the capitalization of the family name.

coverage String describing the geographic data set to be selected.

DESCRIPTION

Creates a new cache and loads object data into the cache to allow quicker recovery of objects. This data comes from the region set by ecs_SetCache. Subsequent calls to ecs_GetObject and ecs_GetNextObject will look in the cache to see if the data is already there.

The function takes several steps. First, a check is done to determine whether the cache already exists. If not, a new one is created and a coverage is allocated. All objects in the coverage are added to the cache and the new cache is added to the internal list of caches. If this is successful, the value TRUE is returned.

EXAMPLE

The programmer who wishes to set the cache for gltp:/GRASS/spearfish/USER1 would call the function in the following way:

```
ecs_LoadCache gltp:/GRASS/spearfish/USER1 Line roads@PERMANENT
```

SEE ALSO

ecs_LoadCache, ecs_ReleaseCache

ecs_ReleaseCache

NAME

ecs_ReleaseCache Release the coverage that was cached by ecs_LoadCache.

SYNOPSIS

```
ecs_ReleaseCache URL Family Coverage
```

ARGUMENTS

URL Uniform Resource Locator

family the Family of the objects of the geographic data set. This must be one of Area, Line, Point, Matrix, Image, Text, Edge, Face, Node or Ring. Note the capitalization of the family name.

coverage String describing the geographic data set to be released.

DESCRIPTION

Releases the cached memory for a particular coverage. Subsequent calls to ecs_GetObject and ecs_GetNextObject for this particular coverage will go to the original geographic datastore rather than to the cache.

EXAMPLE

The programmer who wishes to release the cache that was previously loaded for the coverage roads@PERMANENT with family Line at gltp:/GRASS/spearfish/USER1would call the function in the following way:

```
ecs_ReleaseCache gltp:/GRASS/spearfish/USER1 Line roads@PERMANENT
```

SEE ALSO

ecs_LoadCache, ecs_SetCache

ecs_ReleaseLayer

NAME

ecs_ReleaseLayer Release a layer.

SYNOPSIS

```
ecs_ReleaseLayer URL family coverage
```

ARGUMENTS

URL Uniform Resource Locator

Family Family to which all the objects of the geographic data set belong.

Coverage String describing the geographic data set to be selected.

DESCRIPTION

This function releases the current coverage (layer). The geographic objects are released from the region which was previously selected by ecs_SelectRegion. ecs_ReleaseLayer releases the memory allocated by ecs_SelectLayer.

EXAMPLE

In order to release a layer in the "Line" family in the coverage roads@PERMANENT from the local URL gltp:/GRASS/spearfish/USER1the function would be used as follows:

```
ecs_ReleaseLayer gltp:/GRASS/spearfish/USER1 Line roads@PERMANENT
```

SEE ALSO

ecs_GetNextObject, ecs_GetDictionary, ecs_UpdateDictionary, ecs_SelectRegion

ecs_SelectLayer

NAME

ecs_SelectLayer Specifies the current coverage or layer.

SYNOPSIS

```
ecs_SelectLayer URL family coverage
```

ARGUMENTS

URL Uniform Resource Locator

family String describing the geographic data set to be selected. This must be one of Area, Line, Point, Matrix, Image, Text, Edge, Face, Node or Ring.

Note the capitalization of the family name.

coverage String describing a member of the geographic data set.

DESCRIPTION

This function sets the current coverage (layer). Until this function is called again with a new value or ecs_ReleaseLayer is called, the selected layer will be considered as the current coverage by all other function calls. When geographic objects are retrieved for this coverage, they will be from the region previously selected by ecs_SelectRegion. If ecs_SelectRegion has not been called, the default region will be used.

EXAMPLE

In order to select a layer in the Line family in the coverage roads@PERMANENT from the local URL gltp:/GRASS/spearfish/USER1 the function would have to be called in the following way:

```
ecs_SelectLayer gltp:/GRASS/spearfish/USER1 Line roads@PERMANENT
```

SEE ALSO

ecs_GetNextObject, ecs_GetDictionary, ecs_UpdateDictionary, ecs_SelectRegion

ecs_SetCache

NAME

ecs_SetCache Set a geographic region that will be subsequently loaded into the cache.

SYNOPSIS

```
ecs_SetCache URL Region
```

ARGUMENTS

URL Uniform Resource Locator

Region A Tcl list representing the geographic region in the form {N S E W NS_resolution EW_resolution}

DESCRIPTION

In the case where transmission of data can be somewhat slow, data can be stored on the client side of the connection. ecs_SetCache prepares a store for a specified geographic region for use by ecs_LoadCache.

Example

The programmer who wishes to set the cache for gltp:/GRASS/spearfish/USER1 would call the function in the following way:

```
ecs_SetCache gltp:/GRASS/spearfish/USER1 \
    {4928000 4914000 609000 590000 50 50}
```

SEE ALSO

ecs_LoadCache, ecs_ReleaseCache

ecs_SetClientProjection

NAME

ecs_SetClientProjection Set the projection of the client.

SYNOPSIS

```
ecs_SetClientProjection URL projection
```

ARGUMENTS

URL Uniform Resource Locator

projection Projection descriptor string. (See Projection, on page 27 for more details.)

DESCRIPTION

Sets or changes the client projection. The projection parameter is a cartographic projection descriptor.

EXAMPLE

To set the client projection for gltp:/GRASS/spearfish/USER1, the call would look like:

```
ecs_SetClientProjection gltp:/GRASS/spearfish/USER1 \
     {+proj=merc +ellps=GRS80}
```

SEE ALSO

ecs_SetServerProjection, ecs_SetClientProjection, ecs_GetServerProjection

ecs_SetServerLanguage

NAME

ecs_SetServerLanguage Sets the language in which the server returns information.

SYNOPSIS

```
ecs_SetServerLanguage URL Number
```

ARGUMENTS

URL Uniform Resource Locator

number The number corresponding to the language according to the standard Microsoft country code. For example, English(US) is 001.

DESCRIPTION

This tells the server in which language it should return data. If this is not implemented in the server, an error message is returned. Many servers do not support this function.

EXAMPLE

The programmer who would like to set the language returned by the driver to English for the URL gltp:/GRASS/spearfish/USER1 would call the function in the following way:

```
ecs_SetServerLanguage gltp:/GRASS/spearfish/USER1 1
```

ecs_SetServerProjection

NAME

ecs_SetClientProjection Set the projection of the driver.

SYNOPSIS

```
ecs_SetClientProjection URL projection
```

ARGUMENTS

URL Uniform Resource Locator

projection Projection descriptor string. (See Projection, on page 27)

DESCRIPTION

Sets or changes the driver projection. The projection parameter is a cartographic projection descriptor.

EXAMPLE

To set the driver projection for gltp:/GRASS/spearfish/USER1, the call would look like:

```
ecs_SetServerProjection gltp:/GRASS/spearfish/USER1 \
     {+proj=merc +ellps=GRS80}
```

SEE ALSO

ecs_SetClientProjection, ecs_GetServerProjection

ecs_SelectRegion

NAME

ecs_SelectRegion Set the current geographic region.

SYNOPSIS

```
ecs_SelectRegion URL region
```

ARGUMENTS

URL Uniform Resource Locator

Region A Tcl list representing the geographic region in the form {N S E W NS_resolution EW_resolution}

DESCRIPTION

Sets the current geographic region. Until the function is called again, all geographic objects retrieved will be contained (partially or totally) within this region. The region is set with the client's projection. The result is returned in the standard ecs_Result structure.

EXAMPLE

The programmer who wishes to select a region from the gltp:/GRASS/spearfish/USER1 local URL would call the function in the following way:

```
ecs_SelectRegion gltp:/GRASS/spearfish/USER1 {4928000 \
     4914000 609000 590000 50 50}
```

SEE ALSO

ecs_GetNextObject, ecs_SelectLayer

ecs_UpdateDictionary

NAME

ecs_UpdateDictionary Return an updated list that describes the contents of a datastore.

SYNOPSIS

```
ecs_UpdateDictionary URL ?dictionaryString?
```

ARGUMENTS

URL Uniform Resource Locator

dictionaryString An optional parameter that can be passed to some drivers to tell the server to return only part of the dictionary. Some servers, such as the GRASS server, do not implement this feature but others use it in order to limit the amount of data that is returned with each call to ecs_UpdateDictionary.

DESCRIPTION

Returns a list of available geographic coverages available at the driver end. This command is normally executed within a data dictionary object so it can initialize itself and later refresh itself. The format of the returned value is specific to the driver and can usually only be correctly interpreted by a Data Dictionary object coming from the same source driver.

EXAMPLE

The programmer who wishes to get an updated list that describes the content of a datastore from the local URL gltp:/GRASS/spearfish/USER1 would call the function in the following way:

```
ecs_UpdateDictionary gltp:/GRASS/spearfish/USER1
```

SEE ALSO

ecs_GetDictionary

Chapter 6 Utility library

Utility Library

This chapter introduces the developer to the utility library of functions and macros. These functions and macros are used to facilitate development by grouping into one library all the functions that are useful when developing a new driver. The functions have been divided into 5 types; geometric functions, results preparation functions, regular expression functions, miscellaneous functions and layer functions. To build a new driver for a particular datastore, developers can work with these functions or develop new ones to suit their own needs. Usually developers build new functions in the library for their particular driver.

Geometric functions include all the functions related to performing calculations on geometric objects. Result preparation functions use pointers to render ecs_Result readable in the rest of the application. Regular expression functions are mainly used for the treatment of URL strings and other validation processes. Layer functions are used to manage layers. The miscellaneous functions include functions that do not fit into the other four classes of functions. Macros are used to simplify the code-writing process for developers.

The following list shows all the functions and macros available in the library:

1 Geometric Functions

- ecs_CalcObjectMBR
- ecs_DistanceMBR
- ecs_DistanceObject
- ecs_DistanceSegment
- ecs_GetRegex

2 Results Preparation Functions

- ecs_AddRasterInfoCategory
- ecs_AddText
- ecs_AdjustResult
- ecs_CalcObjectMBR

- ecs_CleanUp
- ecs_CleanUpObject
- ecs_ResultInit
- ecs_SetError
- ecs_SetGeomArea
- ecs_SetGeomAreaRing
- ecs_SetGeomImage
- ecs_SetGeomImageWithArray
- ecs_SetGeomLine
- ecs_SetGeomMatrix
- ecs_SetGeomMatrixWithArray
- ecs_SetGeomPoint
- ecs_SetGeomText
- ecs_SetGeoRegion
- ecs_SetObjAttributeFormat
- ecs_SetObjectAttr
- ecs_SetObjectId
- ecs_SetRasterInfo
- ecs_SetSucces
- ecs_SetText

3 Regular Expression Functions

- ecs_GetRegex
- EcsGetRegError
- EcsRegComp
- EcsRegError
- EcsRegExec

4 Miscellaneous Functions

- ecs_backSlash

- ecs_CopyAndCollapse
- ecs_FindElement
- ecs_freeSplitURL
- ecs_SplitList
- ecs_SplitURL

5 Layer Functions

- ecs_FreeLayer
- ecs_GetLayer
- ecs_SetLayer

6 Macros

- ECSRESULTTYPE
- ECSRESULT
- ECSGEOMTYPE
- ECSGEOM
- ECSAREARING
- ECS_SETGEOMBOUNDINGBOX
- ECS_SETGEOMLINECOORD
- ECS_SETGEOMAREACOORD
- ECS_SETGEOMMATRIXVALUE
- ECS_SETGEOMIMAGEVALUE

Functions

ecs_AddRasterInfoCategory

NAME

ecs_AddRasterInfoCategory adds a raster information category.

SYNOPSYS

```
int ecs_AddRasterInfoCategory (r,no_cat,red,green,blue,label,qty)
    ecs_Result *r;
    long no_cat;
    unsigned int red;
    unsigned int green;
    unsigned int blue;
    char *label;
    unsigned long qty;
```

ARGUMENTS

r is a pointer to a previously-defined structure.

no_cat is the category number.

red s the red component of the category color.

green is the green component category color.

blue is the blue component category color.

label is the category label.

qty is statistical information about the raster (how many points).

DESCRIPTION

This function adds a raster information category.

SEE ALSO

ecs_SetRasterInfo

ecs_AddText

NAME

ecs_AddText concatenates text to an AText item.

SYNOPSYS

```
int ecs_AddText (r,text)
    ecs_Result *r;
    char *text;
```

ARGUMENTS

r is a pointer to a previously-defined structure.

text is the string that holds the text to be concatenated.

DESCRIPTION

This function concatenates text to an AText item.

SEE ALSO

ecs_SetText

ecs_AdjustResult

NAME

ecs_AdjustResult replaces all null strings in ecs_Result with empty strings.

SYNOPSYS

```
int ecs_AdjustResult (r)
    ecs_Result *r;
```

ARGUMENTS

`r` is a pointer to a previously-defined structure.

DESCRIPTION

This function replaces all null strings in ecs_Result with empty strings. This is done to correct a deficiency of RPC regarding data strings: if a string in ecs_Result is NULL, the software crashes. This function returns the value TRUE if the operation works correctly.

ecs_CalcObjectMBR

NAME

ecs_CalcObjectMBR calculates the bounding box of an object.

SYNOPSYS

```
int ecs_CalcObjectMBR (s,r)
    ecs_Server *s;
    ecs_Result *r;
```

ARGUMENTS

s is the structure of the driver.

r is a pointer to a previously-defined structure.

DESCRIPTION

Given an ecs_Result and a previously-defined ecs_Object, this function calculates the bounding rectangle of the object and assigns it to ecs_Result.

SEE ALSO

ecs_DistanceObject, ecs_DistanceMBR, ecs_DistanceSegment

ecs_CleanUp

NAME

ecs_CleanUp performs a complete cleanup and reinitialization of ecs_Result.

SYNOPSYS

```
int ecs_CleanUp (r)
    ecs_Result *r;
```

ARGUMENTS

`r` is a pointer to a previously-defined structure.

DESCRIPTION

This function performs a complete cleanup and reinitialisation of ecs_Result.

SEE ALSO

ecs_CleanUpObject

ecs_CleanUpObject

NAME

ecs_CleanUpObject performs a complete cleanup and reinitialisation of the ecs_Object in ecs_Result.

SYNOPSYS

```
int ecs_CleanUpObject (r)
    ecs_Result *r;
```

ARGUMENTS

`r` is a pointer to a previously-defined structure.

DESCRIPTION

This function performs a complete cleanup and reinitialisation of the ecs_Object in ecs_Result.

SEE ALSO

ecs_CleanUp

ecs_CopyAndCollapse

NAME

ecs_CopyAndCollapse copies a string and eliminates any backslashes that are not in braces.

SYNOPSYS

```
void ecs_CopyAndCollapse(count, src, dst)
    int count;
    register char *src;
    register char *dst;
```

ARGUMENTS

count is the total number of characters to copy from src.

src is the source string.

dst is the destination string.

DESCRIPTION

This function copies a string and eliminates any backslashes that are not in braces. There is no returned value. Count characters get copied from src to dst. During this process, if backslash sequences are found outside braces, the backslashes are eliminated in the copy. After scanning count characters. from source, a null character is placed at the end of dst.

SEE ALSO

ecs_SplitURL, ecs_BackSlash, ecs_freeSplitURL, ecs_FindElement, ecs_SplitList

ecs_DistanceMBR

NAME

ecs_DistanceMBR calculates the distance between a point (posx,posy) and a Minimum Bounding Rectangle (MBR).

SYNOPSYS

```
double ecs_DistanceMBR(xl,yl,xu,yu,posx,posy)
    double xl;
    double yl;
    double xu;
    double yu;
    double posx;
    double posy;
```

ARGUMENTS

x1 is the x coordinate of the first corner of the MBR.

y1 is the y coordinate of the first corner of the MBR.

xu is the x coordinate of the second corner of the MBR.

yu is the y coordinate of the second corner of the MBR.

posx is the x coordinate of the point's position.

posy is the y coordinate of the point's position.

DESCRIPTION

This function calculates the distance between a point (posx,posy) and a Minimum Bounding Rectangle (MBR). It returns the calculated distance.

SEE ALSO

ecs_DistanceObject, ecs_DistanceSegment, ecs_CalcObjectMBR

ecs_DistanceObject

NAME

ecs_DistanceObject calculates the distance between a point (posx,posy) and an ecs_Object.

SYNOPSYS

```
double ecs_DistanceObject(obj,X,Y)
    ecs_Object *obj;
    double X;
    double Y;
```

ARGUMENTS

obj is a geographic object.

X is the x coordinate of the point.

Y is the y coordinate of the point.

DESCRIPTION

This function calculates the distance between a point (posx,posy) and an ecs_Object. The function returns the calculated distance. If an error occurs, the function returns a negative value.

SEE ALSO

ecs_DistanceMBR, ecs_DistanceSegment, ecs_CalcObjectMBR

ecs_DistanceSegment

NAME

ecs_DistanceSegment calculates the distance between a point (posx,posy) and a line segment (xl,yl), (xu,yu).

SYNOPSYS

```
double ecs_DistanceSegment(xl,yl,xu,yu,posx,posy)
    double xl;
    double yl;
    double xu;
    double yu;
    double posx;
    double posy;
```

ARGUMENTS

x1 is the x coordinate of the first point of the line segment.

y1 is the y coordinate of the first point of the line segment.

xu is the x coordinate of the second point of the line segment.

yu is the y coordinate of the second point of the line segment.

posx is the x coordinate of the point’s position.

posy is the y coordinate of the point’s position.

DESCRIPTION

This function calculates the distance between a point (posx,posy) and a line segment (xl,yl), (xu,yu). The calculated distance is returned.

SEE ALSO

ecs_DistanceMBR, ecs_DistanceObject, ecs_CalcObjectMBR

ecs_FindElement

NAME

ecs_FindElement Given a pointer to a list, locates the first (or next) element in the list.

SYNOPSYS

```
int ecs_FindElement(list, elementPtr, nextPtr, sizePtr, bracePtr)
    register char *list;
    char **elementPtr;
    char **nextPtr;
    int *sizePtr;
    int *bracePtr;
```

ARGUMENTS

list is a string containing a Tcl list with zero or more elements (possibly in braces).

elementPtr returns the location of the first significant character of the first element in the list.

nextPtr returns the location of the character just after the white space following the end of the argument (i.e. the next argument or the end of the list).

sizePtr if non-zero, returns the size of the element.

bracePtr if non-zero, returns non-zero/zero to indicate that the argument was/was not in braces.

DESCRIPTION

Given a pointer to a list, this function locates the first (or next) element in the list. The returned value is normally TRUE, which means that the element was successfully located. If FALSE is returned, it means that list did not have proper list structure; interp->result contains a more detailed error message. If TRUE is returned, then *elementPtr points to the first element of list and *nextPtr points to the character just after any white space following the last character that is part of the element. If this is the last argument in the list, then *nextPtr points to the NULL character at the end of list. If sizePtr is non-NULL, *sizePtr contains the number of characters in the element. If the element is in braces, then *elementPtr points to the character after the opening brace and *sizePtr does not include either of the braces. If there are no elements in the list, *sizePtr is zero and both *elementPtr and *termPtr refer to the null character at the end of the list.

Note: this procedure does not collapse backslash sequences.

SEE ALSO

ecs_SplitURL, ecs_BackSlash, ecs_freeSplitURL, ecs_CopyAndCollapse, ecs_SplitList

ecs_FreeLayer

NAME

ecs_FreeLayer frees a specified layer.

SYNOPSYS

```
void ecs_FreeLayer(s,layer)
    ecs_Server *s;
    int layer;
```

ARGUMENTS

s is a pointer to the ecs_Server structure (given by the function or program that makes the call).

layer is the layer position.

DESCRIPTION

This function frees the layer selected by the the function or program that makes the call.

SEE ALSO

ecs_SetLayer, ecs_GetLayer

ecs_freeSplitURL

NAME

ecs_freeSplitURL deallocates all the strings used in SplitURL operations.

SYNOPSYS

```
void ecs_freeSplitURL(type,machine,path)
    char **type;
    char **machine;
    char **path;
```

ARGUMENTS

machine is the machine address containing the URL. If NULL, the server is local.

type is the server type of the DLL to be loaded.

path is the string used by the dynamic database library to set the database server. The string is specific to each kind of server.

DESCRIPTION

This function deallocates all the strings used in SplitURL operations.

SEE ALSO

ecs_splitURL, ecs_BackSlach, ecs_FindElement, ecs_CopyAndCollapse, ecs_SplitList

ecs_GetLayer

NAME

ecs_GetLayer finds a layer in the layer attribute of ecs_Server for a specified selection.

SYNOPSYS

```
int ecs_GetLayer(s,sel)
    ecs_Server *s;
    ecs_LayerSelection *sel;
```

ARGUMENTS

s is a pointer to the ecs_Server structure (given by the function or program that makes the call).

sel is the layer selection structure.

DESCRIPTION

This function finds a layer in the layer attribute of ecs_Server for a specified selection. It then returns the layer position in a table. If the layer does not exist, a negative value is returned.

SEE ALSO

ecs_SetLayer, ecs_FreeLayer

ecs_GetRegex

NAME

ecs_GetRegex allocates and defines a string with the values contained in the regular expression.

SYNOPSYS

```
int ecs_GetRegex(reg,index,chaine)
    ecs_regexp *reg;
    int index;
    char **chaine;
```

ARGUMENTS

reg is a pointer to an ecs_Regexp structure.

index is an index of the table contained in ecs_Regexp.

chaine is the returned value string.

DESCRIPTION

Allocates and defines a string with the values contained in the regular expression.

EXAMPLE

```
|
  static int compiled = 0;
  static ecs_regexp *extractor;
  char substring[100];
  int msg,i;
  char *fullpath = "walk 10km";

  if (!compiled) {
    extractor = EcsRegComp("([0-9]+) *([a-z]+)");
    compiled = 1;
  }

  if (EcsRegExec(extractor,fullpath,NULL) == 0)
    return FALSE;

  for (i=0;i<3;i++) {
```

```
    msg = ecs_GetRegex(extractor,i,substring);
    if (msg) {
     printf("%s\n",substring);
    }

|Will output:
10km
10
km
```

because it is the regular expression extracted from the string passed in the parameter.

SEE ALSO

EcsRegComp, EcsRegError, EcsGetRegError, EcsRegExec

ecs_ResultInit

NAME

ecs_ResultInit initializes ecs_Result.

SYNOPSYS

```
int ecs_ResultInit (r)
    ecs_Result *r;
```

ARGUMENTS

r is a pointer to a previously-defined structure.

DESCRIPTION

This function initializes ecs_Result for the first time. This is performed when the structure is created.

ecs_SetError

NAME

ecs_SetError defines an error code and a message in the ecs_Result.

SYNOPSYS

```
int ecs_SetError (r,errorcode,error_message)
    ecs_Result *r;
    int errorcode;
    char *error_message;
```

ARGUMENTS

r is a pointer to a previously-defined structure.

errorcode is a number representing an error code.

error_message is the string containing the error message.

DESCRIPTION

This function defines an error code and a message in ecs_Result. It does not affect the rest of ecs_Result structure.

SEE ALSO

ecs_SetSuccess

ecs_SetGeomArea

NAME

ecs_SetGeomArea defines a geographical area.

SYNOPSYS

```
int ecs_SetGeomArea (r,lenght)
    ecs_Result *r;
    unsigned int length;
```

ARGUMENTS

r is a pointer to a previously-defined structure.

length is the number of rings in this area.

DESCRIPTION

This function defines a geographical area.

SEE ALSO

ecs_SetGeomLine, ecs_SetGeomPoint, ecs_SetGeomText, ecs_SetGeomAreaRing, ecs_SetGeomMatrix, ecs_SetGeomMatrixWithArray, ecs_SetGeomImageWithArray

ecs_SetGeomAreaRing

NAME

ecs_SetGeomAreaRing defines a ring in an area.

SYNOPSYS

```
int ecs_SetGeomAreaRing (r,position,lenght,centroid_x,centroid_y)
    ecs_Result *r;
    int position;
    unsigned int length;
    double centroid_x;
    double centroid_y;
```

ARGUMENTS

r is a pointer to a previously-defined structure.

position is the position in the ring table.

length is the number of points in the ring.

centroid_x is the horizontal position that defines the centroid of the ring.

centroid_y is the vertical position that defines the centroid of the ring.

DESCRIPTION

This function defines a ring in an area.

SEE ALSO

ecs_SetGeomArea, ecs_SetGeomLine, ecs_SetGeomPoint, ecs_SetGeomText, ecs_SetGeomMatrix, ecs_SetGeomMatrixWithArray, ecs_SetGeomImage, ecs_SetGeomImageWithArray

ecs_SetGeomImage

NAME

ecs_SetGeomImage defines a geographical image.

SYNOPSYS

```
int ecs_SetGeomImage(r, size)
    ecs_Result *r;
    int size;
```

ARGUMENTS

`r` is a pointer to a previously-defined structure.

size is the number of columns in this raster row.

DESCRIPTION

This function defines a geographical image.

SEE ALSO

ecs_SetGeomArea, ecs_SetAreaRing, ecs_SetGeomLine, ecs_SetGeomPoint, ecs_SetGeomText, ecs_SetGeomMatrix, ecs_SetGeomMatrixWithArray, ecs_SetGeomImageWithArray

ecs_SetGeomImageWithArray

NAME

ecs_SetGeomImageWithArray defines an image with an array of previously allocated unsigned integers.

SYNOPSYS

```
int ecs_SetGeomImageWithArray (r, size, array)
    ecs_Result *r;
    int size;
    unsigned int *array;
```

ARGUMENTS

r is a pointer to a previously-defined structure.

size is the size of the array.

array is the table of values.

DESCRIPTION

This function defines an image with an array of previously allocated unsigned integers.

SEE ALSO

ecs_SetGeomArea, ecs_SetAreaRing, ecs_SetGeomLine, ecs_SetGeomPoint, ecs_SetGeomText, ecs_SetGeomMatrix, ecs_SetGeomImage, ecs_SetGeomMatrixWithArray,

ecs_SetGeomLine

NAME

ecs_SetGeomLine defines a geographical line.

SYNOPSYS

```
int ecs_SetGeomLine (r,lenght)
    ecs_Result *r;
    unsigned int length;
```

ARGUMENTS

r is a pointer to a previously-defined structure.

length is the number of points in the Polyline.

DESCRIPTION

This function defines a geographical line.

SEE ALSO

ecs_SetGeomPoint, ecs_SetGeomText, ecs_SetGeomArea, ecs_SetGeomAreaRing, ecs_SetGeomMatrix, ecs_SetGeomMatrixWithArray, ecs_SetGeomImage, ecs_SetGeomImageWithArray

ecs_SetGeomMatrix

NAME

ecs_SetGeomMatrix defines a geographical matrix.

SYNOPSYS

```
int ecs_SetGeomMatrix (r, size)
    ecs_Result *r;
    int size;
```

ARGUMENTS

`r` is a pointer to a previously-defined structure.

size is the number of columns in this raster row.

DESCRIPTION

This function defines a geographical matrix.

SEE ALSO

ecs_SetGeomArea, ecs_SetAreaRing, ecs_SetGeomLine, ecs_SetGeomPoint, ecs_SetGeomText, ecs_SetGeomMatrixWithArray, ecs_SetGeomImage, ecs_SetGeomImageWithArray

ecs_SetGeomMatrixWithArray

NAME

ecs_SetGeomMatrixWithArray defines a matrix with an array.

SYNOPSYS

```
int ecs_SetGeomMatrixWithArray (r, size, array)
    ecs_Result *r;
    int size;
    unsigned int *array;
```

ARGUMENTS

r is a pointer to a previously-defined structure.

size is the size of the array.

array is a table of values.

DESCRIPTION

This function defines a geographical matrix with an array of previously allocated unsigned integers.

SEE ALSO

ecs_SetGeomArea, ecs_SetAreaRing, ecs_SetGeomLine, ecs_SetGeomPoint, ecs_SetGeomText, ecs_SetGeomImage, ecs_SetGeomImageWithArray, ecs_SetGeomMatrix

ecs_SetGeomPoint

NAME

ecs_SetGeomPoint defines a geographical point.

SYNOPSYS

```
int ecs_SetGeomPoint (r,x,y)
    ecs_Result *r;
    double x;
    double y;
```

ARGUMENTS

r is a pointer to a previously-defined structure.

x is the horizontal position of the point to be assigned.

y is the vertical position of the point to be assigned.

DESCRIPTION

This function defines a geographical point.

SEE ALSO

ecs_SetGeomText, ecs_SetGeomLine, ecs_SetGeomArea, ecs_SetGeomAreaRing, ecs_SetGeomMatrix, ecs_SetGeomMatrixWithArray, ecs_SetGeomImage, ecs_SetGeomImageWithArray

ecs_SetGeomText

NAME

ecs_SetGeomText defines geographical text.

SYNOPSYS

```
int ecs_SetGeomText (r,x,y,desc)
    ecs_Result *r;
    double x;
    double y;
    char *desc;
```

ARGUMENTS

r is a pointer to a previously-defined structure.

x is the horizontal starting position of the text.

y is the vertical starting position of the text.

desc is the description string.

DESCRIPTION

This function defines geographical text.

SEE ALSO

ecs_SetGeomPoint, ecs_SetGeomLine, ecs_SetGeomArea, ecs_SetGeomAreaRing, ecs_SetGeomMatrix, ecs_SetGeomMatrixWithArray, ecs_SetGeomImage, ecs_SetGeomImageWithArray

ecs_SetGeoRegion

NAME

ecs_SetGeoRegion defines a geographic region.

SYNOPSYS

```
int ecs_SetGeoRegion (r,north,south,east,west,ns_res,ew_res)
    ecs_Result *r;
    double north;
    double south;
    double east;
    double west;
    double ns_res;
    double ew_res;
```

ARGUMENTS

r is a pointer to a previously-defined structure. north defines the north geographic region boundary.

east defines the east geographic region boundary.

south defines the south geographic region boundary.

west defines the west geographic region boundary.

ns_res defines the horizontal size of raster cells.

ew_res defines the vertical size of raster cells.

DESCRIPTION

This function defines the geographic region.

ecs_SetLayer

NAME

ecs_SetLayer adds a new layer to the layer list in the ecs_Server.

SYNOPSYS

```
int ecs_SetLayer(s,sel)
    ecs_Server *s;
    ecs_LayerSelection *sel;
```

ARGUMENTS

s is a pointer to ecs_Server structure (given by the function or program that makes the call).

sel is the layer selection structure.

DESCRIPTION

This function adds a new layer to the layer list in ecs_Server. It then returns the layer position in a table. If an error occurs during allocation, a negative value is returned.

SEE ALSO

ecs_GetLayer, ecs_FreeLayer

ecs_SetObjAttributeFormat

NAME

ecs_SetObjAttributeFormat defines the objAttributeFormat attribute.

SYNOPSYS

```
int ecs_SetObjAttributeFormat (r)
    ecs_Result *r;
```

ARGUMENTS

`r` is a pointer to a previously-defined structure.

DESCRIPTION

This function defines the objAttributeFormat attribute.

SEE ALSO

ecs_AddAttributeFormat

ecs_SetObjectAttr

NAME

ecs_SetObjectAttr defines the object attribute.

SYNOPSYS

```
int ecs_SetObjectAttr( r,attr)
     ecs_Result *r;
     char *attr;
```

ARGUMENTS

r is the structure that holds the result of the operation.

attr is the string that contains the attribute to be defined. This argument must not be NULL.

DESCRIPTION

This function defines the object attribute. Before calling this function, the developer must invoke ecs_SetGeomText, ecs_SetGeomPoint, ecs_SetGeomLine,ecs_SetGeomArea, ecs_SetGeomMatrix or ecs_SetGeomImage to initialize the geographic object.

SEE ALSO

ecs_SetGeomArea, ecs_SetGeomLine, ecs_SetGeomPoint, ecs_SetGeomText, ecs_SetGeomMatrix, ecs_SetGeomImage

ecs_SetObjectId

NAME

ecs_SetObjectId defines the attribute ID of an object.

SYNOPSYS

```
int ecs_SetObjectId (r,id)
    ecs_Result *r;
    char *id;
```

ARGUMENTS

`r` is a pointer to a previously-defined structure.

`id` is the object identifier.

DESCRIPTION

This function defines the attribute ID of an object. Before calling this function, the developer must invoke, ecs_SetGeomText, ecs_SetGeomPoint, ecs_SetGeomLine, ecs_SetGeomArea, ecs_SetGeomMatrix or ecs_SetGeomImage to initialize the geographic object.

SEE ALSO

ecs_SetGeomArea, ecs_SetGeomLine, ecs_SetGeomPoint, ecs_SetGeomText, ecs_SetGeomMatrix, ecs_SetGeomImage

ecs_SetRasterInfo

NAME

ecs_SetRasterInfo defines the RasterInfo attribute.

SYNOPSYS

```
int ecs_SetRasterInfo (r,width,height)
    ecs_Result *r;
    int width;
    int height;
```

ARGUMENTS

r is a pointer to a previously-defined structure.

width is the width of the raster.

height is the height of the raster.

DESCRIPTION

This function defines the RasterInfo attribute. For this function, you cannot have maxcat < mincat. If this occurs, ecs_AddRasterInfoCategory sets both values to the first value.

SEE ALSO

ecs_AddRasterInfoCategory

ecs_SetSuccess

NAME

ecs_SetSuccess defines a success code and flushes the previously-defined error message.

SYNOPSYS

```
int ecs_SetSuccess (r)
    ecs_Result *r;
```

ARGUMENTS

`r` is a pointer to a previously-defined structure.

DESCRIPTION

This function defines a success code and flushes the previously-defined error message. It does not affect the rest of the ecs_Result structure.

ecs_SetText

NAME

ecs_SetText defines the AText item with a string.

SYNOPSYS

```
int ecs_SetText (r,text)
    ecs_Result *r;
    char *text;
```

ARGUMENTS

r is a pointer to a previously-defined structure.

text is the string that holds the text.

DESCRIPTION

This function defines the AText item with a string.

SEE ALSO

ecs_AddText

ecs_SplitList

NAME

ecs_SplitList splits a list into its constituent fields.

SYNOPSYS

```
int ecs_SplitList(list, argcPtr, argvPtr)
    char *list;
    int *argcPtr;
    char ***argvPtr;
```

ARGUMENTS

list is a pointer to a string with a list structure.

argcPtr is a pointer to the location that indicates the number of elements in the list.

argvPtr is a pointer to a place to store a pointer to an array of pointers to list elements.

DESCRIPTION

This function splits a list into its constituent fields. The returned value is normally TRUE, which means that the list was successfully split. If FALSE is returned, it means that list did not have the proper list structure; interp->result contains a more detailed error message.ArgvPtr contains the address of an array whose elements point to the elements of list, in order.*argcPtr contains the number of valid elements in the array. A single block of memory is dynamically allocated to hold both the argv array and a copy of the list (with backslashes and braces removed in the standard way). The caller must eventually free this memory by calling free()on *argvPtr. Note: *argvPtr and *argcPtr are only modified if the procedure returns the value TRUE.

SEE ALSO

ecs_SplitURL, ecs_BackSlash, ecs_freeSplitURL, ecs_FindElement, ecs_CopyAndCollapse

ecs_SplitURL

NAME

ecs_SplitUrl extracts information from the URL and returns it in the arguments.

SYNOPSYS

```
int textbfecs_SplitURL(textiturl,machine,server,path)
    char *url;
    char **machine;
    char **server;
    char **path;
```

ARGUMENTS

url is the string containing the URL.

machine is the machine address containing the URL. If NULL, the server is local.

server is the server type of the DLL to be loaded.

path is the string used by the dynamic database library to set the database server. This string is specific to each kind of server.

DESCRIPTION

This function extracts information from the URL and returns it in the form of arguments.

SEE ALSO

ecs_freeSplitURL, ecs_BackSlash, ecs_FindElement, ecs_CopyAndCollapse, ecs_SplitList

EcsGetRegError

NAME

EcsGetRegError returns an error message.

SYNOPSYS

```
char * EcsGetRegError()
```

ARGUMENTS

None no arguments are required.

DESCRIPTION

This function is invoked by EcsRegExec or EcsRegComp when an error occurs. This function is similar to EcsRegError except that it does not contain the error string itself.

SEE ALSO

EcsRegComp, EcsRegError, EcsGetRegError, ecs_GetRegex

EcsRegComp

NAME

EcsRegComp compiles a regular expression into internal code.

SYNOPSYS

```
ecs_regexp * EcsRegComp(exp)
char *exp;
```

ARGUMENTS

exp is the string containing the regular expression.

DESCRIPTION

This function compiles a regular expression string into the internal form used for efficient pattern matching. The return value is a token for this compiled form, which can be used in subsequent calls to EcsRegExec. If an error occurs while compiling the regular expression, EcsRegComp returns the value NULL.

EXAMPLE

```
|
static ecs_regexp * extractor;

|
    extractor = EcsRegComp("([0-9]+) *([a-z]+)");
|Now the extractor contains the regular expression in internal form for
efficient pattern matching.
```

SEE ALSO

EcsRegExec, EcsRegError, EcsGetRegError, ecs_GetRegex

EcsRegError

NAME

EcsRegError returns a message when an error occurs.

SYNOPSYS

```
void EcsRegError(string)
    char *string;
```

ARGUMENTS

string the string that describes the error that occurs.

DESCRIPTION

This function is invoked by EcsRegExec or EcsRegComp when an error occurs. It saves the error message so it can be seen by the code that is called.

EXAMPLE

```
|EcsRegError("corrupted pointers’’);|
```

EcsRegExec

NAME

EcsRegExec executes the regular expression pattern matcher.

SYNOPSYS

```
int EcsRegExec(prog,string,start)
   register ecs_regexp *prog;
   register char *string;
   char *start;
```

ARGUMENTS

prog is the compiled regular expression returned previously by EcsRegComp.

string is a string in the form of a regular expression pattern.

start if this string matches a portion of some other string, this argument identifies the beginning of the larger string. If the string does not match another string, then no ^ matches are allowed.

DESCRIPTION

This function executes the regular expression pattern matcher. It returns 1 if the string contains a range of characters that match regexp, 0 if no match is found or -1 if an error occurs. When searching a string for multiple matches of a pattern, it is important to distinguish between the start of the original string and the start of the current search. For example, when searching for the second occurrence of a match, the string argument might point to the character just after the first match; however, it is important for the pattern matcher to know that this is not the start of the entire string, so that it doesn't allow ^ atoms in the pattern to match. The start argument provides this information by pointing to the start of the overall string containing the string. The start pointer should be less than or equal to the string pointer; if the start pointer is less than the string pointer, the RegExec is a sub-string RegExec and no ^ matches are allowed.

EXAMPLE

```
|
static int compiled = 0;
static ecs_regexp *extractor;
```

```
char *fullpath = "Walk 10km";
int msg,i;

  if (!compiled) {
    extractor = EcsRegComp("([0-9]+)*([a-z]+)");
    compiled = 1;
  }

  if (EcsRegExec(extractor,fullpath,NULL) == 0)
    return FALSE;
  else
    return TRUE;
```

|The result is TRUE because 10km matches the pattern to look for in the extractor.

SEE ALSO

EcsRegComp, EcsRegError, EcsGetRegError, ecs_GetRegex

C language macros

Utility functions implemented as C language macros

ECSRESULTTYPE(result) This macro indicates the object type in the ecs_Result structure.

ECSRESULT(result) This macro indicates the path of the object referenced in the structure.

ECSGEOMTYPE(result) This macro indicates the family of a geographic object.

ECSGEOM(result) This macro indicates the path of the referenced geographic object in the structure.

ECSAREARING(result,pos) This macro indicates the path of the structure up to the ring position.

ECS_SETGEOMBOUNDINGBOX(result,lxmin,lymin,lxmax,lymax) This macro assigns a bounding box to a geographic object after ecs_SetGeomText, ecs_SetGeomPoint,ecs_SetGeomLine, ecs_SetGeomArea, ecs_SetGeomMatrix or ecs_SetGeomImage is invoked. The bounding box is assigned to the ecs_Result structure without modifying the current information available in the structure.

ECS_SETGEOMLINECOORD(result,position,lx,ly) This macro assigns a point (lx,ly) at position in the known line segment of ecs_Result. This is done after the ecs_SetGeomLine function is invoked.

ECS_SETGEOMAREACOORD(result,ringpos,position,lx,ly) This macro assigns a point (lx,ly) at the position in the ring ringpos of the area in ecs_Result. This is done after the ecs_SetGeomArea and ecs_SetGeomAreaRing functions are invoked.

ECS_SETGEOMMATRIXVALUE(result,lpos,lval) This macro assigns the value lval at the position lpos in the value table. This is done after the ecs_SetGeomMatrix function is invoked.

ECS_SETGEOMIMAGEVALUE(result,lpos,lval) This macro assigns the value lval at the position lpos in the value table. This is done after the ecs_SetGeomImage function is invoked.

ECSERROR(r) This macro indicates whether ecs_Result contains an error code.

ECSSUCCESS(r) This macro indicates whether ecs_Result contains a success code.

ECSEOF(r) This macro indicates whether ecs_Result contains an EOF message. It is mainly used with the cln_GetNextObject function.

ECSMESSAGE(r) This macro returns the error message contained in ecs_Result (a string).

ECSREGION(r) This macro returns the geographical region contained in ecs_Result if ecs_Result contains this structure. The structure returned is a ecs_Region.

ECSTEXT(r) This macro returns the text string contained in ecs_Result if ecs_Result contains this structure. The structure returned is a string.

ECSRASTERINFO(r) This macro returns the raster information contained in ecs_Result if ecs_Result contains this structure. The structure returned is an ecs_RasterInfo.

ECSRASTERINFONB(r) This macro returns the number of categories in the ecs_RasterInfo structure contained in ecs_Result.

ECSRASTERINFOCAT(r,c) This macro returns the category number c contained in the ecs_RasterInfo of ecs_Result. The structure returned is a ecs_Category.

ECSOBJECT(r) This macro returns the geographic object contained in ecs_Result if ecs_Result contains this structure. The structure returned is an ecs_Object.

ECSOBJECTID(r) This macro returns the Id attribute contained in the ecs_Object structure of ecs_Result.

ECSOBJECTATTR(r) This macro returns the attr attribute contained in ecs_Object structure of ecs_Result.

ECSRASTER(r) This macro returns the raster line table contained in ecs_Object.

Chapter 7 Driver Development

Driver Development

The following chapter presents the different stages involved in the development of a driver, including a description of a Geospatial Library Transfer Protocol Daemon (GLTPD) and its relation with other components such as the port mapper and the driver.

The first sections (“Programming Background” on page 159, “API function Overview” on page 170 and “The driver's components” on page 181) provide the necessary information that will help the programmer understand the OGDI working concepts. It corresponds to the first phase of the development of a driver. The review of the OGDI core technology, the data types, the datastores, the layers and the GLTP servers are described in “Programming Background”. The OGDI functions (API) are then described in terms of their functionality in “API function Overview”. Additional components of a driver are then discussed in “The driver's components” to complete the description.

The second phase in the development of a driver is the programming task, which is presented step by step in “Driver's programming step by step” on page 190. It is important to mention that this section is exclusively based on the skeleton example driver (it can be found on the CD containing the OGDI source code or on the LAS Website at http://www.las.com). The skeleton is a complete example of a custom driver, with useful commentaries in each function header. Although its uses dummy data, it is completely functional, and can be compiled and tested. The programmer should read all the commentaries found in the skeleton to complete his knowledge on how a driver should be coded.

Programming Background

This section presents crucial information the programmer should know before starting the coding process of a driver. The reading of this section is strongly recommended.

Review of the OGDI core technology

The underlying philosophy of OGDI is to encapsulate the many tasks related to geospatial datastore access in a simple and standard API. We should specify that OGDI uses the C language for the portability facilities it offers.

When an application requires access to geospatial data, it calls functions through the API component of OGDI. The driver connected to the needed datastore (geospatial data format or product) is then loaded and used to receive the request, fetch the information from the datastore, translate it into the OGDI transient data structure, and finally return the result to the application. Separate drivers are used for each datastore. The drivers can be accessed directly for local datastores or remotely. For remote procedure calls, the OGDI GLTPD is used together with a network driver to link the application to a remote driver through a TCP/IP network.

In short, OGDI provides a data interoperability solution to access the growing number of geospatial data products and formats.

Data types, Datastore and Layer Definition

As mentioned earlier in this document, the OGDI data structure currently handles two types of geospatial data which are defined as:

Vector Data, which are composed of 4 subtypes of features called families:

- Line features
- Area features (each composed of one or more rings)
- Point features
- Text features

The VRF (Vector Relational Format) driver is an example of a driver that can access vector data from a datastore.

Raster Data, for information pertaining to points at regularly identified intervals, which are composed of two subtypes of objects (families):

- Image objects
- Matrix objects

The Grass driver is an example of a driver that can access raster data and vector data.

There is only one driver associated to a specific datastore (containing data in one specific format), but some drivers can access different types of data in different datastores (vector data and raster data). A datastore is situated in one specific location (path) and represents logically related data. The location and the driver are defined by the URL when the client establishes a connection. Furthermore, the datastore is represented by only one projection of a region. Using OGDI, a client can access different types of data, datastores, many regions and more than one projection of each region.

To fully understand the way an object is retrieved from a datastore, it is important to know what a layer is. A layer is a set of various geographic objects, each of a specific family. The family could be any of the following: Area, Line, Text, Point, Matrix, etc. A family and a string form a layer selection request. The string is a description of what to select and is defined by the following generic form:

```
DatastoreElement@Datastore(Expression).
```

The expression specifies which data to retrieve and so acts as a filter. The terms in the string could vary from one datastore to another, but the role remains the same i.e. specifies what element should be selected from the datastore. For example, VRF contains a string of the form:

```
FEATURE_NAME@COVERAGE(REQUEST)
```

It defines a specific feature name (ex: roads), a coverage type (ex: transportation) and a request which is the operation to perform on the feature table (ex: TYPE = double_lane).

Ex:

```
roads@transportation(roadtype = doubleline)
```

The GLTP server

The GLTP component of OGDI is a utility program that mimics the behavior of the C language API on a remote computer (see C language API, on page 22). To help understand the GLTP protocol, we use the well-known HyperText Transmission Protocol (HTTP), and make some comparisons between the two.

GLTP is based on the Remote Procedure Call (RPC) protocol by opposition to HTTP that is based on its own protocol. Both GLTP and HTTP use TCP/IP, which supplies the two major transport protocols: UDP and TCP. If the client and server communicate using UDP, the interaction is connectionless and if they use TCP, the interaction is connection-oriented. TCP provides all the reliability needed to communicate across the internet. By contrast, UDP offers no guaranty about reliable delivery. GLTP and HTTP both use the TCP transport protocol.

GLTP servers are stateful, and HTTP servers are stateless. Stateful servers keep state information (information that a server maintains about the status of ongoing interactions with clients) that allows them to remember what the client requested previously and to compute an incremental response as each new request arrives. The use of stateful servers permits more efficiency, since keeping information in a server reduces the size of messages that the client and server exchange, and allows the server to respond to request quickly. The motivation to use stateless servers lies in the protocol reliability: if a server uses incorrect state information due to loss of data or bad delivery, it may respond incorrectly.

Remote Procedure Call (RPC) concept

The remote procedure call model draws heavily from the procedure call mechanism found in conventional programming languages. FIGURE 4. “Procedure concept” shows a conventional program consisting of a main that calls one or more procedures which in turn can also call one or more procedures.

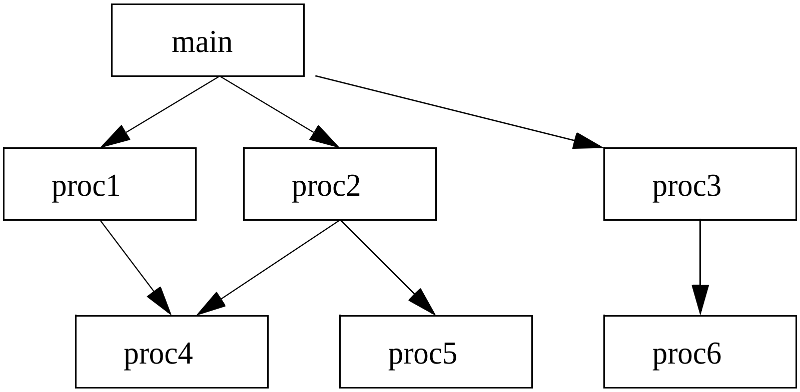


FIGURE 4. Procedure concept

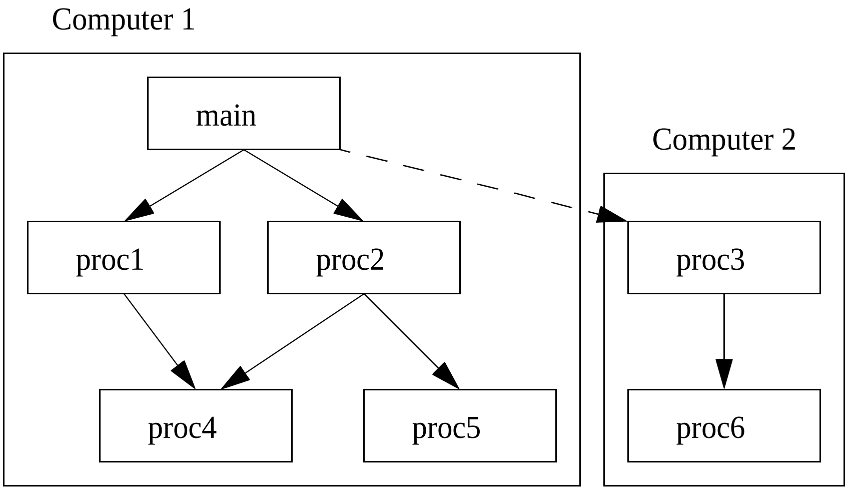


FIGURE 5. Extended to use RPC.

The RPC model uses the same procedural abstraction as a conventional program, but allows a procedure call to span the boundary between two computers. FIGURE 5. “Extended to use RPC.” illustrates how the remote procedure call can be used to divide a program into two pieces that each execute on a separate computer.

A programmer can build a conventional program that solves a particular problem, and then can divide the program into parts that execute on two or more computers. When doing so, the programmer can minimize changes and reduce the chance of introducing errors by adding stubs procedures to the program. The stub procedures implement the necessary communication, and allow the original calling and called procedures to remain unchanged (see stubs in FIGURE 2. “How a network driver connects with the gltpd” on page 25). A program called rpcgen automatically generates the stub's code.

Unlike many TCP/IP protocols, RPC does not use a fixed format for messages. It defines the general format of RPC messages as well as the data items in each field using a language known as the XDR language (see External Data Representation (XDR) concept, on page 166). Each item is encoded using the XDR representation standard.

Sun Microsystems has defined a particular form of RPC that has become a de facto standard. Sun RPC programs do not use well-known protocol ports like conventional clients and servers. Instead they use a dynamic binding mechanism called port mapper which is described in Port mapper, on page 167.

External Data Representation (XDR) concept

Each computer architecture provides its own definition of data. Programmers who create client and server software must contend with data representation because both endpoints must agree on the exact representation for all data sent between them. Sun Microsystems Incorporated devised an external data representation (XDR) that specifies how to represent common forms of data when transferring data across a network. The XDR standard provides definitions for data aggregates (e.g., arrays and structures) as well as for basic data types (e.g., integers and character strings). XDR library routines provide conversion from a computer's native data representation to the external standard and vice versa. Client and server programs can use XDR routines to convert data to external form before sending it, and to internal form after receiving it.

Port mapper

To allow a client to contact remote programs (the GLTP server in our case), the RPC mechanism must include a dynamic mapping service. Each machine that offers an RPC program (i.e., a server) maintains a database of port mappings (see database in FIGURE 6. “Client-Server communication”) and provides a mechanism that allows a caller to map RPC program numbers to protocol ports.

Whenever a remote program begins execution, it allocates a local protocol port that it will use for communication. The remote program then contacts the port mapper (see FIGURE 6. “Client-Server communication”) on its local machine and adds a pair of integers to the database: (RPC program number, protocol port number). Once an RPC program has registered itself, callers on other machines can find its protocol port by sending a request to the port mapper. A caller can always reach the port mapper because the port mapper communicates using the well-known protocol port 111. Once a caller knows the protocol port number the target program is using, it can contact the remote program directly.

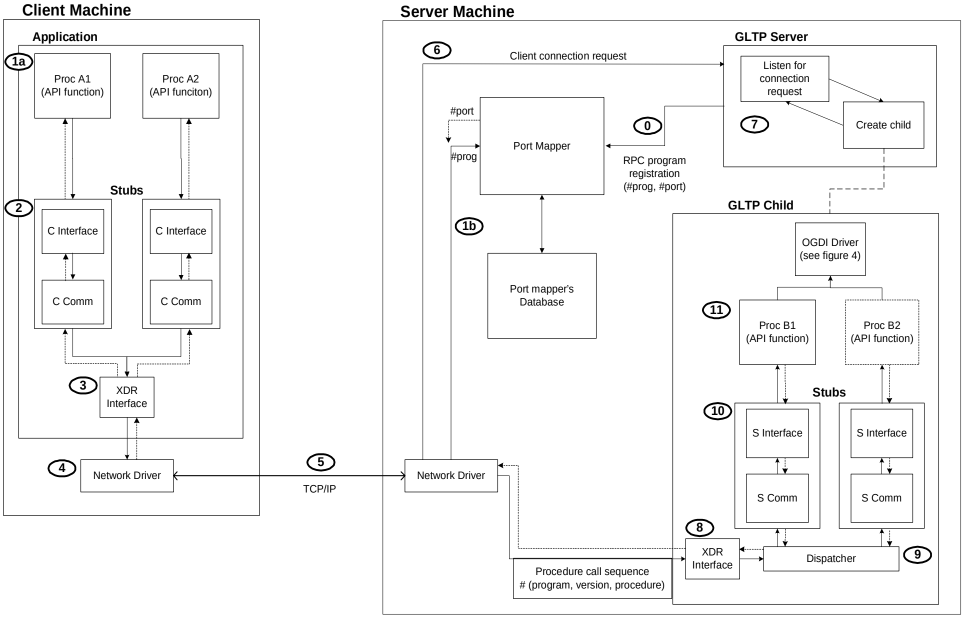


FIGURE 6. Client-Server communication

Step 0: Once executed, the RPC program registers its (program, port) pair in the port mapper. This is the reason why the port mapper must be launched before the GLTPd. This step is executed only once, when the GLTPd is launched which is before any application call is made.

Step 1a: The application calls an API function. For example procedure A1.

Step 1b: The RPC mechanism sends the remote program's number to the port mapper and receives in exchange the port number used to communicate with the remote procedure. The port number is the one the RPC program used to register itself previously.

Step 2: RPC uses the stubs procedure to move the called procedure to a remote machine.

Step 3: The XDR interface encodes all RPC messages in its standard data format.

Step 4: The network driver is responsible for accepting RPC messages (IP datagrams) and for transmitting them over a specific network.

Step 5: The TCP/IP protocol suite transfers the client's data to the server's network driver.

Step 6: The network driver accepts the messages transferred by TCP/IP and transmits them to the server.

Step 7: The GLTPd receives the client connection request, immediately creates a copy of itself and returns to its listening mode until a client requests a new connection. The GLTPd child takes over the communication, loads the driver and processes the client call.

Step 8: The RPC message is decoded by the XDR interface.

Step 9: The dispatcher uses the remote procedure number in the message to decide which stub procedure should receive the call. In our example the stub corresponding to procedure B1 is called.

Step 10: The chosen stub then calls the remote API function the client called (procedure B1).

Step 11: The API function on the server side is executed using the driver's specific function to complete its task. The transient structure (ecs_Result) is then returned to the client using the reversed path (dotted arrows).

Firewall/Proxy server

The GLTPd server can be installed on a system where a firewall server or a proxy server is present. There is a special version called the GLTPd Proxy server (GLTPd proxy program). The GLTPd Proxy server is also a cross platform product that runs on UNIX and Windows. The distribution package comes with an easy installation procedure.

API function Overview

The C language API component is composed of 20 functions prefixed with "cln_". Most of these functions has a svr_function counterpart (used in the server) and a dyn_function counterpart (used in the driver). When an application makes a call to an API function, the code in this cln_function calls the svr_function. The svr_function then calls the corresponding dyn_function (see Driver description, on page 187 for more details). All these functions return the ecs_Result structure that contains the answer related to the function call, except for cln_SetRegionCaches, cln_LoadCache and cln_ReleaseCache, which return an integer that indicates if the function succeeded or failed. Those three API functions are also special because they don't have dyn_functions or svr_functions counterparts. The cln_LoadCache function calls both cln_SelectRegion and cln_SelectLayer to achieve its task and cln_ReleaseCache calls the function cln_ReleaseLayer. On the other hand, cln_SetRegionCaches doesn't call functions on the server, so it only performs local operations. See Chapter 6 Utility library, on page 107 for a complete description of the C-API library functions.

The OGDI API can also be accessed using the Tcl/Tk scripting language. The Tcl/Tk library is composed of 22 functions prefixed with "ecs_". All functions have the same name and the same behavior as the ones in the C-API, except for the Tcl/Tk-API function ecs_SetCache that differs in name from the C-API function cln_SetRegionCaches. The library also includes two additional functions, ecs_GetURLList and ecs_AssignTclAttributeCallback that are specific to the Tcl/Tk interface. See Chapter 6 Utility library, on page 107 for a complete description of the Tcl/Tk-API library functions.

The following sections present definitions of the 20 API functions and of the 2 Tcl/Tk functions. All these functions are grouped by functionality.

Connection operations

Functions of this group are used to connect (or disconnect) an application to (from) the geographic datastore. Up to MAXCLIENT (32) connections can be instantiated simultaneously. The presence of the hostname in the URL specifies if the connection is local or remote. (See Components, on page 20)

CreateClient() creates a client (connects to a geographic datastore).

DestroyClient() deletes a client and unloads the associated driver from memory. This terminates the communication with the geographic datastore.

Datastore information

Functions of this group give information concerning the content of the datastore (dictionary).

GetDictionary() retrieves an [incr Tcl] applet from the driver. The applet describes the contents of a geographic datastore.

UpdateDictionary() returns an updated list that describes the content of a datastore.

Bounding operations

Functions of this group are used to delimit the geographic region in the datastore.

GetGlobalBound() specifies the driver's global geographic region.

SelectRegion() selects the current geographic region.

Layer operations

Functions of this group select (or release) the current layers to work with. Up to MAXLAYER (64) layers can be selected simultaneously.

SelectLayer() specifies the current layer.

ReleaseLayer() releases a layer

Data information

Functions of this group give meta information concerning layers. The GetRasterInfo function is only used when developing a driver that accesses raster data.

GetAttributesFormat() specifies the attribute format of the currently selected vector or raster layer. (See the ecs_ObjAttribute structure in Appendix)

GetRasterInfo() gathers information on the currently selected raster layer. (See ecs_RasterInfo structure in Appendix A, “” on page 196)

Data extraction

Functions of this group give information concerning the objects in the datastore, and are used to retrieve objects. See the structure “ecs_Object” on page 202 in Appendix A that describes all object's types available. An object is composed of the following: ID, attributes, bounding and data.

GetObject() retrieves the object that corresponds to the specified ID in the currently selected layer.

GetNextObject() retrieves the next object in the currently selected layer.

GetObjectIdFromCoord() retrieves the object ID string of the current layer that is nearest to the set of specified coordinates.

Projection operations

Function of this group manipulates data transformation. The driver cartographic projection remains the same for a given datastore and this can't change during a session. This means that all geographical information in the datastore is in a uniform projection. The SetServerProjection is generally not used because it changes the current projection string of the datastore that should be set by the server (driver) and not by the client. The SetClientProjection sets the destination projection string. The source projection is the server's current projection string. The data extraction function's group uses the source and destination projection string to perform data transformation when retrieving objects. The SetClientProjection function doesn't have a _dyn function counterpart. (See Projection, on page 27 for more information concerning projections)

GetServerProjection() returns the server's (datastore's) current projection string.

SetClientProjection() specifies the client's projection string.

SetServerProjection() specifies the current projection string of the driver.

Language definition

This API function isn’t implemented yet. Only English messages are available on the server.

SetServerLanguage() specifies the language in which the server returns information.

Cache operations

Functions of this group are used to load all the data of a layer region in a cache memory to minimize data access time. The cache is on the client side. These functions don’t return an ecs_Result structure, they simply return an integer to indicate a success or a failure. They don’t have dyn_functions counterparts.

SetRegionCaches() or SetCache() specifies the geographic region occupied by caches.

LoadCache l oads data for the region set by the esc_SetRegionCaches command.

ReleaseCache() deletes the cache related to a coverage stored by the cln_Loadcache command.

Tcl/Tk specifics

Functions of this group are only used in TCL/TK applications. These functions don't have dyn_functions counterparts.

AssignTclAttributeCallback() specifies a Tcl callback procedure which is called during calls to GetObject, GetNextObject and GetAttributesFormat.

GetURLList() specifies the list of currently-established connections to geospatial datastores.

The driver's components

This section introduces the three most important structures used by the driver. Furthermore it gives a detailed description of a driver and shows the connection between the files needed to build and compile a driver.

Ecs_Server structure

The ecs_Server structure is very important because it contains all the driver's information. As you will see, this structure is widely used in the server and in the driver. The ecs_Server structure contains many attributes the driver programmer needs to know. Here is the list of the attributes inside ecs_Server that need to be initialized and used by the driver.

void *priv the private geographic information of the driver.

int currentLayer the current layer in use in the driver.

ecs_Region currentRegion the current region of the geographic driver.

ecs_Region globalRegion the global region of the geographic driver.

char *projection the projection string in case the projection is undefined in the driver.

ecs_Result result returned structure to the OGDI user.

All geographic information is handled by this structure. However, this is global information and most of the drivers need to keep more information. For that reason, there is a private structure in ecs_Server. This is simply a pointer to the private information structure handled by the driver. The programmer is responsible for the memory allocation and deallocation of this private structure. There is an example in skeleton.h (ServerPrivateData).

The server handles the following attributes. They must not be modified by the driver.

char *hostname the hostname extracted from the URL.

char *server_type the server type extracted from the URL.

char *pathname the path name extracted from the URL.

ecs_RasterConversion rasterconversion:used to convert rasters in the driver.

ecs_Layer *layer the table of the layer in use in the driver.

int nblayer quantity of layers in layer.

The structure also contains all the pointers to the driver functions (dyn_functions) and the driver must not modify them.They are handled by the server and this will be explained in Data extraction, on page 176.

The ecs_Layer structure

Each time a request is passed to the SelectLayer function, a structure called layer structure is created in memory. A layer structure contains all the necessary information to handle a set of geographic data, whatever the type. To handle a layer and its related information, the OGDI provides three important functions:

ecs_SetLayer creates a layer in the driver and returns its number.

ecs_GetLayer checks if a layer exists and returns its number.

ecs_FreeLayer removes a layer from the set of layers.

The layers are contained in the “layer” attribute of the ecs_Server structure. We also know the number of layers opened and the current layer number, which is the last layer called by ecs_SelectLayer. Here are the attributes available in ecs_Layer that are useful to the driver's programmer:

ecs_LayerSelection sel layer selection information.

int index for GetNextObject the current object extracted.

int nbfeature the number of features in a layer. Optional.

void *priv the private geographic information of the geographic driver for a geographic layer.

The ecs_Layer structure contains a pointer to handle information specific to a driver for a particular layer. The driver's programmer must take the memory allocation and deallocation of the structure in charge. There is an example of this in the skeleton.h (LayerPrivateData).

The LayerMethod structure

The OGDI driver uses a special technique to map every function related to layer operations. The purpose of this technique is to simplify the code. In fact, most of the functions groups use this layer structure to choose the right function according to the current layer's family. So the layer structure eliminates the need to implement a big switch-case block. For example, when the API function GetNextObject is called on a currently selected matrix layer, the driver automatically selects the _GetNextObjectMatrix function to process the task.

The two dimensions layerMethod structure has the following definition:

```
LayerMethod layerMethod[11] = {
  /* 0 */{NULL, NULL, NULL, NULL, NULL, NULL},
  /* Area */{_openAreaLayer, _closeAreaLayer, _rewindAreaLayer,
_getNextObjectArea, _getObjectArea, _getObjectIdArea },
  /* Line */ {_openLineLayer, _closeLineLayer, _rewindLineLayer,
_getNextObjectLine, _getObjectLine, _getObjectIdLine },
  /* Point */{_openPointLayer, _closePointLayer, _rewindPointLayer,
_getNextObjectPoint, _getObjectPoint, _getObjectIdPoint },
  /* Matrix */{_openMatrixLayer, _closeMatrixLayer, _rewindMatrixLayer,
_getNextObjectMatrix, _getObjectMatrix, _getObjectIdMatrix },
  /* Image */{NULL, NULL, NULL, NULL, NULL, NULL},
  /* Text */{_openTextLayer, _closeTextLayer, _rewindTextLayer,
_getNextObjectText, _getObjectText, _getObjectIdText },
  /* Edge */{NULL, NULL, NULL, NULL, NULL, NULL},
  /* Face */{NULL, NULL, NULL, NULL, NULL, NULL},
  /* Node */{NULL, NULL, NULL, NULL, NULL, NULL},
  /* Ring */{NULL, NULL, NULL, NULL, NULL, NULL}
};
```

The layerMethod variable is of type LayerMethod. The LayerMethod structure holds every pointer to geographical access functions. Following is its definition:

```
typedef struct {
  layerfunc*open;
  layervoidfunc*close;
  layervoidfunc*rewind;
  layervoidfunc*getNextObject;
  layervoidfunc*getObject;
```

```
  layervoidfunc*getObjectIdFromCoord;
} LayerMethod;
```

The first dimension of the layerMethod structure (index from 0 to 10) selects the family. All families are defined in the ecs_Family enumeration that can be found in the header file ecs.h.

Note: The index 0 is not used.

The second dimension of the structure gives access to the layer operation function's pointer corresponding to the selected family. For example, to select the close function of the Area family, the following function call should be performed:

```
(LayerMethod[Area].close) (server, layer);
```

For each family supported by the datastore, the programmer will have to fill this structure with appropriate function handlers corresponding to each layer operation. If a family is not present in the datastore, the programmer has to set all function pointers to NULL in the corresponding family. If the client application uses a layer operation API not defined in the family, the driver will return an error and set the error message with the following text "FunctionName is not implemented for this family".

Driver description

When an application makes a call to an API function (see FIGURE 6. “Client-Server communication” on page 167 step 1a), the code in this function calls the svr_function that can be executed locally or remotely. For its part, the svr_function executes basic processing and calls the corresponding dyn_function that is defined in the driver (see FIGURE 7. “Representation of a driver (zoom of the OGDI driver of figure 6)” on page 187). All the driver's dyn_functions are mapped in the ecs_Server structure. The map is performed by the svr_CreateServer function. This function also loads in memory the driver that corresponds to the URL's driver section. (gltp://hostname(optional)/driver/path). The skeleton.c uses object.c and open.c, through the layerMethod structure to access data in the datastore. The driver also uses utils.c which regroups functions specific to a particular driver, and the OGDI function library that is not included in FIGURE 7. “Representation of a driver (zoom of the OGDI driver of figure 6)” on page 187, but defined in Chapter 6 Utility library, on page 107. The result of the application request is sent back using the ecs_Result structure (see Appendix A for a detailed description of the ecs_Result structure).

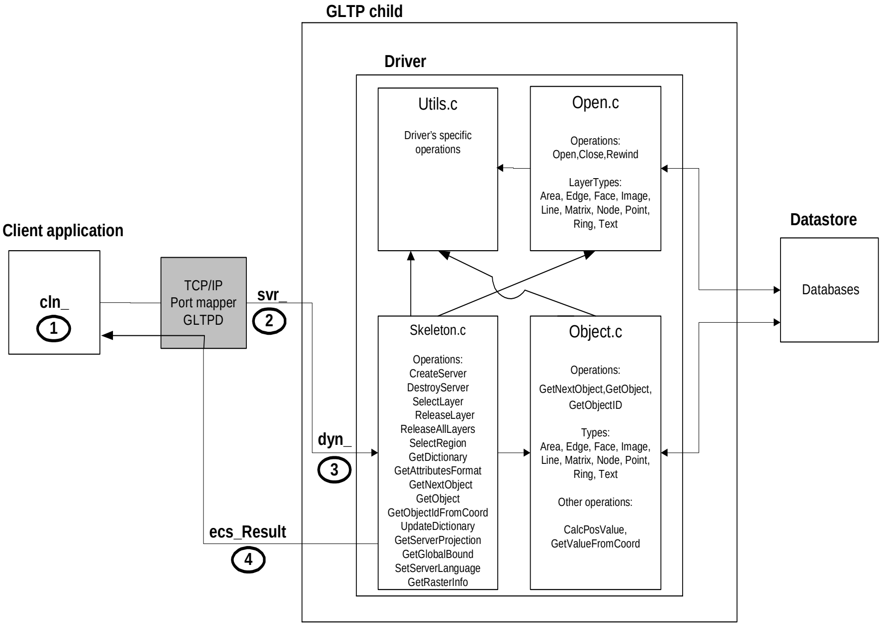


FIGURE 7. Representation of a driver (zoom of the OGDI driver of figure 6)

The driver is divided into two parts: the usual driver part described in driver.c and the “invisible part” that handles global operations (server.c). That must be seen as an object oriented relationship between the server.c and the driver. The server.c is the base class where all common operations, common checks and calls to the driver functions are done. The driver must be seen as an object that inherits from this base class. Because we are working in standard C, this is not totally “Object oriented”. The functions are seen by OGDI as pointers but the ecs_Server structure must be seen as the base classe’s members.

If no dyn_function is defined in the driver, the server will return an error and set the error message in ecs_Result with the following text: “FunctionName not present in dynamic library” or do a default function processing and return the appropriate result, depending on the function.

Here is a list of all API functions that perform default processing if they are not defined in the driver:

- cln_SetServerProjection
- cln_DestroyClient

All others will return an error message if they are not present in the driver.

The following API functions must not be defined in the driver because they use already defined functions to achieve their task.

cln_SetRegionCaches no server function call.

cln_LoadCache calls svr_SelectRegion and svr_SelectLayer.

cln_ReleaseCache calls svr_ReleaseLayer.

cln_SetClientProjection calls svr_GetServerProjection and svr_SetServerProjection.

Driver's files interactions

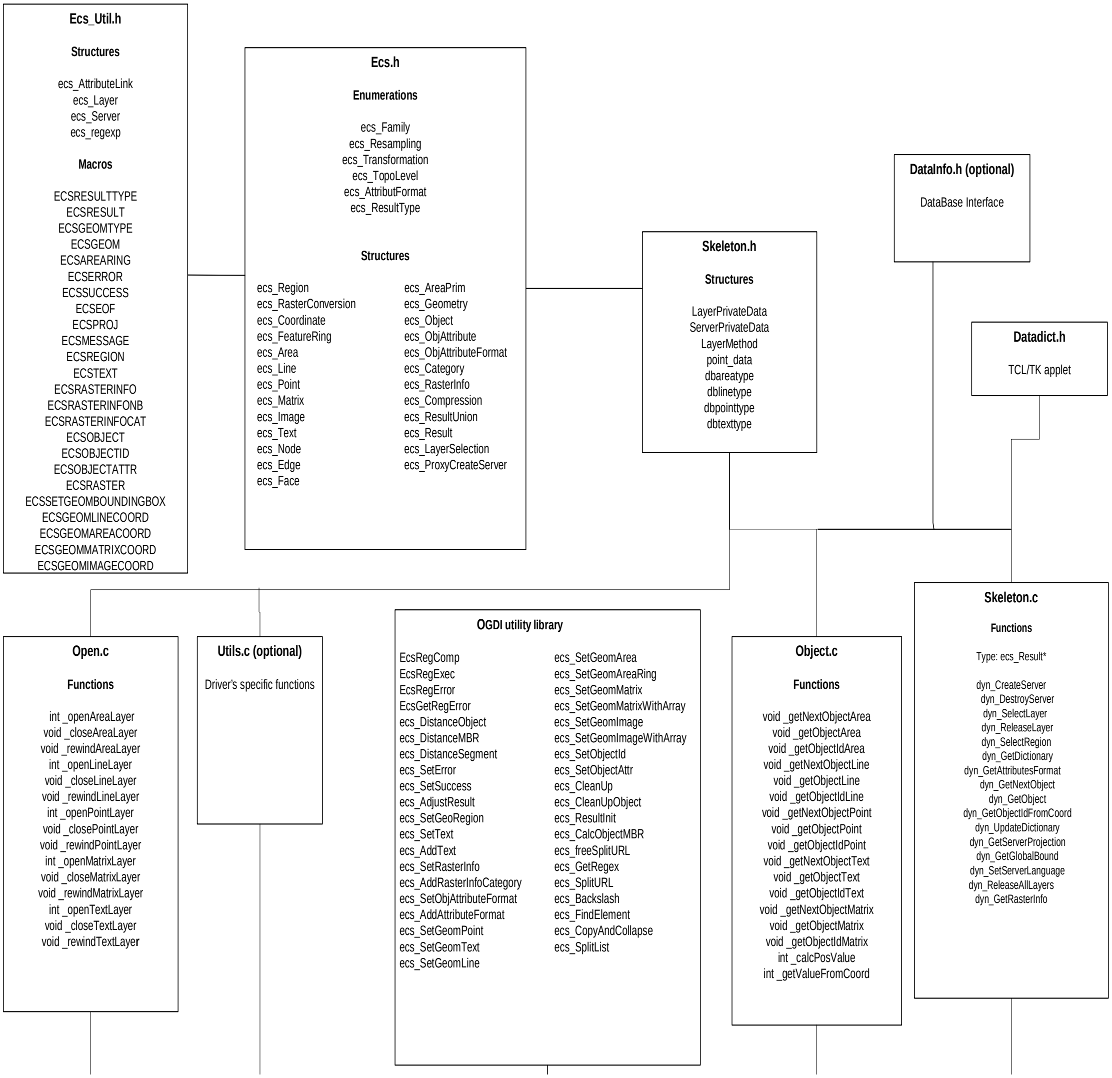


FIGURE 8. Representation of the connection between most of the files needed to build and compile a driver.

Note: OGDI.dll is a library that contains all the client and server functions, plus the utility function library (see Appendix A, "" on page 196). You can find the source code of all the C files required to construct the OGDI.dll in the C-API directory.

Driver's programming step by step

Following is a presentation of the steps involved in the programming phase.

Note: All through the coding process, whenever needed, code all driver specific functions in utils.c.

In the skeleton driver's object file, there is a dummy datastore structure that has to be removed.

OGDI provides utility functions and macros that will help in the development of a driver and in data manipulation. See Appendix for a complete description of these functions and macros.

Step 5.4.3 and 5.4.4 could be coded simultaneously.

(Step 1) Use the skeleton driver

To start the programming, use the skeleton driver that can be found on the CD containing the OGDI source code or on the LAS Website at http://www.las.com.

- Copy the skeleton driver source file in a directory with the name of the new custom driver.
- Rename the file skeleton.c with the name of the new driver. Do the same thing with files skeleton.h and skeleton.def.
- Replace all occurrences of the word skeleton by the name of the new driver in the following files: skeleton.c, skeleton.h, skeleton.def, object.c open.c, utils.c and makefile (Preserve the upper and lower cases).

Note: In the following steps, the word “skeleton” is replaced by “driver”.

(Step 2) Code the driver’s function

Code the dyn_ functions in the driver.c. API function Overview, on page 170 suggests a possible order of operation when developing the functions of a driver, with the exception of the functions in Cache operations, on page 179 and Tcl/Tk specifics, on page 180, which don’t have to be redefined in the driver.

Fill the layerMethod structure with the appropriate type used by this driver (see The LayerMethod structure, on page 185).

For each dyn_function, add the specific driver procedure that will achieve the function task.

(Step 3) Code the datastore function library

The purpose of a data library is to abstract the OGDI driver from data retrieving operation code and it poses as an interface between the driver and the datastore. The datastore function library will contain every function needed to retrieve data from the datastore, so the OGDI driver will only make simple function calls to the datastore functions library. This technique will:

- simplify the OGDI driver code;
- minimize changes in the OGDI driver if the format of the datastore is modified;
- minimize changes in the OGDI driver if the datastore functions library is modified;
- increase the reusability of the OGDI driver code for the development of a new driver (generalization).

The following figure summarizes the relation between the OGDI driver and the datastore when using a datastore function library.

FIGURE 9. Diagram with datastore function library

In the case where all data retrieving operation code is in the OGDI driver, the relation will become,

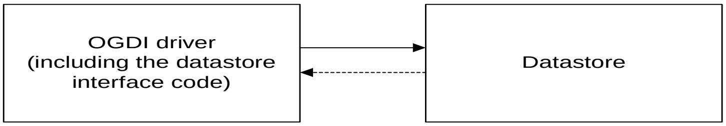


FIGURE 10. Diagram without the datastore function library

(Step 4) Code the Layer oriented-functions

See The LayerMethod structure, on page 185 for more information about the layerMethod structure.

- Code the _open, _close and _rewind functions in open.c for each family.
- Code the _GetNextObject, _GetObjectId and _GetObject functions in object.c for each family.

Appendix A

Implementation Specification

Appendix A first describes the ecs_Result structure and its components with a diagram (FIGURE 11. “Description of the ecs_Result structure and its components”), and then explains the whole ecs_Result structure in the following pages.

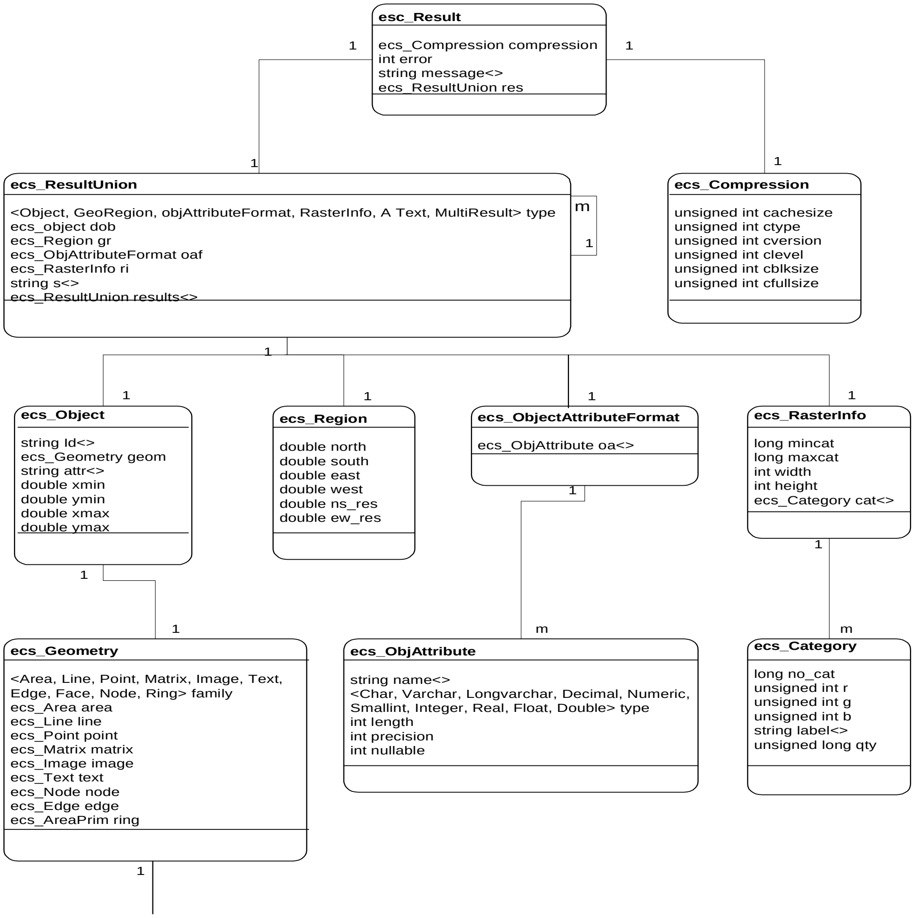
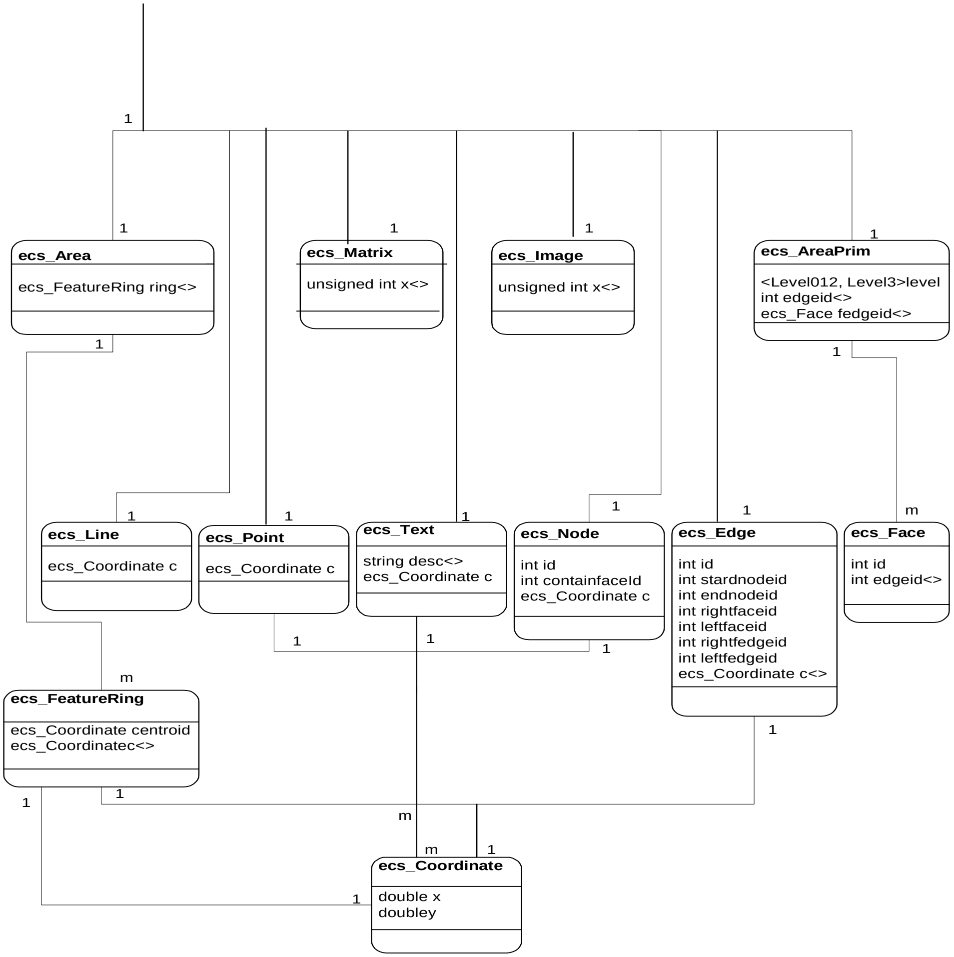


FIGURE 11. Description of the ecs_Result structure and its components

Note: The symbol <> after a variable means that it is an array. So its size is defined at run-time.



In the following section, you will find a definition of every structure presented in FIGURE 11. “Description of the ecs_Result structure and its components”.

ecs_Result

It is common to all the C API commands and contains all the possible answers. The contents of ecs_Result vary depending of the nature of the answers. For example, the structure could contain a string, a list of attributes, a geographical object, etc.

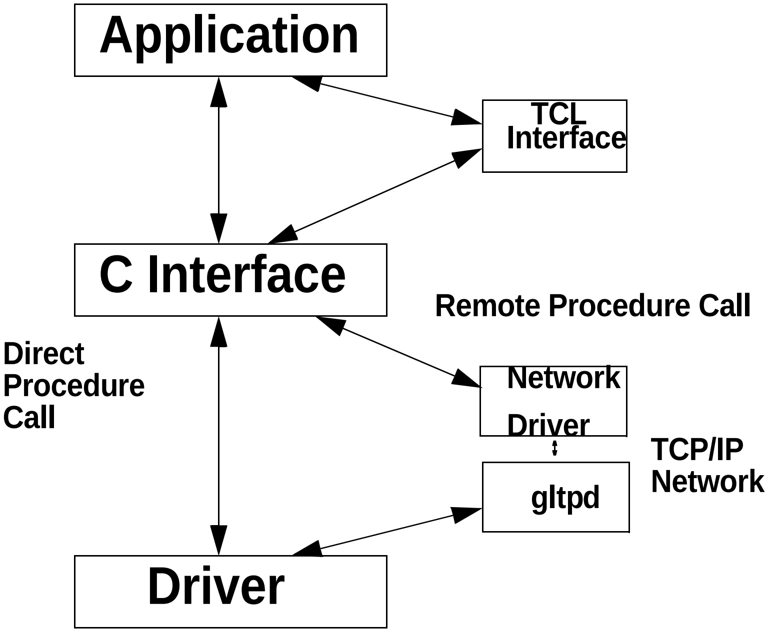
ecs_Compression compression

This attribute contains the compression type of the current object. (see below for more details related to this structure)

int error

This attribute returns an error code. It could be 0 (ECS_SUCCESS), 1 (ECS_ERROR), 2 (End of selection) or 3 (an error appears in an object extraction but continues the extraction).

string message<>



This is an optional message that could be returned if an error occurs. It provides a more precise description of the problem.

ecs_ResultUnion res

This attribute handles all the different types of object the OGDI could return. (see below)

ecs_Compression

This structure contains the necessary parameters to transfer compressed information across the net. It also defines the parameters that allow the transfer of many blocks of information in one operation (MultiResult).

unsigned int cachesize

The maximum size of ecs_Result objects a block could contain for MultiResult. The MultiResult is an attribute of ecs_ResultUnion used as a buffer of geographical objects.

unsigned int ctype

The compression type used during the transfer: 0 for no compression, 1 for Zip compression.

unsigned int cversion

The compression type version.

unsigned int clevel

The level of compression, could be 1 to 9.

unsigned int cblksize

The number of bytes to compress at a time.

unsigned int cfullsize

Used by the server. Not currently used in the compression.

ecs_ResultUnion

This structure handles the different types of information that can be returned by the OGDI. This structure is a union that allows only one structure attribute to be used at the time.

ecs_ResultType type

Indicates the type of data contained in this structure which could be SimpleError, Object, GeoRegion, objAttributeFormat, RasterInfo, Atext, MultiResult. Each of these types will indicate which attribute of this structure will be used. (see below)

ecs_Object dob

The Geographic Object structure. (see below)

ecs_Region gr

The Geographic Region structure. (see below)

ecs_ObjAttributeFormat oaf

The Object Attribute Format structure. (see below)

ecs_Rasterinfo ri

The Raster Information structure. (see below)

string s<>

A string structure

ecs_ResultUnion results<>

In case of MultiResult, this structure contains more than one ecs_ResultUnion structure for the transfer across the network. (see below)

ecs_Object

Contains a geographic object. This structure contains the common information to all geographic objects.

string id<>

The object identifier

ecs_Geometry geom

The geometry structure of the object. (see below for more details related to this structure)

string attr<>

The attribute list.

double xmin

The western limit of the geographical object.

double xmax

The eastern limit of the geographical object.

double ymin

The southern limit of the geographical object.

double ymax

The northern limit of the geographical object.

ecs_Region

Contains the limits of a geographic region

double north

double south

double east

double west

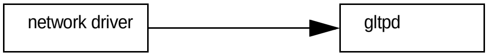
Usually this is known as the minimum bounding rectangle around a region.

double ns_res

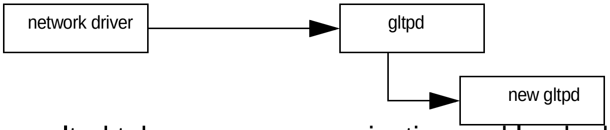
Contains the size of the north-south extent of a pixel in the same coordinate system than the region.

double ew_res

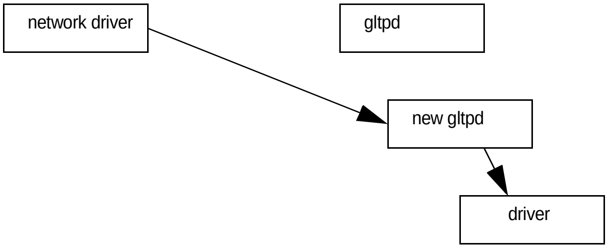
Step 1. Network Driver requests connection to gltpd



Step 2. gltpd creates a new thread (or fork) of itself



Step 3. new gltpd takes over communication and loads driver



Contains the size of the east-west extent of a pixel in the same coordinate system than the region.

ecs_ObjectAttributeFormat

Contains the list of the object attribute format descriptors of the ecs_Object attribute "attr".

ecs_ObjAttribute oa<>

The list of attribute format descriptors. (see below)

ecs_ObjAttribute

The attribute format descriptor of one attribute.

string name<>

The name of the attribute format descriptor.

ecs_ObjAttibuteFormat type

The attribute type descriptor which could be Char, Varchar, Longvarchar, Decimal, Numeric, Smallint, Integer, Real, Float and Double. (see below)

int length

The attribute length. For a string, it contains the maximum length of the string.

int precision

The attribute precision, mainly used for float and double information.

int nullable

Indicates if the value is nullable or not.

ecs_Rasterinfo

Contains the information related to one selected raster.

long mincat

long maxcat

The minimum and maximum categories of the category table.

int width

int height

The width and height of the raster.

ecs_Category cat<>

The category table (see below)

ecs_Category

The description of one category in the category table of ecs_RasterInfo.

long no_cat

The current category number.

unsigned int r

unsigned int g

unsigned int b

The category default color.

string label<>

The label of the category.

unsigned long qty

The number of pixels of this category in the matrix (optional).

ecs_Geometry

The geometry structure of a geographic object that contains a union of all the different types of possible geographic objects in OGDI.

ecs_Family family

The current object family which could be Area, Line, Point, Text, Matrix, Image, Edge, Face, Node and Ring. (see below)

ecs_Area area

An area object geometry description. (see below)

ecs_Line line

A line object geometry description. (see below)

ecs_Point point

A point object geometry description. (see below)

ecs_Matrix matrix

A matrix object geometry description. (see below)

ecs_Image image

An image object geometry description. (see below)

ecs_Text text

A text object geometry description. (see below)

ecs_Node node

A node object geometry description. (see below)

ecs_Edge edge

An edge object geometry description. (see below)

ecs_AreaPrim ring

A ring object geometry description. (see below)

ecs_Area

An area is a set of rings that form a complex area feature. The first area is considered as the main area and the others as islands.

ecs_FeatureRing ring<>

The list of rings that form the area. (see below)

ecs_FeatureRing

A single ring description in the ecs_Area.

ecs_Coordinate centroid

The centroid of this ring. (see below)

ecs_Coordinate c<>

The list of points that form the ring. (see below)

ecs_Line

Contains a polyline geographical object.

ecs_Coordinate c<>

The list of points that form the polyline. (see below)

ecs_Point

Contains a point geographical object

ecs_Coordinate c

The point coordinate itself. (see below)

ecs_Text

Contains a geographical text object. It is a point with a text attached to it.

ecs_Coordinate c

The point coordinate itself. (see below)

String desc<>

The descriptor string.

ecs_Node

Contains a geographical point object with topology.

int id

The identifier of this point.

int containfaceid

Indicates witch face contains this point.

ecs_Coordinate c

The point coordinate itself. (see below)

ecs_Edge

Contains a geographical edge object with topology.

int id

The identifier of the object

int startnodeid

The start node identifier of this edge.

int endnodeid

The end node identifier of this edge.

int rightfaceid

The identifier of the edge right face object.

int leftfaceid

The identifier of the edge left face object.

int rightfedgeid

The identifier of the edge right feature.

int leftfedgeid

The identifier of the edge left feature.

ecs_Coordinate c<>

The list of coordinates of the edge. (see below)

ecs_AreaPrim

Contains an area primitive with topological information.

<LevelO12> <Level3> level

Indicates the level of topology.

int edgeid

The edge id if the level is <LevelO12> (One edge contains the entire area).

ecs_Face fedgeid<>

The list of face id if the level is <Level3>. (see below)

ecs_Face

Contains a face object descriptor.

int id

The identifier of the face.

int edgeid<>

The list of edges id that form this face.

ecs_Coordinate

A geographic coordinate

double x

double y

The coordinate itself.

ecs_Matrix

A matrix block. Part of a larger matrix. Usually used as a buffer of pixel categories.

unsigned int x<>

The matrix buffer.

ecs_Image

An image block. Part of a larger image. Usually used as a buffer of pixel colors.

unsigned int x<>

The image buffer.

Appendix B

Tables

TABLE 1. list of all valid projection acronyms

code	description
aea	Albers Equal Area
aeqd	Azimuthal equidistant
alsk	Alaska Mod.-Stereographics
apian	Apian Globular
bipc	Bipolar Conic
bonne	Bonne
cass	Cassini
cc	Central Cylindrical
cea	Cylindrical Equal Area
collg	Collignon
eck1	Eckert I
eck2	Eckert II
eck3	Eckert III
eck4	Eckert IV
eck5	Eckert V
eck6	Eckert VI
eqc	Equidistant Cylindrical
eqdc	Equidistant Conic
gall	Gall (Stereographic)
gnom	Gnomonic
gs50	50 State U.S. Mod.-Stereographic
gs48	48 State U.S. Mod.-Stereographic
hataea	Hatano Asymmetrical Equal Area
labrd	Laborde
laea	Lambert Azimuthal Equal Area
leac	Lambert Equal Area Conic
lee os	Lee Oblate Stereographics Paci_c
lcc	Lambert Conformal Conic
loxim	Loximuthal
lsat	LANDSAT Space Oblique Mercator
mbtfpp	McBryde-Thomas Flat-Polar Parabolic

code	description
mbtfps	McBryde-Thomas Flat-Polar Sinusoidal
mbtfpq	McBryde-Thomas Flat-Polar Quartic
merc	Mercator
mill	Miller
mil os	Miller Oblate Stereographics Eur-Africa
moll	Mollweides
mtm	Mercator Transverse Modi__ee (Quebec)
nsper	General Vertical Persepective
nzmg	New Zealand Map Grid
ocea	Oblique Cylindrical Equal Area
omerc	Oblique Mercator
ortho	Orthographic
parab	Caster Parabolic
poly	Polyconic (American)
putp2	Putnins P2
putp5	Putnins P5
quau	Quartic Authalic
robin	Robinson
sinu	Sinusoidal
stere	Stereographic
tcc	Transverse Central Cylindrical
tcea	Transverse Cylindrical Equal Area
tmerc	Transverse Mercator
tpers	Tilted Perspective
ups	Universal Polar Stereographic
utm	Universal Transverse Mercator
vandg	Van der Grinten
wink1	Winkel 1

TABLE 2. list of valid ellipsoids

code	**description**
MERIT	MERIT1983
SGS85	SGS85
GRS80	GRS1980(IUGG)
IAU76	IAU1976
airy	Airy1830
APL4.9	Appl.Physics.1965
NWL9D	NavalWeaponsLab.
mod	airy Modi_edAiry
andrae	Andrae1876(Den.)
aust SA	AustralianNatl&S.Amer.1969
GRS67	GRS67(IUGG1967)
bessel	Bessel1841
bess nam	Bessel1841(Namibia)
clrk66	Clarke1866
clrk80	Clarke1880mod.
CPM	Comm.desPoidsetMesures1799
delmbr	Delambre1810(Belgium)
engelis	Engelis1985
evrst30	Everest1830
evrst48	Everest1948
evrst56	Everest1956
evrst69	Everest1969
evrstSS	Everest(Sabah&Sarawak)
fschr60	Fischer(MercuryDatum)1960
fschr60m	Modi_edFischer1960
fschr68	Fischer1968
helmert	Helmert1906
hough	Hough
intl	International1909(Hayford)
krass	Krassovsky
kaula	Kaula1961
MERIT	MERIT1983
SGS85	SGS85

code	description
GRS80	GRS1980(IUGG)
IAU76	IAU1976
airy	Airy1830
APL4.9	Appl.Physics.1965
NWL9D	NavalWeaponsLab.
mod	airy Modi_edAiry
andrae	Andrae1876(Den.)
aust SA	AustralianNatl&S.Amer.1969
GRS67	GRS67(IUGG1967)
bessel	Bessel1841
bess nam	Bessel1841(Namibia)
clrk66	Clarke1866
clrk80	Clarke1880mod.
CPM	Comm.desPoidsetMesures1799
delmbr	Delambre1810(Belgium)
engelis	Engelis1985
evrst30	Everest1830
evrst48	Everest1948
evrst56	Everest1956
evrst69	Everest1969
evrstSS	Everest(Sabah&Sarawak)
fschr60	Fischer(MercuryDatum)1960
fschr60m	Modi_edFischer1960
fschr68	Fischer1968
helmert	Helmert1906
hough	Hough
intl	International1909(Hayford)
krass	Krassovsky
kaula	Kaula1961
lerch	Lerch1979
mprts	Maupertius1738
new intl	NewInternational1967
plessis	Plessis1817(France)
SEasia	SoutheastAsia

code	description
walbeck	Walbeck
WGS60	WGS60
WGS66	WGS66
WGS72	WGS72
WGS84	WGS84

TABLE 3. list of valid units

code	description
km	Kilometer
m	Meter
dm	Decimeter
cm	Centimeter
mm	Millimeter
kmi	International Nautical Mile
in	International Inch
ft	International Foot
yd	International Yard
mi	International Statute Mile
fath	International Fathom
ch	International Chain
link	International Link
us-in	U.S. Surveyor’s Inch
us-ft	U.S. Surveyor’s Foot
us-yd	U.S. Surveyor’s Yard
us-ch	U.S. Surveyor’s Chain
us-mi	U.S. Surveyor’s Statute Mile
ind-yd	Indian Yard
ind-ft	Indian Foot
ind-ch	Indian Chain

Appendix C

Datum change of the OGDI

To use the datum change in the OGDI, the geographical driver must first have a projection with an extension describing the datum to use. Right now, the available datum are nad27 and nad83. To set the datum, simply add to the projection string the attribute « datum ».

Example :

```
+proj=longlat +datum=nad83
```

In the local machine, an environment variable OGDIDATUM must be set to the directory where the tables are set. In Grassland, the directory is /Grassland/nadfiles.

For the OGDI local projection, the things are a little more complex. First of all, to set the datum, it's exactly like the projection string in the geographical driver. However, the table must be set in order to convert the coordinates. The attribute to add in the projection string is «datumconv » with the table name. The module used to convert the points will be choose by the choice of the table. If it's Canada, the driver used to make the conversion will be dtcanada.dll. For all the other tables, the driver will be dtusa.dll. If the table attribute in the projection is not set, the default table will be « conus ».

Example :

```
+proj=longlat +datum=nad27 +datumconv=conus
```

When the converter know all these informations, other points must be considered before the datum conversion. First, the datum are optional but if they are not set in one of the projection, no datum conversion will be made. If both datum are defined but are the same, no datum conversion will be made either. If the datum are different, the conversion will be made only for the points inside the conversion table region. That mean, in a Canadian table, the points in Canada will be convert but the point outside will not. That don't mean the other geographics object are not selected, that mean the datum conversion will not apply for them.

Appendix D

BIBLIOGRAPHY

Douglas E. Comer, David L. Stevens; 1993; Internetworking with TCP/IP Vol. III: Client-Server Programming and Applications; Prentice Hall; 485p.

Index