



PSF Specification
for SDAL[®] Format
Version 1.7

For viewing purposes only. No rights to implement all or any part of the specification disclosed herein are granted or implied. Agreement to the terms and conditions set forth in the SDAL[®] Format Version 1.7 Adopters Agreement (CD-ROM) is required.

Version 0.4
October 1999

Subject to license. ©1999, 2000, 2001 Navigation Technologies Corporation. All rights reserved.

For viewing purposes only. No rights to implement all or any part of the specification disclosed in this Specification are granted or implied. Recipient may only implement all or any part of the Specification by agreeing to the terms and conditions set forth in the SDAL[®] FORMAT VERSION 1.7 ADOPTERS' AGREEMENT (CD-ROM).

Trademark Acknowledgements:

NAVTECH is a registered trademark of Navigation Technologies Corporation.

SDAL and SDAL FORMAT are registered trademarks of Navigation Technologies Corporation.

Table of Contents

PSF Data Organization and Characteristics	1 - 1
CD-ROM Organization	1 - 3
CD-ROM Directory	1 - 3
ISO-9660 Format	1 - 3
Directory Contents	1 - 3
SDAL Global Media File Overview	1 - 4
SDAL Region File	1 - 5
Recorded Voice Explication Data Files	1 - 5
Data Organization and Characteristics.....	1 - 5
Organization of Data	1 - 5
Overview of Physical Data Relationships	1 - 7
Data Structure Representation	1 - 9
Parcelization	1 - 9
Technical Description	1 - 9
SDAL Parcel Types	1 - 9
Parcel Header Structure	1 - 10
Parcel Organization on CD-ROM	1 - 13
General Rules for Parcels	1 - 13
Offset Arrays	1 - 13
SDAL Indexes	1 - 14
Technical Description	1 - 14
Spatial Indexes	1 - 14
PSF B-tree Index	1 - 14
Dense Index	1 - 15
POI Multi-Level Indexes	1 - 15
Global Kd-tree Parcelization Storage Format	1 - 15
Field Type and Structure Definitions	1 - 16
Field Type Definitions	1 - 16
Type Definitions and Structures	1 - 17
Parcel Level Compression	1 - 23
Record Level Compression	1 - 23
Compiler Design Implications of SDAL Data Compression	1 - 23
Overview	1 - 23
Small Huffman Tree Structure	1 - 24
Fast Huffman Tree Structure	1 - 27
Parcel-Level Compression	1 - 29
Parcel-Level Compression Huffman Tree Set Structure	1 - 30
PSF Media and Region Global Data Definitions	2 - 1
Technical Description	2 - 15
Reference Data Parcel Header	2 - 16
Icon Data Parcels	2 - 23
Technical Description	2 - 23
Icon Data Parcel Structure	2 - 23
Compression Information	2 - 25
Technical Description	2 - 25
Region Header Section	2 - 28
Index Root Parcel Section	2 - 31
External Index Records Section	2 - 32
Huffman Tree Section	2 - 33
LZ-Compression Huffman Tree Section	2 - 34
Language Bit Mask Array	2 - 35

Spatially-Organized Data	3 - 1
Spatial Data.....	3 - 3
Cartographic Points	3 - 52
Cartographic Polygons	3 - 54
Cartographic Text Records	3 - 57
Cartographic Text Name Records	3 - 61
Maneuver Sign Records	3 - 74
Maneuver Sign Text Records	3 - 75
Crossroads Parcel	3 - 81
Crossroads Index Parcels	3 - 86
Non-Spatially Organized Data	4 - 1
Non-Spatial Data.....	4 - 3
POI Data	4 - 3
Technical Description	4 - 3
POI Data Parcel Header Information	4 - 3
POI Type Internal Index Table	4 - 5
POI Record Offset Array	4 - 5
Base Name POI Record	4 - 5
Synonyms and Exonyms	4 - 8
Synonym/Exonym POI Record	4 - 8
Multiword Index POI Record	4 - 8
Date-time Record	4 - 9
POI Index Parcels	4 - 9
Character Tree Index	4 - 9
Spatial Index	4 - 10
Non-Spatial Index	4 - 12
Junction Data	4 - 19
Technical Description	4 - 20
Junction Parcel	4 - 20
Technical Description	4 - 20
Referenced By	4 - 20
Data Structures	4 - 21
Parcel Header	4 - 21
Junction Record	4 - 22
Junction Index Files	4 - 22
Technical Description	4 - 22
Referenced By	4 - 23
Junction Index Parcel Header	4 - 23
Nav Feature Entity ID Ordered Junction Index (sparse)	4 - 23
Nav Feature Entity ID Ordered Junction Index Key Record (sparse)	4 - 24
Nav Feature Entity ID Ordered Junction Index (dense)	4 - 24
Nav Feature Entity ID Ordered Junction Index Key Record (dense)	4 - 24
Place Ordered Nav Feature Entity ID Cross-Reference Junction Index (sparse)	4 - 24
Place Ordered Nav Feature Entity ID Cross-Reference Junction Index Key Record (sparse)	4 - 25
Navigable Feature Data	4 - 25
Navigable Feature Alphabetically Ordered Parcels	4 - 25
Technical Description	4 - 25
Data Reference	4 - 26
Navigable Feature Place-Ordered Cross-Reference Parcels	4 - 28
Technical Description	4 - 28
Data Reference	4 - 29
Navigable Feature Place-Ordered Cross-Reference Parcels	4 - 29
Navigable Features Data Index Files	4 - 31
Technical Description	4 - 31

Referenced By	4 - 32
Navigable Feature Character Tree Parcels	4 - 40
Place Data (Administrative Areas, Zones).....	4 - 42
Place Data Parcel	4 - 42
Technical Description	4 - 42
Data Reference	4 - 43
Data Structures	4 - 43
Synonyms and Exonyms	4 - 48
Abbreviated Synonym/Exonym Place Record	4 - 48
Multiwords	4 - 48
Multiword Index Place Record	4 - 48
Place Data Index Files	4 - 49
Technical Description	4 - 49
Referenced By	4 - 49
Index Structures	4 - 50
Data Compilation Guidelines	4 - 53
Postal Code Data.....	4 - 53
Technical Description	4 - 54
Postal Code Data Parcels	4 - 54
Data Reference	4 - 54
Postal Code Data Index Files	4 - 56
Technical Description	4 - 56
Postal Code Index Parcel Header	4 - 56
Postal Code Data Index	4 - 56
Recorded Voice Explication Data Files.....	4 - 57
Technical Description	4 - 57
Glossary	1

This page intentionally left blank.

Section 1 PSF Data Organization and Characteristics

1.1 CD-ROM Organization

This section provides a logical and file-level overview of the SDAL PSF structure on the CD-ROM Media. The SDAL PSF CD-ROM contains the following physical entities:

- CD-ROM Directory
- SDAL Global Media File
- SDAL Region File
- Recorded Voice Explication Data Files

The SDAL Physical Storage Format file system conforms to the ISO-9660 file system format specification. The relative locations of the PSF data stored on the CD-ROM disc are shown in the diagram below.

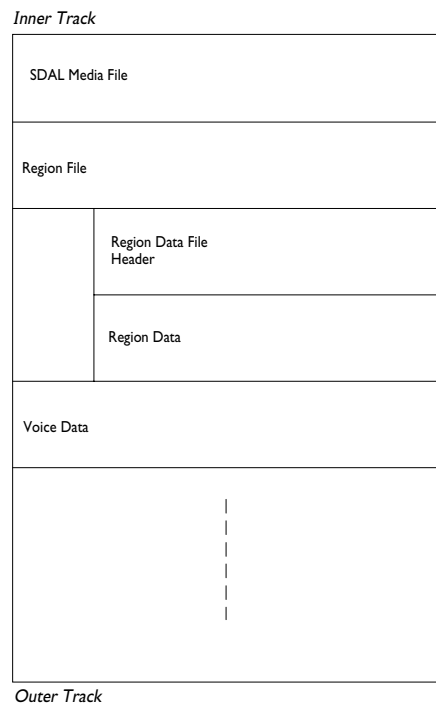


Figure 1-1: CD-ROM Disc Format Diagram

1.1.1 CD-ROM Directory

The SDAL CD Version 1.7.x database contains a two-level directory structure: the root directory and a set of files directly off the root directory.

1.1.1.1 ISO-9660 Format

The ISO-9660 file system format specification may be obtained from several sources including the following Web Sites:

- <http://www.cinram.com>
- <http://www.iso.ch/cate/d17505.html>

1.1.1.2 Directory Contents

The SDAL CD contains at a minimum two database files. It may also contain voice files.

Database Files

The database files are the SDAL Global Media File (0.SDL) and SDAL Region File (e.g., 1.SDL). These are database-specific files based upon the version of the database being referenced within them.

Voice Files

The voice files are not database-specific. They consist of a male and female voice file for each of the supported languages. For example:

File Description	File Name
Dutch, male	DUTM.SDL
Dutch, female	DUTF.SDL
English, male	ENGM.SDL
English, female	ENGFSDL
French, male	FREM.SDL
French, female	FREF.SDL
German, male	GERM.SDL
German, female	GERFSDL
Italian, male	ITAM.SDL
Italian, female	ITAF.SDL
Japanese, male	JPNM.SDL
Japanese, female	JPNFSDL
Spanish, male	SPAM.SDL
Spanish, female	SPAF.SDL
UK English, male	UKEM.SDL
UK English, female	UKEFSDL

1.1.2 SDAL Global Media File Overview

The SDAL Global Media File is a file that stores per-media information. It always corresponds to region zero (filename 0.SDL). The Global Media Header is located at media sector address zero.

The SDAL Global Media File contains these per-media global data categories:

- PSF Version/Type/Date Information
- Index Repository and Key Size information
- Compression Metadata
- Parcel Granularity and Sizes
- Normalized Attribute Array
- Metadata Tables
- Reference Data Index
- Reference Data
- Country Information
- Cartographic Feature Types
- POI Chain ID
- POI Facility Type
- POI Facility Subtype

- Navigable Feature Type
- Navigable Feature Street Type
- Language, Time Zone, and Daylight Savings Time data
- Chain IDs and Subtypes for each Facility Type
- Non-trademarked Icon Data
- Per-media Huffman Compression Trees

An external index to the remaining items is found in the Global Media Index parcel which follows the Global Media Header. The Global Media Index parcel contains parcel ID and record ID extent information.

1.1.3 SDAL Region File

The SDAL Region File is a separate CD-ROM file that stores the region's detailed information. The first region file name is 1.SDL.

The SDAL Region File contains:

- Region File Header
- Spatially-Organized Map Data
- Non-Spatial Data
- Data Indexes

The Region File Header is located at media sector address zero for the region file. This header contains addressing components to the remaining data on the region file. The other data types are explained in the remainder of this document.

1.1.4 Recorded Voice Explication Data Files

SDAL recorded explication voice audio data is included in the SDAL PSF CD-ROM. It is recorded using both male and female voices, in several languages: Dutch, French, English (U.K. and U.S.), German, Japanese, Italian, and Spanish.

For each language/voice pair a separate file is stored on the CD-ROM (e.g., "GERF.SDL" for German/female, "GERM.SDL" for German/male). These files store all ADPCM encoded phrases for the pair with a small table of offsets for the phrases within that file.

1.2 Data Organization and Characteristics

This section of the PSF Specification covers:

- Organization of PSF data and physical data relationships
- Parcelization definition and overview
- PSF data indexing overview
- Field types and structures definitions
- Data compression overview

1.2.1 Organization of Data

Each aspect of a navigation application (Map Display, Vehicle Positioning, Geocoding, Route Calculation, and Route Guidance) all make use of different subsets of map data.

Route Calculation requires data to be organized to facilitate searches for an optimal route between two points. Route calculation data are also layered to facilitate efficient processing of the road network, and are organized so each parcel covers the maximum possible geographic area (within the constraints of a fixed buffer size or a limited number of buffer sizes). This objective is met by:

- Placing little data other than that which is directly relevant to route calculation in the data parcels.
- Constructing these parcels to minimize wasted space within a parcel data record.
- Carrying data in compressed form on the CD-ROM.

The Map Display and Route Calculation aspects require different data. To allow both to perform efficiently, route calculation data and cartographic data are separated on the CD-ROM. Route calculation data, cartographic data, and other types of data are partitioned as follows:

- Route calculation data - Organized to facilitate rapid route calculation, this includes some, but not all the data needed for route guidance (a separate function). Route calculation parcels are organized and accessed spatially and by layer.
- Cartographic data - Organized to facilitate rapid panning and zooming, the cartographic parcels are organized and accessed spatially and by layer.
- Cartographic Cross-reference data - This cartographic-related data, such as cross-references between routing entities and cartographic entities, is not required to display, pan, or zoom map data. However, it is useful for functions such as route highlighting or point-and-click. It is separated from the cartographic data to improve performance of the functions requiring cartographic data but not cartographic cross-reference data. Cartographic cross-reference parcels are organized and accessed spatially and by layer. However, while cartographic and routing data is layered by the “rank” of the roadway, cartographic cross-reference data is layered by a combination of roadway rank and “highlighting layer.” The latter measures the zoom level at which a road segment can be approximated by a straight line.
- Maneuver data - Maneuver parcels contain additional information needed for direction generation and route guidance that is not contained in the route calculation or cartographic parcels. These parcels are spatially organized but not layered.
- Crossroads data - Crossroads data is organized spatially, but not layered.
- POI data - Point-of-Interest data is organized alphabetically, but can be accessed spatially via a spatial index.
- Additional data - The following data is organized and accessed non-spatially: Navigable Features, Places (i.e., Administrative Areas and Zones), Postal Codes, and Junctions. The following diagram provides an overview of the physical organization of SDadata::

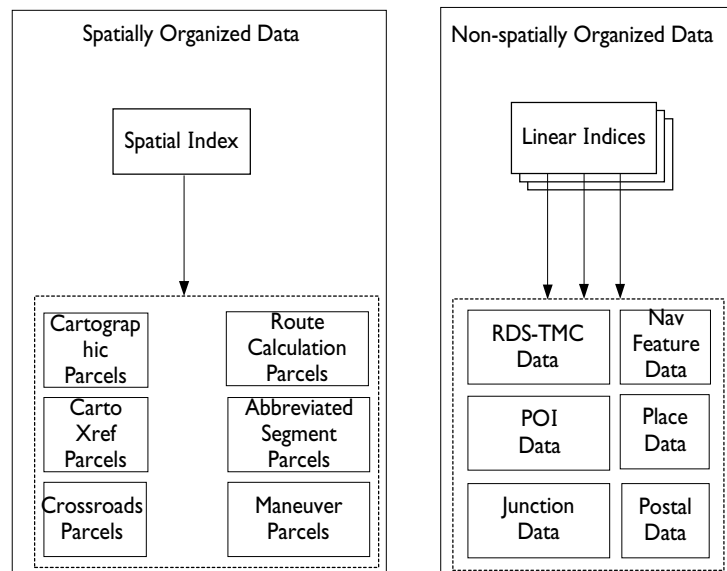


Figure 1-2: Overview of the SDAL Physical Storage Format

1.2.2 Overview of Physical Data Relationships

The following diagrams provide a mid-level overview of the relationships among the physical data record types: within parcels, between parcels, and between spatially and non-spatially organized record types. The physical entities and relationships described here are discussed in detail later. Each arrow represents a reference in one database entity to another database entity. (The latter is at the end with the arrowhead.) These references occur as SDAL Database IDs (DBIDs), which are described in detail later in this section. The entity ID portion of the DBID can reference an entity within a specified parcel. To reference entities outside the parcel, an index into a table of neighboring parcel IDs can replace the full four-byte parcel ID.

The following diagram illustrates the spatial data relationships among the cartographic and route calculation parcels:

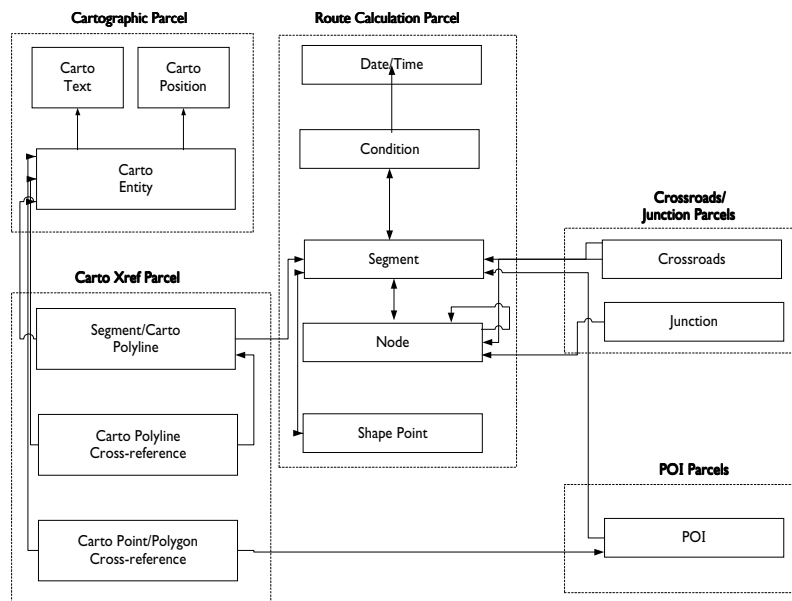


Figure 1-3: Cartographic and Route Calculation parcels

The following diagram illustrates the data relationships for the Point of Interest (POI) parcels:

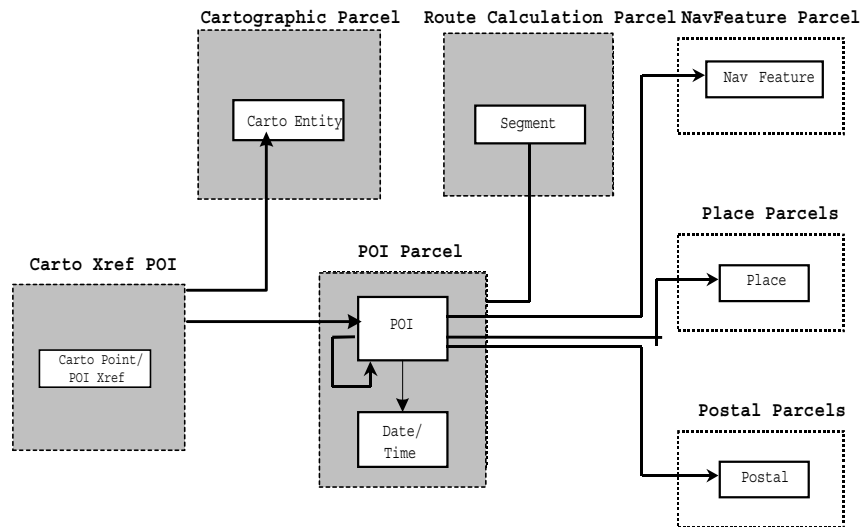


Figure 1-4: Point of Interest

Figure 1-5 illustrates the spatial data relationships for maneuver parcels:

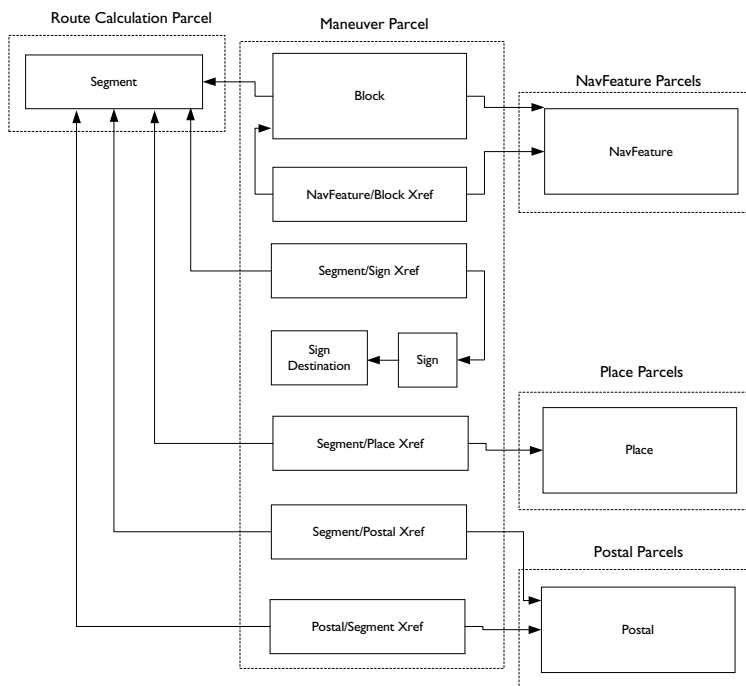


Figure 1-5: Maneuver Information

The next diagram (Figure 1-6) illustrates the data relationships among the non-spatial parcels:

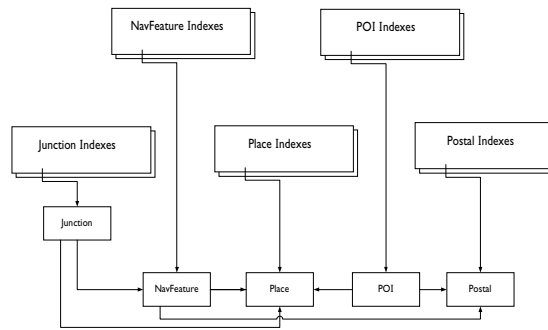


Figure 1-6: SDAL Non-spatial Data Relationships

1.2.3 Data Structure Representation

Data structures with fixed length fields are described in this document as C typedef structures. Data structures with variable length fields cannot be represented this way. For these structures, a field by field description is provided.

1.2.4 Parcelization

A parcel is the basic unit of I/O used in the SDAL format. Parcelization is a method of splitting data into discrete units called parcels. All types of data are formed into parcels using certain rules, as described below.

1.2.4.1 Technical Description

Parcelization requires the division of data into blocks, or parcels. For SDAL spatial map data, a parcel is the physical record containing all the data in a geographic rectangle. For non-spatial data, the parcel contains a sequential group of data records organized in a physical I/O block. The parcel is the basic unit of input/output in the Physical Storage Format (PSF). A parcel contains the records representing the map data plus the parcel overhead. Each parcel is restricted to a maximum uncompressed size of 32 Kbytes. Each parcel has an associated parcel ID which contains the information required to load the parcel from the CD-ROM medium, including: parcel offset (in units specified in global data information) within the region file, parcel size (in units specified in global data information), and a bit flag indicating whether redundant copies of the parcel exist (whose various offsets are specified in a table in global data information). (See “General Rules for Constructing Parcels.”) The structure of the parcel ID is described in the “Field Type Definitions” section of this document.

The placement of data in parcels is guided by parcelization rules, that govern the construction of all types of SDAL data into parcels.

1.2.4.2 SDAL Parcel Types

There are three main categories of SDAL parcels.

- Spatial parcels
- Non-spatial parcels
- Index parcels

Spatial Data Parcels

A spatial parcel contains the records representing all the map data within a rectangle, plus the parcel overhead. Examples of spatial parcels are cartographic and route calculation parcels.

The geographic rectangle represented by any spatial parcel in the database region can be determined from its parcel ID using a single kd-tree (the “Global Kd-tree”). Although there is some flex-

ibility in the way spatial parcelization can be performed, certain parcelization rules must be followed when constructing spatial parcels.

Non-Spatial Data Parcels

A non-spatial parcel contains the records of a particular data type, arranged in entity identifier order (e.g., alphabetically), rather than spatially. Examples of non-spatial parcel data are junctions, administrative areas, postal zones, and navigable features.

Index Parcels

Index parcels contain cross reference and search key information for locating data records in spatial and non-spatial parcels. They do not return data.

1.2.4.3 Parcel Header Structure

All SDAL parcels have an associated parcel header. The structure of the complete parcel header depends on the parcel type, but the initial SDAL Parcel Header structure is common to all parcel types, and is included in all other parcel headers.

Note: The definition of all data types is provided in the *Type Definition and Structures* section.

The following C structure describes the initial parcel header data:

```
/******  
**PclHdr_t: parcel header common to all parcel types  
*****/  
typedef struct  
{  
    /* common to all parcel types */  
    ParcelID_t    parcelid;  
    Ushort_t      usParcelDesc;  
    Uchar_t       ucParcelType;  
    RegionID_t    region;  
    Bool_t        bEndianSwap;    /* set when header endian converted  
                                   */  
    Uchar_t       ucCmpDataSizeHi; /* compressed size in bits, byte 1  
                                   */  
    Ushort_t      usCmpDataSizeLo; /* compressed size in bits, byte  
                                   2&3*/  
    Ushort_t      usCompressType; /* type of parcel compression      */  
    Ushort_t      usCmpData;      /* byte offset to compressed data  
                                   */  
    Ushort_t      usCmpDataUncompSize; /*uncompressed parcel size  
                                   (bytes) */  
    Ushort_t      usExtensionOffset; /* If not 0: Offset to hdr exten-  
                                   sion*/  
}  
PclHdr_t;
```

The valid value for usCompressType is:

```
NO_COMPRESSION  
NO_PARCEL_COMPRESSION  
SZIP_PARCEL_COMPRESSION_EXT  
SZIP_PARCEL_COMPRESSION_INT
```

The valid values for the ucParcelType field are:

Spatial Data Parcels

- 0 Route Calculation Parcel
- 1 Cartographic Parcel

- 2 Cartographic Cross-reference Parcel
- 3 Maneuver Parcel
- 4 POI Spatial Index Parcel
- 5 Abbreviated Segment Parcel
- 6 Crossroads Parcel

Non-Spatial Data Parcels

- 8 Place Parcel
- 9 Alphabetically Sorted Navigable Feature Parcel
- 10 Navigable Features Arranged By Place Parcel
- 11 Navigable Feature Entity ID, Place ID cross-referenced to Place Parcel IDs
- 12 Postal Parcel
- 13 Place DBID, Navigable Feature. Entity ID, Crossroads DBID
- 14 Navigable Feature. Entity ID, Crossroads DBID
- 15 Navigable Feature. Entity ID, Junctions DBID
- 16 POI Multi-Level Parcel – Index of POI Type and Place by ChainID, Postal code and subtype, over full/partial dense parcels
- 17 POI Full-Dense Parcel – Index records containing POI full name and pointing to alpha ordered POI record.
- 18 POI Part-Dense Parcel – Index records containing partial POI name and pointing to alpha ordered POI record.
- 19 Media Header, Reference Data, Metadata, Icons, Voice, Per-Media Compression Trees
- 24 Map Header, Per-Map Compression Trees
- 25 Junction Parcel
- 26 Level 1 Place Partial Name, Place ID
- 27 Level 2 Place Partial Name, Place ID
- 28 Level 3 Place Partial Name, Place ID
- 29 Level 4 Place Partial Name, Place ID
- 30 Level 5 Place Partial Name, Place ID
- 31 Level 6 Place Partial Name, Place ID
- 32 Level 7 Place Partial Name, Place ID
- 33 Country Code, Locale Table, Locale Number
- 34 Highway Only- Navigable Features Parcel
- 35 Place Partial Name, Place ID
- 36 Navigable Partial Name, Alphabetic Sorted
- 37 Place ID, 1st ChildID At Lowest Level, Last ChildID At Lowest Level 1st ChildID At 2nd Lowest Level, Last ChildID At 2nd Lowest Level
- 38 Navigable Features. Fullname Dense Parcel
- 39 Street Type Parcel

- 40 Crossroads Name Parcel. Primary Navigable Feature DBID, Partial Name, Intersecting Navigable Feature DBID, Place/Postal DBID, Crossroad DBID.
- 41 POI Parcel – alphabetically ordered
- 42 Navigable Feature-by-Postal Parcel. Postal Code DBID, Navigable Feature DBID
- 45 Country Nav Parcel – CountryID to NavID index
- 46 Navigable Feature Name Character Tree Parcel
- 47 Segment DBID-to-DNDC ID crossreference
- 48 Segment DNDC ID-to-DBID crossreference
- 49 POI Name Character Tree Parcel

Index Parcels

- 219 DNDC ID-to-DBID crossreference Index
- 220 POI Full Sparse Index – Sparse Index over Type 17
- 221 POI Partial Sparse Index – Sparse Index over Type 18
- 222 Country Nav Parcel Index – B-tree over Type 45
- 223 Navigable Feature-by-Postal Parcel Index
- 224 Navigable Feature Entity ID Index
- 225 Crossroad Names Index
- 226 Street Type Index
- 227 Navigable Feature Full Name Sparse Index
- 228 High Level Place IDx Parcel
- 229 Nav Ftr Next character Alphabetic Dense IDx Parcel B-tree Index
- 230 Levelm Place Alphabetic Parcel B-tree Index
- 232 Level7 Place Alphabetic Parcel B-tree Index
- 233 Level6 Place Alphabetic Parcel B-tree Index
- 234 Level5 Place Alphabetic Parcel B-tree Index
- 235 Level4 Place Alphabetic Parcel B-tree Index
- 236 Level3 Place Alphabetic Parcel B-tree Index
- 237 Level2 Place Alphabetic Parcel B-tree Index
- 238 Level1 Place Alphabetic Parcel B-tree Index
- 239 Junctions B-tree Index
- 247 Junctions Navigable B-tree Index
- 248 Crossroads Navigable B-tree Index
- 249 Crossroads Place B-tree Index
- 250 Postal B-tree Index
- 251 Place B-tree Index
- 252 Navigable Alphabetic -Place B-tree Index
- 253 Place Ordered Navigable Feature B-tree Index
- 254 Navigable Feature Basename B-tree Index

1.2.4.4 Parcel Organization on CD-ROM

CD-ROM Organization of Spatial Data Parcels

The spatial parcel types can be located anywhere in the database, provided they can be accessed using their corresponding parcel IDs from the global kd-tree.

Non-spatial types can be located anywhere relative to the spatial parcel types.

Individual parcels of spatial types can be interleaved with parcels of other spatial types. As an example, if a database has 200 route calculation data parcels and 200 cartographic data parcels, a route calculation parcel can be followed by a cartographic parcel, then by a different route calculation parcel. They must be in the spatial order to be accessible using the global kd-tree.

CD-ROM Organization of Non-Spatial Data Parcels

Non-spatial parcel types can be located anywhere in the database, but some of these types must be located in a specific order relative to each other. The following rule describes the parcel placement:

All parcel types marked “index parcels” (219-255) immediately precede the non-spatial parcel they are indexing. An example is a database with 100 type 40 parcels (crossroads name parcels) and two type 225 parcels (crossroads name index parcels). The two type 225 parcels are located first in the database. The first type 40 parcel follows the second 225 type parcel, with the remaining type 40 parcels following as a single block of parcels.

1.2.4.5 General Rules for Parcels

Parcel size and granularity are specified by fields in the Global Media Header structure, `GlbMediaHeader_t`. The six (6) size index bits of the Parcel ID allow for 64 distinct parcel sizes arranged in units of 2^m bytes where $m = \text{usBlockSize}$ in the `GlbMediaHeader_t` structure. (See “SDAL Media File.”) The bits are used as an index value in a table, `ucaParcelSizes[ ]` in `GlbMediaHeader_t`, stored in global data.

Various types of data can span multiple parcels to maximize space usage. Regardless of this capability, specific data records or tables are not allowed to span parcel boundaries.

An index parcel may contain more than one root only if the complete sub-trees of each of these roots fit into the parcel; i.e., if a complete sub-tree has been parcelized in an index parcel, then another highest level unstored node may be stored as another root in that index parcel, only if that node's complete sub-tree also fits in the parcel.

For certain parcel types, all parcels of the type are required to be of the same size. These types include: all B-tree parcel types and the parcel types indexed by the B-tree parcels, and the Global Kd-tree Index.

No single offset field in the parcel may exceed 32K. In general, 32K is the upper limit on any uncompressed parcel size.

Spatial parcels cannot have a geographic dimension less than 2^9 NTUs. All latitude/longitude used in the SDAL database are stored as integer values of NavTech Units (NTUs). One NTU is equal to 1/100,000 of a degree.

1.2.4.6 Offset Arrays

Within a parcel, when each record of a particular type has a unique entity identifier, one of two methods is used for locating records.

Block Offset Arrays locate data records when entity identifiers are assigned to records of a given type *consecutively* within the parcel; i.e., the first record within the parcel has entity identifier 0, the second 1, etc., with no gaps.

Record Offset Arrays locate records in which gaps do exist, but in which the records are still stored in order by entity identifier.

For example, Record Offset Arrays access segment records and node records in route calculation parcels above the bottom layer, while Block Offset Arrays access most other record types in all parcels.

1.3 SDAL Indexes

SDAL uses indexes to efficiently access parcelized data.

1.3.0.1 Technical Description

The three main types of indexes used in SDAL are the:

- Kd-tree spatial Index accesses spatial parcels.
- B-trees designed for fast searches of ordered record information.
- Dense Indexes to reorder records for fast searches by an alternate key.

1.3.0.2 Spatial Indexes

SDAL uses various types of Spatial Index trees, including the Global KD-tree, described here, and the 2d "Internal Kd-tree" stored in each route calculation and cartographic parcel.

The Global Kd-tree is explained in detail in the "Global KD Tree" section.

In addition to the Global Kd-tree for spatially organized parcels described previously, SDAL data includes various other global index trees for each database region.

1.3.0.3 PSF B-tree Index

The PSF B-tree is designed for fast searches of ordered record information.

Records are ordered by a key value and are parcelized into parcels of a single fixed size, and the records in parcels of a certain type are all of a fixed common size. Keys are of fixed sizes and are ordered. Record parcels are stored in contiguous sequence.

B-tree (index) parcels are of the same single, fixed size as the record parcels they index. They are stored contiguously in breadth-first tree order, and precede the record parcels in a single contiguous storage sequence of index parcels followed by record parcels.

The parcelID of the root B-tree parcel is stored in the region global-data area. Each B-tree index parcel has a header of the following structure:

```
typedef struct
{
    PclHdr_t      pclHdr;
    Ushort_t      usNumKeys;    /* number of keys in this parcel */
    Ushort_t      usKeySize;    /* key size in bytes */
    Ushort_t      usChildParcelPosition; /* position of first child of
                                     this parcel
                                     ** where tree root is position 0 */
    Uchar_t       ucTreeLevel;  /* ** parcels that index records are
                                     ** at level 1; parcels that index
                                     ** level n index parcels are at level
                                     ** n+1 */
    Uchar_t       ucReserved[1]; /* maintain 4-byte alignment */
}
BTreeIdxHdr_t;
```

Following the B-tree parcel header are the keys, stored in a contiguous sequence. Each key is the first key from the record or index parcel it references. For example:

- The first level 1 index parcel contains the first key from record parcel 1, the first key from record parcel 2, etc.
- The first level 2 index parcel contains the first key from the first level 1 index parcel, the first key from the second level 1 index parcel, etc.

At every layer of the B-tree, the B-tree index parcels are completely filled, except for the last parcel at any layer, which may be partially filled.

1.3.0.4 Dense Index

The Dense Index reorders records of a particular type to facilitate ordered searches of the data by some key value:

As an example, navigable feature records are stored in alphabetical order for the whole database. The SDAL PSF contains dense index parcels called NAV_ALPHAPLACE_PARCEL parcels. These contain records with both navigable feature IDs and administrative area IDs, stored alphabetically by navigable feature within the administrative area. Searches for records in the NAV_ALPHAPLACE_PARCEL parcels are done using the B-tree called NAV_FTR_ALPHAPLACE_IDX_PARCEL.

There are other dense index parcels for navigable features, as well as for other data types including POIs, places, etc. Each dense index parcel has a corresponding B-tree for searching for records in the dense index parcel.

The dense index parcels have the parcel header BTreeIdxHdr_t defined in “Field Type and Structure Definitions.” Like the B-tree parcels, the dense index parcels are completely filled, except for the last parcel, which may be partially filled.

1.3.0.5 POI Multi-Level Indexes

A set of index parcels containing several different index record types are called multi-level indexes. The location of each record type is specified by a root index record in the first parcel.

These record types index POI records by type or place, and by subtype, chain and postal code, within type or place.

These records in turn point to one of two types of dense index: One contains full POI names; the other contains only partial names. Each type points to the location of the full POI record in the alphabetically ordered POI parcels.

1.3.0.6 Global Kd-tree Parcelization Storage Format

In general, the global kd-tree may not fit in a single index parcel. A global kd-tree parcel may store a complete or partial tree (a global tree or a sub-tree), or it may store more than one complete tree (sub-trees of a global tree). All global kd-tree parcels must be of a common size and stored in a contiguous sequence starting with the parcel containing the root of the kd-tree (whose parcel ID is stored in global data information). The trees (roots) within each global kd-tree parcel are numbered, by tree identifier, in sequence starting with 0 within each index parcel. Thus, an inter-parcel reference from node to node is:

- by parcel ID, minus region specifier, plus tree ID.
- by relative byte offset Index trees, stored without being compressed in the index parcels.

The index parcel contains the following parcel header IDxPclHdr_t header

```

/*****
    IDxPclHdr_t: parcel header common to index parcel types
    *****/
typedef struct
{
    PclHdr_t      pclHdr;
    Ushort_t      usNumRoots;          /* number of tree roots */
}

```

```
        Ushort_t      usExtensionOffset; /* If non-zero: Offset to extension of header */
    }
    IDxPclHdr_t;
```

Following the header is the offset array to the roots in the parcel, indexed by tree ID:

```
    Ushort16 root_offset []; /* root offset array */
```

Following the offset array are the trees' data contents.

1.4 Field Type and Structure Definitions

The SDAL PSF uses special data formats to maximize data storage on the CD. Some data structures common to many PSF structures use specialized formats.

1.4.0.1 Field Type Definitions

The following methods of encoding a non-negative integer value into a variable number of bytes avoid wasting space when a numeric value (such as a length, an entity ID) can be large, but is often small:

Type 1 Variable Length Unsigned Value

This field ranges in length from one byte to four bytes, and can contain values ranging from zero to 1,114,095.

- If the first byte of this field has a (unsigned) value in the range 0 to 239 (0x00 to 0xEF), the field is one byte in length, and its value is the value contained in this byte.
- If the first byte of this field is in the range 240 to 254 (0xF0 to 0xFE), the field is two bytes in length. Its value is obtained by considering the two bytes as an unsigned Big Endian integer, setting the leftmost four bits of this field to zeros, and adding 240 to it.
- When this field is two bytes in length, its value can range from 240 to 4079.
- When the first two bytes of this field are in the range 0xFF00 to 0xFFEF, the field is three bytes in length. Its value is obtained by considering the second and third bytes as an unsigned Big Endian integer, and adding 4080 to it.
- When this field is three bytes in length, its value can range from 4080 to 65,519.
- When the first two bytes of this field are greater than or equal to 0xFFF0, the field is four bytes in length. Its value is obtained by considering the second through fourth bytes of the field as an unsigned Big Endian integer, setting its leftmost four bits to zeroes, and adding 65,520 to it.
- When this field is four bytes in length, its value can range from 65,520 to 1,114,095.

Type 2 Variable Length Unsigned Value

This field is either one byte or three bytes in length, and can contain values ranging from zero to 65,535.

- If the first byte of this field has an (unsigned) value in the range 0 to 254 (0x00 to 0xFE), the field is one byte in length, and its value is the value contained in this byte.
- When the first byte of this field is equal to 255 (0xFF), the field is three bytes in length, and its value is obtained by considering the second and third bytes as an unsigned Big Endian integer.
- When this field is three bytes in length, its value can range from 255 to 65,535.

This format is used in a limited number of special cases where even byte alignment is needed; usually the Type 1 format is used instead.

Type 3 Variable Length Unsigned Value

This field is either one half byte or one and a half bytes in length, and can contain values ranging from 0 to 270.

- If the first half byte of this field has an (unsigned) value in the range 0 to 14 (0x0 to 0xE), the field is a half byte in length. Its value is equal to that half byte considered as a four bit Big Endian value.
- When the first half byte of this field is equal to 15 (0xF), this field is 1½ bytes in length. Its value is obtained by subtracting 3825 from the field considered as a 12-bit Big Endian value (so 0xF00 signifies 15 and 0xFFF signifies 270).

Type 4 Variable Length Unsigned Value

This field ranges in length from two bytes to three bytes and can contain values ranging from 0 to 1,110,095.

- If the first nibble of this field has an (unsigned) value in the range 0 to 14 (0x0 to 0xE), the field is two bytes in length. Its value is the value considered as a two-byte unsigned Big Endian integer.
- When this field is two bytes in length, its value can range from 0 to 61,439.
- If the first nibble of this field is equal to 15 (0xF), this field is three bytes in length. Its value is obtained by considering the field as a three-byte unsigned Big Endian integer, setting the left-most four bits of it to zeros, and adding 61,440 to it.
- When this field is three bytes in length, its value can range from 61,440 to 1,110,095.

The following is a method for encoding a signed integer.

Type 5 Variable Length Signed Value

This field ranges in length from seven bits to 19 bits, and can contain values ranging from -32768 to +32767.

- If the first bit of this field is 0, the field is seven bits in length, and the remaining six bits represent a signed integer in the range -32 to +31.
- If the first two bits of this field are binary 10, the field is 11 bits in length, and the remaining nine bits represent a signed integer in the range -256 to +255.
- If the first three bits of this field are binary 110, the field is 15 bits in length, and the remaining 12 bits represent a signed integer in the range -2048 to +2047.
- If the first three bits of this field are binary 111, the field is 19 bits in length, and the remaining 16 bits represent a signed integer in the range -32768 to +32767.

The following is a method for encoding a variable length value defined at the parcel level.

1.4.0.2 Type Definitions and Structures

The following definitions and structures are used for several parcel types:

Parcel ID

The ParcelID_t structure is a 32-bit handle used to address a parcel within an SDAL Map file (e.g., 1.SDL). The current bit usage is:

31	30	29-24	23-0
External to region	Redundancy	Size index	Offset in SDAL region file in units of 2 ^m bytes where m = usBlockSize in GlbMediaHeader_t

- The **external-to-region bit** refers to a parcelID located in a different region (i.e., SDAL Map file) than the region of the parcel where it is stored. In this case, the remaining 31 bits are interpreted as an index into a table of parcelID+region structures.
- The **redundancy bit** indicates there are redundant copies of the parcel, whose parcel IDs are indicated by a table stored in the global data information.

- The **6 size index bits** allow for 64 distinct parcel sizes arranged in units of 2^m bytes where $m = \text{usBlockSize}$ in `GlbMediaHeader_t`. The bits are used as an index value in a table, `ucaParcelSizes[ ]` in `GlbMediaHeader_t`, stored in global data.
- The **24 offset bits** represent an offset in the SDAL region file in units of 2^m bytes where $m = \text{usBlockSize}$ in `GlbMediaHeader_t` to the start of the parcel.

Database Identifier

An SDAL Database Identifier (DBID) is a unique identifier (within an entity type) for a database entity.

The `DBID_t` structure is an eight-byte value containing the following bytes in sequence:

1. An 8-byte `ParcelID_t` structure describing the parcel containing the database entity.
2. A (region,layer) specifier, eight bits in length, allowing up to 256 (region,layer) combinations. Region values are between 1 and 48 (with region N corresponding to file “N.SDL” in the root directory of the CDROM), and layer values are between 0 and 4. The 8-bit specifier is encoded as: `gucaRLValueTable[layer][((region-1)>>4) | ((region-1)&0x0f)]`, where `gucaRLValueTable[5][3]` is the following table:

```
{ { 0x10, 0xb0, 0x60 }, { 0x70, 0x20, 0xc0 }, { 0xd0, 0x80, 0x30 }, { 0x40, 0xe0, 0x90 }, { 0xa0, 0x50, 0xf0 } };
```
3. An entity identifier, 24 bits in length which, for most entity types, uniquely identifies a data record only within a parcel; the entire DBID must fully identify one of these records. However, the entity identifiers of Segments, Nodes, POIs and Navigable Features uniquely identify those records within the entire region. For entity types other than these three, only two bytes of the entity identifier are used, since there are fewer than 65,536 entities of a particular type within one parcel.

In many cases, the SDAL PSF stores a database identifier internally in a form other than the `DBID_t` form presented in the SDAL DAPI to conserve space. For example:

- An entity other than a segment, node, POI or navigable feature, within a known parcel, can be identified using two bytes (e.g., bytes 6-7 in the diagram below, or a 2 byte value for the byte offset within the parcel).
- A segment or node, within a known parcel, can be identified using three bytes (e.g., bytes 5-7 in the diagram) in a parcel above the bottom layer, or two bytes (bytes 6-7 minus a minimum parcel entity ID) in the bottom layer, or two bytes (an index into the parcel's record offset array) in a parcel.
- A parcel within a known region can be identified using four bytes (e.g., bytes 0-3 in the diagram).
- An entity of a particular type, known to be in one of a limited number of parcels in spatial proximity to each other, can be identified using a two- or three-byte entity identifier plus a one-byte index into an array of parcel identifiers stored within a specific parcel.

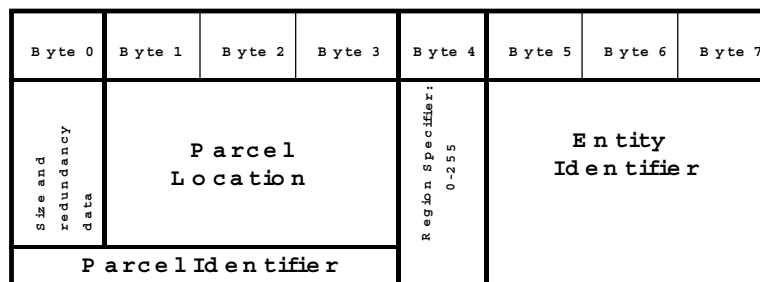


Figure 1-7: DBID (Database ID)

Field Typedefs

```

typedef      char      Bool_t;

typedef      signed char Schar_t;
typedef      unsigned char Uchar_t;
typedef      signed short Sshort_t;
typedef      unsigned short Ushort_t;
typedef      signed long Slong_t;
typedef      unsigned long Ulong_t;

typedef      Ushort_t    BlkOffset_t; /* Block Offset Array */
typedef      Uchar_t     RecOffsetEnt_t[5]; /* 3-byte entity ID;
then 2-byte offset */
typedef      Uchar_t     ExtSegOffsetEnt_t[6]; /* 3-byte entity
ID; 1-byte index
** bitmask; 2-byte offset */

```

Rectangle Definition Structure

```

typedef struct
{
    Slong_t      slMinLong; /* minimum longitude */
    Slong_t      slMinLat; /* minimum latitude */
    Slong_t      slMaxLong; /* maximum longitude */
    Slong_t      slMaxLat; /* maximum latitude */
}
DBRect_t;

```

Parcel Header Common To All Parcel Types

```

typedef struct
{
    /* common to all parcel types */
    ParcelID_t    parcelid;
    Ushort_t      usParcelDesc;
    Uchar_t       ucParcelType;
    RegionID_t     region;
    Bool_t        bEndianSwap; /* set when header endian con-
verted*/
    Uchar_t       ucCmpDataSizeHi; /* compressed size in bits, byte
1*/
    Ushort_t      usCmpDataSizeLo; /* compressed size in bits, byte
2&3*/
    Ushort_t      usCompressType; /* type of parcel compression */
    Ushort_t      usCmpData; /* byte offset to compressed data
*/
    Ushort_t      usCmpDataUncompSize; /* uncompressed parcel size in
bytes */
    Ushort_t      usExtensionOffset; /* If not 0: Offset to hdr exten-
sion*/
}
PclHdr_t;

```

Parcel Header Common To Index Parcel Types

```

typedef struct
{

```

```
PclHdr_t      pclHdr;

Ushort_t      usNumRoots;          /* number of tree roots */
Ushort_t      usExtensionOffset; /* If not 0: Offset to hdr extension*/
}
IDxPclHdr_t;
```

Parcel Header Common To Parcel Types With Pclid Xrefs

```
typedef struct
{
    PclHdr_t      pclHdr;

    /* parcelid table */
    Ushort_t      usPclidOffset;
    Ushort_t      usPclidCount;

    /* parcelid table: external to region */
    Ushort_t      usExtpclidOffset;
    Ushort_t      usExtpclidCount;
    Ushort_t      usExtensionOffset; /* If not 0: Offset to hdr extension*/

    Uchar_t      ucaReserved[2]; /* Maintain 4-byte alignment */
}
XrfPclHdr_t;
```

Parcel Header Common To Spatial Parcel Types

```
typedef struct
{
    XrfPclHdr_t      xrfPclHdr;

    /* common to spatial parcel types */
    DBRect_t      boundingRect;
    DBRect_t      tileRect;
    DBRect_t      ancestorRect;
    Ushort_t      usNeighborKDOffset;
    Ushort_t      usNeighborKDSize;
    Ushort_t      usInternalKDOffset; /* unused for POI parcels */
    Ushort_t      usInternalKDSize; /* unused for POI parcels */
    Ushort_t      usExtensionOffset; /* If not 0: Offset to hdr extension*/

    Uchar_t      ucScale;
    Uchar_t      ucLayer; /* unused for non-layered parcels */
}
SptlPclHdr_t;
```

Parcel Header Structures

The BlkDesc_t structure describes records assigned entity IDs consecutively within the parcel beginning at 0. The structures BldDesc1_t, BlkDesc2_t and BlkDesc3_t are variations of this scheme.

```
typedef struct
{
    Ushort_t      usOffset;          /* Offset (into parcel) of block offset array */
    Ushort_t      usCount;           /* Record count */
    Uchar_t      ucBlkCnt;           /* Number of record blocks */
    Uchar_t      ucRecsPerBlk       /* Recs/block (shift value: 5 -> 2**5 =
```

```

        32) */
        Ushort_t      usMaxSize;      /* Max record size for this record type
        in this parcel */
        Ushort_t      usDataOffset; /* Offset (into parcel) of first data
        record */
        Ushort_t      usMaxBlkSize; /* Maximum block size in bytes */
    }
    BlkDesc_t;

```

The BlkDesc1_t structure describes records assigned entity IDs consecutively within the parcel, beginning at some positive entity ID.

```

typedef struct
{
    Ushort_t      usOffset;      /* Offset (into parcel) of block
                                offset array */
    Ushort_t      usCount;      /* Record count */
    Ulong_t       ulMinIdentifier; /*Min entityID for this entity type
                                in this parcel */
    Uchar_t       ucBlkCnt;      /* Number of record blocks */
    Uchar_t       ucRecsPerBlk /* Recs/block (shift value: 5 ->
                                2**5 = 32) */
    Ushort_t      usMaxSize;      /* Max record size for this record
                                type in this parcel */
    Ushort_t      usDataOffset; /* Offset (into parcel) of first
                                data record */
    Ushort_t      usMaxBlkSize; /* Maximum block size in bytes */
}
BlkDesc1_t;

```

The BlkDesc2_t structure describes records in sequence within the parcel by entity ID, but whose entity IDs are not consecutive.

```

typedef struct
{
    Ushort_t      usOffset;      /* Offset (into parcel) of record off-
                                set array */
    Ushort_t      usCount;      /* Record count */
    Ushort_t      usMaxSize;      /* Max record size for this record type
                                in this parcel */
    Ushort_t      usDataOffset; /* Offset (into parcel) of first data
                                record */
}
BlkDesc2_t;

```

The BlkDesc3_t structure describes records with no entity ID that are accessed directly from elsewhere in the parcel by offset.

```

typedef struct
{
    Ushort_t      usCount;      /* Record count */
    Ushort_t      usMaxSize;      /* Max rec. size for this record type
                                in this parcel */
    Ushort_t      usDataOffset; /* Offset (into parcel) of first data
                                record */
    Uchar_t       ucaReserved[2]; /* Maintain 4-byte alignment */
}
BlkDesc3_t;

```

The `BlkDesc4_t` structure describes records with no entity ID that are accessed directly from elsewhere in the parcel by offset. They are also of fixed size and are not compressed, so no maximum record size is needed.

```
typedef struct
{
    Ushort_t      usCount;          /* Record count */
    Ushort_t      usDataOffset; /* Offset (into parcel) of first data
                                record */
}
BlkDesc4_t;
```

The `BlkDesc5_t` structure describes records of fixed size and can therefore be accessed by index. The record size in bits is the record size in bytes multiplied by eight, unless records can be of a non-integral byte length.

```
typedef struct
{
    Ushort_t      usCount;          /* Record count */
    Ushort_t      usMaxSize;       /* Record size in bytes for this record
                                type */
    Ushort_t      usDataOffset; /* Offset (into parcel) of first data
                                record */
    Uchar_t       ucBitSize;       /* Record size in bits for this record
                                type */
    Uchar_t       ucaReserved[1]; /* Maintain 4-byte alignment */
}
BlkDesc5_t;
```

PSF Database Identifier Typedefs

The following C structures define common PSF database identifier types.

```
/******
**database identifier types
*****/

typedef Uchar_t RegionID_t; /* database region */

typedef Ulong_t ParcelID_t;
/*
** byte 0 == size and redundancy info
** byte 1-3 == sector offset from start of sdal file
*/

typedef struct
{
    ParcelID_t      parcelID;
    RegionID_t      ucRegion;
} ParcelID_Reg_t;

/*
** byte 0 == size and redundancy info
** byte 1-3 == sector offset from start
of sdal file
** byte 4 == database region */

typedef Ulong_t EntityID_t;

typedef Ulong_t DBID_t[2];
```

```

/*
** byte 0    == size and redundancy info
** byte 1-3 == sector offset from start of sdal file
** byte 4    == database region
** byte 5-7 == record entity id
*/

typedef Uchar_tEntityType_t; /* database entity type */

```

1.5 Parcel Level Compression

The SDAL PSF supports up to 255 compression trees for a specified compression algorithm, in the region global data. An index to a compression tree is stored in the parcel headers, along with the compression type code. These two values uniquely define how to decompress the parcel data.

Note: Data from multiple parcel types can be compressed with a single compression tree.

The compression trees are accessed through a table of `CompressedDataElement_t` records stored in region global data. this table points to every Huffman tree used for parcel-level compression and is accessed from the global region header. For detailed information, refer to the *LZ-Huffman tree* heading of the *SDAL Region File* section.

The parcel header at the beginning of the parcel and other data at the end of the parcel are not compressed. Each parcel may contain only one contiguous piece of compressed data.

Parcel IDs refer to the size and location of the compressed parcels, not the uncompressed data.

Internal compression techniques may store the compression tree in each parcel, and no reference to global data is required.

1.6 Record Level Compression

Record-level compression is used to compress specific record types within parcels of the same parcel type, leaving other data in that parcel intact. The Huffman trees are stored as either a separate copy inside each parcel or as one copy in the region global data.

The information about which record types are compressed is stored in a table of `CompressInfo_t` records in the SDAL global media file. It is a table with entries for each record type considered compressible. Each entry has parameters for the specific compression algorithm, extent of compression data, location of the Huffman tree, etc. For detailed information, refer to the *Compression Information* section of the *SDAL Global Media* section. Those Huffman trees stored in the SDAL region file are accessed through the table of `HuffTreeIdx_t` records in region global data. They are described in the *Huffman Tree* section of the *SDAL Region File* section.

1.7 Compiler Design Implications of SDAL Data Compression

1.7.0.1 Overview

The SDAL PSF at this time supports two types of compression:

- Huffman-coded individual-record compression
- LZ-based parcel-level compression

The descriptions here assume familiarity with the basic techniques of Huffman and LZ (Lempel-Ziv) compression.

Huffman-coded individual-record compression adheres closely to the classic Huffman algorithm. The aspects unique to SDAL are that a particularly space-efficient representation is used for the Huffman code trees, and the PSF compiler has the option to make the Huffman code trees internal to the parcel, or external in global data. (External trees are common to all the regions on the media.)

Parcel-level compression employs a variant of sliding-window LZ compression with Huffman encoding of literals, offsets and lengths, using separate Huffman code trees in sets of three. The algorithm is designed to minimize decompression time and memory usage, taking advantage of the fact that LZ-compressible “redundancy” within an SDAL parcel consists mainly of short byte-strings in close proximity to each other within the parcel, and parcels of the same type have similar structures and data patterns. The algorithm also is designed so it does not require dynamic construction or updating of any large or complex data structures at run-time; all Huffman tree sets used are constructed and included in the database by the PSF compiler. Because the similarity of parcels within a type makes it feasible to share common sets of trees among large numbers of parcels, the Huffman tree sets are stored externally in region global data and referenced by multiple parcels.

“Parcel-level” compression does not compress the entire parcel. A small, leading part of the header is always left uncompressed; among other things, it provides parameters required for proper decompression (compressed and uncompressed data sizes, compression algorithm identification, and decode-tree set identification). Also, an uncompressed data area may follow the compressed data at the end of the parcel, allowing post-compression addition of data whose values cannot be determined before compression, such as tables of the parcel IDs of parcels referenced from the compressed parcel.

1.7.0.2 Small Huffman Tree Structure

The key to the Small Huffman tree structure is understanding it is a minimal representation created by building a 256-code Huffman tree in a one-dimensional array of 511 node elements, and then removing all unnecessary data.

In the original array, each encoded 8-bit symbol is assigned to a leaf whose array index is the numeric value of that symbol treated as a binary number, with the nodes (i.e., internal nodes) of the tree residing in the array elements with index values of 256 and up, and the lower-frequency child leaf or node of a parent node as the left child. Leaves having zero frequency of occurrence need not be included in the tree. Each element in the original array contains (at least) left and right child pointers, a parent pointer, and a count of the cumulative frequency of occurrence of all the leaves below it. (The frequency of occurrence is the 8-bit symbol whose value is its array index.) As algorithms for building such trees are well known, the description here indicates only how such a tree is converted to a Small Huffman tree. An example here shows how this is done.

Assume the data set to be Huffman-coded is the single word MASSACHUSETTS. In ISO-8859-1 coding, terminated by a null character (numeric value = 0), the result is the tree structure shown next (A=65, C=67, E=69, H=72, M=77, S=83, T=84, U=85). This results in the original tree having the structure shown (only populated array elements appear):

Original Huffman Tree Array

Index	Index of Left Child	Index of Right Child	Index of Parent	Cumulative Frequency
0	—	—	256	1
65	—	—	259	2
66	—	—	—	0
67	—	—	256	1
68	—	—	—	0

69	—	—	257	1
72	—	—	257	1
77	—	—	258	1
83	—	—	261	4
84	—	—	259	2
85	—	—	258	1
256	0	67	260	2
257	69	72	260	2
258	77	85	261	2
259	65	84	262	4
260	256	257	262	4
261	258	83	263	6
262	259	260	263	8
263	261	262	—	14
510	—	—	—	—

Because a Huffman code is classically decoded one bit at a time by traversal from root to leaf, going left for a “0” bit and right for a “1” bit, and nothing is done at the leaf except to identify the associated symbol, most of the array shown on the previous page is superfluous for decoding purposes. Because the indexes of the elements 0 through 255 are themselves the symbol codes, the decoding algorithm can know any index less than 256 is that of a leaf and thus, by definition, the value to be output. Because the root of the tree is at element 263, and the higher-numbered elements are empty, the elements above 263 are not needed. Moreover, because decoding proceeds only downward through the tree, and the tree is not reconfigured at run time, the parent index values and frequency-of-occurrences are unnecessary. So, for decoding, all that is required is a “Root” pointer and a small part of the tree, as shown next:

Root = 263

256	0	67
257	69	72
258	77	85
259	65	84
260	256	257
261	258	83
262	259	260
263	261	262

As the array index values can range from 0 to 510, nine binary bits must represent each array index. However, because the part of the array included is known to start at element 256 of the original, it is convenient to treat the most significant bit of each array index value as a flag to discriminate between leaves and (internal) nodes. This allows the indexes to be re-based to zero, as follows:

Root = 7

0	0, 0	0, 67
1	0, 69	0, 72
2	0, 77	0, 85
3	0, 65	0, 84
4	1, 0	1, 1
5	1, 2	0, 83
6	1, 3	1, 4
7	1, 5	1, 6

As the leaf/node discriminator flag bits are used in decoding only to detect when a leaf has been reached, and index values are most easily and quickly handled as bytes, it is also convenient to store the flag bits separately in two dedicated bit arrays, one for the left-child indexes and one for the right-child indexes; these are defined in C as arrays of eight signed longs, with the convention that the most significant bit of element 0 is bit 0, and the least significant bit of element 7 is bit 255 (i.e., if the left child of the node in element 'n' of the truncated array is a node rather than a leaf, the flag bit is set with the following C code):

```
laIsLeftANode[ n/32 ] |= (0x00000001 << (31 - (n % 32)));
```

So, the basic tree node array is reduced to an array of 2-byte elements, each byte containing the low-order eight bits of the original child index value (i.e., the value to the right of the comma in the example array above).

The end result is the nominal Small Huffman tree structure, which is defined in C as follows:

```
typedef struct {
    Uchar_t      ucLeftChild;
    Uchar_t      ucRightChild;
```

```

    }
    SmallHuffNode_t;

#define LEAF_CNT 256
#define NODEFLAG_BITARRAY_SZ      8

typedef struct {
    Slong_t      laIsLeftANode[NODEFLAG_BITARRAY_SZ];
    Slong_t      laIsRightANode[NODEFLAG_BITARRAY_SZ];
    Uchar_t      ucRoot;
    Uchar_t      ucReserved; /* for alignment */
    SmallHuffNode_t shnaTreeNode[LEAF_CNT - 1];
}
SmallHuffTree_t;

```

As a final space-saving measure, the tree as actually stored in the PSF can be truncated after the `shnaTreeNode[ ]` element pointed to by `ucRoot`. For the example above, that means truncation after `shnaTreeNode[7]`. As the in-vehicle decoding algorithm never looks at any `shnaTreeNode[ ]` element beyond that pointed to by `ucRoot`. The node flag bit arrays are not similarly truncated because the byte savings are too small to justify the added complexity unless a large fraction of the possible symbol values have zero frequency of occurrence. (This is rare in the actual data).

1.7.0.3 Fast Huffman Tree Structure

The Fast Huffman tree is an offshoot of the Small Huffman tree, modified for faster decoding.

The basis of the modification is to remove the top seven levels of a Small Huffman tree, replacing them with a 128-element Fast Decode Array. This maps each possible 7-bit bit series either to the first leaf index (i.e., symbol) where it would lead if normally decoded. Or it maps to the tree internal node when its decoding would not lead all the way to a leaf. (See the example shown next.)

During decode, the decompression routine extracts seven bits enbloc from the compressed bit-stream, and use their binary value as an index into the Fast Decode Array. Each element of the Fast Decode Array consists of a 1-byte `FlagsAndCount` field and a 1-byte `NodeArrayIndex` field.

If the decoding of some number of bits ‘m’ in that bit series leads in the original Huffman tree to a leaf, then the `FlagsAndCount` field’s terminal/non-terminal discriminator flag is ‘0’. Its count subfield indicates the number of bits that would have been consumed in decoding up to that leaf, and the `NodeArrayIndex` field gives the index of that leaf (i.e., the decoded symbol).

If the decoding of all seven bits in that bit series leads in the original Huffman tree only to an internal node, then the `FlagsAndCount` field’s terminal/non-terminal discriminator flag is ‘1’. Its count subfield indicates the number of bits that would have been consumed in decoding up to that node (i.e., seven), and the `NodeArrayIndex` field gives the index of the node from which further decoding must continue. The remainder of the decoding is done by conventional bit-at-a-time decoding down through the untruncated portion of the Small Huffman tree.

The example below shows this concept using the Small Huffman tree example constructed above for “MASSACHUSETTS”, and a Fast Decode Array of eight elements rather than 128, (both for brevity, and because otherwise no code would reach to an internal node). For reference, the Huffman codes derived from the original Huffman tree are:

000 = M001 = U01 = S100 = A101 = T

1100 = null1101 = C1110 = E1111 = H

Fast Decode Array

Index*	NodeArrayIndex	FlagsAndCount**
000	77	0, 3
001	84	0, 3
010	83	0, 2
011	83	0, 2
100	65	0, 3
101	85	0, 3
110	0	1, 3
111	1	1, 3

* In binary notation

**Notation: <flag>, <count>

Truncated Small Huffman Tree

Index	Index of Left Child	Index of Right Child
0	0, 0	0, 67
1	0, 69	0, 72

The end result is the nominal Fast Huffman tree structure, defined in C as follows:

```

#define FAST_HUFF_IDX_BITLEN_SIZE      3
#define FAST_HUFF_IDX_BITLEN ((0x1 << FAST_HUFF_IDX_BITLEN_SIZE) -1)
#define FAST_HUFF_ARRAY_SIZE (0x1 << FAST_HUFF_IDX_BITLEN)
#define HUFF_NONTERMINAL 0x80
/* 1: ucNodeArrayIdx is index of the internal
   ** Huffman tree node;
   ** 0: ucNodeArrayIdx is index of the virtual
   ** leaf node (the output byte value) */
#define COUNT_MASK ((0x1 << FAST_HUFF_IDX_BITLEN_SIZE) -1)

typedef struct {
    Uchar_t ucNodeArrayIdx;
    Uchar_t ucFlagsAndCount; /* bit 7 is non-terminal flag */
                             /* bits 0-6 are count */
} FastDecodeArrayEl_t;

typedef struct FastHuffTree_tag {
    FastDecodeArrayEl_t fastDecode[FAST_HUFF_ARRAY_SIZE];
    SmallHuffTree_t      smallHuffTree;
    /* can be truncated above highest point
    of entry from fast decode array */
} FastHuffTree_t;

```

Fast Huffman trees are used in parcel-level decompression in sets of three as described further under *Parcel-Level Compression*.

1.7.0.4 Parcel-Level Compression

The header of each compressed parcel must provide the compression-related information below:

```
typedef struct { /* common to all parcel types */
    ParcelID_t      parcelid;
    Ushort_t        usParcelDesc;
    Uchar_t         ucParcelType;
    RegionID_t      region;
    Bool_t          bEndianSwap;      /* set when header endian-converted
                                        */
    Uchar_t         ucCmpDataSizeHi; /* compressed data size in bits,
                                        byte 1 */
    Ushort_t        usCmpDataSizeLo; /* compressed data size in bits,
                                        byte 2&3 */
    Ushort_t        usCompressType;   /* type of parcel compression */
    Ushort_t        usCmpData;        /* byte offset to start of com-
                                        pressed data */
    Ushort_t        usCmpDataUncompSize; /* total size in bytes of parcel
                                        data after decompression, (includ-
                                        ing parts not compressed) */
    Ushort_t        usExtensionOffset; /* If non-zero: Offset to extension
                                        of header*/
} PclHdr_t;
```

The value in the `usCompressType` field low byte must be one of the following:

```
NO_PARCEL_COMPRESSION
SZIP_PARCEL_COMPRESSION_EXT/* external Huffman trees */
SZIP_PARCEL_COMPRESSION_INT/* internal Huffman trees */
```

If the value is `SZIP_PARCEL_COMPRESSION_EXT`, the high byte must contain a non-zero number indicating the external Huffman tree set to be used in decompression. (This is explained in the next section.)

Parcel-level compression employs a fairly straightforward variety of LZ compression with Huffman encoding of literals, LZ back-pointers, and counts using sets of three Fast Huffman trees. An SDAL database can contain up to 255 such tree sets in working memory “external” to the parcels, and any parcel in the database can be compressed using any of those sets. These are referenced via the high byte of `usCompressType` in the parcel header. The LZ references are encoded using back-pointers (the negative offset from the substring-to-be-compressed to the earlier matching substring), which are Huffman-compressed.

The `usCmpDataUncompSize` must be the *total* size of the parcel data after decompression, including any non-compressed data and padding that follow the compressed data in the compressed parcel. The algorithm is:

```
End_Fill_Padding_Size =
    Compressed_Parcel_Size -
    Non_Compressed_Header_Size -
    Compressed_Data_Post_Compression_Size -
    Non_Compressed_End_Data_Size
```

Where “Compressed_Parcel_Size” is the smallest of the possible sector-aligned sizes that can hold the data. With the fill size known, the compiler can then compute:

```
hdr.usCmpDataUncompSize =
    Non_Compressed_Header_Size +
    Compressed_Data_Post-Decompression_Size +
```

```
Non_Compressed_End_Data_Size +  
End_Fill_Padding_Size
```

Note: Non_Compressed_Header_Size = usCmpData.

1.7.0.5 Parcel-Level Compression Huffman Tree Set Structure

After construction of the three Fast Huffman trees to be used in a particular parcel-level compression tree set, the compiler must integrate them into the tree set structure, in which the truncated Fast Huffman trees are concatenated following a header of the following form:

```
typedef struct HuffOffsets {  
    Ulong_t treeOff1; /* offset to Tree 1 */  
    Ulong_t treeOff2; /* offset to Tree 2 */  
    Ulong_t treeOff3; /* offset to Tree 3 */  
    Ulong_t parameter; /* LZ back-pointer length */  
    Ulong_t dataOff;   /* offset to data,  
                        if an internal tree set */  
} HuffOffsets_t;
```

If the concatenated tree set is to be an external tree set, it is then placed in the global data along with the other tree sets. If it is to be a parcel-internal tree set, it is placed in the parcel at the offset given by the parcel header usCmpData field, and its dataOff field is set to the offset to the start of the actual compressed data.

Section 2 PSF Media and Region Global Data Definitions

2.1 SDAL Global Media File

The SDAL Global Media File is a separate file that stores per-media information. It always corresponds to region zero (filename 0.SDL), and begins at media sector address zero.

The SDAL Media File contains the Global Media Header and Global Media Index parcels, followed by the metadata, reference data, and non-trademarked icon data.

The SDAL Media File is divided into sections: Global Media Header, Global Media Index, Compression Data, Reference Data, and Non-Trademarked Icon Data. Each section is further divided into one or more parcels. These parcels are variable-length within a certain granularity and maximum size specified for the PSF. An exception to the PSF-specific granularity and size considerations is made for the first two parcels in the file to allow bootstrapping.

The Global Media Index parcel contains parcel ID and record ID extent information. An external index to the remaining items is found in the Global Media Index parcel which immediately follows the Global Media Header.

The various types of data located in the SDAL Media File can span multiple parcels to maximize space usage. Regardless of this capability, specific data records or tables are not allowed to span parcels.

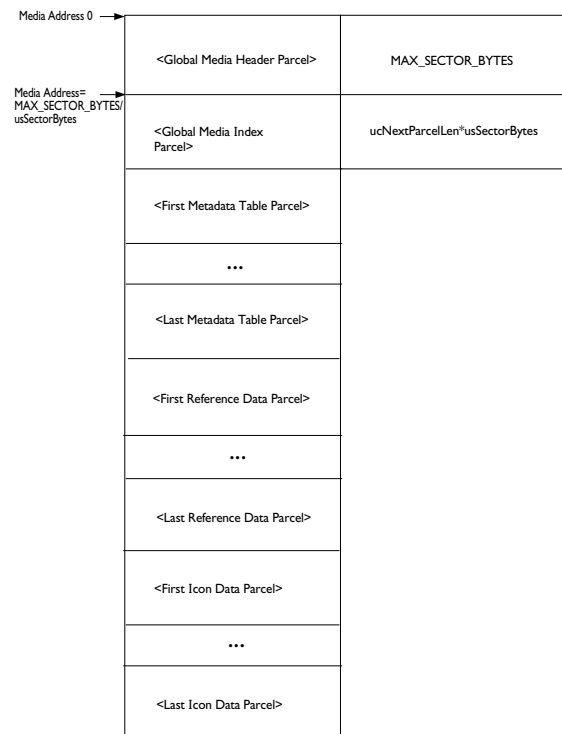


Figure 2-1: Global Media File Structure

The following Global Media Index parcel contains the variable-length data index to the remaining information in the file. It is located `MAX_SECTOR_BYTES` from the start of the SDAL Media File (corresponding to the file-relative sector address found by dividing `MAX_SECTOR_BYTES` by the current number of byte per sector (obtained by left shifting 1 by the value in the `usBlockSizeBits` field in the Global Media Header section obtained from the first parcel)).

Although these parcels are not completely full, using a length that is an integer multiple of the maximum sector size allows the parcels to be read regardless of PSF and physical media sector size considerations. The various data parcels follow, with location and size information encoded in the parcel ID references found in the Global Media Index parcel.

2.1.1 Global Media Header and Index Parcels

2.1.1.1 Technical Description

The Global Media Header Parcel is always located at the beginning of the SDAL Media File and contains a fixed-length data structure and Index Offset and Parcel Count files for each data type in the file. The Global Media Index Parcel is located immediately after the Global Media Header parcel and contains the variable-length data index for the remaining information in the file.

The layout for the Global Media Header and Global Media Index Parcels are illustrated in Figure 1-2:

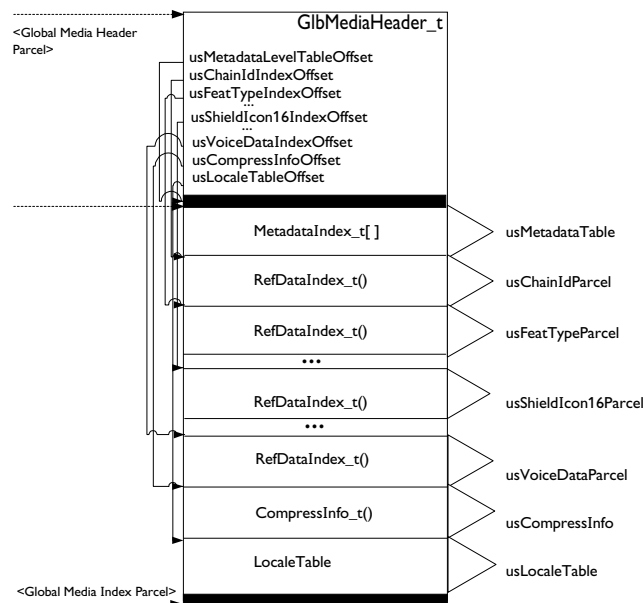


Figure 2-2: Global Media Header and Index Parcels

2.1.1.2 Global Media Header Parcel

The Global Media Header Parcel is located in the SDAL Media File corresponding to sector 0 relative to the starting sector of the file. The header parcel has a length equal to the largest sector size in any SDAL PSF (defined by the *media-independent* constant MAX_SECTOR_BYTES). This is the 2 Kbyte sector size used for CD-ROM media.

2.1.1.3 Global Media Header Structure

The Global Media Header section is completely contained within the first parcel of the SDAL Media File. This parcel contains the fixed-length Global Media information only. The Global Media Header contains the offset and count information required to locate and load the variable-length index information from the Global Media Index parcel. These offsets are relative to the beginning of the Global Media Index parcel.

Because the Global Media Header is fixed in length, a C structure is imposed on it. These constants are referenced to the C structure that follows.

```

#define          GLB_NAA_LEN      4                                /*
                                                                **bits 31-29: Speed Cat
                                                                **bits 28-27: Lane Cat
                                                                **bits 26-17: Access Characteris-
                                                                tics
                                                                **bits 16-0: Display Characteris-
                                                                tics
                                                                */

#define          GLB_NAA_NUM      256                             /* Num.of normalized attributes
                                                                */

#define          GLB_MAX_AFFIX    8                               /* Number of street prefix/suf-
                                                                fix names (N, S, W, E, ...)*/
#define          GLB_MAX_NODE_COST 32                             /* Maximum number of permanent
                                                                node cost values in database */

#define          GLB_MAX_EXT_IDX  64                             /* Maximum number of external
                                                                index parcel types */

#define          GLB_MAX_INT_IDX  32                             /* Maximum number of parcel types
                                                                with in-parcel indices */

#define          MAX_PARCEL_SIZES 128                             /* 4 size bits in parcel ID */
#define          MAX_SECTOR_BYTES 2048                           /* Maximum size of one sector and
                                                                of Global Media Header parcel */

#define          MAX_ADMIN_LEVEL  7                               /* Maximum number of unique
                                                                administrative levels in hierar-
                                                                chy */

typedef struct
{
    /* Normalized attribute array */
    Uchar_t      ucaNormalizedAttrs[GLB_NAA_NUM][GLB_NAA_LEN];

    /* Index type metadata */
    Uchar_t      ucaExtIdxBitmap[GLB_MAX_EXT_IDX / 8];
    Uchar_t      ucaIntIdxBitmap[GLB_MAX_INT_IDX / 8];

    /*Variable length information found in Global Media Index parcel*/

    /* Point to metadata structures in Media Index parcel */
    Ushort_t     usMetadataLevelTableOffset;
    Ushort_t     usMetadataLevelTableCount;

    Ushort_t     usMetadataModuleTableOffset;
    Ushort_t     usMetadataModuleTableCount;
    Ushort_t     usMetadataHashvalue;

    /* Point to RefDataIndex_t structures in Media Index parcel */
    Ushort_t     usChainIDIndexOffset;
    Ushort_t     usChainIDParcelCount;
    Ushort_t     usChainIDMaxIndex;

```

```

/* Point to RefDataIndex_t structures in Media Index parcel */
Ushort_t      usFeatTypeIndexOffset;
Ushort_t      usFeatTypeParcelCount;
Ushort_t      usFeatTypeMaxIndex;

/* Point to RefDataIndex_t structures in Media Index parcel */
Ushort_t      usFacTypeIndexOffset;
Ushort_t      usFacTypeParcelCount;
Ushort_t      usFacTypeMaxIndex;

/* Point to RefDataIndex_t structures in Media Index parcel */
Ushort_t      usAttrValueIndexOffset;
Ushort_t      usAttrValueParcelCount;
Ushort_t      usAttrValueMaxIndex;

/* Point to RefDataIndex_t structures in Media Index parcel */
Ushort_t      usLangCodeIndexOffset;
Ushort_t      usLangCodeParcelCount;
Ushort_t      usLangCodeMaxIndex;

/* Point to RefDataIndex_t structures in Media Index parcel */
Ushort_t      usCountryInfoIndexOffset;
Ushort_t      usCountryInfoParcelCount;
Ushort_t      usCountryInfoMaxIndex;

/* Point to RefDataIndex_t structures in Media Index parcel */
Ushort_t      usTimeZoneIndexOffset;
Ushort_t      usTimeZoneParcelCount;
Ushort_t      usTimeZoneMaxIndex;

/* Point to RefDataIndex_t structures in Media Index parcel */
Ushort_t      usDSTIndexOffset;
Ushort_t      usDSTParcelCount;
Ushort_t      usDSTMaxIndex;

/* Point to GlbIconHdr_t structures in Media Index parcel */
Ushort_t      usShieldIconIndexOffset;
Ushort_t      usShieldIconDataMaxIndex;

/* Point to GlbIconHdr_t structures in Media Index parcel */
/* POI type icons follow highway shield icons */
Ushort_t      usPOIIconIndexOffset;
Ushort_t      usPOIIconDataMaxIndex;

Ushort_t      usVoiceDataIndexOffset; /* always zero */
Ushort_t      usVoiceDataParcelCount; /* always zero */
Ushort_t      usVoiceDataMaxIndex;    /* always zero */

/* Entity-level compression metadata */
/* Point to CompressInfo_t structures in Media Index parcel */
Ushort_t      usCompressInfoOffset;
Ushort_t      usCompressInfoCount;
Ushort_t      usHuffTreeOffset;
Uchar_t       ucHuffTreeCount;
Uchar_t       ucReserved1[3];

/* Locale table (per-continent, therefore per-media)*/
/*Points to LocaleTable_t structures*/
Ushort_t      usLocaleTableOffset;

```

```

Ushort_t      usLocaleTableLen;

/* Parcel size information */

Uchar_t      ucaParcelSizes[MAX_PARCEL_SIZES]; /* may not be
full*/
Uchar_t      ucParcelSizeCount; /* 1 to MAX_PARCEL_SIZES */
Uchar_t      ucNextParcelLen; /* sector length of Media ** Index
parcel */

Ushort_t      usBlockSize; /* 2**x for PSF block size */
Ushort_t      usBlockNbrBits; /* Number of bits of block number */
Ushort_t      usBlockSizeBits; /* Number of bits of block size **
index */

/* PSF version and media type information */
Uchar_t      ucPSFMajor;
Uchar_t      ucPSFMinor;
Uchar_t      ucPSFIssue;
Uchar_t      ucPSFPoint;
Uchar_t      ucPSFType;
Uchar_t      ucPSFDay; /* Binary 1-31 */
Uchar_t      ucPSFMonth; /* Binary 1-12 */
Ushort_t      usPSFYear; /* Binary 1996-65535 */

/* Node cost table */
Ushort_t      usaNodeCost [GLB_MAX_NODE_COST];

Ulong_t      Unused;

/* TPD vendor mask */
Ulong_t      ulTPDVendorMask;
ParcelID_t    firstIconParcelID;

/* Point to RefDataIndex_t structures in Media Index parcel */
Ushort_t      usAffixIndexOffset;
Ushort_t      usAffixParcelCount;
Ushort_t      usAffixMaxIndex;

/* Generic structures for adding more reference data types */
/* Point to RefDataIndex_t structures in Media Index parcel */
Ushort_t      usGenericRefDataIndexOffset [MAX_GENERIC_REF_DATA];
Ushort_t      usGenericRefDataParcelCount [MAX_GENERIC_REF_DATA];
Ushort_t      usGenericRefDataMaxIndex [MAX_GENERIC_REF_DATA];

/* Parcel size of parcels in this specific parcel type */
/* It can vary from database to database */
Ushort_t      usPOIMultiLevelParcelSize;

Ushort_t      usMaxChainsPerFacility;
Ushort_t      usMaxSubtypesPerFacility;
}
GlbMediaHeader_t;

```

Note: The *usBlockSize* field above is important because it determines the logical sector size used for the rest of the database. The sector size is obtained by left shifting 1 by the field's value. The practical values of the *usBlockSize* field can be 8, 9, 10, or 11, corresponding to possible sector sizes of 256 bytes, 512 bytes, 1 Kbyte, or 2 Kbytes.

Decreasing the value of *usBlockSize* leads to increasing the number of possible parcel sizes, up to the MAX_PARCEL_SIZES limit. Also, parcel IDs of same-sized parcels show these parcels using correspondingly more sectors. An example of this is the *ucNextParcelLen* field. Assuming it carries parcel size of 4 Kbytes, its value can be 2 for *usBlockSize* value of 11 and 16 for *usBlockSize* value of 8.

Normalized Attribute Array

The Normalized Attribute Array is an array of commonly occurring attribute combinations associated with a routing segment. The Normalized Attribute Array is used because attribute values for a specific list of attributes (rank, lane category, speed category, speed limit access characteristics and other segment attributes) are not completely independent of each other. For example, the rank of a segment indicates it is an inter-metropolitan highway, it is normally in a high speed category. It has been determined that the list of combinations of these attributes actually found in the data is small enough that it saves space by storing these combinations in one table (the Normalized Attribute Array) and storing an index into that list in the Segment Record, discussed previously. This allows a single index to be carried in the Segment Record, instead of all the associated attribute values.

The Global Normalized Attribute Array contains the “top 256” attribute combinations *worldwide*, thus allowing them to be stored in the SDAL Media File. Any other combinations would be stored local to the parcel in which they are used. There is, thus, a single Global Normalized Attribute Array stored in the SDAL Media File, while there may be several different local normalized attribute arrays in different Route Calculation parcels.

The Global Normalized Attribute Array is the *ucaNormalizedAttrs[]* field of the Global Media Header structure *GlbMediaHeader_t*, while the local Normalized Attribute Array is the *ucaAttribArray[]* array of the Route Calculation parcels. Both the Local and Global Normalized Attribute Arrays have the same two-dimensional format. Each array stores the 256 most commonly used attribute combinations. The first dimension of the array ranges from 0-255 and is the index to the attribute combinations. The second dimension contains the bit values for these attributes, as described below.

The Normalized Attribute arrays contain sets of values for the following fields:

- Speed Category
- Lane Category: Not applicable, One lane, Two or three lanes, Four or more lanes
- Access Characteristics: Automobiles, Buses, Taxis, Carpool, Pedestrians, Bicycles, Trucks, Through traffic, Deliveries, Emergency vehicles
- Display Characteristics: Paved, Private, Frontage Road, Bridge, Tunnel, Ramp, Tollway, POI Access, Controlled Access, Roundabout, Intersection Internal, Undefined Traffic Area, Boat Ferry, Rail Ferry, Special Traffic Figure, Indescribable, Maneuver

Normalized Attribute Array Format

Each entry in the normalized attribute array has the following format.

Field (Length)	Description												
Display Characteristics:	(17 bits)(Bits 0-16, where 0 is the high order bit) Type of segment or polyline												
	<table><tr><th><u>Value</u></th><th><u>Description</u></th></tr><tr><td>000000000000000001</td><td>Paved</td></tr><tr><td>000000000000000010</td><td>Private</td></tr><tr><td>0000000000000000100</td><td>Frontage</td></tr><tr><td>00000000000000001000</td><td>Bridge</td></tr><tr><td>000000000000000010000</td><td>Tunnel</td></tr></table>	<u>Value</u>	<u>Description</u>	000000000000000001	Paved	000000000000000010	Private	0000000000000000100	Frontage	00000000000000001000	Bridge	000000000000000010000	Tunnel
<u>Value</u>	<u>Description</u>												
000000000000000001	Paved												
000000000000000010	Private												
0000000000000000100	Frontage												
00000000000000001000	Bridge												
000000000000000010000	Tunnel												

00000000000100000	Ramp
00000000001000000	Tollway
00000000010000000	POI access
00000000100000000	Controlled access
00000001000000000	Roundabout
00000010000000000	Intersection internal
00000100000000000	Undefined traffic area internal
00001000000000000	Boat ferry
00010000000000000	Rail ferry
00100000000000000	Special traffic figure
01000000000000000	Indescribable
10000000000000000	Maneuver

Access Characteristics: (10 bits)(Bits 17-26) Allowable types of vehicle

<u>Value</u>	<u>Description</u>
0000000001	Automobiles
0000000010	Buses
0000000100	Taxis
0000001000	Carpool
0000010000	Pedestrians
0000100000	Bicycles
0001000000	Trucks
0010000000	Through traffic
0100000000	Deliveries
1000000000	Emergency vehicles

Lane Category: (2 bits)(Bits 27-28) Number of lanes.

<u>Value</u>	<u>Description</u>
000	Not Applicable
001	1 Lane
010	2-3 Lane
011	4 or More Lanes

Speed Category: (3 bits) (Bits 29-31) Speed limit range.

<u>Value</u>	<u>Description</u>
000	131 - 255 miles
001	101 - 130 miles
010	91 - 100 miles
011	71 - 90 miles
100	51 - 70 miles
101	31 - 50 miles

110	11 - 30 miles
111	0 - 10 miles

2.1.1.4 Global Media Index Parcel

The Global Media Index Parcel contains arrays of fixed size external index structures for each unique global data type found in the SDAL Media File. In this context, external means each index element resolves to a particular data parcel ID. This parcel ID pertains to the set of variable-length data tables, reference data, and non-trademarked icons stored in parcels following the Global Media Index parcel. The Global Media Index Parcel also contains compression data and the per-continent locale table, which are discussed in the following section.

Record data tables, reference data, and trademarked icons are segregated into separate data parcels and have unique characteristics requiring distinct index data structures. An array of index elements is required since the complete set of any particular data classification can span multiple parcels. This is true for two reasons:

- Any particular data set can be larger than the largest possible parcel size. However, most of the reference data uses a 16-bit index for future expansion, which has the potential for reference data sets containing up to 64K records.
- Multiple sets of the same data classification are packed into the minimum number of data parcels to increase space utilization. Additional reference data should be placed in a parcel until it is optimally full. For example, Country Information can also be placed in a parcel without requiring the entire Country Information data set to fit in the remaining space. In this case, the Country Information is split between the parcel in question and its immediate successor. Such a split occurs at record boundaries; it is not possible for records to span parcels.

In general, each element in an external index array locating the particular subset of data is found in a particular data parcel.

Metadata Index

The database contains one or more “record data levels” and a list of Access Library releases which correspond to a given “record data level” present in the database. This data level provides a degree of isolation between the database release and the Access Library release - there are multiple occurrences of identical record data levels for different Access Library releases on a single database. Thus, record data level 1 corresponds to the earliest Access Library release and is incremented when SDAL PSF format and Access Library accumulate enough changes to warrant a new data level. This process does not have to correspond one-to-one to new PSF or Access Library releases. Thus, multiple new Access Library releases may occur before a new record data level comes into effect.

This design results in two arrays in the global media portion of a database: one which lists Access Library versions and specifies the matching record data level to be used with it, and a table which relates the record data level to the parcel locations for various modules on the media. The structure defining the first array is:

```
typedef struct {
    Ushort_tusSdalLibraryVer; /* Major + Minor version */
    Ushort_tusMetadataLevel; /* Unique metadata level id */
} MetadataLevel_t;
```

The structure defining the second array is:

```
typedef struct {
    Ushort_t      usMetadataLevel; /* Unique metadata level id */
    Ushort_t      usModuleId;      /* Module within level. Ranges 1 -
                                   64K, but not 0 */
    Ushort_t      usModuleLength; /* Module length */
    Ushort_t      usFlags;        /* Flags */
}
```

```

ParcelID_t      moduleParcel1; /* Parcel location in reference
                                data */
ParcelID_t      moduleParcel2; /* Parcel location elsewhere in the
                                database*/
} MetadataModule_t;

```

The MetadataLevel_t and MetadataModule_t arrays are accessed using the usMetadataModuleLevelOffset and usMetadataModuleTableOffset fields in Global Media Header. The number of elements in these arrays is contained in the usMetadataModuleLevelCount and usMetadataModuleTableCount fields in the Global Media Header. The record data index layout is illustrated in: Global Media Metadata Structure.

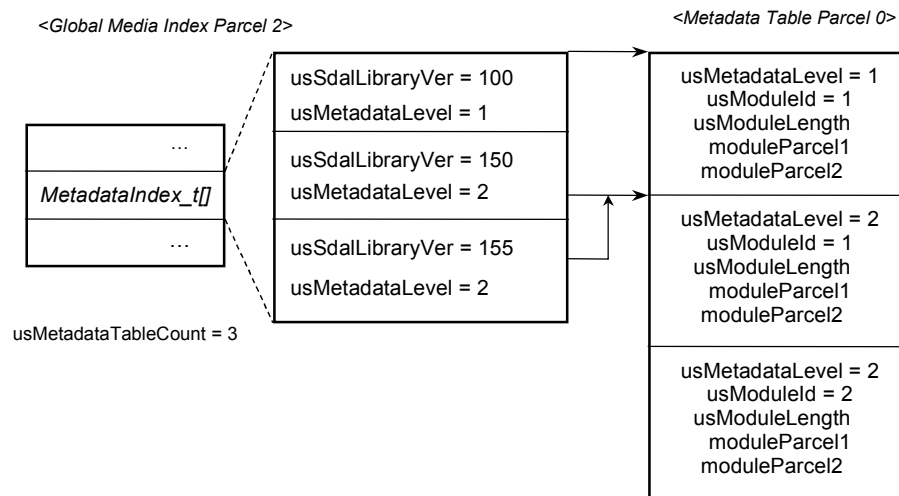


Figure 2-3: Global Media Metadata Structure

Multiple versions of Access Library can share the same record data if they specify the same value of the *usMetadataLevel* field. For each MetadataLevel, its corresponding MetadataModule_t records are differentiated by their ModuleId field.

Each data object code module is stored in a unique parcel with a PclHdr_t parcel header. These modules are stored with the reference data parcels. The *moduleParcel1* field is the parcel ID where this module is stored within Global Media Data. The *usModuleLength* field is the length of actual metadata information in each of these parcels.

Reference Data Indexes

Reference data indexes are stored in RefDataIndex_t structures in the Global Media Index parcel. This data structure indicates the parcel where a subset of a particular type of reference data can be found in the *ParcelID* field. The subset interval in record index order is described by the *usLastIndex* field, which contains the record index of the end of the subset interval stored in the parcel. The initial index of the interval described by the first RefDataIndex_t index array element is implicitly zero. When the entire reference data set can be stored in a single parcel, only one index array element is required and the *usLastIndex* field can be ignored. This is illustrated in: Single Parcel Reference Data Structure.

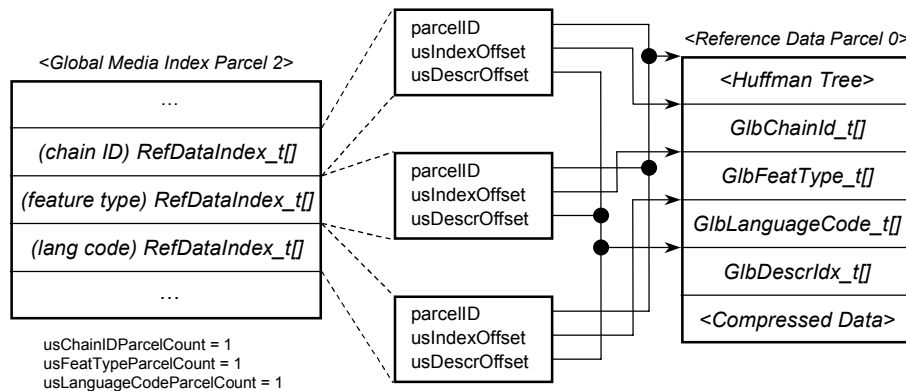


Figure 2-4: Single Parcel Reference Data Structure

The RefDataIndex_t index array element also contains offsets to the fixed portion of the reference data section and the language-code-to-description-string array. These offsets are stored in us...IndexOffset fields in the Global Media Header.

The number of elements in the RefDataIndex_t array is contained in the us...ParcelCount field in the Global Media Header. When the reference data set spans multiple parcels, the usLastIndex field of the all but the last index array element is less than the us...MaxIndex field copied from the Global Media Header into global memory. This value indicates the last valid reference data record index and is used at run time to validate the record index of a query. In the multiple parcel case, the next subset interval can be obtained from the next array element, with an implicit starting index formed by incrementing the usLastIndex field from the previous index array element by one. This case is illustrated in Figure 2-5: Multiple Parcel Reference Data Structure for Facility Subtype reference data.

```
typedef struct
{
    ParcelID_t    parcelID;           /* where reference index and data
                                     located */
    Ushort_t      usIndexOffset;      /* from beginning of parcel */
    Ushort_t      usDescrOffset;      /* from beginning of parcel */
    Ushort_t      usLastIndex;        /* index of last record in this par-
                                     cel */
    Uchar_t       ucReserved[2];
}
RefDataIndex_t; /* 12 bytes */
```

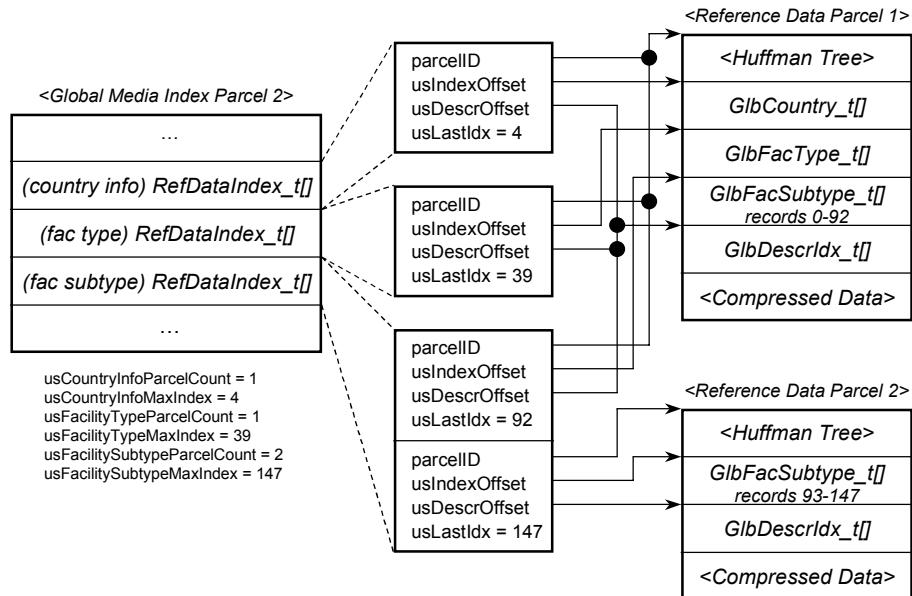


Figure 2-5: Multiple Parcel Reference Data Structure

POI and Highway Shield Icon Indexes

Indexes for POI and highway shield icons are also stored in the Global Media Index parcel. Unlike other reference data, a record exists for every type of icon, not for each parcel. The icon index records are implicitly ordered by the index used to access them. The structure of the icon index record is:

```
typedef struct
{
    Ushort_t      usParcelID32Idx;
    Ushort_t      usParcelID16Idx;
    Ushort_t      usIconOffset32; /* from start of parcel */
    Ushort_t      usIconOffset16; /* from start of parcel */
} GlbIconHdr_t;
```

The ParcelID fields are relative indexes into the reference data parcels, originating from the *firstIconParcelID* field in the global media header. There is no length field, as it can be computed from information stored in the icon record itself. The parcels containing the actual Icon data are described in a later section of this document.

Compression Information

While not strictly an index type, compression information metadata is also located in the Global Media Index Parcel. Compression information is a metadata table representing which entities in this particular PSF are compressed and what type of compression algorithm is used. For Huffman compression, additional information is required. One Huffman-specific parameter is whether compression for the specified entities is performed at the record, block/subparcel, or parcel extent. All other compression types compress entire parcels only. The location of the Huffman decompression tree is also required in the compression information. An exception to this are the per-parcel Huffman decompression trees, which are located separately in the Region Global Data section.

The compression metadata table takes the form of an array of fixed-size `CompressInfo_t` data structures (see sample code below).

A bitmap is used for the entity type. This is required for per-parcel compression where multiple entity types are resident in a single parcel type. It is also required when Huffman trees are used across multiple entity types.

```
#define HUFF_COMPRESS_TYPE 0
#define FAST_HUFF_COMPRESS_TYPE 1
#define TWICE_HUFF_COMPRESS_TYPE 2
#define LZ_HUFF_COMPRESS_TYPE 3
#define LZ_FAST_HUFF_COMPRESS_TYPE 4
#define LZ_TWICE_HUFF_COMPRESS_TYPE 5
#define LZ_COMPRESS_TYPE 6
#define NO_COMPRESSION 7

#define PARCEL_COMPRESS_EXTENT 0
#define BLOCK_COMPRESS_EXTENT 1
#define RECORD_COMPRESS_EXTENT 2

#define HUFF_TREE_PER_MEDIA 0
#define HUFF_TREE_PER_REGION 1
#define HUFF_TREE_PER_PARCEL 2

typedef struct
{
    Ulong_t    ulReserved[1];
    Uchar_t    ucCompressType; /* Huffman or LZ */
    Uchar_t    ucHuffTreeLocation; /* media, region, or per-parcel */
    Uchar_t    ucHuffCompressExtent; /* parcel, block/subparcel, or
        record */
    Uchar_t    ucReserved[1];
} CompressInfo_t; /* 8 bytes */
```

Locale Table

Locale information is also stored in the Global Media Index Parcel. Because the table is effective for character collation on a continent-wide basis, it is kept as per-region information.

The locale table is an array used for sorting names (of places, navigable features, etc.) .The locale table structure `LocaleTable_t` is:

```
typedef struct {
    Ushort_t    TableSize; /* Size of table in bytes */
    Ushort_t    CharArray[256]; /* Conversion array */
    Uchar_t     Strings[1]; /* Beginning of extra string data */
} LocaleTable_t;
```

The *usLocaleTableOffset* field of the Global Media Header structure `GlbMediaHeader_t` contains the offset to the locale table in the Global Media Index parcel the *usLocaleTableLen* field of the `GlbMediaHeader_t` structure contains the length of the locale table and must equal the *TableSize* field of the `LocaleTable_t` structure.

The locale table implements a mapping from the original unsigned character set {0,..., 255} to the set consisting of the union of {0, ..., 32767} and {null-terminated unsigned character strings}. 0 must map to 0. Each original unsigned character is an index into `CharArray[ ]`. The value of `CharArray[ ]` is either an element of {0,...,32767}, or an offset to a null-terminated unsigned character string as described below:

In the case of ligatures, these map to a string of variable length. In this case, the corresponding locale table entry contains a two's complement value, the negative of the offset from the beginning of the locale table to the null-terminated character string corresponding this ligature.

The locale table determines the sort order of names in the database as follows: An array of unsigned integers is formed from the name, so each original unsigned character {0,...,255} of the name maps in sequence to one (respectively, more than one) unsigned integers of the array. If the value of CharArray[] element indexed by the original unsigned character of the name is ≤ 32767 then there is one corresponding unsigned integer, equal to the value in CharArray[]. If the value of CharArray[] element indexed by the original unsigned character of the name is > 32767 , there is more than one corresponding unsigned integer, and these unsigned integers are the non-zero values in the corresponding Strings[] string. Two names are compared by examining the corresponding arrays of unsigned integers one at a time starting with the first array element.

Consider, for example, a locale table containing the mappings:

'0' -> '0'

'1' -> '1'

...

'9' -> '9'

'A' -> 'A'

'B' -> 'B'

...

'Z' -> 'Z'

'A-accent' -> 'A'

'ligature AE' -> extra string "AE"

In the example locale table mapping would result in 'A-accent' sorting equal to 'A', and 'ligature AE' sorting equal to the string "AE".

2.2.2 SDAL Media File Reference Data Parcels

Reference data is stored in one or more parcels in the SDAL Media file. The following Reference Data Parcel types are supported:

- Chain ID
- Feature
- Facility
- Attribute Value
- Language Code
- Country Information
- Time Zone
- Daylight Savings Time (DST)

Regardless of type, reference data relates a compact index stored with the map data entities to a DNDC attribute value associated with a multiple-language set of variable-length strings describing the attribute value.

2.2.2.1 Technical Description

Figure 2-6: Reference Parcel Layout illustrates the detailed structure of a Reference Data Parcel. It begins with the parcel header (containing the Huffman compression tree). The figure also shows three different types of reference data being stored in a single parcel, each sharing a common pool of Huffman-compressed description string data and Huffman decompression tree.

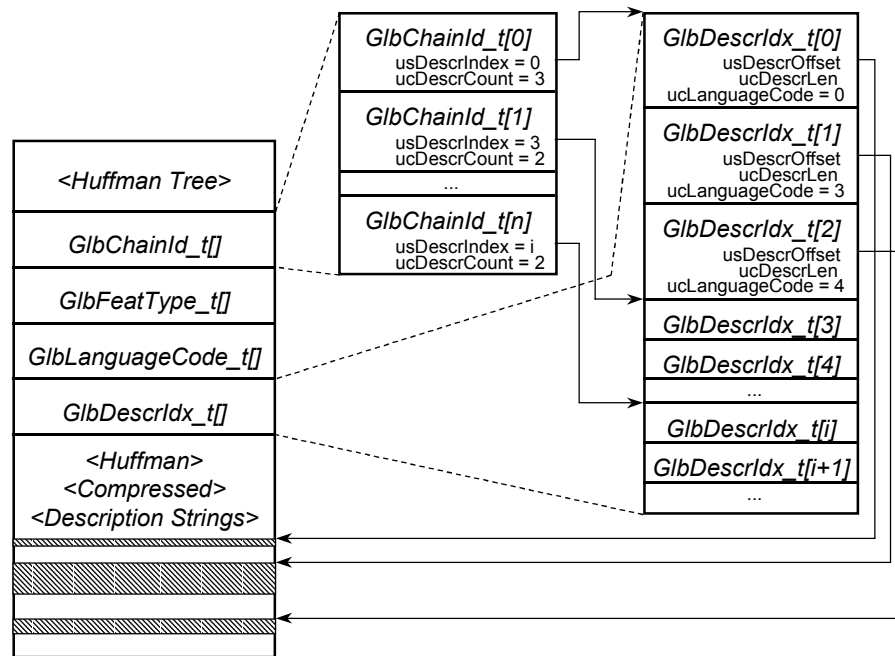


Figure 2-6: Reference Parcel Layout

2.2.2.2 Reference Data Parcel Header

Reference data employs Huffman compression of character string or bitmap data. The compression tree is stored in each parcel in the following parcel header:

```
typedef struct
{
    PclHdr_t      pclHdr;
    SmallHuffTree_t huffmanTree;
}
RefPclHdr_t;
```

Each set of reference data is structured as an array of fixed-length data structures, each of which point to an interval of fixed length GlbDescrIdx_t data structures, which identify the language codes of the referenced description strings.

The *ucDescrOffset* field of the GlbDescrIdx_t is a byte offset from the beginning of the parcel to a particular packed description string. The description strings are stored in a compressed locale-dependent format using the ISO8859-1 character set for North American and European locales. For this reason, a compressed string length field, *ucDescrLen*, is maintained to limit the decompression scope to the desired string only. A Huffman compression scheme allows effective compression of small (≤ 80 character) strings. The *ucLanguageCode* field indicates the language of the particular description string.

```
typedef struct
{
    Ushort_t      usDescrOffset; /* to packed description string in this
                                parcel */
    Uchar_t       ucDescrLen;    /* length of packed description string */
    Uchar_t       ucLanguageCode;
}
GlbDescrIdx_t; /* 4 bytes */
```

2.2.2.3 Reference Data Parcel Types

Reference data is stored in one or more parcels in the SDAL Media file. The following Reference Data Parcel types are supported:

- Chain ID: Names of chains (e.g., Denny's, Marshall Field's, Kroger's).
- Feature: Physical features (e.g., river, lake, landmark).
- Facility: Type of facility (e.g., restaurant, department store, grocery store).
- Attribute Value: Facility subtype, (e.g., cuisine type).
- Chain ID list for each Facility type and Facility subtype list for each Facility type.
- Language Code
- Country Information: Country specific information for all countries associated with the region.
- Time Zone
- Daylight Savings Time (DST): Indicates whether Daylight Savings Time is applicable to the Time Zone information.

Chain ID Section

Chain ID reference data is an array of fixed size structures indexed by the internal SDAL chain ID value.

An interval (index, count) of contiguous fixed-length GlbDescrIdx_t records used to indirectly obtain the variable-length compressed description strings is also stored in the fixed data portion of the Chain ID reference data.

The Chain ID is fixed in length and a C structure can be imposed:

```
/* this define goes into the ucFlags field of GlbChainID_t */
#define CHAIN_ID_IN_THIS_DATABASE      0x02
typedef struct
{
    Ushort_t      usDNDCChain; /* DNDC chain ID value */
    Ushort_t      usDescrIndex; /* index in GlbDescrIdx_t array */
    Uchar_t       ucFlags;
    Uchar_t       ucDescrCount; /* number of array (languages) for this
                                value */
}
GlbChainID_t;
```

Feature Type Section

Feature Type reference data is an array of fixed size structures indexed by the internal SDAL feature type value.

An interval (index, count) of contiguous fixed-length GlbDescrIdx_t records used to indirectly obtain the variable-length compressed description strings is also stored in the fixed data portion of the Feature Type reference data.

The Feature Type is fixed in length and a C structure can be imposed:

```
typedef struct
{
    Ulong_t       ulDNDCFeature; /* DNDC feature type value */
    Ulong_t       ulSimilarDNDCFeature;
    Ushort_t      usDescrIndex; /* index in GlbDescrIdx_t array */
    Uchar_t       ucDescrCount; /* number of array (languages) for this
                                value */
}
GlbFeatureType_t;
```

Facility Type Section

Facility Type reference data is an array of fixed size structures indexed by the internal SDAL facility type value.

An interval (index, count) of contiguous fixed-length GlbDescrIdx_t records used to indirectly obtain the variable-length compressed description strings is also stored in the fixed data portion of the Facility Type reference data.

The Facility Type is fixed in length and a C structure can be imposed:

```
/* these defines go into the ucFlags field of GlbFacilityType_t */
#define FACTYPE_IN_CURRENT_RELEASE 0x01
#define FACTYPE_IN_THIS_DATABASE 0x02
typedef struct
{
    Ushort_t      usDNDCFacility; /* DNDC facility type value */
    Ushort_t      usFirstChild;
    Ushort_t      usLastChild;
    Ushort_t      usParent;
    Ushort_t      usDescrIndex; /* index in GlbDescrIdx_t array */
    Uchar_t       ucFlags;
    Uchar_t       ucDescrCount; /* number of array (languages) for this
                                value */
}
GlbFacilityType_t;
```

Attribute Value Section

Attribute Value reference data is an array of fixed size structures indexed by the simple sequential index.

An interval (index, count) of contiguous fixed-length GlbDescrIdx_t records used to indirectly obtain the variable-length compressed description strings is also stored in the fixed data portion of the Attribute Type reference data.

The Attribute Type *usAttrType* provides classification for the Attribute Value records in reference data. Currently, there is only one record describing Cuisine Type information.

The Attribute Value record is fixed in length and a C structure can be imposed:

```
typedef struct
{
    Ushort_t      usAttrType; /* SDAL internal value for this
                              type */
    Ushort_t      usDNDCAttrValCode; /* DNDC code of this value */
    Ushort_t      usDescrIndex; /* index in GlbDescrIdx_t array */
    Uchar_t       ucFlags;
    Uchar_t       ucDescrCount; /* number of arrays (languages) for
                              this Value */
}
GlbAttrValue_t;
```

Chain IDs and Subtypes for Facility Type Section

ChainIDs in Facility Type data and Subtypes in Facility Type data share the same structure. They are both arrays of fixed size structures indexed by simple sequential index. The records are fixed in length and a C structure can be imposed:

```
typedef struct
{
    Ushort_t      usFacilityType;
    Ushort_t      usChainCountForFac;
    Ushort_t      usDescrIndex; /* index in GlbDescrIdx_t array */
}
```

```

Uchar_t      ucDescrCount;    /* count in GlbDescrIdx_t array */
} GlbChainsForFacility_t;

typedef struct
{
Ushort_t      usFacilityType;
Ushort_t      usSubTypeCountForFac;
Ushort_t      usDescrIndex;    /* index in GlbDescrIdx_t array */
Uchar_t      ucDescrCount;    /* count in GlbDescrIdx_t array */
} GlbSubTypesForFacility_t;

```

An interval (index, count) of contiguous fixed-length GlbDescrIdx_t records used to indirectly obtain the variable-length table of Chain or Subtype IDs is also stored in the fixed data portion of the ChainID or Subtypes for Facility Type reference data. The data referred to by the *usDescrOffset* field of each GlbDescrIdx_t record is a table of 2-byte numeric entries that represent indexes into the Chain ID data for Chain IDs or Attribute Value data for Subtypes. Their count is given by *usChainCountForFac* or *usSubTypeCountForFac* fields. The *ucDescrLen* field of each GlbDescrIdx_t record is a count of these bytes used by these 2-byte records. The *ucLanguageCode* field should be ignored.

Language Code Section

Language Code reference data is an array of fixed size structures indexed by the internal SDAL language code value.

An interval (index, count) of contiguous fixed-length GlbDescrIdx_t records used to indirectly obtain the variable-length compressed description strings is also stored in the fixed data portion of the Language Code reference data.

The Language Code is fixed in length and a C structure can be imposed:

```

#define MARC_CODE_LEN3

typedef struct
{
Ushort_t      usDescrIndex;    /* index in DescrIndex_t array */
Uchar_t      ucDescrCount;    /* number of array elements (languages)
    for this value*/
Uchar_t      ucaMarcCode[ MARC_CODE_LEN];
}
GlbLanguageCode_t;

```

Affix Section

Affix data is a list of commonly used prefixes and suffixes. Affix reference data is an array of fixed size structures indexed by the internal SDAL affix value.

An interval (index, count) of contiguous fixed-length GlbDescrIdx_t records used to indirectly obtain the variable-length compressed description strings is also stored in the fixed data portion of the Language Code reference data.

The Language Code is fixed in length and a C structure can be imposed:

```

#define AFFIX_INSTANCE_EXISTS    0x80
typedef struct
{
Ushort_t usDescrIndex; /* index in GlbDescrIdx_t array */
Uchar_t ucDescrCount; /* number of GlbDescrIdx_t entries */
Uchar_t ucFlags;
}
GlbAffix_t;

```

Country Information Section

Country Information reference data is an array of fixed size structures indexed by the internal SDAL country code value.

Intervals (index, count) of contiguous fixed-length GlbDescrIdx_t records used to indirectly obtain the variable-length compressed description strings for country name as well as the name of each administrative level is also stored in the fixed data portion of the Country Information reference data.

The Country Information is fixed in length and a C structure can be imposed:

```
#define    MAX_SPEED_CAT      8
#define    SPEED_CAT_LEN     2  /* 8 bit min. followed by 8 bit max. */

typedef struct
{
    /*
     ** Country name followed by admin level description string
     ** index and language count values
     */
    Ulong_t      ulCountryDescrIndex;
    Ulong_t      ulaAdminDescrIndexes[ DB_MAX_ADMIN_LVL];
    Uchar_t      ucCountryDescrCount;
    Uchar_t      ucaAdminDescrCounts[ DB_MAX_ADMIN_LVL];
    EntityID_t   placeID;          /* 24-bit parcel ID offset por-
                                   tion */
    Ushort_t     usPhoneCountryCode; /* Phone number country code */
    Uchar_t      ucNumAdminLevels;   /* Number of admin levels */
    Uchar_t      ucSignificantLevels; /* Admin levels significant for a
                                   place */
    Uchar_t      ucMinMunicipalLevel; /* highest level considered
                                   municipal */
    Uchar_t      ucNationalLangCode; /* National language code */
    Uchar_t      ucDrivingSide;      /* Driving side */
    Uchar_t      ucMeasurementSystem; /* Measurement system */
    Uchar_t      ucHouseNumberPos;   /* House number position in
                                   street address*/
    Uchar_t      ucCurrencyType;     /* Type of currency */
    Uchar_t      ucCurrencyPrecision; /* Decimal precision of monetary
                                   amts */
    Uchar_t      ucaSpeedCat[ SPEED_CAT_LEN * MAX_SPEED_CAT];
}
GlbCountry_t;
```

Time Zone Section

Time Zone reference data is an array of fixed size structures indexed by the internal SDAL time zone index value.

An interval (index, count) of contiguous fixed-length GlbDescrIdx_t records used to indirectly obtain the variable-length compressed description strings is also stored in the fixed data portion of the Time Zone reference data. The strings are the ASCII representation of the Time-Zone data.

The Time Zone is fixed in length and a C structure can be imposed:

```
typedef      struct
{
    Sshort_t    sNumMinutes; /* +/- minutes from GMT */
    Ushort_t    usDescrIndex; /* index in DescrIndex_t array */
    Uchar_t     ucDescrCount; /* number of array elements (languages)
                               for this value*/
}
```

```

    }
    GlbTimeZone_t;

```

Daylight Savings Time (DST) Section

DST reference data is an array of fixed size structures indexed by the internal SDAL DST index value.

An interval (index, count) of contiguous fixed-length GlbDescrIdx_t records can be used to indirectly obtain the variable-length compressed description strings stored in the fixed data portion of the DST reference data.

The DST format is fixed in length and a set of C structures can be imposed. The GlbDST_t structure is the most inclusive one stored in global reference data:

```

typedef struct
{
    Uchar_t      ucType;          /* Type of date          */
    Schar_t      scValue1;        /* First date value      */
    Schar_t      scValue2;        /* Second date value     */
    Sshort_t     sValue3;         /* Third date value      */
}
GlbDate_t;

typedef struct
{
    GlbDate_t     startDate;      /* Start date             */
    GlbDate_t     endDate;        /* End date               */
    Ushort_t      usStartTime;    /* Start time             */
    Ushort_t      usEndTime;      /* End time               */
    Uchar_t       ucTimeZone;     /* Time zone code         */
    Uchar_t       ucDST;          /* Daylight savings time code */
    Uchar_t       ucAttrs;        /* Date attributes        */
}
GlbDateTime_t;

typedef struct
{
    GlbDateTime_t DST;            /* DTM structure          */
    Ushort_t      usDescrIndex;   /* index in DescrIndex_t array */
    Uchar_t       ucDescrCount;   /* number of array (languages) for this
                                value */
}
GlbDST_t;

```

The GlbDate_t structure contains a date type and its associated values. The *ucType* field contains one of the following codes to indicate the type of date.

```

DB_DATE_NOT_APPLICABLE      /* Date not applicable*/
DB_DAY_OF_WEEK               /* Day of week bitmap*/
DB_ABSOLUTE_DATE            /* Absolute date */
DB_DAY_OF_YEAR               /* Day of year */
DB_DAY_OF_MONTH              /* Day of month */
DB_DAY_OF_WEEK_OF_MONTH     /* Day of week of month */
DB_DAY_OF_WEEK_OF_YEAR      /* Day of week of year */
DB_WEEK_OF_MONTH            /* Week of month */
DB_WEEK_OF_YEAR             /* Week of year */
DB_MONTH_OF_YEAR            /* Month of year */
DB_DAY_OF_MONTH_OF_YEAR     /* Day of month of year */
DB_DAY_OF_WEEK_OF_MONTH_OF_YEAR /* Day of month of year */

```

The following table indicates which values are used for each date type:

Type	scValue1	scValue2	scValue3
DB_DATE_NOT_APPLICABLE	n/a	n/a	n/a
DB_DAY_OF_WEEK	Day Bitmap	n/a	n/a
DB_ABSOLUTE_DATE	Month of Year	Day of Month	Year
DB_DAY_OF_YEAR	n/a	n/a	Day of Year
DB_DAY_OF_MONTH	Day of Month	n/a	n/a
DB_DAY_OF_WEEK_OF_MONTH	Day Bitmap	Week of Month	n/a
DB_DAY_OF_WEEK_OF_YEAR	Day Bitmap	Week of Year	n/a
DB_WEEK_OF_MONTH	Week of Month	n/a	n/a
DB_WEEK_OF_YEAR	Week of Year	n/a	n/a
DB_MONTH_OF_YEAR	Month of Year	n/a	n/a
DB_DAY_OF_MONTH_OF_YEAR	Day of Month	Month of Year	n/a
DB_DAY_OF_WEEK_OF_MONTH_OF_YEAR	Day Bitmap	Week of Month	Month of Year

The following table lists the domain for each date value:

Value	Domain
Day Bitmap	See day of week bit flags defined below.
Day of Month	-31 to 31 (excluding 0)
Day of Year	-366 to 366 (excluding 0)
Week of Month	-5 to 5 (excluding 0)
Week of Year	-53 to 53 (excluding 0)
Month of Year	-12 to 12 (excluding 0)
Year	1 to 9999 (century and year)

Note: Negative values indicate a date value relative to the end of a particular timeframe (e.g., end of month). The actual value is computed as follows:

end of timeframe + value + 1

The following bit flags are used for the day of week bitmap.

DB_SUNDAY	/* Sunday	*/
DB_MONDAY	/* Monday	*/

```

DB_TUESDAY      /* Tuesday      */
DB_WEDNESDAY    /* Wednesday    */
DB_THURSDAY     /* Thursday     */
DB_FRIDAY       /* Friday       */
DB_SATURDAY     /* Saturday     */

```

The *startDate* and *endDate* fields contain the starting and ending dates as defined by the *GlbDate_t* data structure.

The *usStartTime* and *usEndTime* fields contains the starting and ending times in 24-hour format (hhmm). These times are expressed in terms of local time, not Greenwich Mean Time (GMT).

The *ucTimeZone* and *ucDST* fields can be used to obtain the information required to compute local time from GMT.

The *ucTimeZone* field contains a code defining the time zone.

The *ucDST* field contains a code defining when daylight savings time is in effect.

The following constants define the bit flags for the *ucAttrs* field.

```

DB_EXCLUDE_DATE  Exclude the date range specified by startDate and
                  endDate
DB_DAWN_TO_DUSK   Time period is from dawn to dusk
DB_DUSK_TO_DAWN   Time period is from dusk to dawn

```

2.2.3 Icon Data Parcels

The Non-Trademarked Icon Data Parcels (Icon Data Parcels) are a distinct data type comprising the SDAL Media File. They contain the various POI and highway shield icons used throughout the database.

2.2.3.1 Technical Description

Icon data consists of an array of fixed size structures indexed by the values in *GlbIconHdr_t* records in Global Media Index Parcel. Icon data is stored directly in one or more Icon Data Parcels.

2.2.3.2 Icon Data Parcel Structure

The icon data is represented in an efficient format. Specifically, a packed format is used in the PSF, represented by one of the following C structures. The first C structure is used for 16 pixel by 16 pixel icons, and the second C structure is used for 32 pixel by 32 pixel icons.

```

/* two global media Non-Trademark icon record definitions */
/* 16 by 16 pixel icon */
typedef struct
{
    Uchar_t ucBitsPerPixel;
    Schar_t scTransparColorIdx;          /* which color is transparent, or -1 */
    union
    {
        struct
        {
            Uchar_t ucPixels[ (16 * 16 * 1) / 8]; /* 1 bit-per-pixel */
            Uchar_t ucaColors[2];
        } oneBit;
        struct
        {
            Uchar_t ucPixels[ (16 * 16 * 2) / 8]; /* 2 bits-per-pixel */
            Uchar_t ucaColors[4];
        }
    }
}

```

```

    } twoBit;
    struct
    {
        Uchar_t ucPixels[ (16 * 16 * 3) / 8]; /* 3 bits-per-pixel */
        Uchar_t ucaColors[8];
    } threeBit;
    struct
    {
        Uchar_t ucPixels[ (16 * 16 * 4) / 8]; /* 4 bits-per-pixel */
        Uchar_t ucaColors[16];
    } fourBit;
    struct
    {
        Uchar_t ucPixels[ (16 * 16 * 4) / 8]; /* 4 bits-per-pixel */
        Uchar_t ucTranspar[ (16 * 16 * 1) / 8]; /* 1 bit transparency */
    } fiveBit;
    } iconPixels;
    }
    DBPSFIcon16_t;

/* 32 by 32 pixel icon */
typedef struct
{
    Uchar_t ucBitsPerPixel;
    Schar_t scTransparColorIdx; /* which color is transparent, or
        -1 */
    union
    {
        struct
        {
            Uchar_t ucPixels[ (32 * 32 * 1) / 8]; /* 1 bit-per-pixel */
            Uchar_t ucaColors[2];
        } oneBit;
        struct
        {
            Uchar_t ucPixels[ (32 * 32 * 2) / 8]; /* 2 bits-per-pixel */
            Uchar_t ucaColors[4];
        } twoBit;
        struct
        {
            Uchar_t ucPixels[ (32 * 32 * 3) / 8]; /* 3 bits-per-pixel */
            Uchar_t ucaColors[8];
        } threeBit;
        struct
        {
            Uchar_t ucPixels[ (32 * 32 * 4) / 8]; /* 4 bits-per-pixel */
            Uchar_t ucaColors[16];
        } fourBit;
        struct
        {
            Uchar_t ucPixels[ (32 * 32 * 4) / 8]; /* 4 bits-per-pixel */
            Uchar_t ucTranspar[ (32 * 32 * 1) / 8]; /* 1 bit transparency */
        } fiveBit;
    } iconPixels;
}
DBPSFIcon32_t;

```

The one-bit through four-bit types contain an array of color values, *ucaColors*, which correspond to the pixel values. Any of these color values can be transparent when the *scTransparColorIdx* field is set to a valid index in the color value array. A -1 value for this field indicates no pixels are transparent.

Unlike other reference data, a record exists for every type of icon, not for each parcel. The order of actual icon records in the reference data parcels does not have to match the order of these icon index records because icon records are sorted in the order of most frequent use in Cartographic data.

All media data parcels containing icon records do not contain any other reference data and have the following header.

```
typedef struct
{
    /* common to all parcel types */
    ParcelID_t   parcelid;
    Ushort_t     usParcelDesc;
    Uchar_t      ucParcelType;
    RegionID_t    region;
    Bool_t        bEndianSwap;          /* set when header endian converted
                                         */
    Uchar_t      ucCmpDataSizeHi;       /* compressed data size in bits,
                                         byte 1 */
    Ushort_t     usCmpDataSizeLo;       /* compressed data size in bits,
                                         byte 2&3 */
    Ushort_t     usCompressType;         /* type of parcel compression */
    Ushort_t     usCmpData;             /* byte offset to compressed data
                                         */
    Ushort_t     usCmpDataUncompSize;   /* size of uncompressed parcel in
                                         bytes */
    Ushort_t     usExtensionOffset;     /* If non-zero: Offset to extension
                                         of header*/
}
PclHdr_t;
```

2.2.4 Compression Information

Compression information metadata, located in the Global Media Index Parcel, is a metadata table representing which entities in this particular PSF are compressed and the type of compression algorithm used.

When Huffman compression is used, additional information is required. For example:

- One Huffman-specific parameter must indicate whether compression for the specified entities is performed at the record, block/subparcel, or parcel extent. All other compression types compress entire parcels only.
- The location of the Huffman decompression tree is also required in the compression information.

2.2.4.1 Technical Description

The compression metadata table takes the form of an array of fixed-size *CompressInfo_t* data structures (see sample code below).

A bitmap is used for the entity type. This is required for per-parcel compression where multiple entity types are resident in a single parcel type. It is also required when Huffman trees are used across multiple entity types.

Per-media and per-region Huffman trees are loaded into the in-memory global data structure at media and region initialization time (respectively) from the SDAL Media File or SDAL Map File. As a

result, the records in the compressed information array are sorted by compression type and Huffman tree location so per-media Huffman information precedes per-region Huffman information, which precedes all other types. This allows the appropriate amount of memory to be allocated to hold the compression metadata table and Huffman trees in global data space. It also allows the compression information array index to access the corresponding Huffman tree.

Sample

```
#define HUFF_COMPRESS_TYPE 0
#define FAST_HUFF_COMPRESS_TYPE 1
#define TWICE_HUFF_COMPRESS_TYPE 2
#define LZ_HUFF_COMPRESS_TYPE 3
#define LZ_FAST_HUFF_COMPRESS_TYPE 4
#define LZ_TWICE_HUFF_COMPRESS_TYPE 5
#define LZ_COMPRESS_TYPE 6
#define NO_COMPRESSION 7

#define PARCEL_COMPRESS_EXTENT 0
#define BLOCK_COMPRESS_EXTENT 1
#define RECORD_COMPRESS_EXTENT 2

#define HUFF_TREE_PER_MEDIA 0
#define HUFF_TREE_PER_REGION 1
#define HUFF_TREE_PER_PARCEL 2

typedef struct
{
    Ulong_t    ulReserved[1];
    Uchar_t    ucCompressType;          /* Huffman or LZ */
    Uchar_t    ucHuffTreeLocation;      /* media, region, or per-parcel */
    Uchar_t    ucHuffCompressExtent;    /* parcel, block/subparcel, or
                                         record */
    Uchar_t    ucReserved[1];
} CompressInfo_t;                    /* 8 bytes */
```

2.3 SDAL Region File

The SDAL region file contains all the attributed geometric data for a specific geographic area, along with the necessary control and indexing information to access that data. The SDAL region file contains:

- Region Global Data
- Spatial Data
- Non-Spatial Data

A region file exists as a separate file on the CD-ROM. The filename of the first region is “1.SDL”. All data is organized into parcels. Data is accessed and indexed at the parcel level.

2.3.1 SDAL Region Global Data

The SDAL region global data is contained in the first two parcels of the region file, at media sector addresses zero and one of the file. The per-region global data located in the SDAL region file consists of:

- Index Root Parcel IDs
- Region Bounding Rectangle
- Region Bounding Polygon
- Locale-dependent Translation Tables
- Maximum Related Entity Counts
- Maximum Parcel Record Counts
- Per-region Huffman Compression Trees
- Parcel-level Huffman Compression Trees

2.3.2 Spatial Data

The SDAL region file contains spatial data accessed via the Global KD-Tree Spatial Index. The spatial data consists of:

- Route Calculation Data
- Cartographic Data and Cartographic Cross-reference data
- Maneuver Data
- Point of Interest Data

2.3.3 Non-Spatial Data

The SDAL region file contains non-spatial data accessed via b-tree Indexes. The non-spatial data consists of:

- Crossroads Data
- Junction Data
- Navigable Feature Data
- Place Data
- Postal Code Data

2.4 SDAL Region Global Data Structure

The SDAL region file contains region data and index parcels that constitute the majority of the file. The SDAL region file also contains the region global data, located in the first two parcels of the file. The region global data contains all of the region-specific global information for that particular SDAL region file.

2.4.1 Region Global Data Parcels

The region global data parcels are located at media sector addresses zero and one relative to the start of the SDAL region file. The length of the first region global data parcel is the sector size used for the PSF associated with the media type in use. It contains the fixed-size region header section. The length of the second parcel (in sectors) is specified in the region header section. The second region global data parcel contains any region-specific variable-length information that is required. The organization of the region global data parcels is illustrated below.

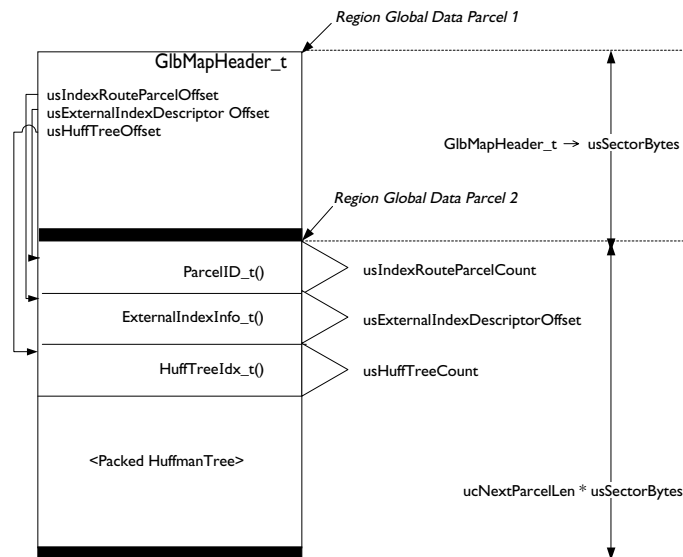


Figure 2-7: SDAL Region Global Data Layout

2.4.1.1 Region Header Section

The first parcel of the region global data contains the fixed size region header structure. The region header contains the offset and count information required to locate and process the variable-length data portions. These offsets are relative to the beginning of the second region global data parcel. Because the region header is fixed in size, a C structure can be imposed on it.

```

/*****
** Region file data structures
*****/

#define NUM_LAYER_ITEMS 20
#define PER_MEDIA_HUFF_TREE_CNT 3

typedef struct
{
/* replaces topLevelPlaceParcelId */
PlaceLevelIdx_t placeLevelIndex[DB_MAX_ADMIN_LVL];
ParcelID_t firstNFParcelId;

/* maximum parcel record count section */
Ushort_t usMaxNodeParcelRecs;
Ushort_t usMaxTypeAlphaPoiParcelRecs;

Ushort_t usMaxPlaceAlphaPoiParcelRecs;
Ushort_t usMaxTypeplaceAlphaPoiParcelRecs;
Ushort_t usMaxCrossroadParcelRecs;
Ushort_t usMaxPlaceParcelRecs;
Ushort_t usMaxJunctionParcelRecs;

Ushort_t usMaxAlphaNavFeatParcelRecs;
Ushort_t usMaxPlaceAlphaNavFeatParcelRecs;
Ushort_t usMaxMnvrParcelBlockRecs;
Ushort_t usMaxPoiParcelRecs;

/* maximum related entity count per region section */
Ushort_t usMaxManeuverPclsPerNavFeat;
Ushort_t usMaxPostalCodesPerNavFeat;
Ushort_t usMaxAliasesPerName; /* max. of POI/Place values
*/

Ushort_t usMaxPosPerPolyline;
Ushort_t usMaxPosPerPolygon;
Ushort_t usMaxTextPerPolyline;
Ushort_t usMaxTextPerPolygon;
Ushort_t usMaxSegsPerPolyline;
Ushort_t usMaxPolylinesPerSeg;

Ushort_t usMaxPoiPerSeg;
Ushort_t usMaxShapesPerSegment;
Ushort_t usMaxEntIDPIDXrefParcelRecs;
Ushort_t usMaxArmToArm; /* entries in supernode arm-
to-arm table */
Ushort_t usMaxArmToArmSegs; /* segs in supernode arm-to-
arm path */
Ushort_t usMaxAbbrSegPerSegment;

Uchar_t ucMaxSegsPerNode;
Uchar_t ucMaxChildNodes; /* supernode child nodes */
Uchar_t ucMaxChildSegs; /* supernode internal seg-
ments */

Uchar_t ucMaxSegsPerCond;
Uchar_t ucMaxFNsPerSegment;
Uchar_t ucMaxCondsPerSegment;
Uchar_t ucMaxDtTmPerCond;
Uchar_t ucMaxRouteIDsPerSignDest;
Uchar_t ucMaxTextLinesPerSignDest;

```

```

Uchar_t      ucMaxPlacesPerCrossroad;
Uchar_t      ucMaxPlacesPerJunction;
Uchar_t      ucReserved1[1];
Uchar_t      ucIndexRootParcelCount;
Uchar_t      ucReplicInfoCount;      /* located for best align-
                                     ment */
Uchar_t      ucHuffTreeCount;      /*for per-region huffman
                                     trees*/
Ushort_t     usRegionPolyPointCount; /* located for best align-
                                     ment */
DBRect_t     regionMBRect;          /* region MBR */
DBRect_t     regionKDRect;          /*rounded up to nearest tile
                                     bdy*/
ParcelID_t   idXrefParcelID;
Ushort_t     usMITTreeIndexOffset; /* in next parcel */
Ushort_t     usMITTreeCount;      /* of trees and index
                                     structs */
Ushort_t     usExternIndexDescriptorOffset; /* in next parcel */
Ushort_t     usExternIndexDescriptorCount; /* of index structs */
Ushort_t     usExternIndexStartNumber; /* same value as
                                     GLB_EXT_FIRST_NONSPATIAL_IDX
                                     */
Ushort_t     usRegionPolyPointOffset; /* in next parcel */
Ushort_t     usIndexRootParcelOffset; /* in next parcel */
Ushort_t     usReplicInfoOffset;    /* in next parcel */

Ushort_t     usLanBitMaskTableOffset; /* in next parcel */
Ushort_t     usLangBitMaskCount;    /*number of unique bitmasks
                                     */
Uchar_t      ucPlaceLangIndexWidth /*bytes in place lang. idx
                                     */
Uchar_t      ucPOILangIndexWidth;  /*bytes in POI lang.idx */

Ushort_t     usHuffTreeOffset;      /*to 1st per-region huff.
                                     tree*/
Uchar_t      ucNextParcelLen;      /*** sector length of second
                                     *** map header parcel */

Uchar_t      ucMaxRoutingLayer;
Uchar_t      ucMaxCartoLayer;
Uchar_t      ucDrivingSide;
Ushort_t     usaNTULayers[DB_MAX_CARTO_LAYERS];
Uchar_t      ucaRoutingLayerMap[NUM_LAYER_ITEMS];
Xchar_t      xcaDBName[51];
Xchar_t      xcaDBType[30];
Xchar_t      xcaGDFVersion[30];
Ulong_t      ulDBSerialNum;

DBRect_t     NonLHRegionMBR;        /* non-long haul MBR */

Ushort_t     usCountryNavOffset;
Ushort_t     usCountryNavCount;    /* number of countries in
                                     CountryNav */

Ushort_t     usNumNFPcls;
Ushort_t     usNumPPOIPcls;        /* number of data parcels
                                     for NF and POIs */

ParcelID_t   nfEntIDXrefParcelID;

}
GlbMapHeader_t;

```

2.4.1.2 Index Root Parcel Section

The Index Root Parcel Section is a variable-sized array of parcel IDs located in the second Region Header Parcel. These parcel IDs identify the parcel containing the root of each index tree. The Index Root Parcel IDs are stored in the order in which bits are set in the *ucaExtIndexBitmap* member of the Global Media Header. That is, each parcel ID corresponds with a set bit in the bit-map, as illustrated in the diagram below. The Index Root Parcel ID section is addressed directly from the Region Header using the *usIndexRootParcelOffset* field. The extent of this section is defined by the *ucIndexRootParcelCount* field. The format of the index information in these parcels is PSF-dependent as is the total number of indexes (and thus root parcel IDs).

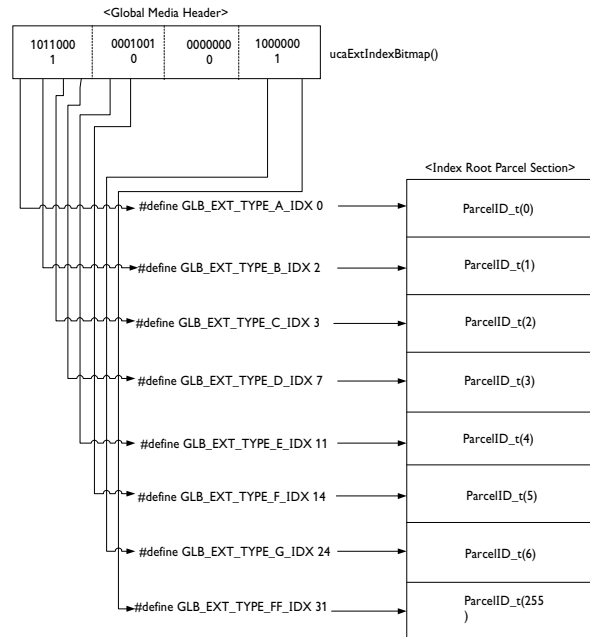


Figure 2-8 Index Bitmap and Root Parcel ID Array

The following list defines the position assignments for parcel ids in the array of root index parcels. These are in the 2nd sector of global map area. The first 8 are for spatial parcels, so they all point to the global kd-tree.

```
#define GLB_ROUTING_IDXID_POS 0
#define GLB_CARTO_IDXID_POS 1
#define GLB_CARTOXREF_IDXID_POS 2
#define GLB_MANEUVER_IDXID_POS 3
#define GLB_POI_IDXID_POS 4
#define GLB_ABBRSEG_IDXID_POS 5
#define GLB_CROSSROADS_IDXID_POS 6
/* number 8 is the first non-spatial */
#define GLB_NFSOUNDEX_IDXID_POS 8
#define GLB_NFALPHA_IDXID_POS 9
#define GLB_NFPLACE_IDXID_POS 10
#define GLB_PLACE_IDXID_POS 11
#define GLB_NFALPHAPLACE_IDXID_POS 12
#define GLB_POSTAL_IDXID_POS 13
#define GLB_PLACECROSSROADS_IDXID_POS 14
#define GLB_NFCROSSROADS_IDXID_POS 15
#define GLB_NFJUNCTIONS_IDXID_POS 16
```

```

#define GLB_POI_TYPEALPHA_IDXID_POS 17
#define GLB_POI_PLACEALPHA_IDXID_POS 18
#define GLB_POI_TYPEPLACEALPHA_IDXID_POS 19
#define GLB_POI_TYPESOUNDEX_IDXID_POS 20
#define GLB_POI_PLACESOUNDEX_IDXID_POS 21
#define GLB_POI_TYPEPLACESOUNDEX_IDXID_POS 22
#define GLB_JUNCTIONS_IDXID_POS 23
/* maintain LEVELn in consecutive sequence! */
#define GLB_LEVEL1_PLACE_ALPHA_IDXID_POS 24
#define GLB_LEVEL2_PLACE_ALPHA_IDXID_POS 25
#define GLB_LEVEL3_PLACE_ALPHA_IDXID_POS 26
#define GLB_LEVEL4_PLACE_ALPHA_IDXID_POS 27
#define GLB_LEVEL5_PLACE_ALPHA_IDXID_POS 28
#define GLB_LEVEL6_PLACE_ALPHA_IDXID_POS 29
#define GLB_LEVEL7_PLACE_ALPHA_IDXID_POS 30
#define GLB_HIGHWAY_IDXID_POS 32
/* municipal place index */
#define GLB_LEVELM_PLACE_ALPHA_IDXID_POS 33
#define GLB_EXT_NAV_FTR_NEXTCHARALPHA_DENSE_IDXID_POS 34
#define GLB_EXT_HLPLACE_IDXID_POS 35

```

2.4.1.3 External Index Records Section

The external index records section is a variable-sized array of records located in the second region header parcel, referenced by the `usExternalIndex` and `GlbMapHeader_t` fields. These are records of fixed size describing the format of the non-spatial B-tree external indexes in the database. They are stored following the index root parcel IDs and are also accessed, based on the settings of the Index Bitmap field. Further, these records are only used to describe non-spatial indexes, so only a subset of the `#define` list of `GLB_EXT_*` constants refer to records in this array. Index values 0 - 7 refer to all spatial parcels., with the first external index record referenced to by the eighth `GLB_EXT_*` constant. Each record is in the form of the `ExternalIndexInfo_t` structure.

```

typedef struct
{
    Uchar_t    ucSize;                /*index record size, total of com-
                                     ponent sizes */
    Uchar_t    ucKeySize;             /* key size */
    Uchar_t    ucRecordDBIDSize;      /* Record DBID size */
    Uchar_t    ucExtractionSize;      /* index extraction information
                                     size */
    Uchar_t    ucFilterSize;          /* filter size */
    Uchar_t    ucNameSize;            /* size of name portion of key */
    Uchar_t    ucRecordDBIDType;      /* type of record DBID */
    Uchar_t    ucNumFilterParts;      /* number of parts of filter */
    Ushort_t   usRecsPerPcl;          /* number of records per parcel */
    struct
    {
        Uchar_t ucFilterPartType;
        Uchar_t ucFilterPartSize;    /* size in bits */
        Uchar_t ucUnknownToLib;
    }
    aFilterPart[MAX_FILTER_PARTS];    /* could be variable length. */
    Uchar_t    ucReserved[2];         /* for alignment. Change if adding
                                     new fields */
}ExternalIndexInfo_t;
#define MAX_FILTER_PARTS 8
#define FILTERPART_LAST 7 /* last filter type */
#define EII_REJECT_UNKNOWN_NONZERO(0x1)
/*

```

```

** this bit of ucUnknownToLib indicates to the Access
** Library to reject records when ucFilterPartType is
** unknown to library (i.e. > FILTERPART_LAST compiled in
** library.
*/

```

This ExternalIndexInfo_t structure describes a dense index record with this format:

Key	DBID	Extraction	FilterParts
-----	------	------------	-------------

The first usNameSize bytes of the Key field are interpreted as a Name string of the specified length (used for alphabetical searches of an index). The Key field is also assumed to be the record used in the sparse B-tree over these dense index records. This removes the need for a second descriptor for the B-tree records over these index records.

The DBID field (if non-zero in length) contains a reference to a record in a data parcel and is in a form defined by the following #define list of constants:

```

#define RECORDDBID_0_0    0 /* no record DBID */
#define RECORDDBID_4_4    1 /* 4-byte PID, 4-byte EID */
#define RECORDDBID_4_3    2 /* 4-byte PID, 3-byte EID */
#define RECORDDBID_4_2    3 /* 4-byte PID, 2-byte EID */
#define RECORDDBID_4_0    4 /* 4-byte PID, 0-byte EID */
#define RECORDDBID_2_4    5 /* 2-byte (sequential) PID, 4-byte EID*/
#define RECORDDBID_2_3    6 /* 2-byte (sequential) PID, 3-byte EID*/
#define RECORDDBID_2_2    7 /* 2-byte (sequential) PID, 2-byte EID*/
#define RECORDDBID_2_0    8 /* 2-byte (sequential) PID, 0-byte EID*/
#define RECORDDBID_INDEX  9 /* PID is current parcel, EID based on
                           offset */
#define RECORDDBID_0_3    10 /* no PID, 3-byte EID */

```

The FilterPart of the index record is a byte aligned field containing bit aligned filter information, as described in the aFilterPart array. This array defines the order in which the filter bits can be found, their length, and the type of information in each filter. The following types of filter information are supported:

```

#define FILTERPART_LANGUAGE      0
#define FILTERPART_MULTIWORD    1
#define FILTERPART_DUPLICATE     2
#define FILTERPART_PLACEENTITY   3
#define FILTERPART_POSTALENTITY  4
#define FILTERPART_INTERSECTIONTYPE 5
#define FILTERPART_PRIMDIRONSIGN 6
#define FILTERPART_INTERSECTDIRONSIGN 7

```

2.4.1.4 Huffman Tree Section

The Huffman tree section is a fixed-sized array of HuffTreeIdx_t records located in the second parcel of the region global data. Each element in the array is used to define the location of a variable-length huffman tree (of type SmallHuffTree_t *) directly following the HuffTreeIdx_t array. The offset is from the beginning of the second parcel of the region global data.

These global Huffman trees define the Huffman decompression of region data and are associated with particular data entities as described in the compression metadata tables located in the CompressInfo_t array found in the global media index parcel. These trees appear in the same order as the contiguous set of CompressInfo_t elements whose ucHuffTreeLocation field has the value HUFF_TREE_PER_REGION, and correspond to those entries in a one-to-one fashion. This section can be reached directly from the region header using the usHuffTreeOffset field. The extent of this section is obtained from the ucHuffTreeCount field.

```

typedef struct
{
    Ushort_t      usOffset;
    Ushort_t      usLength;
}
HuffTreeIdx_t;

```

The diagram below illustrates how the `CompressInfo_t` array located in the global media index parcel is used to locate the corresponding per-media and per-region Huffman trees. The elements of the `CompressInfo_t` array are sorted in per-media order followed by per-region Huffman tree order.

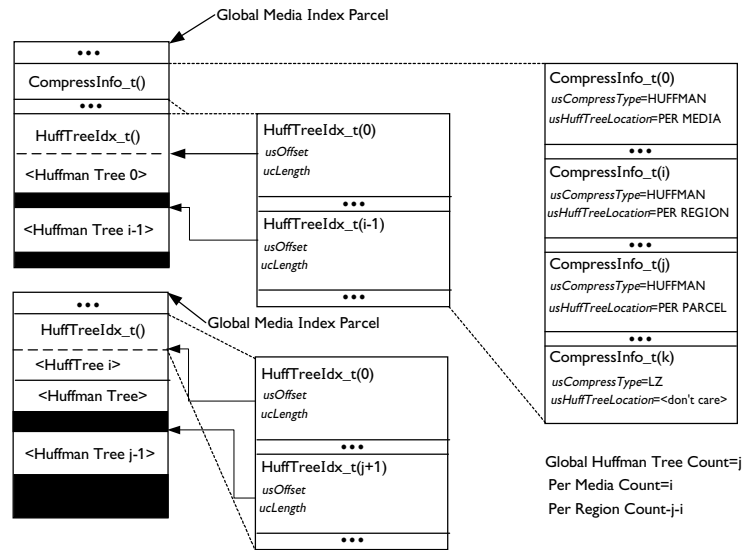


Figure 2-9: Compression Information and Global Huffman Trees

2.4.1.5 LZ-Compression Huffman Tree Section

The Huffman trees used for parcel-level compression have a different structure than the Huffman trees described in the previous section. The tree structure description is in the *Compiler Design Implications of SDAL Data Compression* heading in the *Data Organization and Characteristics* section. In global data, these trees are located in the second parcel of the region global data, possibly extending into the third and subsequent parcels.

The second parcel of the region global data contains a table of `CompressedDataElement_t` structures, which correspond to the trees in a one-to-one fashion. The compression tree index in compressed parcels directly indexes this table.

```

typedef struct
{
    Uchar_t      ucDataID;           /*compiler assigned table ID          */
    Uchar_t      usCompressRouting; /*from CompressionTypes_t            */
    Ushort_t     usOffset;           /*data offset from start of parcel    */
    Ushort_t     usLength;           /*data length in bytes                */
    Uchar_t      ucaReserved[2];     /*maintain 4-byte alignment           */
    ParcelID_t   parcelID;           /*parcel containing data              */
}

```

This table can be accessed by using the `usMITTreeIndexOffset` and `usMITTreeCount` fields of the `GlbMapHeader_t` structure. The parcel-level Huffman trees are accessed through the `parcelID`, `usOffset`, and `usLength` fields of the corresponding `CompressedDataElement_t` structure.

The external index array elements are referenced using the following constants:

```
#define GLB_INT_ROUTING_IDX 0
#define GLB_INT_CARTO_IDX 1

#define GLB_EXT_ROUTING_IDX 0
#define GLB_EXT_CARTO_IDX 1
#define GLB_EXT_CARTOXREF_IDX 2
#define GLB_EXT_MANEUVER_IDX 3
#define GLB_EXT_POI_IDX 4
#define GLB_EXT_ABBRSEG_IDX 5
#define GLB_EXT_CROSSROADS_IDX 6
/* just 1 for nav-alpha data */
#define GLB_EXT_NFSOUNDEX_IDX 8
/* just 1 for nav-alpha data */
#define GLB_EXT_NFALPHA_IDX 9
/* just 1 for nav-place data */
#define GLB_EXT_NFPLACE_IDX 10
#define GLB_EXT_PLACE_IDX 11
#define GLB_EXT_NFALPHAPLACE_IDX 12
#define GLB_EXT_POSTAL_IDX 13
#define GLB_EXT_PLACECROSSROADS_IDX 14
#define GLB_EXT_NFCROSSROADS_IDX 15
#define GLB_EXT_NFJUNCTIONS_IDX 16
#define GLB_EXT_POI_TYPEALPHA_IDX 17
#define GLB_EXT_POI_PLACEALPHA_IDX 18
#define GLB_EXT_POI_TYPEPLACEALPHA_IDX 19
#define GLB_EXT_POI_TYPESOUNDEX_IDX 20
#define GLB_EXT_POI_PLACESOUNDEX_IDX 21
#define GLB_EXT_POI_TYPEPLACESOUNDEX_IDX 22
#define GLB_EXT_JUNCTIONS_IDX 23
/* maintain LEVELn in consecutive sequence! */
#define GLB_EXT_LEVEL1_PLACE_ALPHA_IDX 24
#define GLB_EXT_LEVEL2_PLACE_ALPHA_IDX 25
#define GLB_EXT_LEVEL3_PLACE_ALPHA_IDX 26
#define GLB_EXT_LEVEL4_PLACE_ALPHA_IDX 27
#define GLB_EXT_LEVEL5_PLACE_ALPHA_IDX 28
#define GLB_EXT_LEVEL6_PLACE_ALPHA_IDX 29
#define GLB_EXT_LEVEL7_PLACE_ALPHA_IDX 30
#define GLB_EXT_HIGHWAY_IDX 32
/* municipal place index */
#define GLB_EXT_LEVELM_PLACE_ALPHA_IDX 33
#define GLB_EXT_NAV_FTR_NEXTCHARALPHA_DENSE_IDX 34
```

2.4.1.6 Language Bit Mask Array

The language bitmask array is an array of 4-byte elements whose location and number are defined in the `GlbMapHeader_t` structure. Each language code defined in global data corresponds to a bit in the language bitmask, based on its language index value. Bit `DB_BASENAME_LANGUAGE` is reserved for enabling identification of a default language used when no language is specified in a search query, or no match can be found with a specified language.

For a named entity in the database, each of its exonyms points to a bitmask in this array with a single bit set for the language code of the exonym. The basename records point to a bitmask with a bit set for each language in which there are no exonyms, as well as the `DB_BASENAME_LANGUAGE`.

The language bitmask array contains one entry for every unique combination of bitmask required by the database to enable this.

This language bit mask array makes it possible to easily identify if an exonym exists for a basename and if so, which exonym is suitable for the current preferred language, and which is the preferred default record (if no language is specified).

Section 3 Spatially-Organized Data

3.1 Spatial Data

Some of the data in the PSF region file is stored and accessed spatially. The spatially organized data in the PSF includes:

- Global kd-tree
- Route calculation data parcels
- Cartographic data parcels
- Cartographic cross-reference data parcels
- Maneuver data parcels
- Crossroads data parcels
- POI spatial index parcels (not POI data parcels)

Spatially organized data has the following characteristics:

- The parcelization of the data occurs spatially. A spatial parcel will contain all of a particular entity's data within a defined geographic rectangle. The data within a parcel is organized spatially, usually in approximate Peano key order
- All spatial parcels in the PSF region file are indexed by the global kd-tree, though other b-tree indexes may also provide access to this data

To facilitate access to all types of map data in all contexts, the different types of spatially organized parcels often contain pointers to other nearby or overlapping parcels. For example, a maneuver parcel references an overlapping layer 0 route calculation parcel.

3.1.1 Global KD Tree

The SDAL PSF contains a two dimensional (2d) global kd-tree spatial parcelization index whose structure corresponds to the parcelization that is a refinement of all parcelizations for all spatially parcelized data types. What this means is the following: if a separate kd-tree were constructed for each spatial parcel type and layer, each of these separate kd-trees would be a subset of the global kd-tree – that is, each of these kd-trees would cover the same geographic area as the global kd-tree, and each node in each of these trees would correspond to a node in the global kd-tree. This is possible because the parcelization rules do not allow overlapping parcels. Two spatial parcels must be disjoint, equal, or one must be entirely contained in the other.

This kd-tree contains implicit bounding rectangle information for each parcel—implicit because the parcelization rule creates parcel boundaries that can be encoded with a few bits (five bits per division because rectangles are split on $n/32$ boundaries). Each entry (i.e., each internal node or leaf node in the kd-tree) also contains the parcel identifiers of any parcels corresponding to that rectangle.

Figure 3-1: Geographic Rectangle Divided into Parcels shows a rectangular area which has been parcelized into six parcels (A through F). The numbers indicate the parcel splitting which has occurred:

1. Vertical cut at 14/32 tile division.
2. Horizontal cut at 18/32 tile division.
3. Horizontal cut at 16/32 tile division.
4. Vertical cut at 12/32 tile division.
5. Horizontal cut at 16/32 tile division.

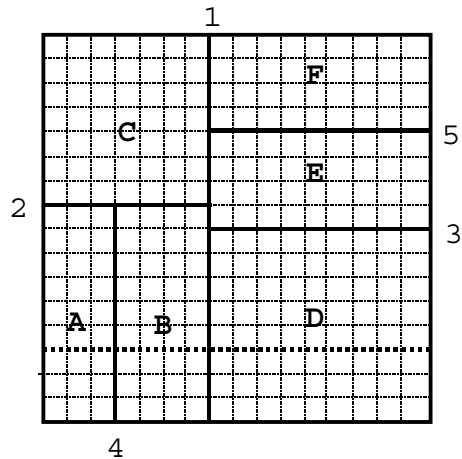


Figure 3-1: Geographic Rectangle Divided into Parcels

Figure 3-2: KD-Tree Representation of Figure 1: Geographic Rectangle Divided into Parcels Parcelization shows how the above parcelization can be represented as a kd-tree. The numbered nodes of the kd-tree contain parcel splitting information.

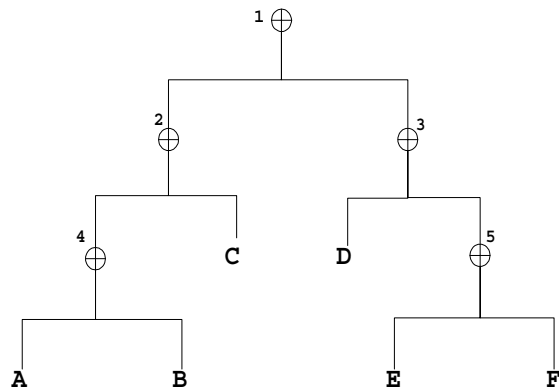


Figure 3-2: KD-Tree Representation of the previous figure's Parcelization

Each node in the kd-tree can contain rectangle splitting information, or information for parcel types at that node, or both. Figure 3-3: KD-Tree SDAL Table provides an example of how the above kd-tree information would appear in the SDAL PSF.

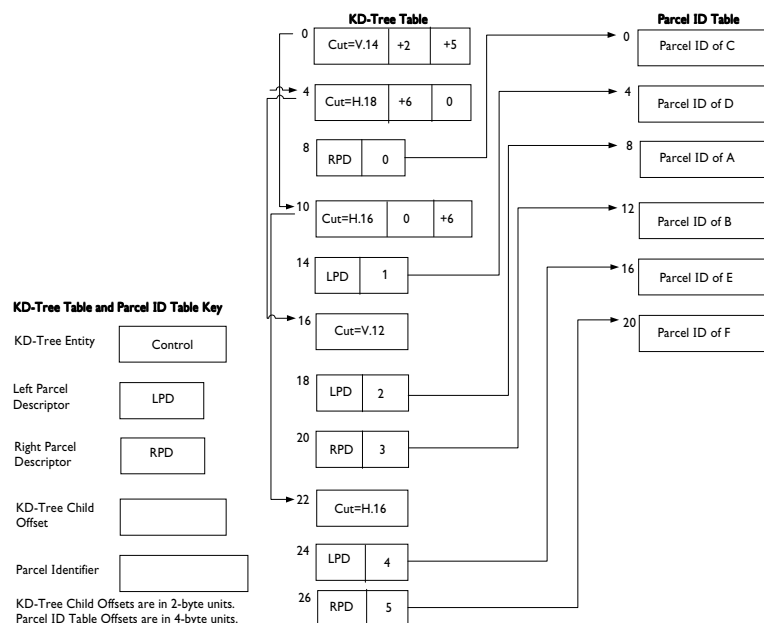


Figure 3-3: KD-Tree SDAL Table

3.1.1.1 KD-Tree Representation of Multiple Parcelizations

A global kd-tree node may refer to a data parcel of some type (e.g., layer 1 Route Calculation) and also have descendant nodes which refer to data parcels of another type (e.g., layer 0 Cartographic).

All spatially organized parcel types are parcelized in parallel. Therefore, they can be indexed via a single kd-tree: the global kd-tree. This kd-tree is two-dimensional, based on the following parcelization rule: For any two spatial parcels A and B of same or differing type or layer, it is always the case that exactly one of the following is true: (1) the geographic area covered by A is disjoint from that covered by B, (2) the geographic area covered by A is equal to that covered by B, (3) the geographic area covered by A contains that covered by B, or (4) the geographic area covered by A is contained in that covered by B.

Figure 3-4: Parcelization Across Layers shows how parcels can be further split at lower layers. For example, Parcel B at layer 3 is split into parcels B1 and B2 at layer 2, and parcels B11, B12, and B21 at layer 1.

The line splitting a rectangle into two subrectangles must be chosen so that all data in the south or west subrectangle is associated with segments and nodes *having smaller entity IDs* than those associated with data in the north or east subrectangle. This rule allows the kd-tree used to search the parcels spatially also to be used as a binary tree for searching the parcels by segment or node entity ID.

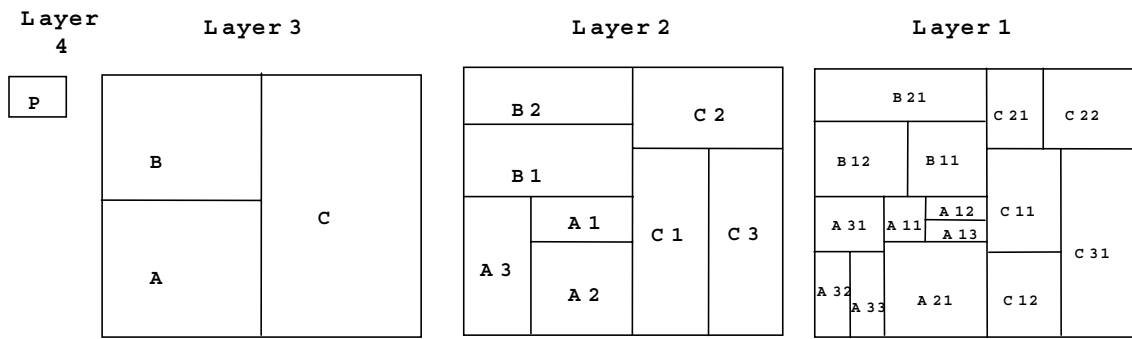


Figure 3-4: Parcelization Across Layers

Figure 3-5 KD-Tree Representation of Figure 4 shows the multiple layer parcelization as a kd-tree representation.

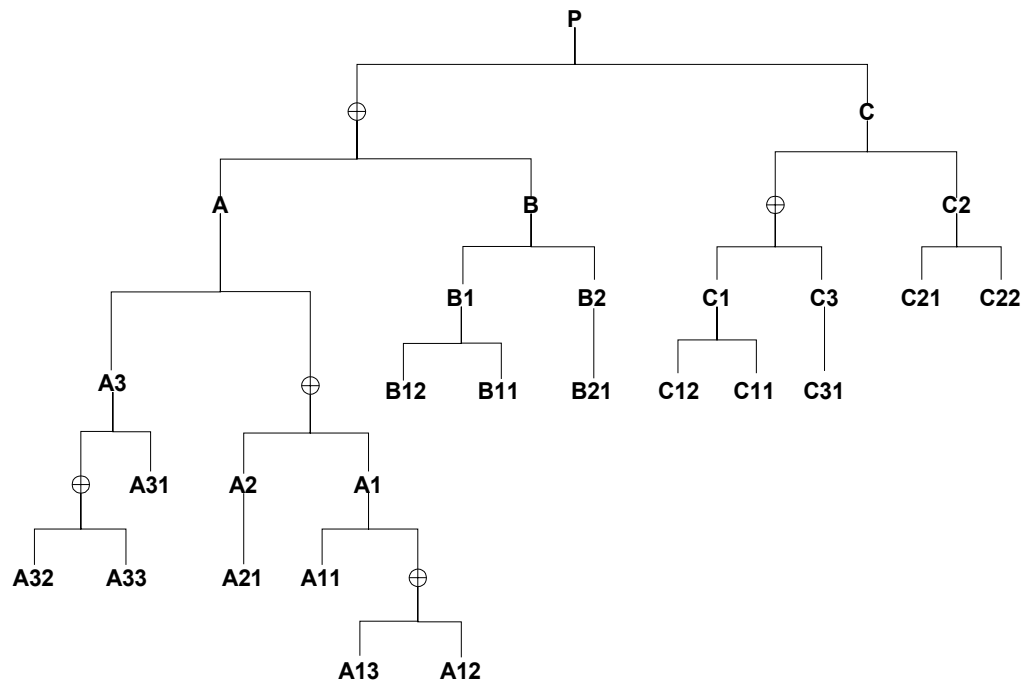


Figure 3-5: KD-Tree Representation of Figure 4: Parcelization Across Layers Parcelization

3.1.1.2 Global KD Tree Entry Structure

The following describes the structure of an entry in the kd-tree:

- In general, each cut splitting a rectangle is described with a kd-tree entry of two or more bytes: two bytes of control information, zero or six bytes of segment and node entity IDs (described later in this section), zero to eight bytes to represent offsets to the left and right children (one, two, or four bytes per offset), and zero or more bytes of Parcel Identifiers.
- A one- or two-byte child offset represents an offset within the current parcel (from the current entry to the child entry, in two-byte units).

- A four-byte offset represents a child in another kd-tree index parcel. This offset is made up of a two-byte offset from the initial kd-tree parcel to the kd-tree parcel containing the child, plus a two-byte tree identifier, identifying the tree within that parcel.
- Five bits of the control information are dedicated to the location of the line that cuts the parent rectangle into left and right sub-rectangles (since this cut can be at any $1/32$ division of a rectangle's enclosing di-tile).

Whenever either child of a kd-tree (internal or leaf) node corresponds to a parcel, a parcel descriptor (one byte) is present in the kd-tree, containing the parcel's type and layer, and other information. A parcel descriptor is followed by a four-byte Parcel ID.

Whenever any of the left or right descendants of a kd-tree entry are layer 0 routing parcels, the entry contains—explicitly or implicitly—a segment entity ID and a node entity ID (24 bits each). Each segment / node entity ID represents the minimum segment / node entity ID present in the right sub-tree of the current kd-tree entry.

Both left and right child offsets are required to have the same size (a four byte offset for a non-pruned child uses the rightmost, i.e., least significant, bytes).

The descriptor list does not store table indexes for parcel IDs but instead stores parcel IDs directly since there is no other reference to it in the index parcel; however, these parcel IDs are not necessarily aligned in storage.

Finally, the entire descriptor / identifier pair list is rounded up to 2-byte-multiple total length because child offsets are in units of 2-bytes.

Node (and segment) entity IDs are stored in the global KD-Tree to support a map from node (or segment) entity ID to parcel ID of given layer and type. If bit 12 of the control bits is set then the six (6) bytes immediately following the control bytes (and preceding the child offset information) consist of three (3) bytes segment entity ID followed by three (3) bytes of node entity ID; these entity IDs are the minimal entity IDs contained in the right child.

The order of fields in a kd-tree entry is as follows:

- Control Data

Byte 0								Byte 1							
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1
0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5
Control Bits															

Figure 3-6: Two Byte KD Tree Control Data

Control Bit Definitions:

- Bit 0: Left Parcel Descriptor List Present
- Bit 1: Right Parcel Descriptor List Present
- Bit 2: 0 = Vertical Cut; 1 = Horizontal Cut
- Bits 3-7: Location of Cut
- Bits 8-9: 00 = No Left Child; Empty Parcel
01 = 4-byte Left Child Offset Present
10 = 1-byte Left Child Offset Present
11 = 2-byte Left Child Offset Present
- Bits 10-11: 00 = No Right Child; Empty Parcel

01 = 4-byte Right Child Offset Present

10 = 1-byte Right Child Offset Present

11 = 2-byte Right Child Offset Present

Bits 12-13: 00 = No Entity IDs Present

01 = Entity IDs Present

10 = Implicit Entity IDs equal to zero

11 = Implicit Entity IDs equal to 0xFFFFFFFF

- Segment Entity ID and Node Entity ID, if present
- Left and Right Child Offsets, if present

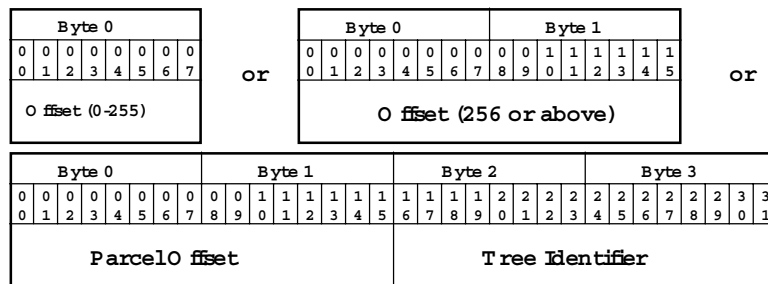


Figure 3-7: KD-Tree Child Offset

- List of Parcel Descriptors and Identifiers, if present

Parcel descriptor

0	0	0	0	0	0	0	0
0	1	2	3	4	5	6	7

Parcel descriptor Bit Definitions:

Bit 0: Last Parcel descriptor in List

Bits 2-4: Parcel Type

Bits 5-7: Layer

Parcel descriptor plus Parcel Identifier

Byte 0	Byte 1	Byte 2	Byte 3	Byte 4
Parcel Descriptor	Parcel Identifier			

Figure 3-8: Parcel Descriptors and Identifiers

3.1.1.3 Global KD Tree Parcelization

The Global KD tree may not fit into a single index parcel. An index parcel may store a complete or partial tree (sub-tree), or it may store more than one complete tree (sub-tree of a global tree).

The roots are numbered (by tree identifier) in sequence starting with 0 within each index parcel. Thus, an inter-parcel reference to a node is by parcel ID (minus region specifier) plus tree ID. Intra-parcel reference to a node is by byte offset. Index trees are stored uncompressed in the index parcels.

All index parcels in the global tree have identical size and redundancy information, which allows the child parcel's parcel ID to be constructed at runtime from this parcel ID offset, and the second two bytes of the offset give the tree ID.

Index Parcel Structure

The first data in the parcel is the IdxPclHdr_t header (byte packed).

```

/*****
**IdxPclHdr_t: parcel header common to index parcel types
*****/
typedef struct
{
    PclHdr_t    pclHdr;
    Ushort_t    usNumRoots;          /* number of tree roots */
    Ushort_t    usExtensionOffset; /* If non-zero: Offset to extension **
                                   of header*/
}
IdxPclHdr_t;

```

Following the header is the offset array to the roots in the parcel, indexed by tree id:

```

Ushort16    root_offset []; /* root offset array */

```

Following the offset array are the trees.

3.2 Route Calculation Data

Route calculation data is stored in parcels containing only the data needed for route calculation including node, condition, and segment information. Direction generation, map-matching, or map display information require data in other parcels.

A Peano key is constructed from each route calculation parcel's minimum latitude and longitude (i.e., the latitude and longitude of the parcel's southwest corner), and the parcels are stored on the media in order by that key.

The Global kd-tree is used to access route calculation parcels spatially. The Global kd-tree contains node and segment entity ID information, so it can also be used to locate the route calculation parcel containing a given entity ID.

Route calculation data consists of data parcels containing:

- Node records
- Ghost node records
- Segment records
- Shape point records (layer 0 only)
- Condition records
- Date / time records
- Normalized attribute array data

3.2.1 Route Calculation Data Parcel Layers

The SDAL PSF Route Calculation data is parcelized in multiple layers. The lowest level layer (layer zero) of the route calculation data contains all route calculation data within the spatial boundaries of the parcel. Higher layers of the data contain successively less detailed information and consequently each higher layer parcel represents larger geographic areas.

Data associated with a segment (e.g., node, shape, condition data) is contained in higher layer parcels if the associated segment is present at those layers.

The arterial network at any layer must be connected. For each navigable segment, there must be a navigable path in the network to every other navigable segment.

3.2.2 Routing Parcel Structure

Routing parcels contain data needed for route calculation only (direction generation, map-matching, or map display require data in other parcels); this includes node, condition and segment information. The parcelization method used for this data has the goal of maximizing the amount of relevant route calculation information allowed in memory at once, to minimize paging during route calculations.

3.2.2.1 Route Calculation Parcel Header Information

The following structures define the initial, fixed portion of the parcel data for the route calculation parcels.

Routing Parcel Header for Layer 0 parcels

```
typedef struct
{
    SptlPclHdr_t  spatialHdr;
    BlkDesc1_t    nodeDesc;           /* Node/supernode block offset
                                      descriptor */
    BlkDesc1_t    segmentDesc;       /* Segment block offset descriptor
                                      */
    BlkDesc2_t    extSegDesc;        /* External segment record offset
                                      */
}
```

```

        descriptor */
        BlkDesc_t    conditionDesc;    /* Condition block descriptor */
        BlkDesc2_t   ghostNodeDesc;    /* Except/Ghost node block descrip-
        tor */
        BlkDesc3_t   deltaShapeDesc;   /* Shape point block descriptor */
        BlkDesc3_t   dateTimeDesc;     /* Date-time block descriptor */
        BlkDesc4_t   attribArrayDesc;  /* Local Normalized Attribute Array
        */
        BlkDesc4_t   spatialIndexArrayDesc; /* Spatial index bitmask array
        block descriptor */
        BlkDesc4_t   aggrSegArrayDesc;  /* Aggregated Segment Entity ID
        Array */
        Ushort_t     usMaxArmToArm;     /* Max entries in Supernode Arm-Arm
        table */
        Uchar_t      ucMaxSegsPerNode;  /* Max segments connected to a node
        */
        Uchar_t      ucMaxChildNodes;   /* Max child nodes per supernode */
        Uchar_t      ucMaxChildSegs;    /* Max internal segs per supernode
        */
        Uchar_t      ucMaxArmArmSegs;   /* Max segs in Supernode Arm-Arm
        Path */
        Uchar_t      ucMaxCondsPerSeg;  /* Max Conditions per Segment
        Record */
        Uchar_t      ucMaxSegsPerCond;  /* Max Segs related to a condition
        */
        Uchar_t      ucMaxDtTmPerCond;  /* Max Date-time recs per Condition
        */
        Uchar_t      ucMaxNFsPerSegment; /* Max Nav Feature IDs per segment
        */
        Uchar_t      ucaReserved[2];    /* Maintain 4-byte alignment */
    }
    RoutingParcelHeader0_t;

```

Routing Parcel Header for parcels above layer 0

```

typedef struct
{
    SptlPclHdr_t    spatialHdr;

    BlkDesc2_t      nodeDesc;           /* Node/supernode block offset
    descriptor */
    BlkDesc2_t      segmentDesc;        /* Segment block offset descriptor */
    BlkDesc_t       conditionDesc;      /* Condition block descriptor */
    BlkDesc2_t      ghostNodeDesc;      /* Except/Ghost node block descriptor
    */
    BlkDesc3_t      abbrevSegDesc;      /* Abbrev. seg/node block descrip-
    tor*/
    BlkDesc3_t      dateTimeDesc;       /* Date-time block descriptor */
    BlkDesc4_t      attribArrayDesc;    /* Local Normalized Attribute Array
    */
    BlkDesc4_t      aggrSegArrayDesc;   /* Aggregated Segment Entity ID Array
    */
    Ushort_t        usMaxAbbrevSegs;    /* Max Abbrev Segs per Aggregated Seg
    */
    Ushort_t        usMaxArmToArm;      /* Max entries in Supernode Arm-Arm
    table */
    Uchar_t         ucMaxSegsPerNode;   /* Max segments connected to a node
    */
    Uchar_t         ucMaxChildNodes;    /* Max child nodes per supernode */

```

```

    Uchar_t    ucMaxChildSegs;    /* Max internal segments per supernode */
    Uchar_t    ucMaxArmArmSegs;   /* Max segments in Supernode Arm-Arm Path */
    Uchar_t    ucMaxCondsPerSeg;  /* Max Conditions per Segment Record */
    Uchar_t    ucMaxSegsPerCond;  /* Max Segs related to a condition */
    Uchar_t    ucMaxDtTmPerCond; /* Max Date-time recs per Condition */
    Uchar_t    ucMaxNFsPerSegment; /* Nav Feature IDs per segment */
}
RoutingParcelHeader1_t;

```

Per-parcel Data

The locations of the following arrays, and their sizes, are defined in the parcel header:

```

ParcelID_t    aPclId[];          /* Parcel ID Array */
Uchar_t       ucaAttribArray[];  /* Local Normalized attribute array */
Uchar_t       ucaAggrSegIDArray[] /* Each entry is a 3 byte aggr seg entity ID */
ExternalPcl_t aExtPcl[];        /* External parcel Array-usually not present */

```

3.2.2.2 Data Reference

Figure 3-9 illustrates the layout and data relationships of the parcel header at layer 0.

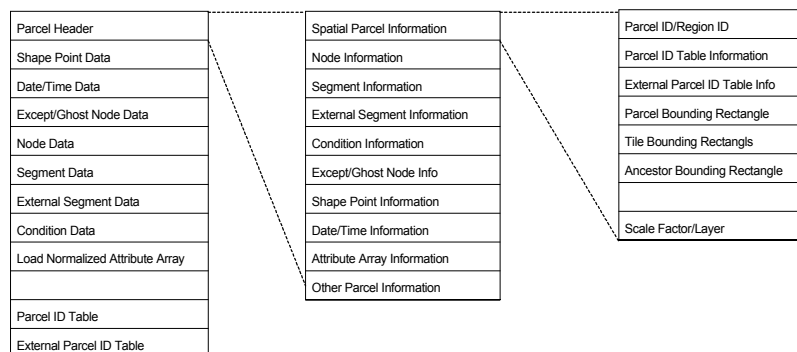


Figure 3-9: Bottom Layer Route Calculation Parcel Header Information :

Figure 3-10 illustrates the layout and data relationships of the parcel header at layer 0:

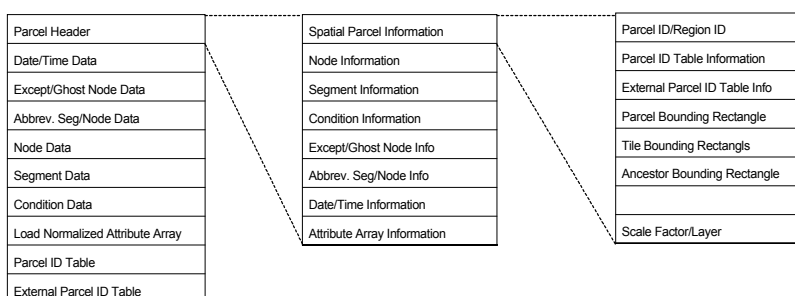


Figure 3-10: Upper Layer Route Calculation Parcel: Header Information

3.2.2.3 Local Normalized Attribute Array

The Local Normalized Attribute Array (`ucaAttribArray[ ]`) contains segments' normalized attributes not contained in the Global Normalized Attribute Array. Each array entry has the same format as the Global Normalized Attribute Array entries. (See "SDAL Media File, Normalized Attribute Array.")

3.2.2.4 Associated Parcel Identifiers

Internal to the Region

An array of associated parcel IDs is included in the parcel, to allow them to be accessed directly without the need for reading a separate index.

The following parcel IDs may be associated with a routing parcel:

1. For each ghost node, the ID of the parcel containing the corresponding node record.
2. For supernodes in parcels above layer 0, the ID of the layer 0 parcel containing the supernode.
3. For aggregated segments, the ID of the abbreviated segment parcel containing the aggregated segment's abbreviated nodes and segments

The parcel ID array is accessed via the following parcel header fields (defined in the `XrPclHdr_t` structure, contained in the `SptlPclHdr_t` structure)

```
Ushort_t  usPclidOffset; /* Offset to beginning of parcel ID array */
Ushort_t  usPclidCount; /* Number of parcel IDs in the array */
```

3.2.2.5 Initial Location of Parcels

A Peano key is constructed from each route calculation parcel's minimum latitude and longitude, and the parcels are stored on the media in order by that key.

3.2.2.6 Parcel Internal Formats

The information in data records is in a packed form with no padding bytes and generally no unused fields, so each uncompressed data record is in the form of a variable-length character array. This packed data is the source from which the logical records passed to the application software by SDAL are constructed.

For most records (except for nodes and segments above the bottom layer), a table of offsets locates the beginning of each block of 2^K records. This reduces the number of variable-length records that must be examined (compared with a sequential search for the desired record).

At the bottom layer, Node Records are stored in Peano key order within the parcel based on latitude and longitude, and Segment Records are stored in approximate Peano key order based on the

latitude and longitude of the segment's "Left" node. All nodes and all segments in the region are given a unique entity identifier in this order.

Condition and Date-time records are stored in Peano key order based on the Peano key of either the node connecting the primary segment with the first of the related segments, or the primary segment's left node for a one-segment condition. Storage of records in Peano key order reduces the time spent searching for records within a parcel, since entities in spatial proximity are often represented by records near each other within a parcel.

3.2.2.7 Spatial Searches within a Bottom Layer Parcel

Each routing parcel is divided into eight subrectangles, and this subdivision is represented by a kd-tree, each of whose leaf nodes corresponds to one subrectangle. This kd-tree narrows the list of segment records to be examined in a spatial search. Each of the subrectangles represented by a kd-tree leaf node contains, roughly, about the same number of shape points and nodes as other such subrectangles. Shape point information in the bottom layer is stored as a contiguous array of three-byte delta position records (ranging from -1024 to +1023 NavTech Units from the position of the previous shape point). Each segment record contains a count of shape point records and a pointer to its first shape point.

The following describes a kd-tree entry for a bottom layer route calculation parcel. Cuts are defined at any $1/32$ division of a rectangle's minimum enclosing $2^1 \times 2^1$ di-tile (so, for example, a division exactly in half is represented as a $16/32$ cut). Each internal or leaf node in the kd-tree represents a subrectangle of the entire parcel. Each subrectangle corresponding to a leaf node is assigned an index in the range 0-7. Each segment in the parcel is checked against each subrectangle, and the segment is assigned the index of any subrectangle it intersects. A segment can, of course, intersect multiple subrectangles, so an 8-bit bitmask (each bit of which corresponds to a subrectangle) is associated with each segment.

Figure 3-11 shows the layout of an internal kd-tree entry. The eight control bits are defined as:

- Bit 0 0 = Vertical Cut
 1 = Horizontal Cut
- Bit 1 0 = left is Offset
 1 = Left is Index
- Bit 2 0 = Right is Offset
 1 = Right is Index
- Bits 3-7 (5 bits) Location of Cut ($n/32$)

The second byte of the internal kd-tree entry defines the Offset or Index. An Offset is the distance from the beginning of the current entry, in two-byte units, to the beginning of its left or right child entry. An Index is in the range 0 - 7, and is the one assigned to the rectangle represented by the left or right child of the current entry. The Offset/Index Bits are defined as:

- Bits 8-11 (4 bits) Left Offset or Index
- Bits 12-15 (4 bits) Right Offset or Index

Byte 0								Byte 1							
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1
0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5
Control								Left Offset or Index				Right Offset or Index			

Figure 3-11: Internal kd-tree Entry

3.2.3 Routing Node Records

The node record stores positional information such as longitude, latitude, and a Z-level, for end nodes of route calculation segments. The node record also contains attributes providing additional information about the node, but does not include shape points.

3.2.3.1 Technical Description

The node entity consists of two virtual entities—the regular node and the supernode.

- Regular nodes represent dead ends or simple intersections of two or more segments.
- Supernodes collapse complex intersections, consisting of many segments, into a single node. For example, the segments making up a roundabout can be collapsed into a single supernode (i) as illustrated below:

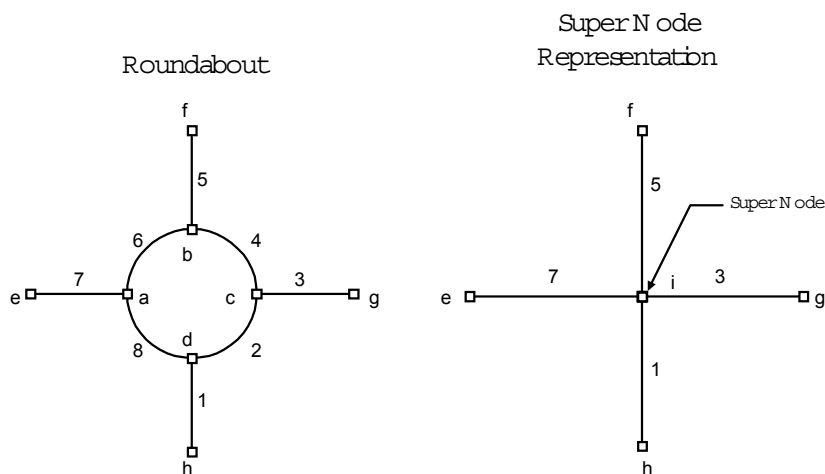


Figure 3-12: Example of collapsing a roundabout into a single supernode

The supernode identifies all the segments that participate in a complex intersection and uses this information to simplify route calculation for the intersection. Although the geometry of segments 1, 3, 5, and 7 appears different in the supernode representation, all the segments' attributes, including the segment length, remain the same as they would for the full representation of the intersection. Regular nodes a, b, c, and d are subordinate to supernode i.

3.2.3.2 Data Reference

Figure 3-13 illustrates the relationships of the node data in the bottom layer route calculation parcel.

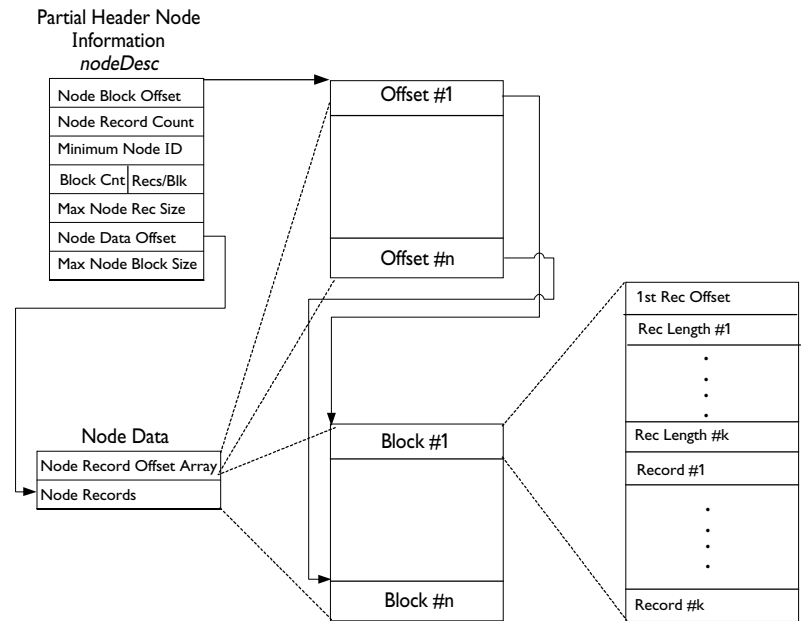


Figure 3-13: Route Calculation Parcel: Node Data (at Bottom Layer)

Figure 3-14 illustrates the relationships of the node data within the route calculation parcel above the bottom layer.

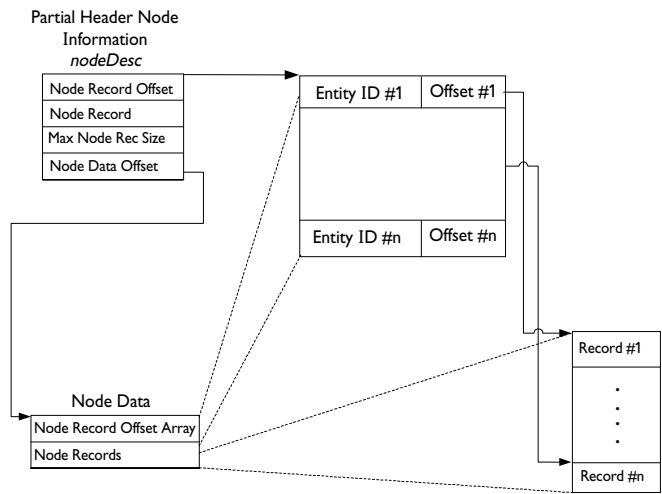


Figure 3-14: Route Calculation Parcel: Node Data (above Bottom Layer)

3.2.3.3 Node Record

The node record is a character array of variable length. Since fields are not at fixed positions, this record is not described by a C-language structure.

Field (Length) Description

The following nine fields occupy two bytes:

Node Type (3 bits) (Bits 0-2, where bit 0 is the high order bit):

	<u>Value</u>	<u>Description</u>
	000	Regular node not subordinate to supernode
	001	Regular node subordinate to supernode
	010	Bottom layer supernode
	011	Supernode above bottom layer
Rank	(3 bits) (Bits 3-5)	The highest routing layer in which this node appears.
	<u>Value</u>	<u>Description</u>
	000	Layer 0
	001	Layer 1
	010	Layer 2
	011	Layer 3
	100	Layer 4
Max Signif	(3 bits) (Bits 6-8)	Highest layer at which this node has three or more segments connected to it, its adjacent segments have a mismatch on at least one aggregation attribute, or the node is a dead end.
	<u>Value</u>	<u>Description</u>
	000	Layer 0
	001	Layer 1
	010	Layer 2
	011	Layer 3
	100	Layer 4
Unused (1bit)	(Bit 9)	
In Process	(1 bit) (Bit 10)	Set if all segments attached to this node are in-process data.
Attributes	(2 bits) (Bits 11-12)	
	Bit 11:	Node is a mini-roundabout.
	Bit 12:	Node is a supernode.
Node Cost Flag	(1 bit) (Bit 13)	Set to 1, if the record contains arm-to-arm node costs (impedances).
Z-level Flag	(1 bit) (Bit 14)	Set to 1, if the record contains a Z-level field. If not set, the Z-level defaults to zero.
Named Flag	(1 bit)	Set to 1, if this node is named. A “named” node represents a junction with a name, found in a junction record, and reached via the “Named Node to Junction” cross-reference in a maneuver parcel.
Latitude	(2-3 bytes) (Type 4 Variable Length Unsigned Value)	To compute the value for the latitude field: subtract the minimum latitude (in NTUs) of the parcel's bounding rectangle from the latitude value of the node, and right shift the result by a number of bits equal to the parcel's scale factor.

Longitude	(2-3 bytes) (Type 4 Variable Length Unsigned Value) To compute the value for the longitude field: subtract the minimum longitude (in NTUs) of the parcel's bounding rectangle from the longitude value of the node, and right shift the result by a number of bits equal to the parcel's scale factor.
Long Haul	(0-1 bytes) Type IntValP associated with Parcel Descriptor array entries pdRouting_LH and pdRouting_LH_2 nd . When present, it occupies the high order bits of the byte, and the remaining bits of the byte are unused.
Seg Count	(1 byte) Count of the number of segments attached to this node.
SegID Array	(variable) The number of entries given by the Seg Count field. (Type 1 Variable Length Unsigned Value) Represents a segment attached to this node, as follows: (1) If less than this parcel's segment count, and a bottom layer parcel, this is both an index to the array of segment records for this parcel, and (by the addition of the parcel's Minimum Segment Entity ID) the segment's entity identifier. (2) If less than this parcel's segment count, and not a bottom layer parcel, this is an index to this parcel's segment offset array, from which both the segment's identifier and an offset to the segment record can be obtained. (3) If not less than this parcel's segment count, this value minus this parcel's segment count is an index to this parcel's external segment offset array, from which can be obtained both the segment's identifier and an offset to the segment record.
Name Chg Bits	Length: $(((((\text{Seg Count}) * ((\text{Seg Count}) - 1)) >> 1) + 7) >> 3)$ bytes. A triangular array with bits indicating whether the set of names attributed as "Name on Road Sign" changes between each pair of arms. Bits are defined as follows, where bit 0 refers to the leftmost or most significant bit in the array and N is the number of arms:
Bit 0:	Name change from segment 0 to segment 1
Bit 1:	Name change from segment 0 to segment 2
...	
Bit N-2:	Name change from segment 0 to segment N-1
Bit N-1:	Name change from segment 1 to segment 2
Bit N:	Name change from segment 1 to segment 3
...	
Bit 2N-4:	Name change from segment 1 to segment N-1
Bit 2N-3:	Name change from segment 2 to segment 3
...	
Bit $(N(N-1)/2)-1$:	Name change from segment N-2 to N-1
The 0-based index of the name change bit for the path from <i>Arm I</i> to <i>Arm J</i> is therefore given by the formula $((N(N-1)/2) - ((N-I)(N-I-1)/2) + J - I - 1)$.	
If the following is present, the Z-level Flag must be set to 1:	

Z-level (1 byte) The Z-level value (-5 to 4); if not present, the Z-level defaults to zero.

If the following is present, the Node Type field must have a binary value of 011:

SupNode Pcl (2 bytes) An index to the Parcel ID table for the layer 0 parcel containing this supernode.

The following four fields are present when the Node Type field has a binary value of 010 or 011:

Highlight Layer Length: three bits. The Cartographic Highlighting Layer associated with this supernode. Values are in the range 0-5.

Rndabout Flg Length: one bit. When set, this node represents a roundabout.

Spc Traff Flg Length: one bit. When set, this node represents a Special Traffic Figure.

Mnvr Bit Flag Length: two bits.

Value	Description
-------	-------------

00:	All maneuver bits are zero.
-----	-----------------------------

01:	All maneuver bits are set.
-----	----------------------------

10:	Maneuver bits are in the Mnvr Bit Array.
-----	--

The following field is present when the Node Type field has a binary value of 010 or 011 and the Mnvr Bit Flag is set to binary 10:

Mnvr Bit Array Length: The following number of bytes: $((((\text{Seg Count})^2) + 7) \gg 3)$; A bit map containing one bit for each possible arm-to-arm path within a supernode.

Each one-bit entry represents an arm-to-arm path within the supernode, as follows: Let the segments in the SegID Array be $S_0, S_1, \dots, S_{N-1}$, where N is equal to Seg Count. The entry representing the arm-to-arm path from segment S_i to segment S_j is shown by bitmap entry number $(N \times i) + j$.

Each one-bit entry is interpreted as follows: If it represents the path from segment S_i to S_j it's set to 1, for some non-ramp segment S_{int} on that path. The set of road names of S_{int} attributed as "name on road sign" is non-empty, and has an empty intersection with the union of the set of names of S_i and S_j attributed as "name on road sign".

If the following field is present, either the Node Cost Flag must be set to 1, or the Node Type must have a binary value of 010:

Arm-Arm Map $(((((\text{Seg Count})^2) \ll 1) + 7) \gg 3)$ Bit map containing two bits for each possible arm-to-arm path within a supernode.

Each two-bit entry represents an arm-to-arm path within the supernode, as follows: Let the segments in the SegID Array be $S_0, S_1, \dots, S_{N-1}$, where N is equal to Seg Count. The entry representing the arm-to-arm path from segment S_i to segment S_j is shown by bitmap entry number $(N \times i) + j$.

Each two-bit entry is interpreted as follows: A binary value of 10 implies a node cost of zero. A binary value of 11 implies an "infinite" node cost. A binary value of 00 or 01 is concatenated to the beginning of a Node Cost Tbl entry to form a Node Cost Value Table Index.

If the following field is present, the Node Cost Flag must be set to 1:

Node Cost Tbl (Length is variable) (Four bits). The number of entries equals the number of entries in the Arm-Arm Map with a binary value of 00 or 01. The table length is rounded up to the next byte if it ends on a half-byte boundary. The Arm-Arm Map entry and the corresponding Node Cost Tbl entry are concatenated to form an index to the Node Cost Value Table (usaNodeCost [] in global data). This index is always in the range 0-31, since the high order bit of the six-bit index is zero.

If the following field is present, the Node Type must have a binary value of 001:

Supernode ID (1-2 bytes) (Type 1 Variable Length Unsigned Value) Represents the supernode this node is subordinate to, as follows:

- (1) If less than this parcel's node count, *and* a bottom layer parcel, this is both an index to the node record array for this parcel, as well as (by the addition of the parcel's Minimum Node Entity ID) the supernode's entity identifier.
- (2) If less than this parcel's node count, and *not* a bottom layer parcel, this is an index to this parcel's node record offset array, from which can be obtained both the supernode's identifier and an offset to the supernode record.
- (3) If not less than this parcel's node count, this value minus this parcel's node count is an index to this parcel's ghost node offset array, from which can be obtained both the supernode's identifier and an offset to the (ghost) supernode record.

If the next five fields are present, the Node Type must have a binary value of 010:

Node Count (1 byte) (Type 1 Variable Length Unsigned Value) Number of regular nodes subordinate to this supernode.

Node IDs (variable) Number of entries given by the Node Count field. Represents a node subordinate to this supernode, in the following way:

- (1) If less than this parcel's node count, the entry is both an index to the node record array for this parcel, as well as (by the addition of the parcel's Minimum Node Entity ID) the node's entity identifier.
- (2) If not less than this parcel's node count, this value minus this parcel's node count is an index to the ghost node record offset array.

Int Seg Cnt (1 byte) Number of segments internal to this supernode.

Int Seg Array (Type 1 Variable Length Unsigned Value) (Internal Segment Array) Number of segments is the count in the Int Seg Cnt field. Represents a segment, as follows:

- (1) If less than this parcel's segment count, *and* a bottom layer parcel, this is both an index to the segment record array for this parcel, as well as (by the addition of the parcel's Minimum Segment Entity ID) the segment's entity identifier.
- (2) If less than this parcel's segment count, and *not* a bottom layer parcel, this is an index to this parcel's segment record offset array, from which can be obtained both the segment's identifier and an offset to the segment record.

- (3) If not less than this parcel's segment count, this value minus this parcel's segment count is an index to this parcel's external segment offset array, from which both the segment's identifier and an offset to the segment record can be obtained.

Arm-Arm Tbl (Arm-to-Arm Table) Number of entries found using the Arm-Arm Bitmap field. Each arm-to-arm path with a non-infinite impedance in the Arm-Arm Bitmap has an entry in the Arm-Arm Tbl.

Each entry in the Arm-to-Arm table contains the following three fields:

Path Length (multiple of 4 bits) The length in meters of this path through the supernode; in the parcel descriptor table entry with index equal to `pdRouting_PathBitLen`; always a multiple of four bits, all fields entry are half-byte aligned.

Path Seg Cnt (Type 3 Variable Length Unsigned Value) Count of the number of segments making up this path within the supernode.

Path Seg Array (Type 3 Variable Length Unsigned Value) Ordered list of the segments making up this path through the supernode; constitutes an index to this node's Int Seg Array.

Figure 3-15 illustrates the structure of a routing node record.

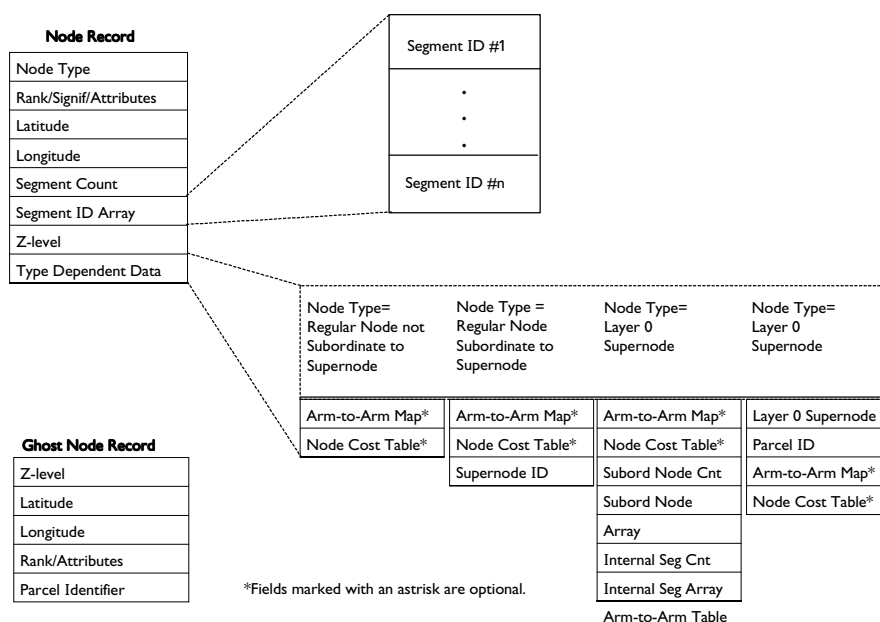


Figure 3-15: Route Calculation Parcel: Node Record

3.2.3.4 Node Block Offset Array

Node records at the bottom layer are accessed through the Node Block Offset Array. Each entry contains a block offset pointing to the beginning of a block of $2^K + 1$ length fields followed by 2^K node records. The entity identifier is obtained by adding the index of the record to the minimum node entity identifier for this parcel.

The length fields are all Type 1 Variable Length Unsigned Values. The first length field is the offset from the beginning of the block to the first node record in the block. The following 2^K length fields are the lengths of each of the 2^K compressed node records.

```
BlkOffset_t aNodeBlk[]; /* Node Block Offset Array */
```

3.2.3.5 Node Record Offset Array

Node records above the bottom layer are accessed through the Node Record Offset Array. Each entry contains both an entity identifier and an offset to the corresponding node record.

```
RecOffsetEnt_t aNodeOffsetArray[]; /* Node Record Offset Array */
```

3.3.4 Routing Segment Records

The segment record stores information about a portion or segment of the road network.

3.3.4.1 Technical Description

The segment record includes information about the end nodes of the segment and routing attributes. A frame of reference established for the segment is identified by the end nodes as the left node and the right node, and the sides of the segment as the upper side and the lower side. The left node is the node with the lowest longitude. If the longitudes of the two nodes are identical, it is the node with the lowest latitude. If the longitude and latitude values of the end nodes are identical, the left node is the node with the lowest Z-level. If the segment is rotated so it is oriented horizontally from left node to right node, the upper side is the side above the segment and the lower side is the side below the segment.

There are two types of segment entities—regular segments and aggregated segments.

- Regular segments are the fundamental units of the navigable road network. They contain only two nodes, one at each end.
- Aggregated segments are combined regular segments used at route calculation layers above layer 0. They provide a simplified representation of the road network which can help speed up route calculation. An aggregated segment represents a set of contiguous regular segments with sufficiently similar characteristics.

3.3.4.2 Data Reference

Figure 3-16 shows the relationships of the segment data in the bottom layer route calculation parcel.

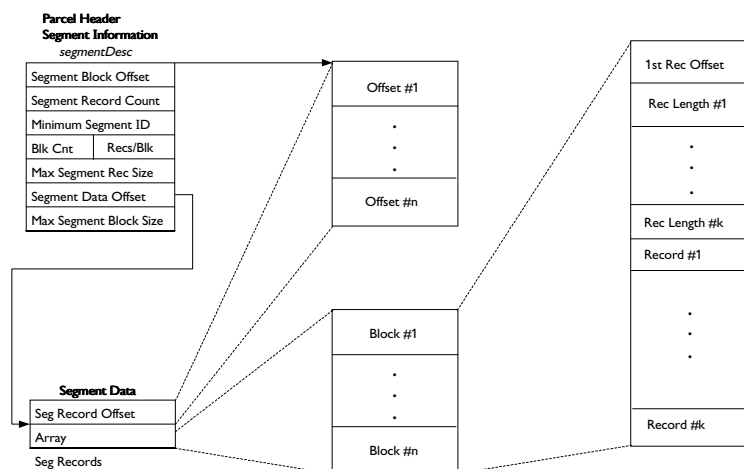


Figure 3-16: Route Calculation Parcel: Segment Data at Bottom Layer

The next diagram illustrates the relationships of the segment data within the route calculation parcel above the bottom layer.

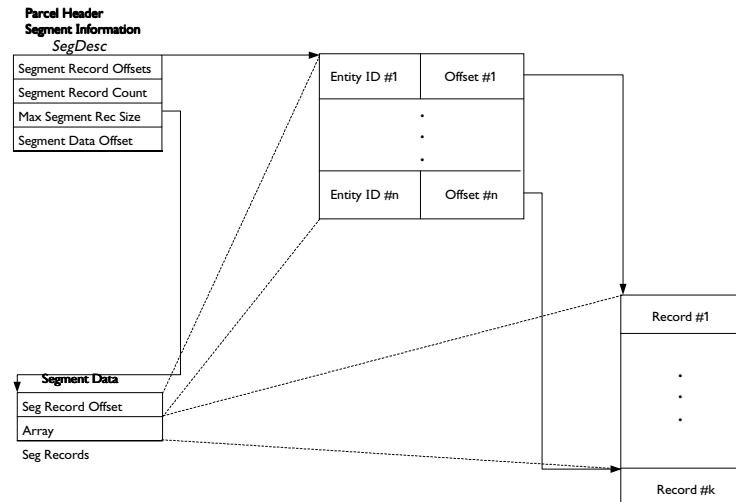


Figure 3-17: Route Calculation Parcel: Segment Data (above Bottom Layer)

3.3.4.3 Segment Record

The segment record is a character array of variable length. Since fields are not at fixed positions, this record is not described by a C-language structure.

Field (Length)	Description
----------------	-------------

The following seven fields occupy one byte:

Aggr Segment	(1 bit) (Bit 0, where 0 is the high order bit) Set to 1 if this represents an aggregated segment. (See Segment Aggregation specification.)
--------------	--

POI Flag	(1 bit) (Bit 1) Set to 1 if POIs are associated with this segment.
----------	--

Sign Flag	(1 bit) (Bit 2) Set to 1 if signs are associated with this segment.
-----------	---

Dir of Travel	(2 bits) (Bits 4-5) Direction of travel along segment.
---------------	--

<u>Value</u>	<u>Description</u>
01	Travel from left node to right node
10	Travel form right node to left node
11	Travel in both directions

Left Super Flg	(1 bit) (Bit 6) Set to 1 if the left node of this segment is a supernode.
----------------	---

Right Super Flg	(1 bit) (Bit 7) Set to 1 if the right node of this segment is a supernode.
-----------------	--

The following eight fields occupy one byte:

LNode Reverse	(1 bit) (Bit 0, where 0 is the high order bit) Set to 1 if this represents an aggregated segment, and the Left Node of the aggregated segment is actually the Right Node of the aggregated segment's leftmost constituent segment.
---------------	--

RNode Reverse	(1 bit) (Bit 1) Set to 1 if this represents an aggregated segment, and the Right Node of the aggregated segment is actually the Left Node of the aggregated segment's rightmost constituent segment.
---------------	--

Agg Seg ID Flag	(1 bit) (Bit 2) Set to 1 if this is part of an aggregated segment at a higher layer, and the Agg Seg ID Map and Agg Seg ID Array fields are present in the record. (See Segment Aggregation specification.)
-----------------	---

In Process	(1 bit) (Bit 3) Set to 1 if the record represents in-process data.
------------	--

Shape Count Flag	(1 bit) (Bit 4) Set to 1 if the record contains a Shape Pt Cnt field.
------------------	---

Cond Count Flag	(1 bit) (Bit 5) Set to 1 if the record contains a Cond Count field.
-----------------	---

Multi-Dig Flag	(1 bit) (Bit 6) Set to 1 if this segment is part of a multi-digitized roadway.
----------------	--

Internal Flag	(1 bit) (Bit 7) Set to 1 if the record represents an internal segment of a supernode.
---------------	---

Left Bearing	(1 byte) The Left Bearing and Right Bearing fields contain the bearings of the segment at its left and right nodes. The bearing values are calculated in degrees and scaled down to a range of 0-255. Due north is represented by the value 0. East is the value 64. South is the value 128. West is the value 192.
--------------	---

Right Bearing	(1 byte) See Left Bearing above.
---------------	----------------------------------

The following four fields occupy one byte:

Rank	(3 bits) (Bit 0, where 0 is the high order bit); the highest route calculation layer in which the segment appears.
------	--

<u>Value</u>	<u>Description</u>
000	Layer 0
001	Layer 1

	010	Layer 2
	011	Layer 3
	100	Layer 4
Divider Loc	(3 bits) (Bits 3-5) Bit flags specifying the existence of divider.	
	<u>Value</u>	<u>Description</u>
	000	No divider
	001	Segment is divided
	010	Left node is divided
	100	Right node is divided
Divider Legal	(1 bit) (Bit 6) Set to 1 if divider is legal, otherwise it is physical.	
Detailed City	(1 bit) (Bit 7) Set to 1 if segment was attributed based on detailed city inclusion rules, otherwise, it was attributed based on inter-town inclusion rules.	

The following three fields occupy one byte:

HiLite Layer	Length: three bits; the cartographic Highlighting Layer of this segment.
LHTZDS Flag	Length: one bit. If any of the lengths of the Time Zone, Long Haul or Driving Side fields are non-zero, set to 1, otherwise set to 0.
NavIndex Cnt	Length: four bits; the number of Navigable Feature Index entries in the NavID Array.

The following three fields occupy 0-1 bytes.

Time Zone	This is a field of type <i>IntValP</i> associated with Parcel Descriptor array entries <code>pdRouting_TZ</code> and <code>pdRouting_TZ_2nd</code> .
Long Haul	This is a field of type <i>IntValP</i> associated with Parcel Descriptor array entries <code>pdRouting_LH</code> and <code>pdRouting_LH_2nd</code> .
Driving Side	This is a field of type <i>IntValP</i> associated with Parcel Descriptor array entries <code>pdRouting_DS</code> and <code>pdRouting_DS_2nd</code> .
Left Node ID	(1-2 bytes) (Type 1 Variable Length Unsigned Value) Represents a node in one of the following ways: (1) If less than this parcel's node count, <i>and</i> a bottom layer parcel, the entry is an index to the node record array for this parcel, and (by the addition of the parcel's Minimum Node Entity ID) the node's entity identifier. (2) If less than this parcel's node count, and <i>not</i> a bottom layer parcel, this entry is an index to this parcel's node record offset array, from which both the node's identifier and an offset to the node record can be obtained. (3) If not less than this parcel's node count, this value minus this parcel's node count is an index to this parcel's exception/ghost node offset array, from which both the node's identifier and an offset to the exception/ghost node record can be obtained.
Right Node ID	(1-2 bytes) (Type 1 Variable Length Unsigned Value) Represents a node in one of the following ways:

- (1) If less than this parcel's node count, *and* a bottom layer parcel, the entry is both an index to the node record array for this parcel, as well as (by the addition of the parcel's Minimum Node Entity ID) the node's entity identifier.
- (2) If less than this parcel's node count, and *not* a bottom layer parcel, this entry is an index to this parcel's node record offset array, from which both the node's identifier and an offset to the node record can be obtained.
- (3) If not less than this parcel's node count, this value minus this parcel's node count is an index to this parcel's exception/ghost node offset array, from which both the node's identifier and an offset to the exception/ghost node record can be obtained.

If the next field is present, the Left Super Flg bit must be set to 1; only valid for Layer 0 parcels:

- Left SubNode (2 bytes) The subordinate node ID of the supernode which is the actual left node of the segment.
- (Type 1 Variable Length Unsigned Value) Represents a node in one of the following ways:
- (1) If less than this parcel's node count, the entry is both an index to the node record array for this parcel, as well as (by the addition of the parcel's Minimum Node Entity ID) the node's entity identifier.
 - (2) If not less than this parcel's node count, this value minus the parcel's node count is an index to this parcel's ghost node offset array, from which both the node's identifier and an offset to the ghost node record can be obtained.

If the next field is present, the Right Super Flg bit must be set to 1; only valid for Layer 0 parcels:

- Right SubNode (2 bytes) The subordinate node ID of the supernode; the actual right node of the segment.
- (Type 1 Variable Length Unsigned Value) Represents a node in one of the following ways:
- (1) If less than this parcel's node count, the entry is both an index to the node record array for this parcel, as well as (by the addition of the parcel's Minimum Node Entity ID) the node's entity identifier.
 - (2) If not less than this parcel's node count, this value minus the parcel's node count is an index to this parcel's ghost node offset array, from which both the node's identifier and an offset to the ghost node record can be obtained.

If the following field is present, the Cond Count Flag must be set to 1:

- Cond Cnt (1 byte) Count of Condition records attached to this segment.
- Segment Cost (1-3 bytes) (Type 1 Variable Length Unsigned Value) Estimated time in seconds required to traverse the segment.
- Length (1-3 bytes) (Type 1 Variable Length Unsigned Value) Length of the segment in meters.
- Attrib Index (1-3 bytes) (Type 1 Variable Length Unsigned Value) An index to a Normalized Attribute Array. This is in the range 0-255 if used as an index to the global Normalized Attribute Array.

If larger than 255, its value, minus 256, constitutes an index to this parcel's Local Normalized Attribute Array.

If the next field is present, the Shape Count Flag must be set to 1; only valid for Layer 0 parcels:

Shape Pt Cnt (1-3 bytes) (Type 1 Variable Length Unsigned Value) Number of Delta Shape records associated with this segment.

If the following field is present, the Shape Pt Cnt field must be greater than zero. This field is only valid for Layer 0 parcels:

Delta Shape (2 bytes) Offset of the first delta shape position record for this segment. The number of delta shape position records associated with this segment is equal to the value in the Shape Pt Cnt field.

NavID Array The number of entries is in the NavIndex Cnt field. Each represents one Navigable Feature associated with the segment. Only features attributed as "Name on Road Sign" for this segment are included. If the segment is unnamed or has no names attributed as "Name on Road Sign", this is empty.

NavID Index Type *IntValP* associated with Parcel Descriptor array entries *pdRouting_NavID* and *pdRouting_NavID_2nd*.

Cond IDs (variable) Length is 2 bytes multiplied by the value in Cond Cnt. Each two-byte entry is the entity ID of a condition record.

If the following field is present, the Aggr Segment bit must be set to 1:

Abbr Seg PclID (2 bytes) (See the Segment Aggregation specification.)

If the following two fields are present, the Agg Seg ID Flag bit must be set to 1:

Agg Seg ID Map (1-2 bytes) (See the Segment Aggregation specification.)

Agg Seg ID Array (variable) (See the Segment Aggregation specification.)

Figure 3-18 illustrates the structure of a routing segment record for segments at the bottom layer.

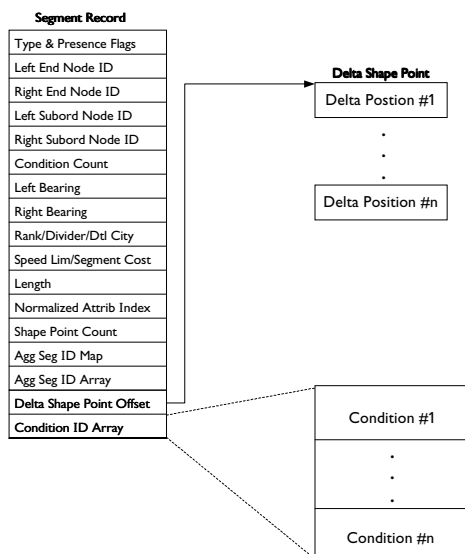


Figure 3-18: Route Calculation Parcel: Bottom Layer Segment Record

Figure 3-19 illustrates the structure of a routing segment record for segments above the bottom layer.

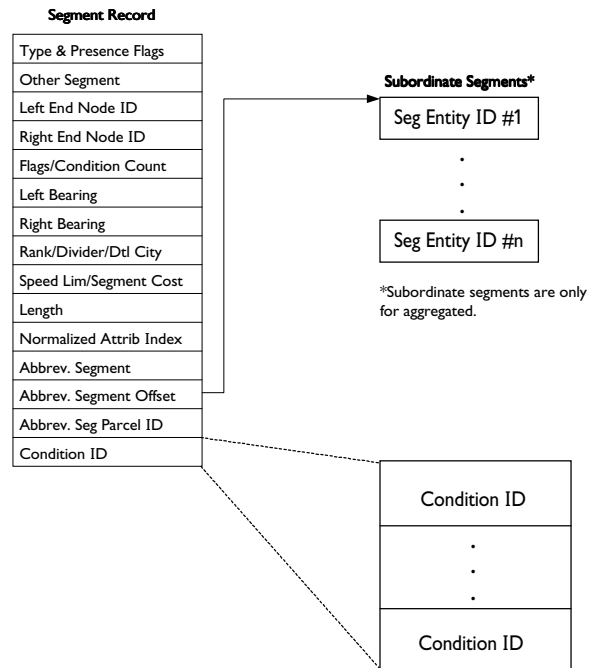


Figure 3-19: Route Calculation Parcel: Segment Record above Bottom Layer

Segment Block Offset Array

Segment records at the bottom layer are accessed through the Segment Block Offset Array. Each entry contains a block offset pointing to the beginning of a block of $2^K + 1$ length fields followed by 2^K segment records. The entity identifier in this case is obtained by adding the index of the record to the minimum segment entity identifier for this parcel.

The length fields are all Type 1 Variable Length Unsigned Values. The first length field is the offset from the beginning of the block to the first segment record in the block. The following 2^K length fields are the lengths of each of the 2^K compressed segment records.

```
BlkOffset_t aSegBlk[]; /* Segment Block Offset Array */
```

Segment Record Offset Array

Segment records above the bottom layer are accessed via the Segment Record Offset Array. Each entry contains both an entity identifier and an offset to the corresponding segment record.

```
RecOffsetEnt_t aSegOffsetArray[]; /* Segment Record Offset Array */
```

External Segment Record Offset Array

External segment records at the bottom layer (those whose left nodes are not within this parcel's boundaries) are accessed through the External Segment Record Offset Array. Each entry contains a three-byte entity identifier, a one-byte Segment Record Spatial Index Bitmask Array, described below, and a two-byte offset to the corresponding segment record.

```
ExtSegOffsetEnt_t aExtSegOffsetArray[]; /* External Segment Record Offset Array */
```

Segment Record Spatial Index Bitmask Array

This array contains a one-byte bitmask for each segment record in the parcel. Each of the eight bits in the bitmask corresponds to a subrectangle of the parcel. The subrectangle and its corresponding index are defined in the internal kd-tree. This array exists only at layer 0.

3.3.4 Segment Aggregation

A segment aggregate represents a set of contiguous regular segments with sufficiently similar characteristics. Aggregated segments help speed route calculation by providing a simplified representation of the road network.

3.3.4.1 Technical Description

Navigable segments of all functional classes are included in the lowest layer of route calculation data. In each succeeding higher layer, the lowest-ranked remaining functional class is dropped. This usually results in the creation of a number of bivalent nodes—intersection nodes between exactly two segments. If all significant attributes for these two segments, relevant to navigation, are equal it is possible to drop the bivalent node. Segments formed this way are called aggregated segments.

The following figure details the segment record fields used for segment aggregation, and shows the relationships among the segment aggregation data:

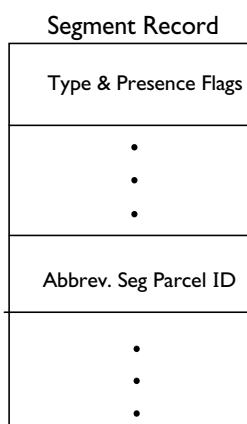


Figure 3-20: Aggregated Segments

3.3.4.2 Physical Storage of Aggregated Segments

When aggregation occurs, segment records and node records internal to the aggregated segment exist in an abbreviated form.

- Each abbreviated segment contains the segment identifier, length, transit time, and bearing.
- Each abbreviated node contains the node identifier and position.

These abbreviated records are accessible only through the aggregated segment record containing the attributes common to the abbreviated segments. It is not possible to access an abbreviated segment or node record within a parcel except through the identifier of the aggregated segment.

3.3.4.3 Aggregated Segment Structures

As previously discussed, set the (1-bit) Aggr Segment field of the Segment Record to 1 if the segment represents an aggregated segment. In this case, the Segment Record contains additional fields which are detailed next.

The associated abbreviated segment and node records are stored in a separate parcel called an abbreviated segment parcel.

Segment Aggregation Fields of the Segment Record

Field	Description
-------	-------------

If the following field is present, the Aggr Segment bit must be set to 1:

Abbr Seg PclID	(2 bytes) An index to this parcel's Parcel ID Table. This identifies the Abbreviated Segment Parcel containing the abbreviated segment/node records for this aggregated segment.
----------------	--

If the following two fields are present, the Agg Seg ID Flag bit must be set to 1:

Agg Seg ID Map	(1 byte) Composed of one or more bitmap components; the maximum number of bitmap components is equal to the Maximum Database Layer (currently 4) minus the current database layer.
----------------	--

Each bitmap component (from high-order to low-order bit) is composed of a string of zero or more bits set to 1, followed by one bit set to 0. However, in the last bitmap component of this field, a bit set to 1 corresponding to the Maximum Database Layer need not be followed by a bit set to 0.

In the following, a bitmap component containing one or more bits set to 1 is called a *positive component*; one containing only a single bit, with that single bit set to 0, is called a *zero component*.

- Each positive component is associated with one entry in the Agg Seg ID Array, and the number of bits set to 1 in that positive component gives the number of consecutive database layers where the associated aggregated segment exists and contains the current segment.
- Each zero component is associated with one database layer, and indicates there is no aggregated segment containing the current segment in that layer.

The database layer associated with the first (highest-order) bitmap component is the layer above the current layer, and subsequent database layer associations go consecutively up from there until the maximum database layer is reached.

Agg Seg ID Array	(variable) (Type 1 Variable Length Unsigned Value) Represents an index to the Aggregated Segment Entity ID Array for this parcel. The number of entries is the number of positive components in the Agg Seg ID Map field. The first entry is for the first (highest-order) positive component in the Agg Seg ID Map field, the second is for the second positive component in the Agg Seg ID Map field, etc.
------------------	--

Abbreviated Segment Parcel Structure

For segment records representing aggregated segments, the associated abbreviated segment/node records are stored in the abbreviated segment parcel. It contains aggregated segment records (defined below) for each segment record representing an aggregated segment. Each aggregated segment record contains the corresponding abbreviated segment/node array.

Abbreviated Segment Parcel Header Information

The following structure defines the initial, fixed portion of the parcel data for the abbreviated segment parcels.

```
/*
** Abbreviated Segment Parcel Header
*/
```

```

typedef struct
{
    SptlPclHdr_t  spatialHdr;

    BlkDesc2_t    segmentDesc;    /* Segment block offset descriptor */
    Ushort_t      usMaxAbbrevSegs; /* Max Abbrev Segs per Aggregated Seg */
    Uchar_t       ucaReserved[2]; /* Maintain 4-byte alignment */
}
AbbrSegmentParcelHeader_t;

```

Aggregated Segment Record Offset Array

Aggregated segment records are accessed via the Aggregated Segment Record Offset Array. Each entry contains both an entity identifier and an offset to the corresponding aggregated segment record.

```

RecOffsetEnt_t aSegOffsetArray[]; /* Aggregated Segment Record Offset
                                   Array */

```

Aggregated Segment Record

Field	Description
Abbr Seg Cnt	(variable) (Type 1 Variable Length Unsigned Value) Represents the number of abbreviated segments associated with this aggregated segment. The number of associated abbreviated nodes is one greater than this value.
Abbrev Array	(variable) The number of records in this array is $2N+1$, where N is the value in the Abbr Seg Cnt field. This array contains two types of records: Abbreviated Segment records, and Abbreviated Node records. Abbreviated Segment and Node records always alternate. First, an Abbreviated Node record, followed by an Abbreviated Segment record, alternating until the last record is reached; it is always an Abbreviated Node record. The layer 0 routing Parcel ID of each abbreviated record in this list is determined as follows: (1) If a segment or node contains a Parcel ID Index field, that segment or node has that Layer 0 Parcel ID. (2) If a segment or node record does not contain a Parcel ID Index, the record is assumed to have the same Layer 0 Parcel ID as the previous segment or node in the list that did contain a Parcel ID Index. (The first node record always contains a Parcel ID Index.) The Abbr Seg Cnt field in the aggregated segment record contains the number of Abbreviated Segment records in the corresponding list of abbreviated segment/node records; it is equal to one less than the number of Abbreviated Node records in this list.

Abbreviated Segment Record

Field	Description
The following two fields (Pcl ID Flag, Seg ID) occupy three bytes:	
Pcl ID Flag	(1 bit) (Bit 0, where 0 is the high order bit) If the record contains a Parcel ID field, set to 1.
Seg ID	(23 bits) (Bits 1-23) The entity ID of the abbreviated segment.
Fraction	(1 byte) A number in the range 0-255 representing both the length of the seg-

ment and the segment cost as a fraction (N/256) of the corresponding field of the aggregated segment.

Layer 0 Parcel ID (1-3 bytes) (Type 1 Variable Length Unsigned Value) An index to this parcel's Parcel ID array. It represents the Route Calculation parcel *at layer 0* containing this segment. If present, the Pcl ID Flag bit must be set to 1.

Abbreviated Node Record

Field	Description
-------	-------------

Node ID	(3 bytes) Contains the entity ID of the abbreviated node.
---------	---

The following six fields (Max Significance, In Parcel, Reverse Flag, Pcl ID Flag, Z-level Present, Supernode Flag) occupy one byte:

Max Significance	(3 bits) (Bits 0-2, where 0 is the high order bit) Highest layer at which this node has three or more segments connected to it, its adjacent segments have a mismatch on at least one aggregation attribute, or the node is a dead end.
------------------	---

<u>Value</u>	<u>Description</u>
--------------	--------------------

000	Layer 0
-----	---------

001	Layer 1
-----	---------

010	Layer 2
-----	---------

011	Layer 3
-----	---------

100	Layer 4
-----	---------

In Parcel	(1 bit) (Bit 3) Set to 1 if the node is within the current parcel's bounding rectangle.
-----------	---

Reverse Flag	(1 bit) (Bit 4) Set to 1 if this node represents the Right Node of the <i>following</i> abbreviated segment (in the list of abbreviated nodes and segments). If <i>not</i> set, this node is the following abbreviated segment's Left Node.
--------------	---

Pcl ID Flag	(1 bit) (Bit 5) Set to 1 if the record contains a Parcel ID field.
-------------	--

Z-level Present	(1 bit) (Bit 6) Set to 1 if the record contains a Z-level field.
-----------------	--

Supernode Flag	(1 bit) (Bit 7) Set to 1 if this record represents a supernode.
----------------	---

Latitude	(variable) (Either 4-byte signed integer or Type 4 Variable Length Unsigned Value) If this node's In Parcel Flag is set, this field is 2-3 bytes long. To compute the value for the latitude field: subtract the minimum latitude (in NTUs) of the parcel's bounding rectangle from the latitude value of the node and right shift the result by a number of bits equal to the parcel's scale factor.
----------	---

If this node's In Parcel Flag is not set, this field is four bytes in length and contains the node's latitude in NTUs.

Longitude	(variable) (Either 4-byte signed integer or Type 4 Variable Length Unsigned Value) If this node's In Parcel Flag is set, this field is 2-3 bytes long. To compute the value for the longitude field: subtract the minimum longitude (in NTUs) of the parcel's bounding rectangle from the longitude value of the node and right shift the result by a number of bits equal to the parcel's scale factor.
-----------	--

If this node's In Parcel Flag is not set, this field is four bytes in length and con-

tains the node's longitude in NTUs.

Layer 0 Parcel ID (1-3 bytes) (Type 1 Variable Length Unsigned Value) An index to this parcel's Parcel ID array. This represents the Route Calculation parcel at *layer 0* containing this node. Set the Parcel ID flag to 1 so this field can be present.

Z-level (1 byte) If present, the Z-level Present flag must be set to 1; the value (-5 to 4). When not present, the value defaults to zero.

3.4.5 Exception/Ghost Segments and Nodes

An exception segment's left endpoint (node) is in a different layer 0 route calculation parcel than its right node. The segment exists as a regular segment in the parcel that contains its left node. It exists as an exception segment in the parcel that contains its right node but not its left node. A ghost segment is a segment that passes through a parcel but has no node in that parcel. Whenever a segment is contained in a parcel, as a regular segment or an exception segment, if either of its endpoint nodes is not contained in that parcel, that node is represented in the parcel as a ghost node.

3.4.5.1 Technical Description

Segment entity identifiers are unique within the entire region, and every segment entity has the same entity identifier wherever it occurs. This convention makes it easier to correlate instances of the same segment occurring in different layers or neighboring parcels. At the bottom layer, segment entity identifiers are assigned consecutively within a parcel. This convention saves space in the parcel.

These two conventions result in the need to treat ghost segments and some exception segments differently from other segments. The same segment identifier cannot be part of two or more different consecutive ranges of identifiers for the two or more parcels in which the segment is found. This is done by assigning a segment a unique, consecutive identifier within the bottom layer parcel in which its Left Node is found. In other bottom layer parcels containing this segment, it is referred to as an external segment, and is stored in a separate external segment record array.

An exception segment in a bottom layer parcel is stored in the parcel's external segment array. A ghost segment in a bottom layer parcel is also stored in the parcel's external segment array. Above the bottom layer, the consecutive identifier criterion does not apply, and all segments are stored in the same record array.

The following example illustrates the duplication of exception and ghost segments in the parcels that they intersect:

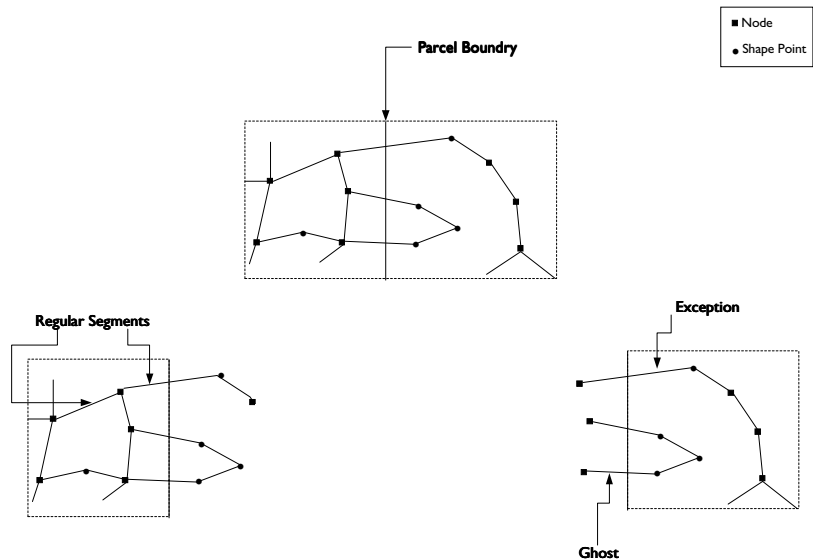


Figure 3-21: Exception and Ghost Segments

Exception Segments

All information for exception segments is duplicated in each of the two parcels in which the segment has a node. Both parcels have:

- A record for the segment
- Node and shape point data for the segment.

When either or both the endpoint nodes for a segment fall outside the parcel's boundaries, they are carried in an abbreviated form (called a Ghost Node record) in that parcel. These node records carry only position information and the location of the full node record.

Ghost Segments

Ghost segments are treated the same way as exception segments. Parcels above the bottom layer do not carry ghost segments.

3.4.5.2 Ghost Node Record

The ghost node record is defined as a character array of variable length. Since fields are not at fixed positions, this record is not described by a C-language structure.

Field	Description
The following four fields occupy seven bytes:	
Z-level	(4 bits) (Bits 0-3 where 0 is the high order bit) The Z-level value (-4 to 5).
In Process	(1 bit) Bit 4) Set to 1 if the record represents in-process data.
Latitude	(25 bits) (Bits 5-29) To compute the value for the latitude field: subtract the minimum latitude (in NTUs) of the parcel's bounding rectangle from the latitude value of the node and right shift the result by a number of bits equal to the parcel's scale factor.
Longitude	(26 bits) (Bits 30-55) To compute the value for the longitude field: subtract the minimum longitude (in NTUs) of the parcel's bounding rectangle from the longitude value of the node and right shift the result by a number of bits equal to the parcel's scale factor.

The following three fields (Rank, Attributes, Max Signif) occupy one byte:

Rank (3 bits) (Bits 0-2 where 0 is the high order bit) Values 0 (lowest layer) to 4 (highest layer) in which this node appears.

<u>Value</u>	<u>Description</u>
000	Layer 0
001	Layer 1
010	Layer 2
011	Layer 3
100	Layer 4

Attributes (2 bits) (Bits 3-4)

<u>Bit</u>	<u>Attribute</u>
Bit 3	Node is a mini-roundabout.
Bit 4	Node is a supernode.

Max Signif (3 bits) (Bits 5-7) Highest layer at which this node has three or more segments connected to it, its adjacent segments have a mismatch on at least one aggregation attribute, or the node is a dead end.

<u>Value</u>	<u>Description</u>
000	Layer 0
001	Layer 1
010	Layer 2
011	Layer 3
100	Layer 4

Parcel ID (Type 1 Variable Length Unsigned Value) An index to the Parcel ID table for the parcel in which the ghost node resides.

Ghost Node Record Offset Array

Ghost Node records are accessed via the Ghost Node Record Offset Array, where each entry contains both an entity identifier and an offset to the corresponding ghost node record.

```
RecOffsetEnt_taGhostOffsetArray[]; /* Ghost Node Record Offset Array */
```

3.4.6 Routing Data Shape Points

Shape point records store positional information in terms of longitude, latitude, and Z-level, that define the shape of a segment. Shape points represent bends or kinks in a segment or intersections of segments at different grades, and do not include segment end nodes.

3.4.6.1 Technical Description

The delta shape point record is defined as a character array. There is no Delta Shape Point Offset Array because all records in this array are accessed directly as offsets. The first delta shape point for a segment is accessed via the delta shape field of the segment record. Subsequent delta shape points for a segment follow sequentially. This record is not compressed.

3.4.6.2 Routing Shape Point

Field	Description
-------	-------------

The following four fields take up 2-5 bytes:

Internal Pt	(1 bit) (High order bit) Set to 1 if this point is internal to the Routing Parcel only. It is present only because its offset from the previous record is outside the range -32768 to +32767.
Z-level Present	(1 bit) Set to 1 if this record contains a Z-level field. If the Z-level field is not present, the value of this record's Z-level defaults to zero.
Latitude	(7 - 19 bits) (Type 5 Variable Length Signed Value) Represents the scaled change in latitude between this point and either the preceding shape point or node position. To compute the value for the latitude field: subtract the minimum latitude (in NTUs) of the parcel's bounding rectangle from the latitude value of the node and right shift the result by a number of bits equal to the parcel's scale factor.
Longitude	(7 - 19 bits) (Type 5 Variable Length Signed Value) Represents the scaled change in longitude between this point and either the preceding shape point or node position. To compute the value for the longitude field: subtract the minimum longitude (in NTUs) of the parcel's bounding rectangle from the longitude value of the node and right shift the result by a number of bits equal to the parcel's scale factor; always ends on a byte boundary.

If the following field is present, the Z-level Present bit must be set to 1:

Z-level	(1 byte) The Z-level value (-5 to 4). When not present, the Z-level defaults to zero.
---------	---

3.4.6.3 Data Compilation Guidelines

Shape point information is present only for the bottom layer of data and information is not stored in upper layer routing parcels.

3.4.7 Routing Condition Records

Routing condition records store information about restrictions or additional attributes associated with a segment.

3.4.7.1 Technical Description

Routing condition information includes a condition type defining the type of information stored in the record, and modifier fields providing additional information about the condition.

Conditions can specify information for a single segment or a series of connected segments referred to as a maneuver. For example, a road construction condition is defined as a single segment, but a restricted driving maneuver condition is defined in terms of two or more connected segments. The condition record also contains scope information to specify the lanes of the segment affected by the condition and the types of vehicles affected by the condition.

The Routing Segment Record contains a field for the number of Date/Time records associated with it, and an array containing the offset to each of these Date/Time records. See the "Routing Date/Time Records" section.

3.4.7.2 Data Reference

The following diagrams illustrate the relationships of the condition data within the route calculation parcel.

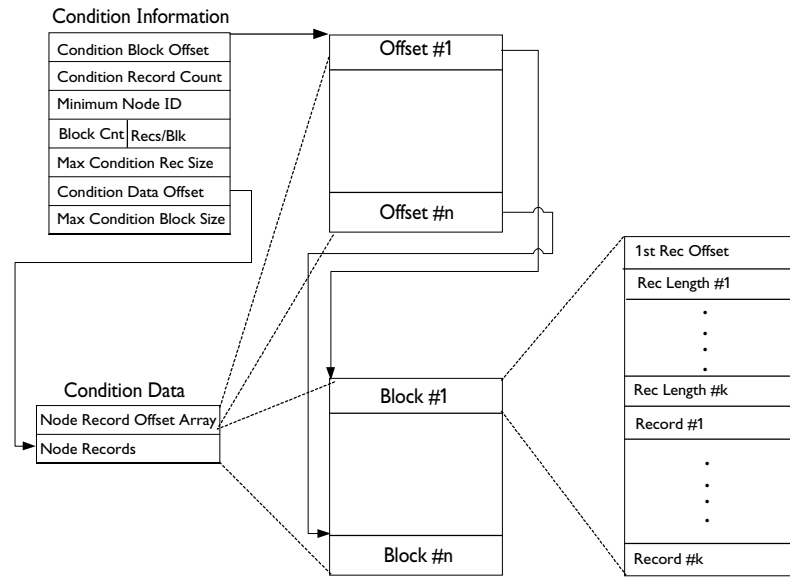


Figure 3-22: Route Calculation Parcel: Condition Data

Figure 3-23 illustrates the layout of the Condition record:

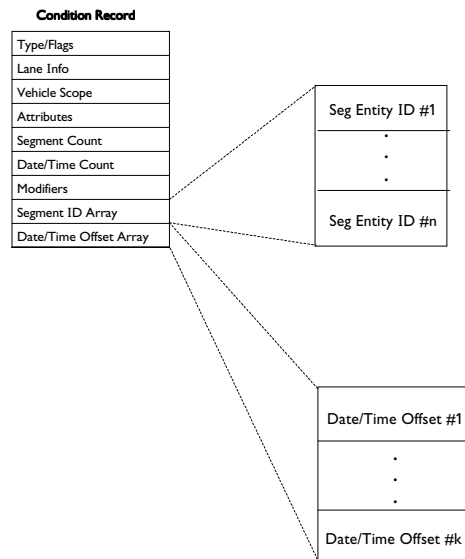


Figure 3-23: Route Calculation Parcel: Condition Record

3.4.7.3 Condition Record

The condition record is defined as a character array of variable length. Since fields are not at fixed positions, this record is not described by a C-language structure.

Field	Description
Cond Type	(2 bytes) Numeric code indicating the type of condition:
<u>Value</u>	<u>Description</u>
1	Access restriction

	2	Carpool restriction
	3	Construction status
	4	Direction of travel
	5	Gate
	6	Instruct flag
	7	Restricted driving maneuver
	8	Tollbooth
Presence Bits	(1 byte)	Bit flags indicating whether certain other fields in the record are present. Set to 1 if the corresponding field is present in the record.

<u>Bit</u>	<u>Description</u>
0x00	Modifier1 field is present
0x01	Modifier2 field is present
0x02	Modifier3 field is present
0x03	Modifier4 field is present

Lane Info	(4 bytes)	The condition's segment lanes can be expressed as relative or absolute lane information. If present, the Cond Attrs field relative lane information (0x0001) or absolute lane information flags must be set to 1. If the relative lane information flag is set, relative lane information is stored separately for each direction of travel on the road.
-----------	-----------	--

- The first and second bytes contain information for the lanes on the upper half of the road that travel in the same direction.
- The third and fourth bytes contain information for the lanes on the lower half of the road that travel in the opposite direction.
- For a one way road, only the first and second bytes are used.

Relative lane information is expressed in terms of the side of the road and lanes affected by the condition.

- The side of the road is described as "inner" or "outer", where "inner" is the side closest to the lanes traveling in the opposite direction.
- A condition can be applied, for example, to the "inner 2 lanes" or the "outer 3 lanes." The first and second bytes contain the side and number of lanes affected by the condition on the upper half of the road. The third and fourth contain the side and number of lanes affected by the condition on the lower half of the road. For one way roads, the "upper" lanes are considered the "inner" lanes regardless of the direction of travel.

The valid values for the first and third bytes are:

1	Inner Lanes
2	Outer Lanes

If the Cond Attrs field specifies absolute lane information (0x0002), then:

- The four bytes contain absolute lane information expressed in terms of a 32-bit bitmap.
- Bits set represent the lanes for which the condition applies. For a road with travel in both directions, the left most 16 bits of the bitmap (bits 31 - 16) correspond to the upper half of the road; the right most 16 bits (bits 15 - 0) correspond to the lower half of the road. The division between the two directions of travel is between bits 16 and 15. The bits closest to this center division on both sides are used when the number of bits exceeds the number of actual lanes. For example, a four lane road with two lanes in each direction of travel uses bits 17, 16, 15, and

14, with bit 17 representing the uppermost lane. A one way road uses as many of the most significant bits as there are actual lanes. For example, a three lane one way road uses bits 31, 30, and 29, with bit 31 representing the upper-most lane.

Vehicle Scope (2 bytes)

<u>Value</u>	<u>Description</u>
0x0001	Automobiles
0x0002	Buses
0x0004	Taxis
0x0008	Carpools
0x0010	Pedestrians
0x0020	Bicycles
0x0040	Trucks
0x0080	Through traffic
0x0100	Deliveries
0x0200	Emergency vehicles

Cond Attrs (2 bytes)

<u>Value</u>	<u>Description</u>
0x0001	Relative lane information exists
0x0002	Absolute lane information exists
0x0004	Date/time information exists
0x0008	Additional maneuver segments exist

Seg Cnt (1 byte) Number of segments associated with this condition.

DtTm Cnt (1 byt) Number of date-time records attached to this condition.

The data format of the following four modifier fields are defined following the Condition record definition.

Modifier1 (2 bytes) If present, bit 0 of the Presence Bits must be set to 1.

Modifier2 (2 bytes) If present, bit 1 of the Presence Bits must be set to 1.

Modifier3 (4 bytes) If present, bit 2 of the Presence Bits must be set to 1.

Modifier4 (4 bytes) If present, bit 3 of the Presence Bits must be set to 1.

SegID Array Length calculated as follows: Number of entries is equal to the value in the Seg Cnt field. Each entry is 1-2 bytes or 5-7 bytes in length:

(1) If the first byte of an entry is not equal to zero, the field is 1-2 bytes in length, and contains a Type 1 Variable Length Unsigned Value 1 greater than an index to one of the segment arrays for this parcel:

(1a) If the value *minus 1* is less than this parcel's segment count, and this is a bottom layer parcel, the entry's value *minus 1* is both an index to the segment record array for this parcel, and (by the addition of the parcel's Minimum Segment Entity ID) the segment's entity identifier.

(1b) If the value *minus 1* is less than this parcel's segment count, and this is not a bottom layer parcel, the entry's value,

minus 1, is an index to this parcel's segment offset array, from which both the segment's identifier and an offset to the segment record can be obtained.

(1c) If the value *minus 1* is not less than this parcel's segment count, the entry's value, *minus 1*, minus the parcel's segment count represents an index to the parcel's external segment offset array, from which both the segment's identifier and an offset to the segment record can be obtained.

- (2) If the first byte is equal to zero, the following three bytes contain the entity identifier of the segment. The bytes following the entity identifier reference the segment's parcel identifier. These bytes are a Type 1 Variable Length Unsigned Value, and constitute an index to this parcel's Parcel ID Array.

DtTm Array Length: Two bytes multiplied by the value in the DtTm Cnt field. Each entry is the offset of a date-time record.

Condition Block Offset Array

Condition records are accessed through the Condition Block Offset Array. Each entry contains a block offset pointing to the beginning of a block of $2^K + 1$ length fields followed by 2^K Condition records. The entity identifier equals the index of the record (within the parcel).

The length fields are all Type 1 Variable Length Unsigned Values. The first length field is the offset from the beginning of the block to the first condition record in the block. The following 2^K length fields are the lengths of each of the 2^K compressed condition records.

```
BlkOffset_t aConditionBlk[]; /* Condition Block Offset Array */
```

3.4.7.4 Condition Modifier Field Formats

The Modifier1 and Modifier2 fields store 16-bit condition modifier values. The Modifier3 and Modifier4 fields store 32-bit condition modifier values. The number of modifier fields used and the contents of these fields varies based on the type of condition being defined.

Additional information about each of the condition types is contained in the modifier fields. This table summarizes the use of the modifiers by condition type:

Condition Type	Modifier 1	Modifier 2	Modifier 3	Modifier 4
Access Restriction	n/a	n/a	n/a	n/a
Carpool Restriction	Minimum Passengers	n/a	n/a	n/a
Construction Status	Construction Type	n/a	n/a	n/a
Direction of Travel	Direction	n/a	n/a	n/a
Gate	Gate Type	n/a	n/a	n/a
Instruction Flag	n/a	n/a	n/a	n/a
RDM	RDM Type	n/a	n/a	n/a
Toll Booth	n/a	n/a	Toll Cost	n/a

Valid Modifier1 values for Construction Type:

- 1 Open to traffic
- 2 Closed to traffic

Valid Modifier1 values for Direction are:

- 1 From left node
- 2 Toward left node
- 3 Both directions

Valid Modifier1 values for Gate Type are

- 1 Key access
- 2 Permission required
- 3 Emergency vehicle access

Valid Modifier1 values for RDM Type are.

- 1 Legal
- 2 Physical
- 3 Logical

3.4.8 Routing Date/Time Records

The date/time record stores date and time information associated with a condition or a POI, including:

- Time periods during which a condition is in effect.
- Time periods during which a POI is open for public use.

3.4.8.1 Technical Description

The date/time record is a character array of variable length. All date/time records are accessed directly as offsets to the parcels. The maximum length for this record is 16 bytes.

3.4.8.2 Data Reference

The dateTimeDesc field in the routing parcel header is a BlkDesc3_t structure pointing to the date/time data. Additionally, the DtTm Array of the condition record contains offsets to date/time records.

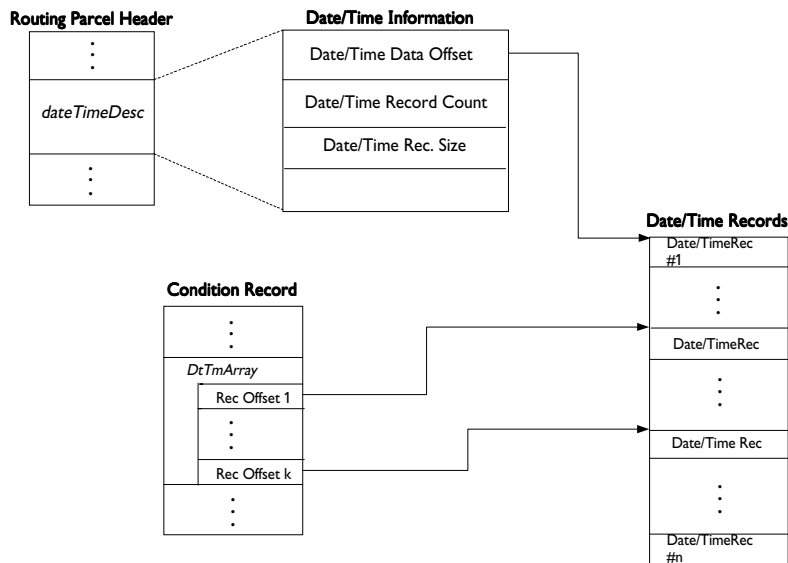


Figure 3-24: Route Calculation Parcel: Data Relationship for Date/Time Data

3.4.8.3 Date/Time Record

Field	Description
Presence Bits	(1 byte) Flags indicating the presence or absence of other fields in the record. Set to 1 if the corresponding field is to be present in the record.
<u>Bit</u>	<u>Meaning if set to 1</u>
0x01	Start Time and End Time fields are present.
0x02	End Date Val3 field is present.
0x04	End Date Val2 field is present.
0x08	End Date Val1 field is present.
0x10	Start Date Val3 field is present.
0x20	Start Date Val2 field is present.
0x40	Start Date Val1 field is present.
0x80	Unused.

The following two fields occupy one byte:

Start Type	(4 bits) (Bits 0-3, where 0 is the high order bit) Start Date Type. Start Type can have the following values:
<u>Value</u>	<u>Description</u>
0	Date not applicable
1	Day of week bitmap
2	Absolute date
3	Day of year

	4	Day of month
	5	Day of week of month
	6	Day of week of year
	7	Week of month
	8	Week of year
	9	Month of year
	10	Day of month of year
	11	Day of month of year
End Type	(4 bits) (Bits 4-7) End Date Type. End Type can have the following values:	
	<u>Value</u>	<u>Description</u>
	0	Date not applicable
	1	Day of week bitmap
	2	Absolute date
	3	Day of year
	4	Day of month
	5	Day of week of month
	6	Day of week of year
	7	Week of month
	8	Week of year
	9	Month of year
	10	Day of month of year
	11	Day of month of year
Attributes	(1 byte) Date Attributes. The valid bit flags are:	
	<u>Value</u>	<u>Description</u>
	0x01	Exclude the date range specified by startDate and endDate
	0x02	Time period is from dawn to dusk
	0x04	Time period is from dusk to dawn

If the following is present, bit 0x40 of the Presence bits must be set to 1:

Start Date Val1 (1 byte) Start Date Value 1. (See below.)

If the following is present, bit 0x20 of the Presence bits must be set to 1:

Start Date Val2 (1 byte) Start Date Value 2. (See below.)

If the following is present, bit 0x10 of the Presence bits must be set to 1:

Start Date Val3 (2 bytes) Start Date Value 3. (See below.)

If the following is present, bit 0x08 of the Presence bits must be set to 1:

End Date Val1 (1 byte) End Date Value 1. (See below.)

If the following is present, bit 0x04 of the Presence bits must be set to 1:

End Date Val2 (1 byte) End Date Value 2. (See below.)

If the following is present, bit 0x02 of the Presence bits must be set to 1:

End Date Val3 (2 bytes) End Date Value 3. (See below.)

If the following two fields are present, bit 0x01 of the Presence Bits must be set; these two occupy three bytes:

Start Time (12 bits) (Bits 0-11, where bit 0 is the high order bit) Starting time in 24- hour format (hhmm), expressed in terms of local time, not Greenwich Mean Time (GMT).

End Time (12 bits) (Bits 12-23) Ending time in 24-hour format (hhmm), expressed in terms of local time, not Greenwich Mean Time (GMT).

TZ Index (0-1 byte) A field of type *IntValP* associated with Parcel Descriptor array entries *pdRouting_TZ* and *pdRouting_TZ_2nd*. When present, this occupies the most significant bits of the byte, and the remaining bits are unused.

Start and End Date Values

The following table indicates which start/end date values are valid for each date type. This table is applicable for both start dates and end dates:

<u>Type</u>	<u>Date Value 1</u>	<u>Date Value 2</u>	<u>Date Value 3</u>
0	n/a	n/a	n/a
1	Day Bitmap	n/a	n/a
2	Month of Year	Day of Month	Year
3	n/a	n/a	Day of Year
4	Day of Month	n/a	n/a
5	Day Bitmap	Week of Month	n/a
6	Day Bitmap	Week of Year	n/a
7	Week of Month	n/a	n/a
8	Week of Year	n/a	n/a
9	Month of Year	n/a	n/a
10	Day of Month	Month of Year	n/a
11	Day Bitmap	Week of Month	Month of Year

The next table lists the domain for each date value:

<u>Value</u>	<u>Domain</u>
Day Bitmap	See day of week bit flags defined below
Day of Month	-31 to 31 (excluding 0)
Day of Year	-366 to 366 (excluding 0)
Week of Month	-5 to 5 (excluding 0)
Week of Year	-53 to 53 (excluding 0)
Month of Year	-12 to 12 (excluding 0)
Year	1 to 9999 (century and year)

Note: Negative values indicate a date value relative to the end of a particular timeframe (e.g., end of month). The record value is computed as follows:

$$\text{record value} = (\text{end of time frame} - \text{actual value} + 1) * -1$$

The following bit flags are used for the day of week bitmap:

<u>Value</u>	<u>Description</u>
0x01	Sunday
0x02	Monday
0x04	Tuesday
0x08	Wednesday
0x10	Thursday
0x20	Friday
0x40	Saturday

3.5 Cartographic Data

There are four types of cartographic entities: Point, Polyline, Polygon, and Name Text. The Point cartographic entity represents POIs and other single point features, such as tollbooths. The Polyline cartographic entity represents linear features such as roads and railroads. The Polygon cartographic entity represents polygonal features such as parks and lakes. The Name Text cartographic entity contains feature name text and text placement information used to position the text on a map display.

3.5.1 Technical Description

There are four types of cartographic entities: the Point, Polyline, and Polygon, and Name Text.

- **Point** The Point entity stores the location of the point, which is expressed in terms of longitude and latitude.
- **Polyline** The set of points defining the polyline are stored as individual cartographic delta positions, referenced by the polyline entity.
- **Polygon** The set of points defining the polygon are stored as individual cartographic delta positions, referenced by the polygon entity. To aid in polygon filling, these points are referenced in counterclockwise order for the outer boundary of the polygon and clockwise order for any enclaves.
- **Name Text** The Name Text cartographic entity contains feature name text and text placement information used to position the text on a map display.

Each of these entities contain attributes specific to that particular type of entity. The attributes provide the application with the information it needs to determine the display characteristics of the entity (e.g., line width, color, fill pattern, icon, etc.).

A parcel is populated with up to a set limit of data which is subdivided or subparceled using an internal (or local) KD-tree.

The internal KD-tree node has the following 2-byte control bit structure followed by up to 14 optional bytes:

Bit	Definition
Bit 0	Vertical Cut
Bit 1	Horizontal Cut
Bits 3-7	Location of Cut
Bit 8	Indicate if KD-tree Child Offsets present
Bit 9	Polyline record ID and Count present
Bit 10	Polygon record ID and Count present

Bit 11 Point record ID and Count present

The remaining 14 bytes of the KD-tree indicate the parcel structure (for each of the following that is present):

2 bytes	Left/Right Child offsets
2 bytes	Polyline Record count
2 bytes	Polyline Initial Entity ID
2 bytes	Polygon Record count
2 bytes	Polygon Initial Entity ID
2 bytes	Point Record count
2 bytes	Point Initial Entity ID

3.5.1.1 Cartographic Parcel Layers

The same parcelization method used for route calculation parcels is used to create cartographic parcels. Cartographic data is layered and currently has the same number of layers as route calculation data.

These parcels contain map display data including points, polylines, polygons and feature name text. Delta shape points are associated with the polylines and polygons.

Data records within cartographic parcels are accessed in blocks in the same way as records (other than segments and nodes) in the route calculation parcels.

3.5.2 Cartographic Parcel Header Information

Cartographic parcel headers contain data information needed to identify and construct a polygon, polyline, or point.

3.5.2.1 Technical Description

Polylines, polygons, and points are all carto entities. They can not have overlapping ID ranges within a parcel. The first polygon has entity ID 0. The entity IDs of the first polyline and first point are stored in each parcel's header.

3.5.2.2 Data Reference

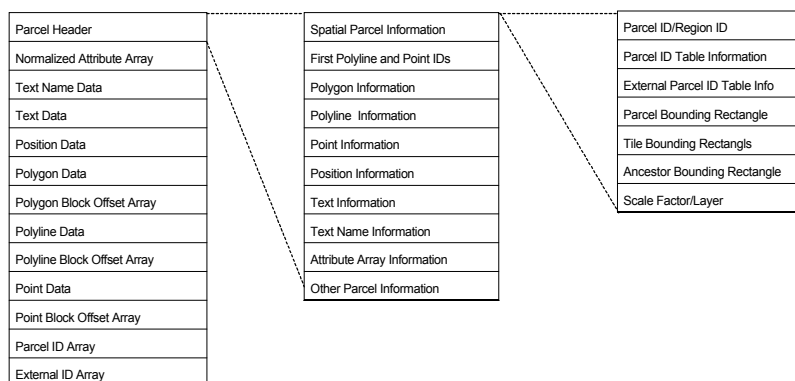


Figure 3-25: Cartographic Parcel Header Information

3.5.2.3 Parcel Header Information

The following structure defines the initial, fixed portion of the parcel data.

```

/*
**Cartographic Parcel Header
*/
typedef struct
{
    SptlPclHdr_tspatialHdr;

    EntityID_t  firstPolylineId;
    EntityID_t  firstPointId;
    BlkDesc_t   polygonDesc;          /* Area (polygon) block descriptor */
    BlkDesc_t   lineDesc;             /* Polyline block descriptor */
    BlkDesc_t   pointDesc;            /* Point block descriptor */
    BlkDesc3_t   positionDesc;         /* Position block descriptor */
    BlkDesc3_t   textDesc;             /* Text block descriptor */
    BlkDesc3_t   textNameDesc;         /* Text Name block descriptor */
    BlkDesc4_t   attribArrayDesc;      /* Local Normalized Attribute Array */
    BlkDesc4_t   longHaulAttrDesc;     /* Long Haul Attr Value array descriptor */
    BlkDesc4_t   polygonLHBitmapDesc; /* Polygon Long Haul Attrib. Bitmap */
    BlkDesc4_t   polylineLHBitmapDesc; /* Polyline Long Haul Attrib. Bitmap */
    BlkDesc4_t   pointLHBitmapDesc;    /* Point Long Haul Attribute Bitmap */
    Uchar_t      ucMaxPosPerPolygon;    /* Max position records for a polygon in this parcel */
    Uchar_t      ucMaxPosPerPolyline;  /* Max position records for a polyline in this parcel */
    Uchar_t      ucMaxTextPerPolygon;   /* Max Text records per polygon */
    Uchar_t      ucMaxTextPerPolyline; /* Max Text records per polyline */
    Uchar_t      ucaReserved[2];       /* Maintain 4-byte alignment */
}
CartoParcelHeader_t;

```

Per-parcel Data Arrays

The locations in the parcel of the following arrays, and the array sizes, are defined in the parcel header:

```

Ushort_t      usaInternalKdTable[]; /* Internal KD-Tree table */
ParcelID_t     aPclId[];           /* Parcel ID Array */
Uchar_t        ucaAttribArray[];   /* Local Normalized Attribute Array */
ExternalPcl_t  aExtPcl[];          /* External parcel array, usually not present */
Uchar_t        ucaLHAttribArray[]; /* Array of one-byte long haul attribute */
values */
Uchar_t        ucaPolygonLHBitmap[];
/* Indexed by Polygon entity ID;
** Each entry is polygonLHBitmapDesc.ucCount bits
** long, which is also equal to the value in the
** parcel descriptor entry with index equal to
** pdCarto_PolygonLHIdxBitLen.
** Entry value = index into ucaLHAttribArray[] */

```

```

Uchar_t      ucaPolylineLHBitmap[];
              /* Indexed by 0-based relative polyline index;
              ** Each entry is polylineLHBitmapDesc.ucCount bits
              ** long, which is also equal to the value in the
              ** parcel descriptor entry with index equal to
              ** pdCarto_PolylineLHIdxBitLen.
              ** Entry value = index into ucaLHAttribArray[] */

Uchar_tucaPointLHBitmap[];
              /* Indexed by 0-based relative Point index;
              ** Each entry is pointLHBitmapDesc.ucCount bits long,
              ** which is also equal to the value in the parcel
              ** descriptor entry with index equal to
              ** pdCarto_PointLHIdxBitLen.
              ** Entry value = index into ucaLHAttribArray[] */

```

3.5.3 Cartographic Polylines

The polyline record represents linear map display features such as roads, walkways, and railroads.

3.5.3.1 Technical Description

The full format polyline record is a character array of variable length.

The short format polyline record is a character array of variable length, used to show an overpass or bridge as a short line.

3.5.3.2 Data Reference

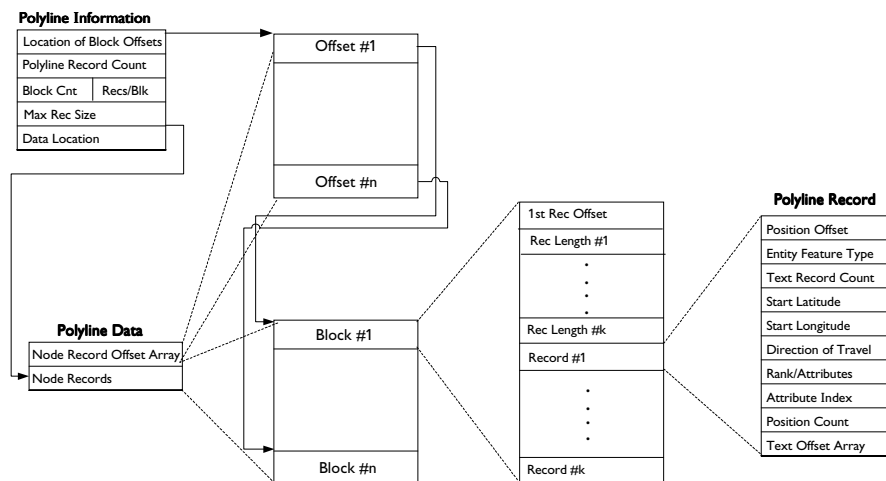


Figure 3-26: Cartographic Parcel Polyline Data

3.5.3.3 Polyline Record (Full Format)

Field (Length) Description

The following two fields (Record Format, Position Offset) occupy two bytes:

Record Format	(1 bit) (High order bit) Set to 0 for a Full Format polyline record.
Position Offset	(15 bits) (Bits 1 - 15) Offset of the first position record defining this polyline.
Text Rec Cnt	(1 byte) Count of the Text records attached to this polyline.
Entity Type	(1-3 bytes) (Type 1 Variable Length Unsigned Value) Type of feature the polyline represents. Values are:

<u>Value</u>	<u>Description</u>
0	Navigable road or walkway
2	Ocean or coastline
3	River
4	Canal or water channel
25	Railroad
31	Junction

Start Latitude	(2-3 bytes) (Type 4 Variable Length Unsigned Value) To compute the value for the latitude field: subtract the minimum latitude (in NTUs) of the parcel's bounding rectangle from the latitude value of the initial point defining this polyline, and right shift the result by a number of bits equal to the parcel's scale factor.
----------------	---

Start Longitude	(2-3 bytes) (Type 4 Variable Length Unsigned Value) To compute the value for the longitude field: subtract the minimum longitude (in NTUs) of the parcel's bounding rectangle from the longitude value of the initial point defining this polyline, and right shift the result by a number of bits equal to the parcel's scale factor.
-----------------	--

The following five fields occupy one byte:

Navigable	(1 bit) (High order bit) Set to 1 if the polyline represents a navigable feature.
-----------	---

Note: River or Administrative area is set to non-navigable.

Dir of Travel	(2 bits) (Bits 1 - 2) Set to 0 if the Entity Type field indicates this polyline represents a non-navigable feature.
---------------	---

These indicate the direction of travel for the road:

<u>Bit</u>	<u>Travel Direction</u>
0x00	Non-navigable
0x01	From left node to right node
0x10	From right node to left node
0x11	Both directions

Unused	(1 bit) (Bit 3)
--------	-----------------

In Process	(1 bit) (Bit 4) Set to 1 if this polyline represents in-process data.
------------	---

Route Type	(3 bits) (Bits 5 - 7) Route type for roads in the US and Europe are as follows:
------------	---

<u>Bit</u>	<u>Route Type (US)</u>
0x1	Interstate Road

0x2	Federal Road
0x3	State Road
0x4	County Road
<u>Bit</u>	<u>Route Type (Europe)</u>
0x1	Interstate or EGT Level 1 Road
0x2	Federal or EGT Level 2 Road
0x3	State or EGT Level 3 Road
0x4	County or EGT Level 4 Road
0x5	EGT Level 5 Road

The following five fields (Rank through Attrib Index) are not present if the Entity Type for this polyline represents a non-navigable feature.

The first four fields (Rank, Divider Loc, Divider Legal, and Multi-digitized) occupy one byte:

Rank	(3 bits) (Bits 0 - 2, where bit 0 is the high order bit)
<u>Bit</u>	<u>Rank</u>
0x0	Level 1
0x1	Level 2
0x2	Level 3
0x3	Level 4
0x4	Level 5
Divider Loc	(3 bits) (Bits 3 - 5) Indicates location of a divider in the roadway. A divider prevents a vehicle from making left turns in right-side driving countries, right turns in left-side driving countries, and U-turns at divided intersections and in the middle of divided links.
<u>Bit</u>	<u>Divider Location</u>
0x0	Link Only
0x1	Link and Both Nodes
0x2	Link and Reference Node
0x3	Link and Non-Reference Node
0x4	No Divider
Divider Legal	(1 bit) (Bit 6) Represents a legal restriction applied to a link, such as painted line, that prohibits turns by statute only.
Multi-digitized	(1 bit) (Bit 7) Set to 1, if another polyline represents same road.
Attrib Index	(1-2 bytes) (Type 1 Variable Length Unsigned Value) An index to a Normalized Attribute Array. If this field is in the range of 0-255, it is an index to the <i>global</i> Normalized Attribute Array. If this field contains a value larger than 255, this field's value, minus 256, constitutes an index to this <i>parcel's</i> Normalized Attribute Array.
Position Cnt	(1-3 bytes) (Type 1 Variable Length Unsigned Value) Count of Position records for this polyline.

Txt Offst Array (2 bytes each) Offsets of the text records attached to this polyline. The number of offsets is determined by the value of the text rec cnt field.

3.5.3.4 Polyline Record (Short Format)

Field (Length) **Description**

The following two fields (Record Format, Delta Latitude) occupy one byte:

Record Format (1 bit) (High order bit) Set to 1 for a Short Format polyline record.

Delta Latitude (7 bits) (Bits 1 - 7) Scaled change in latitude between the starting point and the following point in the polyline.

To compute the value for the latitude field: Subtract the following point's latitude (in NTU's) from the start latitude, and right shift this difference by the number of bits equal to the parcel's scale factor.

The following two fields (Next Point Flag, Delta Longitude) occupy one byte:

Next Point Flag (1 bit) (High order bit) If this polyline contains an additional point after this point, this flag is set to 1.

Delta Longitude (7 bits) (Bits 1 - 7) Scaled change in longitude between the starting point and the following point in the polyline.

To compute the value for the longitude field: Subtract the following point's longitude (in NTU's) from the start longitude, and right shift this difference by the number of bits equal to the parcel's scale factor.

Full Record Index (1-2 bytes) (Type 1 Variable Length Unsigned Value) Represents the 0-based index, within this parcel, of the full polyline record associated with this patch polyline. Remaining polyline attributes for the patch can be obtained from the full record.

Start Latitude (2-3 bytes) (Type 4 Variable Length Unsigned Value) To compute the value for the latitude field: subtract the minimum latitude (in NTUs) of the parcel's bounding rectangle from the latitude value of the starting point, and right shift the result by a number of bits equal to the parcel's scale factor.

Start Longitude (2-3 bytes) (Type 4 Variable Length Unsigned Value) To compute the value for the longitude field: subtract the minimum longitude (in NTUs) of the parcel's bounding rectangle from the longitude value of the starting point, and right shift the result by a number of bits equal to the parcel's scale factor.

The following two fields are present only if the Next Point Flag is set. Whenever the "Next Point Flag 2" bit is set for a delta lat/long pair, there is an additional delta lat/long pair following it. The "Next Point Flag 2 bit is set to 0 for the last delta lat/long pair.

Delta Latitude 2 (1 byte) Scaled change in latitude between the previous point and the current point in the polyline.

To compute the value for the latitude field: Subtract the new point's latitude (in NTU's) from the previous point's latitude, and right shift this difference by the number of bits equal to the parcel's scale factor.

The following two fields occupy one byte:

Next Point Flag 2 (1 bit) (High order bit) Set Next Point Flag 2 to 1, if this polyline contains an additional point after this point.

Delta Longitude 2 (7 bits) (Bits 1 - 7) Scaled change in longitude between the previous point and the current point in the polyline.

To compute the value for the longitude field: subtract the new point's longitude (in NTUs) from the previous point's longitude, and right shift this difference by the number of bits equal to the parcel's scale factor.

Polyline Block Offset Array

Polyline records are accessed through the Polyline Block Offset Array. Each entry contains a block offset pointing to the beginning of a block of $2^K + 1$ length fields followed by 2^K polyline records. The entity identifier is equal to the index of the record within the parcel.

The length fields are all Type 1 Variable Length Unsigned Values. The first length field is the offset from the beginning of the block to the first polyline record in the block. The following 2^K length fields are the lengths of each of the 2^K polyline records.

```
Blkoffset_taPolylineBlk[]; /* Polyline Block Offset Array */
```

3.5.4 Cartographic Points

Point records represent POIs and other single point features, such as tollbooths.

3.5.4.1 Technical Description

The location of the point, expressed in terms of longitude and latitude, is stored within the point entity, along with such attributes as POI facility type and Chain ID. The Point record is a character array.

3.5.4.2 Data Reference

Figure 3-27 illustrates the relationships of the point data within the cartographic parcel.

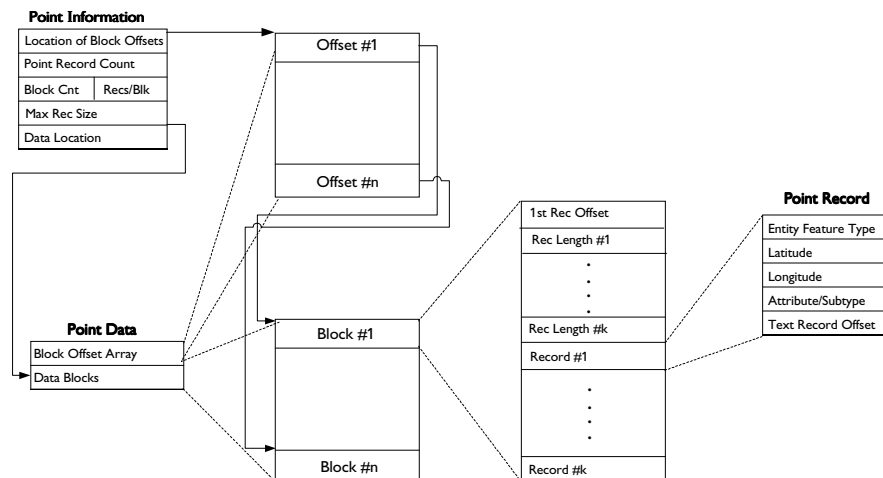


Figure 3-27: Cartographic Parcel Point Data

3.5.4.3 Point Record

Field (Length)	Description
Entity Type	(1-3 bytes) (Type 1 Variable Length Unsigned Value) Type of feature the point represents.
Latitude	(2-3 bytes) (Type 4 Variable Length Unsigned Value) To compute the value for the latitude field: subtract the minimum latitude (in NTUs) of the parcel's bounding rectangle from the latitude value of the starting point, and right shift the result by a number of bits equal to the parcel's scale factor.
Longitude	(2-3 bytes) (Type 4 Variable Length Unsigned Value) To compute the value for the longitude field: subtract the minimum longitude (in NTUs) of the parcel's bounding rectangle from the longitude value of the starting point, and right shift the result by a number of bits equal to the parcel's scale factor.

The following two fields (Attribs, Subtype) occupy one or two bytes.

Attribs	(4 bits) (Bits 0 - 3, where bit 0 is the high order bit) Indicates the relationship of Point feature to POI.														
	<table> <tr> <th>Bit</th><th>Point Feature</th></tr> <tr> <td>0x01</td><td>Point feature represents a POI.</td></tr> <tr> <td>0x02</td><td>Point feature represents a nationally important POI.</td></tr> <tr> <td>0x04</td><td>Point feature represents named place POI that is a capital.</td></tr> <tr> <td>0x08</td><td>Point feature represents named place POI that is a country capital.</td></tr> <tr> <td>0x10</td><td>Point feature is a partial long haul point</td></tr> <tr> <td>0x20</td><td>Point feature is long haul data</td></tr> </table>	Bit	Point Feature	0x01	Point feature represents a POI.	0x02	Point feature represents a nationally important POI.	0x04	Point feature represents named place POI that is a capital.	0x08	Point feature represents named place POI that is a country capital.	0x10	Point feature is a partial long haul point	0x20	Point feature is long haul data
Bit	Point Feature														
0x01	Point feature represents a POI.														
0x02	Point feature represents a nationally important POI.														
0x04	Point feature represents named place POI that is a capital.														
0x08	Point feature represents named place POI that is a country capital.														
0x10	Point feature is a partial long haul point														
0x20	Point feature is long haul data														
Subtype	(4 bits or 12 bits) (Bits 4 - 7, or 4 - 15) Set the first four bits to 0xF if there is no POI Chain ID associated with this POI. This is then four bits in length.														
	Set the first four bits of this field in the range of x0-0xE, if this field represents a POI Chain ID. This is 12 bits in length. See the appendix "Chain IDs" for valid values of the Subtype field.														

The following two fields occupy 2 bytes.

Pop Cat Flag	(1 bit) (High order bit) Set this bit to 1 if the Pop Category field is present.														
Txt Offst	(15 bits) (Bits 1 - 15) Contains the offset of the text record attached to this point.														
Pop Category	(1 byte) Present only if the Pop Cat Flag is set to 1.														
	<table> <tr> <th>Value</th><th>Population Category</th></tr> <tr> <td>0</td><td>Not Applicable</td></tr> <tr> <td>1</td><td>< 50,000</td></tr> <tr> <td>2</td><td>>= 50,000 and < 100,000</td></tr> <tr> <td>3</td><td>>= 100,000 and < 200,000</td></tr> <tr> <td>4</td><td>>= 200,000 and < 500,000</td></tr> <tr> <td>5</td><td>>= 500,000 and < 1,000,000</td></tr> </table>	Value	Population Category	0	Not Applicable	1	< 50,000	2	>= 50,000 and < 100,000	3	>= 100,000 and < 200,000	4	>= 200,000 and < 500,000	5	>= 500,000 and < 1,000,000
Value	Population Category														
0	Not Applicable														
1	< 50,000														
2	>= 50,000 and < 100,000														
3	>= 100,000 and < 200,000														
4	>= 200,000 and < 500,000														
5	>= 500,000 and < 1,000,000														

Point Block Offset Array

Point records are accessed through the Point Block Offset Array. Each entry contains a block offset pointing to the beginning of a block of $2^K + 1$ length fields followed by 2^K point records. The entity identifier is equal to the index of the record within the parcel.

The length fields are all Type 1 Variable Length Unsigned Values. The first length field is the offset from the beginning of the block to the first point record in the block. The following 2^K length fields are the lengths of each of the 2^K point records.

```
Blkoffset_t aPointBlk[]; /* Point Block Offset Array */
```

3.5.5 Cartographic Polygons

Polygon records are used to represent polygonal features such as parks and lakes.

3.5.5.1 Technical Description

The set of points defining the polygon are stored as individual cartographic delta point entities, cross-referenced to the polygon entity. These points are ordered in counterclockwise order for the outer boundary of the polygon and clockwise order for any enclaves.

The polygon record is defined as a character array of variable length.

3.5.5.2 Data Reference

Figure 3-28 illustrates the relationships of the polygon data within the cartographic parcel.

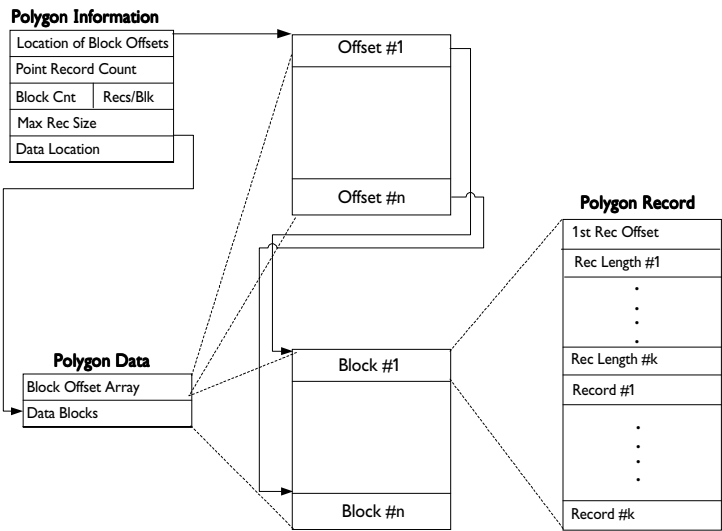


Figure 3-28: Cartographic Parcel: Polygon Data

3.5.5.3 Polygon Record

Field (Length)	Description
Entity Type	(1-3 bytes) (Type 1 Variable Length Unsigned Value) Defines the type of feature the polygon represents.

	<u>Value</u>	<u>Description</u>
	1	Undefined traffic area
	2	Ocean or coastline
	3	River
	4	Canal or water channel
	5	Lake
	6	Bay or harbor
	7	Island
	8	City administrative area
	9	Native American reservation
	10	Built-up area
	11	County administrative area
	12	Country administrative area
	13	Land administrative area
	14	State administrative area
	15	National park or monument
	16	State park
	17	City or county park
	18	Woodlands
	19	Golf course
	20	Shopping centre
	21	University or college
	22	Cemetery
	23	Sports complex
	24	Hospital
	26	Airport
	27	Military base
	28	Pedestrian zone
	29	Aircraft road
	30	Industrial complex
Poly Area	(2 bytes)	Area of the polygon in square meters.
		This value is scaled as follows: The leftmost six bits represent an exponent and the rightmost 10 bits represent a mantissa (normalized so its high order bit is non-zero).
		To obtain the value represented, do the following:
		(1) Subtract nine from the exponent.
		(2a) If the result is positive, left-shift the mantissa by that number of bits.
		(2b) If the result is negative, right-shift the mantissa by the absolute value of that number of bits.

Unused (1 byte)

The following two fields (Initial Pen State, Text Rec Cnt) occupy one byte:

Initial Pen State (1 bit) (High order bit) A value of 1 indicates the drawing starts with pen up.

Text Rec Cnt (7 bits) (Bits 1 - 7) Count of the text records attached to this polygon.

Start Latitude (2-3 bytes) (Type 4 Variable Length Unsigned Value)

To compute the value for the latitude: subtract the minimum latitude (in NTUs) of the parcel's bounding rectangle from the latitude value of the starting point, and right shift the result by a number of bits equal to the parcel's scale factor.

Start Longitude (2-3 bytes) (Type 4 Variable Length Unsigned Value)

To compute the value for the longitude: subtract the minimum longitude (in NTUs) of the parcel's bounding rectangle from the longitude value of the starting point, and right shift the result by a number of bits equal to the parcel's scale factor.

Position Cnt (1-3 bytes) (Type 1 Variable Length Unsigned Value) Count of Position records for this polygon.

Position Offset (2 bytes) Offset of the first position record defining this polygon.

Txt Offset Array (2 bytes each) Array containing the offsets of the Text records attached to this polygon.

Polygon Block Offset Array

Polygon records are accessed through the Polygon Block Offset Array. Each entry contains a block offset pointing to the beginning of a block of $2^K + 1$ length fields followed by 2^K polygon records. The entity identifier is equal to the index of the record within the parcel.

The length fields are all Type 1 Variable-Length Unsigned Values. The first length field is the offset from the beginning of the block to the first polygon record in the block. The following 2^K length fields are the lengths of each of the 2^K polygon records.

```
BlkOffset_taPolygonBlk[]; /* Polygon Block Offset Array */
```

3.5.6 Cartographic Position Records

A position record stores a delta point for a cartographic polyline or polygon. the latitude and longitude values contained in this entity are expressed as deltas from the previous point. The first delta point is located relative to a starting point defined in the polyline or polygon records.

3.5.6.1 Technical Description

The position record is a fixed-length character array. The record is always accessed directly through an offset.

3.5.6.2 Data Reference

The following illustrates the relationships of the position data within the cartographic parcel.

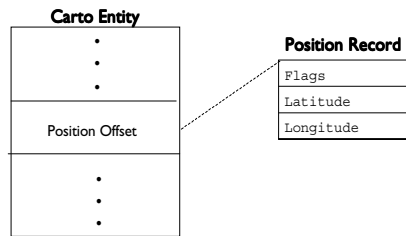


Figure 3-29: Cartographic Parcel: Position Data

3.5.6.3 Position Record

Field (Length)	Description
----------------	-------------

The following four fields occupy three bytes:

Internal Pt	(1 bit) (High order bit) Set to 1, if this position is internal to the Carto Parcel only; present because its offset from the previous Position record is outside the range -1024 to +1023.
Pen Toggle	(1 bit) (Bit 1) Set to 1, if the line between this point to the next point is not drawn.
Latitude	(11 bits) (Bits 2 - 12) Scaled change in latitude between this position and the preceding position or the polygon/polyline starting position. To compute the value for the latitude field: Subtract the new point's latitude (in NTU's) from the previous point's latitude, and right shift this difference by the number of bits equal to the parcel's scale factor. This field ranges in value from +1023 to -1024.
Longitude	(11 bits) (Bits 13 - 23) Scaled change in longitude between this position and the preceding position or the polygon/polyline starting position. To compute the value for the longitude field: subtract the new point's longitude (in NTUs) from the previous point's longitude, and right shift this difference by the number of bits equal to the parcel's scale factor. This field ranges in value from +1023 to -1024.

3.5.7 Cartographic Text Records

Cartographic Text Records contain placement information that can position the text on the map display. For route names, this record contains additional information specifying the highway shield icon to be displayed.

3.5.7.1 Technical Description

The text record is a character array of variable length. All text records are accessed directly as offsets. Since text records are only accessed through carto entity records, the entity type (polygon, line, or point) is already known.

The cartographic text record defines the placement information for text to be placed on a map display: a target position for text placement, and a bounding placement rectangle for the text.

Figure 3-30 and Figure 3-31 show the use of the target position and bounding rectangle for placement of text on a map display. The first diagram illustrates the cartographic text position when the text is placed above the target position.

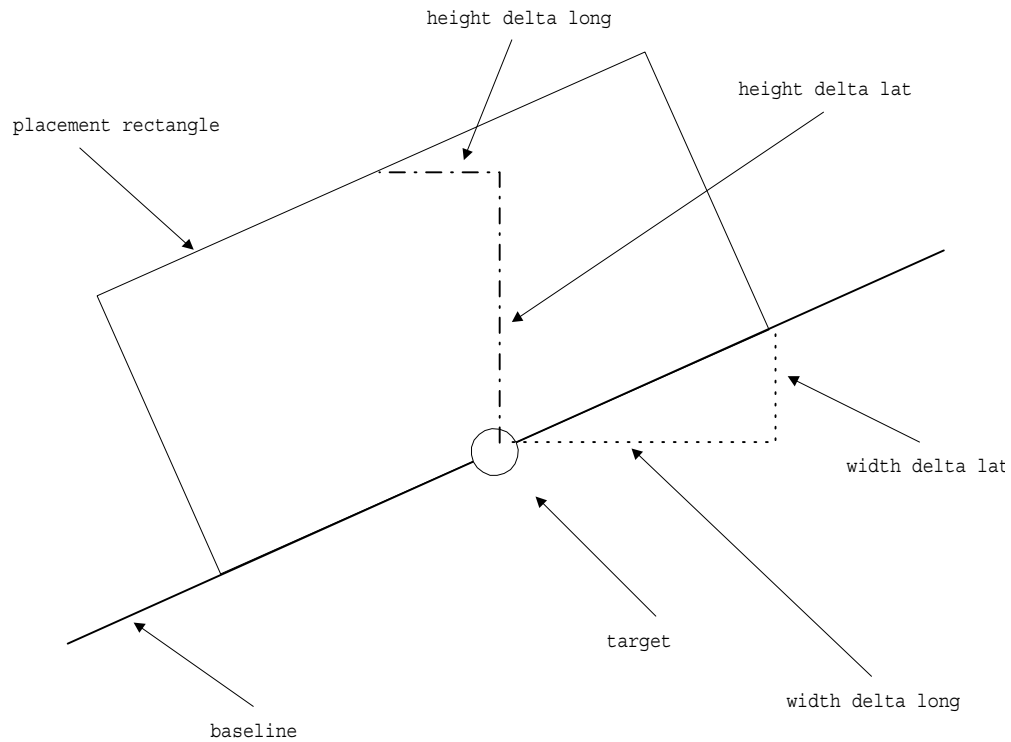


Figure 3-30: Text Placement Above the Target Position

The next diagram illustrates the cartographic text position when the text is placed below the target position.

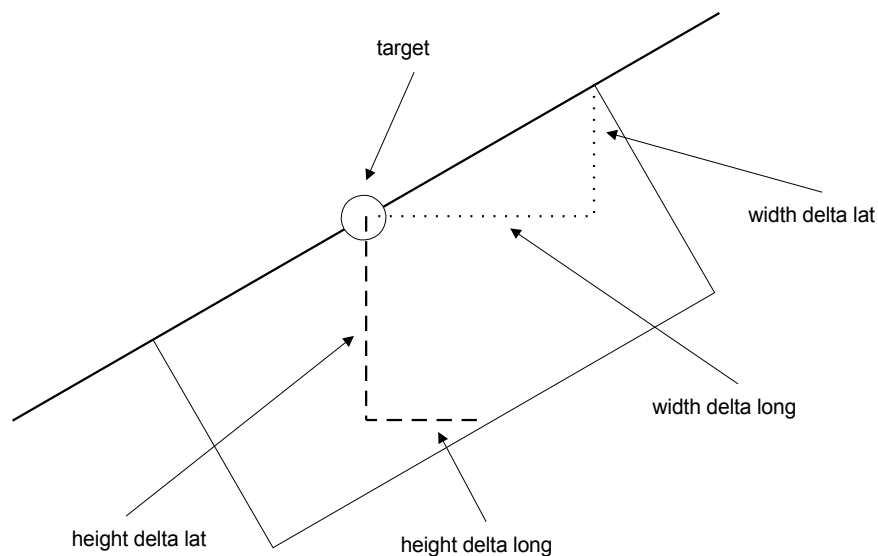


Figure 3-31: Text Placement Below the Target Position

3.5.7.2 Data Reference

The following diagram illustrates the relationships of the text data within the cartographic parcel.

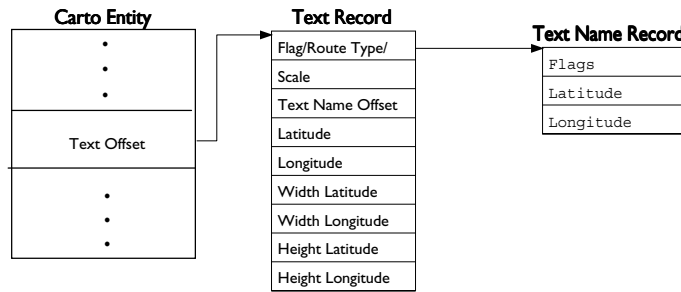


Figure 3-32: Cartographic Parcel: Text Data

3.5.7.3 Text Record

Field (Length) Description

The following four fields (Linear, Route Type, Side, Include Scale) occupy one byte:

Linear (1 bit) (High order bit) A value of 1 indicates the text is to be placed in a line centered above or below (see "side" field) a line (the "baseline") running through the point defined by the provided target latitude and longitude and width latitude and longitude. Example: names of roads and narrow non-navigable features.

A value of 0 indicates the text is to be placed in a rectangle centered on the provided target latitude and longitude. Example: road shields and the names of large polygons.

Route Type (3 bits) (Bits 1 - 3) Type of road shield to be used. For non-shield names, set to 0.

Route types for roads in the US and Europe are as follows:

Bit Route Type (US)

0x1 Interstate Road

0x2 Federal Road

0x3 State Road

0x4 County Road

Bit Route Type (Europe)

0x1 Interstate or EGT Level 1 Road

0x2 Federal or EGT Level 2 Road

0x3 State or EGT Level 3 Road

0x4 County or EGT Level 4 Road

0x5 EGT Level 5 Road

Side (1 bit) (Bit 4) For linear names, this bit indicates on which side of the line placement should be made.

Value Description

0 Lower

	1	Upper (when the placement angle is -90 degrees a value of 1 indicates "east")
	For non-linear placements, side is unused.	
Include Scale	(3 bits) (Bits 5 - 7)	Zoom factor at or below which this placement should be included.
	<u>Value</u>	<u>Description</u>
	0	Include if the data is displayed at the recommended zoom level for this layer.
	1	Include if the data is displayed zoomed out (i.e., less magnification) to twice the recommended zoom level for this layer, or closer.
	2-6	Include if the data is displayed zoomed out (i.e., less magnification) to 4 times the recommended zoom level for this layer, or closer; and so on for values of 3 and more.
	7	Always include.
Text Name	(2 bytes)	Offset of the Text Name Record for this Text Record.
Target Lat	(2-3 bytes) (Type 4 Variable Length Unsigned Value)	Latitude of the target placement for the text. To compute the value for the latitude: subtract the minimum latitude (in NTUs) of the parcel's bounding rectangle from the latitude value of the starting point, and right shift the result by a number of bits equal to the parcel's scale factor.
Target Long	(2-3 bytes) (Type 4 Variable Length Unsigned Value)	Longitude of the target placement for the text. To compute the value for the longitude field: subtract the minimum longitude (in NTUs) of the parcel's bounding rectangle from the longitude value of the starting point, and right shift the result by a number of bits equal to the parcel's scale factor.
Width Lat	(1 byte)	Delta latitude of one half of the width of the area available for the text. To compute the value for the latitude: subtract the latitude (in NTUs) of this point from the latitude value of the target point, and right shift the result by a number of bits equal to the parcel's scale factor. For linear placements, this is the delta latitude from the target point to the right edge of the placement area along the baseline. For non-linear placements, this is the delta latitude from the target point to the center of the right edge of the placement area. For placements where text collision avoidance is not provided, this value will be zero.
Width Long	(1 byte)	Delta longitude of one half of the width of the area available for the text To compute the value for the longitude: subtract the longitude (in NTUs) of this point from the longitude value of the target point, and right shift the result by a number of bits equal to the parcel's scale factor. For linear placements, this is the delta longitude from the target point to the right edge of the placement area along the baseline. For non-linear placements, this is the delta longitude from the target point to the center of the right edge of the placement area. This value is always non-negative.

Height Lat	For placements where text collision avoidance is not provided, this value is zero.
	(1 byte) Delta latitude of the height of the area available for the text.
	To compute the value for the latitude: subtract the latitude (in NTUs) of this point from the latitude value of the target point, and right shift the result by a number of bits equal to the parcel's scale factor.
	For linear placements above the baseline, this is the delta latitude from the target point on the baseline to the center of the top edge of the placement area.
	For linear placements below the baseline, this is the delta latitude from the center of the bottom edge of the placement area to the target point on the baseline.
Height Long	For non-linear placements, this is twice the delta latitude from the target point to the center of the top edge of the placement area. This value is always non-negative.
	For placements where text collision avoidance is not provided, this value is zero.
	(1 byte) Delta longitude of the height of the area available for the text.
	To compute the value for the longitude: subtract the longitude (in NTUs) of this point from the longitude value of the target point, and right shift the result by a number of bits equal to the parcel's scale factor.
	For linear placements above the baseline, this is the delta longitude from the target point on the baseline to the center of the top edge of the placement area.
	For linear placements below the baseline, this is the delta longitude from the center of the bottom edge of the placement area to the target point on the baseline.
	For non-linear placements, this is twice the delta longitude from the target point to the center of the top edge of the placement area.
	For placements where text collision avoidance is not provided, this value is zero.

3.5.8 Cartographic Text Name Records

Cartographic Text Name Records store the name text associated with a cartographic entity.

3.5.8.1 Technical Description

The text name record is a character array of variable length, accessed via the text record.

3.5.8.2 Data Reference

Figure 3-33 illustrates the relationships of the text name data within the cartographic parcel.

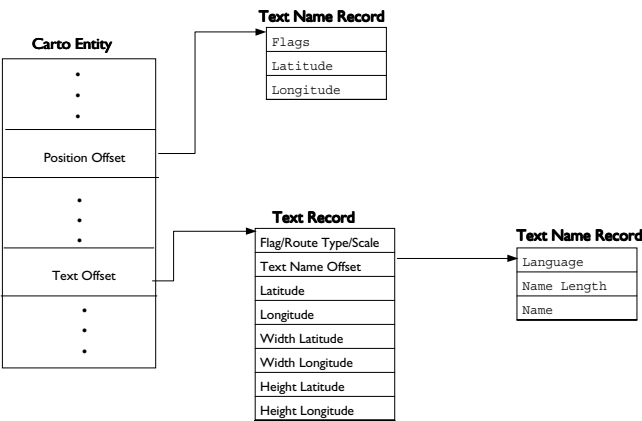


Figure 3-33: Cartographic Parcel: Text Data

3.5.8.3 Text Name Record

Field (Length) Description

The following three fields occupy one byte:

Has Icon Index (1 bit) (High order bit) Set to 1, if fields Rte Num Offset, Rte Num Len, Text Color, and Icon Index are to be present.

Unused (3 bits) (Bits 1 - 3)

Language (4 bits) (Bits 4 - 7) Language code for the name.

The following four fields (Rte Num Offset, Rte Num Len, Text Color, Icon Index) occupy three bytes and are present if the Has Icon Index field is set to 1.

Rte Num Offset (4 bits) (Bits 0 - 3, where 0 is the high order bit) Offset to the name string (in characters) to the character at which the route number begins.

Rte Num Len (4 bits) (Bits 4 - 7) If the name is a route number, this contains the number of characters, minus 1, that should appear on the highway shield.

Text Color (4 bits) (Bits 8 - 11) Color used for the text drawn on the shield.

Bit	Color of Shield Text
0x0	Black
0xE	Yellow
0xF	White

Icon Index (12 bits) (Bits 12 - 23) Global Media File icon index for the appropriate shield for this country, route type, and number of digits.

Name Length (1 byte) *Uncompressed* length of the Name field in bytes.

Name (Name Length bytes) Text of the feature name.

3.6 Cartographic Cross Reference Data

Cartographic Cross-reference data associates cartographic entities with other entities in the database. Relationship data between cartographic polylines and routing segments as well as polygon or cartographic point to POI references are contained in these parcels. Also included is data necessary for optimized hi-lighting of routing segments including those for supernode paths and aggregated routing segments.

3.6.1 Technical Description

Cartographic cross-reference parcels are spatially organized, and the same parcelization method used for other spatial parcel types is used. Cartographic cross-reference data is not included with cartographic data to maximize the geographic area covered by each cartographic parcel.

The cartographic cross-reference parcel data is divided into five major sets:

- Polygon to POI
- Cartographic Point to POI
- Polyline to Routing Segment
- Routing Segment to Polyline
- Aggregated Segment and Supernode Path Hi-lighting

The first three of these sets, polygon to POI, cartographic point to POI, and polyline to routing segment cross references, are similarly represented using a record offset array which maps cartographic entity IDs to variable-length lists of fixed-size reference records. Each record of an offset array table contains fields for a parcel ID table index, cartographic entity ID, and a count and first record index of fixed-size reference records.

The routing segment to polyline cross reference table has fields for routing segment ID, polyline ID virtual table index, reverse flag, polyline position array starting index, and polyline position count. This table is shared with the polyline to routing segment cross-references, and also contains aggregated routing segment cross-references that either reference cartographic polylines or position data within the same cross-reference parcel. The polyline ID virtual table is a mapping between the index values in the routing segment to polyline cross-reference table to a polyline DBID. This is a virtual table since all fields are always zero bits wide.

The aggregated segment and supernode path highlighting cross-reference tables have fields for the first position delta index, position delta count, starting position latitude, and starting position longitude. These records exist for aggregated segments and supernode paths considered cartographically significant at a particular layer.

3.6.2 Cartographic Cross-reference Parcel Structure

The data within cartographic cross-reference parcels are per-parcel, table-driven, bit-aligned values in Base-Relative Positional-Prediction Difference (BRPPD) representation. This representation stores the value differences from an index-relative adjusted base value.

3.6.2.1 Base-Relative Positional-Prediction Difference (BRPPD) Representation

The space needed to store an array of values having a limited range can be reduced by storing the base value of the range in a header area, and representing each value by its difference from the base, so that for value V , the stored difference DIFF is $V - \text{the Base value}$. The DIFF value is stored using the minimum number of bits sufficient to represent the largest difference value, which is also stored in the header.

Records are also grouped by linearity. For each group, the header contains the base value for the group, the field length of the DIFF field, a count of records in the group, and a multiplier M for the group. Thus, for the element j of the group i , the DIFF value is $V - (\text{Base}[i] + j * m[i])$. Making the

differences also base-relative by storing a MinVal value in the header (which is also a bit-variable field) is the BRPPD representation. This representation yields:

$$\begin{aligned} \text{Diff}[i,j] &= V - (\text{Base}[i] + j * m[i]) - \text{MinVal} \\ V &= \text{Base}[i] + j * m[i] + \text{Diff}[i,j] + \text{MinVal} \end{aligned}$$

Note: The MinVal is not currently used because it normally does not save any space.

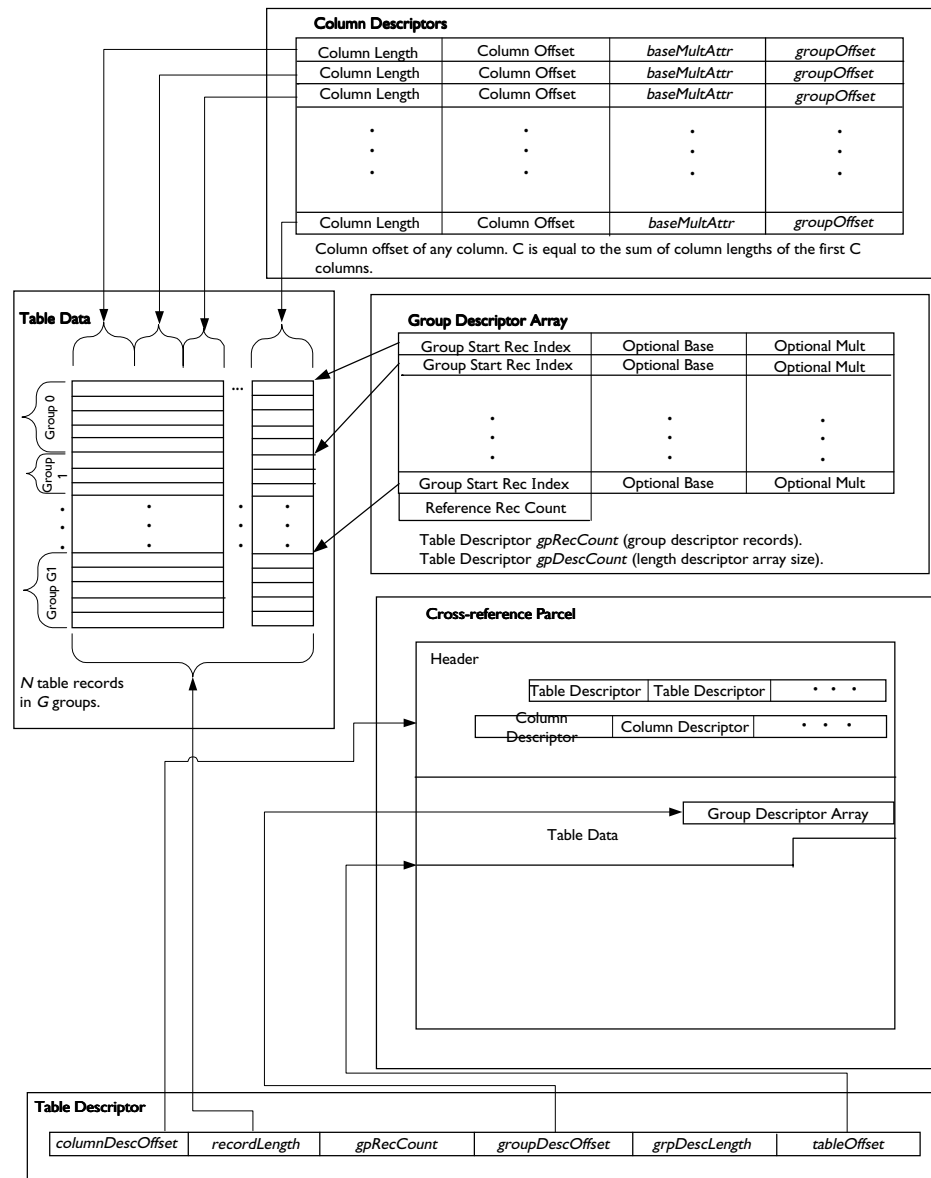
In the cartographic cross-reference parcels, the BRPPD data is organized as follows:

Table Descriptor (one per table for each xref offset array or xref records array)

Offset to Column Descriptor Array					
	Table Record Length (bits)				
		Group or Record Count			
			(Record count if offset to Group Desc. Array = 0)		
			Offset to Group Descriptor Array		
				Group Descriptor Array Element size (bits)	
					Offset to start of table
cccc	nnnn	gggg	dddd	ssss	tttt

Column Descriptor List (one entry per table column)

Column Bit Length					
	Column Offset in Table (bits)				
		Base Multiplier Length			
				Column Info Offset in Group Descriptor List	
				Minimum Value	
ll	oo	t1	vv	mm	mm mm mm mm mm



3.6.3 Parcel Header

The following structures define the initial, fixed portion of the parcel data.

```
/*
** enumerated list defining the indices of the column descriptor array
** corresponding to all the columns of the cross reference data tables
**/
enum CartoXrefColDescTypes
{
    /* e_polygonXODesc - polygon offset array table */

```

```
e_polygonXOPiti, /* Piti = Parcel Id Table Index */
e_polygonXOEntId,
e_polygonXOXrecCnt,
e_polygonXOFxri, /* Fxri = First-Xref-Record Index */

/* e_lineXODesc - polyline offset array table */
e_lineXOPiti, /* Piti = Parcel Id Table Index */
e_lineXOEntId,
e_lineXOXrecCnt,
e_lineXOFxri, /* Fxri = First-Xref-Record Index */

/* e_pointXODesc - carto point offset array table */
e_pointXOPiti, /* Piti = Parcel Id Table Index */
e_pointXOEntId,
e_pointXOXrecCnt,
e_pointXOFxri, /* Fxri = First-Xref-Record Index */

/* e_polygonPoiDesc - polygon to POI reference table */
e_polygonPoiPiti,
e_polygonPoiEntId,

/* e_pointPoiDesc - carto point to POI reference table */
e_pointPoiPiti,
e_pointPoiEntId,

/* e_lineSegIdxDesc - polyline to segment record index table */
e_lineSegIdx,

/* e_entIdMapDesc - polyline DBID mapping virtual table */
e_entIdMapEntSerN,
e_entIdMapEntId,
e_entIdMapPiti,

/* e_segPolyDesc - segment to polyline reference table */
e_segPolySegId,
e_segPolyLineId,
e_segPolyRvrsFlag,
e_segPolyStPos,
e_segPolyPosCnt,

/* e_segPolyXtenDesc - extended segment to polyline reference table */
e_segPolyXtenSegId,
e_segPolyXtenLineId,
e_segPolyXtenRvrsFlag,
e_segPolyXtenStPos,
e_segPolyXtenPosCnt,

/* e_lineDesc */
e_linePosIndex,
e_linePosCount,
e_lineStartLat,
e_lineStartLong,

/* e_positionDesc - delta positions table */
e_positionLat,
e_positionLong,

/* e_supernodeDesc - supernode table */
e_supernodeEntity,
```

```

    e_supernodeArmCount,
    e_supernodeArmIndex,

/* e_armDesc - arm-to-arm supernode path index table */
    e_armSegment1Entity,
    e_armSegment2Entity,
    e_armPathIndex,

/* e_pathDesc - supernode path table */
    e_pathPosIndex,
    e_pathPosCount,
    e_pathStartLat,
    e_pathStartLong,

    e_CartoXrefTableColDescCount/* column descriptor array record count */
};

/*
** Reference table descriptor - one record for each table
*/
typedef struct
{
    Ushort_t    columnDescOffset; /* byte offset in parcel of first column
                                ** descriptor for this table */
    Ushort_t    recordLength;      /* bit length of record in table */
    Ushort_t    grpRecCount;       /* If groupDescOffset is nonzero, this is
                                ** the number of group descriptor elements
                                ** If groupDescOffset is zero, this is the
                                ** number of records in the table */
    Ushort_t    groupDescOffset;  /* byte offset in parcel of first group
                                ** descriptor for this table. Zero for
                                ** no group descriptor entries */
    Ushort_t    grpDescLength;     /* byte length of group descriptor entry
                                */
    Ushort_t    tableOffset;      /* byte offset in parcel of table */
} TableDescriptor_t;

/*
** Reference data column descriptor - one record for each column of every
    table
*/
typedef struct
{
    Uchar_t    columnLength;       /* bit length of column [0-32] */
    Uchar_t    columnOffset;      /* bit offset of column in record */
    Uchar_t    baseMultAttr;      /*
                                ** signedBase:1,** base value in group
                                ** descriptor is
                                ** ** signed
                                ** decimalPoint:3,** decimal point loca-
                                ** tion in multiplier
                                ** baseSize:2,** byte size of base in
                                ** group descriptor
                                ** multSize:2,** byte size of multiplier
                                ** in group
                                ** ** descriptor
                                */

```

```

        Uchar_t    groupOffset;        /* byte offset to base and/or multiplier
                                        in
                                        ** group descriptor */
    } ColumnDescriptor_t;
    /*
    ** Cartographic Cross-reference Parcel Header
    */
    typedef struct
    {
        SptlPclHdr_t spatialHdr;

        TableDescriptor_t  polygonXODesc; /* polygon-POI xref offset array
                                           desc*/
        TableDescriptor_t  lineXODesc;    /* line-seg xref offset array desc */
        TableDescriptor_t  pointXODesc;   /* point-POI xref offset array desc
                                           */
        TableDescriptor_t  polygonPoiDesc; /* polygon xref POI ID table desc */
        TableDescriptor_t  pointPoiDesc;  /* point xref POI ID table desc */
        TableDescriptor_t  lineSegIdxDesc; /* line xref segpoly idx table desc
                                           */
        TableDescriptor_t  entIdMapDesc;   /* virtual table to map indices from
                                           ** seg-poly records to polyline par-
                                           cel
                                           ** ID table indexes and entity IDs */
        TableDescriptor_t  segPolyDesc;    /* seg-polyline xref table desc */
        TableDescriptor_t  segPolyXtenDesc; /* extension for line-seg xref on
                                           ** segs not in seg-poly */
        TableDescriptor_t  lineDesc;       /* local polyline records */
        TableDescriptor_t  positionDesc;   /* positions of local polylines */
        TableDescriptor_t  supernodeDesc;  /* supernode entity IDs */
        TableDescriptor_t  armDesc;        /* supernode arms */
        TableDescriptor_t  pathDesc;       /* supernode path */
        ColumnDescriptor_t ColDescAry[ e_CartoXrefTableColDescCount ];

    } CartoXrefParcelHeader_t;

```

Per-Parcel Data Arrays

The location of the following array and its size are defined in the parcel header:

```
ParcelID_t    aPclId[];        /* Parcel ID Array */
```

The Group Descriptor Array consists of a two-byte starting index, followed by optional base and/or multiplier pairs. A trailing record count entry follows the last group descriptor array entry. The baseMultAttr field of a column descriptor defines the existence of a group descriptor array for that column:

```
char    aGroupDescArray[];    /* Group Descriptor Array */
```

3.6.3.1 Segment to Polyline Section Cross-reference Record Array (and External Segment to Polyline Section Cross-reference Record Array)

Cross-reference records are in order by Segment Entity ID. A Cartographic cross-reference parcel contains at least one segment to polyline cross-reference record for every segment whose left node's location is within the bounding rectangle of that parcel. The number of segment to polyline cross-reference records for a given segment is equal to the number of polylines used to represent that segment in the current cartographic layer; this number exceeds one only when the segment is split by a cartographic parcel or subparcel boundary. External segment to polyline section cross-reference records exist for segments whose left node is not located within the bounding rectangle of the parcel, but which are referenced by a polyline cross-reference record in this parcel.

Whenever there are multiple segment to polyline records for one segment, they are stored in an order so the first one represents the portion of the segment starting at the left node. Each successive record represents a section of the segment farther from the left node than the previous record.

Field	Description
Seg Entid	The entity ID of the segment.
Polyline ID	A search index into this parcel's Polyline DBID Virtual Map Table.
Reverse Flag	Set this bit to 1 if the position along the polyline represented by the Start Position is closer to the right node of the segment than the position represented by the Start Position plus Position Count.
Start Position	An <u>index</u> to the array of positions for this polyline corresponding to the beginning of the segment.
Position Count	Added to the Start Position value, this gives the index to the array of positions for this polyline corresponding to the end of the segment.

3.6.3.2 Polygon Cross-reference Record Offset Array

Field	Description
Carto Pcl ID	An index to the Parcel ID Table for a Cartographic Parcel ID.
Carto EntID	A Cartographic Entity ID (for a polygon).
Xref Rec Cnt	Count of POI reference records for this Carto entity.
First Xref	Index to first POI reference record for this Carto entity.

3.6.3.3 Polygon Cross-reference Record

Field	Description
POI PclID	An index to the Parcel ID Table for a POI Parcel ID.
POI EntID	POI Entity ID.

3.6.3.4 Polyline Cross-reference Record Offset Array

Field	Description
Carto Pcl ID	An index to the Parcel ID Table for a Cartographic Parcel ID.
Carto EntID	A Cartographic Entity ID (for a polyline).
Xref Rec Cnt	Count of segment record index reference records for this Carto entity.
First Xref	Index to first segment record index reference record for this Carto entity.

3.6.3.5 Polyline Cross-reference Record

A Cartographic Cross-reference parcel contains one polyline cross-reference record for each polyline whose start location is within the bounding rectangle of that parcel.

Field	Description
SegXref	Segment record index of a segment to a polyline cross-reference record or external segment to a polyline cross-reference record, or external segment to a polyline cross-reference record.

3.6.3.6 Point Cross-reference Record Offset Array

Field	Description
Carto Pcl ID	An index to the Parcel ID Table for a Cartographic Parcel ID.

Carto EntID	A Cartographic Entity ID (for a Carto point).
Xref Rec Cnt	Count of POI reference records for this Carto entity.
First Xref	Index to first POI reference record for this Carto entity.

3.6.3.7 Point Cross-reference Record:

Field	Description
POI PclID	An index into the Parcel ID Table for a POI Parcel ID.
POI EntID	POI Entity ID.

3.6.3.8 Polyline Entity DBID Virtual Map

Field	Description
Search Index	Entity ID Search Index.
EntID	Polyline Entity ID.
PclID	Index to the Parcel ID Table for a Polyline parcel ID. If zero, this record refers to a polyline in the current (Carto Xref) parcel.

3.6.3.9 Polyline Record

Field	Description
Position Index	The <i>index</i> of the first position record defining this polyline.
Position Cnt	The count of Position records for this polyline.
Start Latitude	To compute the value for the latitude field: subtract the minimum latitude (in NTUs) of the parcel's bounding rectangle from the latitude of the polyline and right shift the result by a number of bits equal to the parcel's scale factor.
Start Longitude	To compute the value for the longitude field: subtract the minimum longitude (in NTUs) of the parcel's bounding rectangle from the longitude of the polyline and right shift the result by a number of bits equal to the parcel's scale factor.

3.6.3.10 Position Record

Field	Description
Latitude	The scaled change in latitude between this position and either the preceding position or the polyline starting position. To compute the value for the latitude field: subtract the latitude of the previous position/starting point from the latitude of the position and right shift the result by a number of bits equal to the parcel's scale factor.
Longitude	The scaled change in longitude between this position and either the preceding position or the polyline starting position. To compute the value for the longitude field: subtract the longitude of the previous position/starting point from the longitude of the position and right shift the result by a number of bits equal to the parcel's scale factor.

3.6.3.11 Supernode Array

Field	Description
Node EntID	A Node Entity ID (for a supernode).
Arm Cnt	Count of Arm records for this supernode.

First Arm	Index to first Arm record for this supernode.
-----------	---

3.6.3.12 Supernode Arm Array

Field	Description
Seg1 EntID	The segment entity ID for the arm of the supernode at the entrance to the path.
Seg2 EntID	The segment entity ID for the arm of the supernode at the exit from the path.
Path Index	Index to Supernode Path record for this supernode and arm pair.

3.6.3.13 Supernode Path Record

Field	Description
Position Index	The <i>index</i> of the first position record defining this path.
Position Cnt	The count of Position records for this path.
Start Latitude	Left-shifting this value by a number of bits equal to this parcel's scale factor, and adding it to the minimum latitude of this parcel's bounding rectangle gives the latitude of the initial point defining this path.
Start Longitude	Left-shifting this value by a number of bits equal to this parcel's scale factor, and adding it to the minimum longitude of this parcel's bounding rectangle gives the longitude of the initial point defining this path.

3.7 Maneuver Data

Maneuver data is parcelized in parallel with the route calculation data. Route calculation data contains only part of what is required for direction generation. The maneuver parcels contain the additional data needed for this function. Maneuver data records in maneuver parcels are accessed in blocks the same way as records (other than segments and nodes) are in the route calculation and cartographic parcels.

3.7.0.1 Technical Description

Maneuver data consists of the following four entities:

- **Block** The Block entity provides a logical relationship between a segment, a navigable feature name, and an address range. A segment may have multiple names and be represented in multiple blocks. The Block entity allows an address range, or ranges, to be paired with one of the names associated with a segment. In addition to the segment and navigable feature ID, the Block entity contains the address ranges for the upper and lower sides of the segment and the type of address (e.g., base, county, commercial, etc.).
- **Sign** The Sign entity stores information about road signs located along the segment. This information includes a list of the segment IDs associated with each of the destinations displayed on the sign as well as the exit number for each destination. The text for each of the destinations is stored in one or more of the Sign Text entities.
- **Sign Route ID** The Sign Route ID entity stores the route ID information for a particular sign destination or line of text. In addition to the route ID, this entity contains a text type defining whether the route ID corresponds to the road the exit provides access to, or is a route toward which the exit leads.
- **Sign Text** The Sign Text entity stores a line of text for a sign destination. In addition to free format text, this entity contains a text type defining whether the sign text describes the road the exit provides access to or a destination toward which the exit leads.

3.7.1 Maneuver Block Records

The block record provides a logical relationship between a segment, a navigable feature name, and an address range.

3.7.1.1 Technical Description

A segment can have multiple names and be represented in multiple blocks. The Block record allows an address range, or ranges, to be paired with one of the names associated with a segment.

In addition to the segment and navigable feature ID, the Block record contains the address ranges for the upper and lower sides of the segment as well as the type of address (i.e., base, county, commercial, etc.).

The Block record is a character array of variable length. Block records are organized in segment ID order, so blocks associated with a particular segment can be found with a binary search of the Block Record Offset Array.

3.7.1.2 Referenced By

Figure 3-34 illustrates the relationships of the block data within the maneuver parcel.

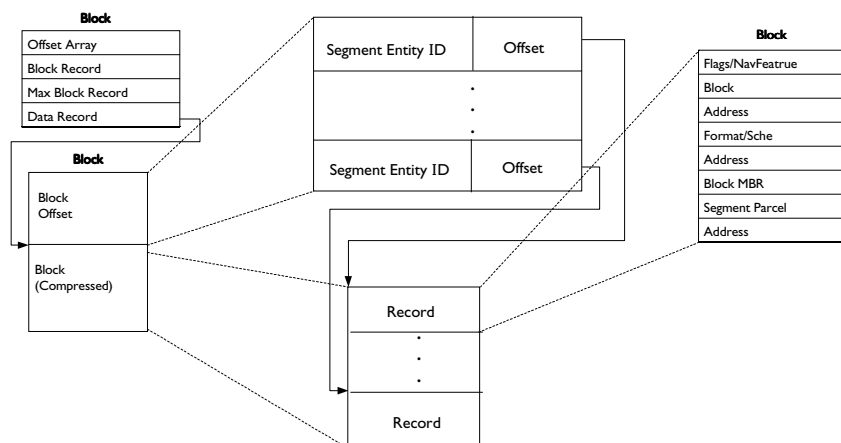


Figure 3-34: Maneuver Parcel: Block Data

3.7.1.3 Block Record

Field (Length)	Description
The following five fields occupy two bytes if UnName Flag is set to 0. Otherwise these fields occupy one byte:	
MBR Flag	(1 bit) If set to 1, the MBR2 field is present.
UnName Flag	(1 bit) If set to 1, this block is unnamed.
UnAddr Flag	(1 bit) If set to 1, this block is unaddressed.
Direction	(3 bits) Binary value
	001 'N'
	010 'E'
	011 'S'

100 'W'

'000' blank.

Nav Feature ID (10 bits, or 2 bits if UnName Flag is set to 1) An index to this parcel's Nav Feature to Block Index Array, from which the Nav Feature ID can be obtained. If UnName Flag is 1, this field is unused.

The following is not present if the UnName Flag is set to 1:

Block Attrs (1 byte)

The following is not present if the UnAddr Flag or the UnName Flag is set to 1:

Address Type (1 byte)

The following four fields occupy two bytes and are not present if the UnAddr Flag is set to 1 or the UnName Flag is set to 1:

Upper Format (4 bits) Format of the addresses on the upper side of the segment.

Upper Scheme (4 bits) Scheme of the addresses on the upper side of the segment.

Lower Format (4 bits) Format of the addresses on the lower side of the segment.

Lower Scheme (4 bits) Scheme of the addresses on the lower side of the segment.

The following four fields (Uppr Left Len, Uppr Right Len, Lowr Left Len, Lowr Right Len) occupy three bytes and are not present if the UnAddr Flag or the UnName Flag is set to 1:

Uppr Left Len (6 bits) Length of the upper left address.

Uppr Right Len (6 bits) Length of the upper right address.

Lowr Left Len (6 bits) Length of the lower left address.

Lowr Right Len (6 bits) Length of the lower right address.

MBR1 (2 bytes) The first byte of this field is a latitude index (0-255) and the second byte of this field is a longitude index.

Let D be the larger of two dimensions of this parcel's minimum enclosing tile. If LT is the latitude index, the southwest corner of the minimum bounding rectangle of this block has a latitude $\geq$ the tile's minimum latitude plus $(LT \times D)/256$, and $<$ the tile's minimum latitude plus $((LT + 1) \times D)/256$. The longitude index has an analogous meaning.

The following field (MBR2) is present if the MBR Flag bit is set to 1.

MBR2 (2 bytes) This field is defined the same way as the MBR1 field, except it represents the northeast corner of the block's MBR. If MBR2 is not present, then its value defaults to that of MBR1.

Seg Parcel ID (1-3 bytes) (Type 1 Variable Length Unsigned Value) An index to the parcel ID table.

The next four fields are not present if the UnAddr Flag or the UnName Flag is set to 1:

Uppr Left Adr (as specified by Uppr Left Len) Contains the upper left address.

Uppr Right Adr (as specified by Uppr Right Len) Contains the upper right address.

Lowr Left Adr (as specified by Lowr Left Len) Contains the lower left address.

Lowr Right Adr (as specified by Lowr Right Len) Contains the lower right address.

Block Record Offset Array

The Block Record Offset Array defines offsets into the block data to the start of block records.

```
RecOffsetEnt_t aBlockOffset[]; /* Block Record Offset Array */
```

3.7.2 Maneuver Sign Records

Information about road signs located along the segment is stored in the sign record.

3.7.2.1 Technical Description

The sign record stores information about each destination appearing on the sign. This information includes:

- Segment ID for the destination segment
- Exit number associated with the destination
- Number of segments that must be traversed to get to the destination segment from the segment where the sign is located.

The sign record does not contain the actual sign text, which is broken out as a separate record in the PSF.

3.7.2.2 Referenced By

The signDesc field in the Maneuver Parcel Header is a BlkDesc_t structure pointing to the maneuver sign data. The data relationships for sign data are illustrated in Figure 3-35.

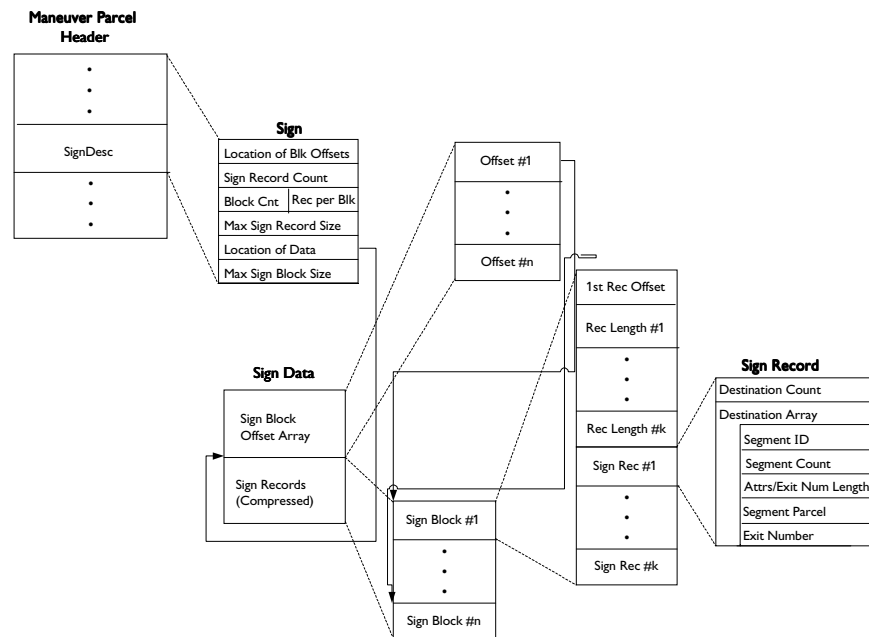


Figure 3-35: Maneuver Parcel Sign Data

3.7.2.3 Sign Record

The Sign record is a character array of variable length.

Field (Length)	Description
Num Dests	(1 byte) Number of destinations for this sign.
Dest Array	Each entry contains the following six fields:
Segment ID	(3 bytes) Entity ID of the destination segment.
Seg Cnt	(1 byte) Number of segments between the sign and the destination.

The following two fields (Dest Attrs, Exit Num Len) occupy one byte:

Dest Attrs	(4 bits) The following bit settings are used:
	0x1 Branch Route IDs are present
	0x2 Toward Route IDs are present
	0x4 Branch Text is present
	0x8 Toward Text is present
Exit Num Len	(4 bits) Length of the Exit Number field in bytes.
Segment Parcel	(Type 1 Variable Length Unsigned Value) Represents an index to the parcel ID Table for the destination segment parcel ID.
Exit Number	Number of bytes in the Exit Num Len field) Text of the Exit Number.

Sign Block Offset Array

Sign records are accessed via the Sign Block Offset Array, each entry of which contains a block offset pointing to the beginning of a block of $2^K + 1$ Length fields followed by 2^K Sign records.

The entity identifier is equal to the index of the record (within the parcel). The length fields are all Type 1 Variable Length Unsigned Values. The first length field is the offset from the beginning of the block to the first Sign record in the block, and the following 2^K length fields are the lengths of each of the 2^K compressed Sign records.

```
BlkOffset_t    aSignBlk[];                    /* Sign Block Offset Array */
```

3.7.3 Maneuver Sign Text Records

The sign text record stores a line of text for a sign destination.

3.7.3.1 Technical Description

The Sign Text record is a character array of variable length. In addition to free format text, this contains a text type defining whether the sign text describes:

- The road the exit provides access to
- A destination the exit leads towards

3.7.3.2 Referenced By

The `signTextDesc` field in the maneuver parcel header is a `BlkDesc_t` structure pointing to the maneuver sign data. The data relationships for sign text data are shown in Figure 3-36.

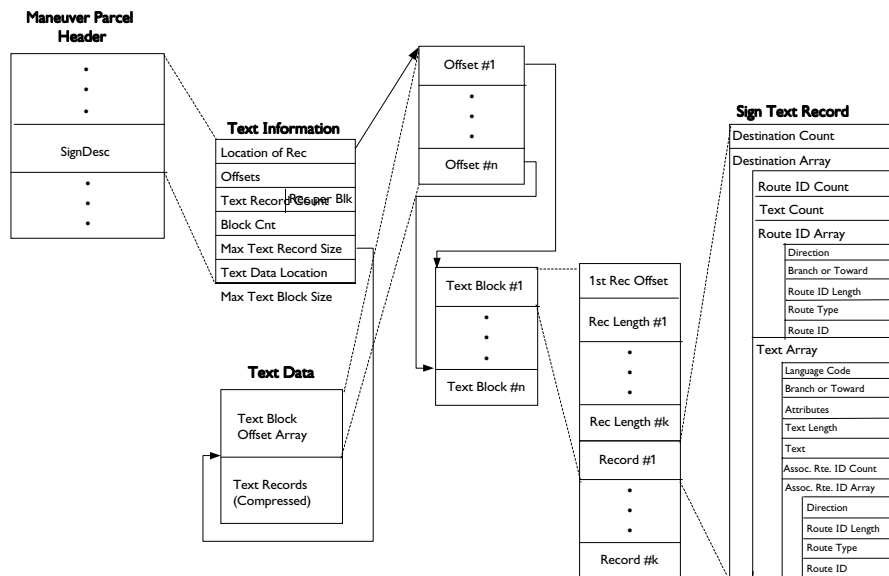


Figure 3-36: Maneuver Parcel: Sign Text Data

3.7.3.3 Sign Text Record

Field (Length)	Description
Num Dests	(1 byte) Number of destinations for this sign.
Dest Array	Each entry contains the following fields:
<u>Field</u>	<u>Description</u>
Entry Length	(Type 1 Variable Length Unsigned Value) Represents the length in bytes of the current destination array entry, so the start of the next entry can be found quickly.
Num RIDs	(1 byte) Number of Route ID entries for this destination.
Num Texts	(1 byte) Number of sign text line entries for this destination.
RID Array	The number of entries in this array is given by the Num RIDs field.
	Each entry contains the following fields:
<u>Field</u>	<u>Description</u>
The next three fields (Route Dir, Br or To, Route ID Len) occupy one byte:	
Route Dir	(3 bits) Contains one of the four cardinal directions (or no direction), in encoded form, as follows:
0x0	No direction
0x1	'N'
0x2	'E'

0x3	'S',
0x4	'W'
Br or To	(1 bit) Branch or Toward Indicator, in which the value 0 indicates "branch to" and the value 1 indicates "toward".
Route ID Len	(4 bits) Size of the route ID information.
Route Type	(1 byte) Valid values:
0	N/A
1	Level 1 route
2	Level 2 route
3	Level 3 route
4	Level 4 route
5	Level 5 route
6	Level 6 route
Route ID	(Number of bytes in the Route ID Len field) The Route ID.
Text Array	(variable) Number of entries given by the Num texts field. Each entry contains the following fields:
<u>Field</u>	<u>Description</u>
Ent Len	(1 byte) Length in bytes of the current RID Array entry, so that the start of the next entry can be found quickly.

Lang. Code (1 byte) Describes the language of the sign text, and contains a valid language code as defined in the reference data parcels.

The next three fields (Br or To, Attrs, Text Len) occupy one byte:

Br or To	(1 bit) Branch or Toward Indicator, in which the value 0 indicates "branch to" and the value 1 indicates "toward".
Attrs	(1 bit) If set to 1, route IDs are associated with the sign text.
Text Len	(6 bits) Size of the Text information.
Text	(Number of bytes in the Text Len field); contains the Sign Text.
Num ARIDs	(1 byte) Number of entries in the ARID Array; present only if the Attrs bit is set to 1.
ARID Array	(variable: The number of entries given by the Num ARIDs); present only if the Attrs bit is set to 1.

Each entry contains the following four fields:

The next three fields (Route Dir, Route ID Len, Route Type) occupy one byte:

Route Dir	(3 bits) Contains one of the four cardinal directions (or no direction), in encoded form, as follows:
0x0	No direction
0x1	'N'
0x2	'E'
0x3	'S'

0x4	'W'
Route ID Len (5 bits)	Size of the route ID information.
Route Type (1 byte)	Valid values:
0	N/A
1	Level 1 route
2	Level 2 route
3	Level 3 route
4	Level 4 route
5	Level 5 route
6	Level 6 route
Rte ID	(Number of bytes in the Route ID Len field)

Sign Text Block Offset Array

Sign Text records are accessed via the Sign Text Block Offset Array, each entry of which contains a block offset pointing to the beginning of a block of $2^K + 1$ Length fields followed by 2^K Sign Text records.

The entity identifier is equal to the index of the record (within the parcel). The length fields are all Type 1 Variable Length Unsigned Values. The first length field is the offset from the beginning of the block to the first Sign Text record in the block, and the following 2^K length fields are the lengths of each of the 2^K compressed Sign Text records.

```
BlkOffset_t    aSignTextBlk[];    /* Sign Text Block Offset Array */
```

3.8 Crossroads Data

Crossroads data is used primarily to determine a starting or ending point prior to route calculation. For a given an intersection, it is necessary to find the node or supernode at that intersection.

3.8.0.1 Technical Description

Crossroads data is stored in the SDAL database spatially. Each physical location, representing an intersection or overpass, appears once in the database. There can be multiple crossroads at that location (e.g., if any of the roadways at the crossroads have multiple names). However, the crossroads at location L of Navigable Feature A and Navigable Feature B is given a single crossroads record ID, regardless of the number of Places containing that crossroads data.

When a supernode has been created in the SDAL database (e.g., to represent the intersection of doubly-digitized roads, or a roundabout), crossroads records are generated at the supernode rather than the underlying subordinate nodes. The underlying subordinate data is used to determine the attributes of the crossroads (overpass, intersection at grade, entrance or exit). When a roundabout is named, crossroads records are generated for the intersection of each named arm of the supernode within the roundabout. Even though the roundabout is completely internal to its supernode, it is not present after the supernode is collapsed. Since junctions can cover large areas and be complicated interchanges with many arms, supernodes representing junctions are ignored when generating crossroads data.

Where there is a navigable path comprising one or more ramp segments from roadway #1 to roadway #2, roadway entrances and exits are supported as follows:

- A crossroads record for roadway #1 and roadway #2, attributed as an *exit*, is created where a navigable path from roadway #1 to roadway #2 intersects roadway #2.

- A crossroads record for roadway #1 and roadway #2, attributed as an *entrance*, is created where a navigable path from roadway #1 to roadway #2 intersects roadway #1.

Signs at such entrances and exits are also added to the crossroads data in the form of Sign record DBIDs.

3.8.0.2 Spatial Access

Crossroads records are stored in the parcel in Peano Key ordered by position providing a minimally-sized kd-tree structure. The structure divides the parcel's minimal enclosing di-tile into 16 equal rectangles. Four bits are required to define the kd-tree, and for each of the 16 rectangles, (parcel header field) ucXrdsEntSize bits are needed to define the rectangle's range of crossroads records.

The field ucXrdsEntSize is the number of bits needed to represent the largest crossroads entity ID in the parcel (for example, 255 requires 8 bits, 511 requires 9 bits). The total size of the information required for an internal spatial search is therefore $16 * \text{ucXrdsEntSize} + 4$, which would, for example, be 21 bytes for a parcel containing 1000 crossroads records.

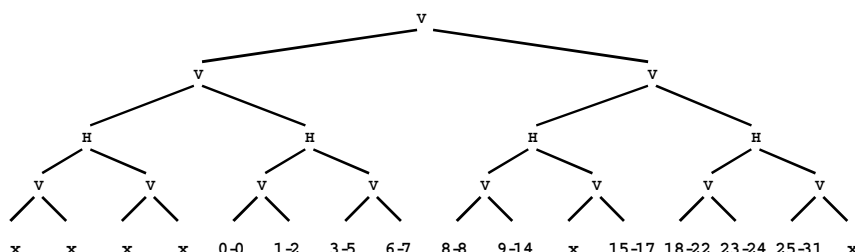
Assume the following represents the di-tile covering a crossroads parcel, and the number of crossroads records in each of its 16 subrectangles is as follows:

0	0	3	2	0	3	7	0
0	0	1	2	1	6	5	2

In this example, there are 31 crossroads in the parcel, so five bits are required to represent a crossroads entity ID. The size of the kd-tree is therefore $16 * 5 + 4 = 84$ bits = 11 bytes. Four bits define the cuts (vertical, vertical, horizontal, vertical), and five bits define the entity ID range in each subrectangle:

KD-Tree: Cuts: VVHV (Binary 0010) IDs: 0, 0, 0, 0, 1, 3, 6, 8, 9, 15, 15, 18, 23, 25, 32, 32.

For example, entry #8 is 9 and entry #9 is 15, which means that rectangle #9 contains entity IDs 9-14. Entry #9 is 15 and entry #10 is 15, which means that rectangle #10 contains no entity IDs.



B

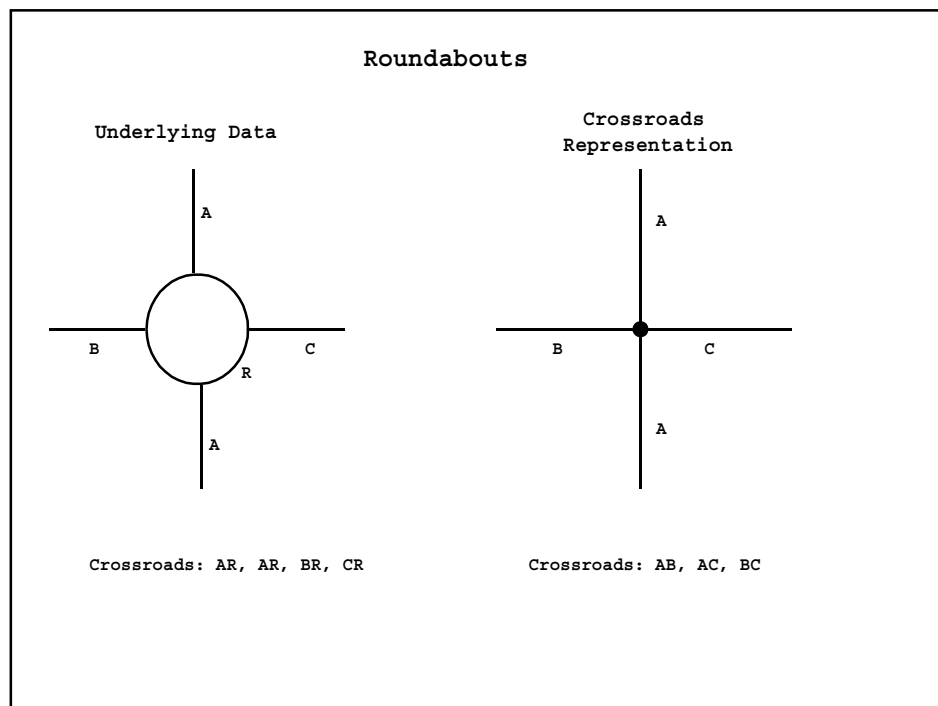


Figure 3-37: Crossroads Data for Roundabouts

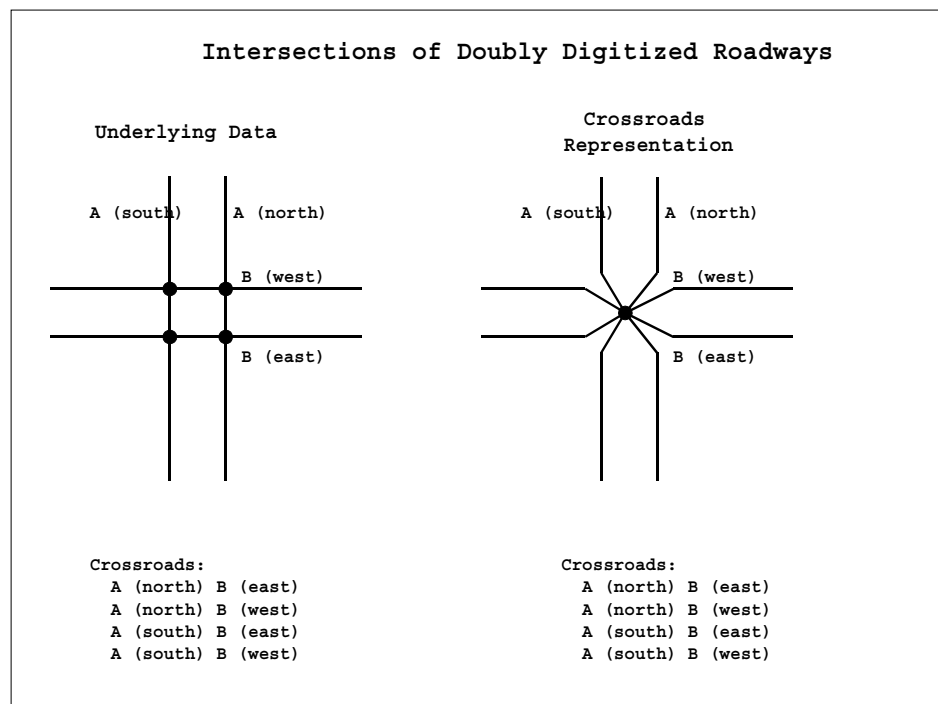


Figure 3-38: Crossroads Data for Doubly Digitized Roadways

3.8.1 Crossroads Parcel

Crossroads store information about the intersection of two navigable features. This information is used primarily for specifying the source or destination of a route. The Crossroads entity contains:

- The navigable feature IDs of the primary and intersecting features and the direction-on-sign associated with these features, where applicable. The direction-on-sign is the official directional identifier assigned to a highway and can be one of the four compass directions (N, S, E, W) or blank.
- The location of the intersection in terms of a Node ID for intersections at grade, or as a relative location along the primary segment for underpasses and overpasses.
- An attribute field containing bit flags that provide additional information about the crossroads, such as whether there are any sign records associated with the crossroads.

3.8.1.1 Technical Description

Crossroads parcels are spatially organized and are searchable via the global kd-tree.

3.8.1.2 Referenced By

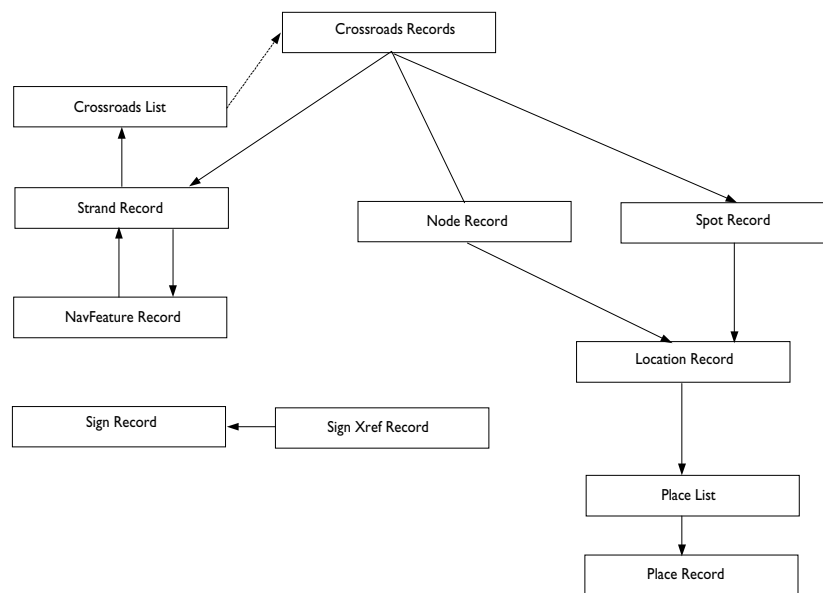


Figure 3-39: Crossroads Parcel Overview

3.8.1.3 Parcel Header

The following structure defines the initial, fixed portion of the parcel.

```

typedef struct
{
    SptlPclHdr_t  spatialHdr;      /* Common parcel header */
    BlkDesc3_t    xroadsDesc;      /* Crossroads Record Array */
    BlkDesc3_t    xroadsListDesc; /* Crossroads ID List Array */
    BldDesc3_t    placeDesc;       /* Place Record Array */
    BldDesc3_t    placeListDesc;  /* Place Index List Array */
    BldDesc3_t    strandDesc;     /* Place Record Array */

```

```

        BldDesc3_t    navFeatIDDesc;    /* Nav. Feature ID array */
        BldDesc3_t    locationDesc;     /* Location Record array */
        BldDesc3_t    nodeDesc;         /* Node Record array */
        BldDesc3_t    spotDesc;         /* Segment-Spot Record array*/
        BldDesc3_t    signXrefDesc;     /* Crossroads-Sign Xref array */
        BldDesc3_t    signDesc;         /* Sign Record array */
    }
    CrossroadsHeader_t;

    enum pcldescXroads{
        pdXroads_NavEnt,                /* Nav Feat EntID */
        pdXroads_PclIndex,              /* Pcl ID Tbl Index */
        pdXroads_StrandCnt,             /* Cnt of Strands/Nav Feat */
        pdXroads_PlcListIndex,          /* index to Place List */
        pdXroads_PlaceEnt,              /* Place EntID */
        pdXroads_PlcIndex,              /* index to Place Array */
        pdXroads_NavIndex,              /* index to Nav Feat Array */
        pdXroads_XrdsCnt,               /* Cnt of xrds/strand */
        pdXroads_XrdsListIndex,         /* index to Xrds List */
        pdXroads_StrandIndex,           /* index to Strand Array */
        pdXroads_NodeEnt,               /* Node EntID */
        pdXroads_LocIndex,              /* index to Location Array */
        pdXroads_SegEnt,                /* Segment EntID */
        pdXroads_Lat,                   /* Latitude Offset */
        pdXroads_Long,                  /* Longitude Offset */
        pdXroads_LocPlcCnt,             /* Cnt of Places/Location */
        pdXroads_NodeSpotIndex,         /* Node Record Index */
        pdXroads_XrdsEnt,               /* Crossroads EntID */
        pdXroads_SignEnt,               /* Place EntID */
        pdXroads_SignCnt,               /* Cnt of xrds/strand */
        pdXroads_SignListIndex          /* index to Xrds List */
    };

```

Per-parcel Data Arrays

```

    ParcelID_t        aPclId[];        /* Parcel ID array */

```

Crossroads Navigable Feature ID Array

Each entry in this array has the format shown below. If the length (in parentheses) of the field contains a name, the length is in that parcel descriptor table entry.

Field	Description										
NavFeatEntID	(pdXroads_NavEnt) Type IntP (Parcel-dependent, variable length) representing a Navigable Feature Entity ID.										
NavFeatPclID	(pdXroads_PclIndex) Type IntP (Parcel-dependent, variable length) representing a Navigable Feature Parcel ID, in the form of an <i>index</i> to this parcel's Parcel ID Array.										
Direction	(1 byte) Direction-on-Sign indicates the official directional identifiers assigned to highways; typically either "North/South" or "East/West."										
	<table> <tr> <th>Bit</th><th>Direction</th></tr> <tr> <td>000</td><td>blank</td></tr> <tr> <td>001</td><td>N</td></tr> <tr> <td>010</td><td>E</td></tr> <tr> <td>011</td><td>S</td></tr> </table>	Bit	Direction	000	blank	001	N	010	E	011	S
Bit	Direction										
000	blank										
001	N										
010	E										
011	S										

	100	W
StrandCount	(pdXroads_StrandCnt) Type IntP (Parcel-dependent, variable length) giving the number of Strand records associated with this Navigable Feature in this parcel.	
StrandListIndex	(pdXroads_StrandIndex) Type IntP (Parcel-dependent, variable length) representing an index to an array of Strand records. The index of the first of "Strand Count" strand records for this Navigable Feature.	

Crossroads Place Array

Each entry has the format shown below. If the length (in parentheses) contains a name, the length is in that parcel descriptor table entry.

Field	Description
PlaceEntID	(pdXroads_PlaceEnt) type IntP (Parcel-dependent, variable length) representing a Place Entity ID.
PlacePclID	(pdXroads_PclIndex) Type IntP (Parcel-dependent, variable length) representing a Place Parcel ID, in the form of an <i>index</i> to this parcel's Parcel ID Array.

Place List Array

Each entry has the format shown below. If the length (in parentheses) contains a name, the length is in that parcel descriptor table entry.

Field	Description
PlaceIndex	(pdXroads_PlcIndex) Type IntP (Parcel-dependent, variable length) representing an index to the Place Array.
PlaceType	(1 byte) Administrative area or postal zone.
<u>Bit</u>	<u>Indicates</u>
0x1	Administrative area
0x2	Postal zone

Crossroads Strand Array

Each strand record contains a Navigable Feature record index for the strand's primary Navigable Feature. It contains a count of crossroads and a Crossroads Array. Each Crossroads Array entry contains either:

- A Crossroads entity ID, referencing a crossroads record within the same parcel.
- A Crossroads entity ID plus an index to the Parcel ID table, referencing a crossroads record in another crossroads parcel.

Only the first and last entry in the Crossroads Array can reference crossroads in another parcel. Crossroads array entries are stored in the forward direction.

- For a one way strand, the forward direction is defined to be driving direction.
- For a two-way strand, the forward direction is defined arbitrarily (but consistently w-hen the strand extends across two or more crossroads parcels)

Each entry has the format shown below. If the length (in parentheses) contains a name, the length is in that parcel descriptor table entry.

Field	Description
NavFeatureIndex	(pdXroads_NavIndex) Type IntP (Parcel-dependent, variable length) representing an index to the Navigable Feature Array.
FirstExternalFlag	(1 bit) If set to 1, the first entry in the list of crossroads for this strand is external to this parcel, and is followed by a parcel ID table index.
LastExternalFlag	(1 bit) If set to 1, the last entry in the list of crossroads for this strand is external to this parcel, and is followed by a parcel ID table index.
Unused	(6 bits)
XrdsCount	(pdXroads_XrdsCnt) Type IntP (Parcel-dependent, variable length) giving the number of Crossroads associated with this Strand in this parcel.
XrdsListIndex	(pdXroads_XrdsListIndex) Type IntP (Parcel-dependent, variable length) representing an index to an array of Crossroads Entity IDs.

Crossroads Node Array

Each entry has the format shown below. If the length (in parentheses) contains a name, the length is in that parcel descriptor table entry.

Field	Description
NodeEntID	(pdXroads_NodeEnt) Type IntP (Parcel-dependent, variable length) representing a Node Entity ID.
NodePclID	(pdXroads_PclIndex) Type IntP (Parcel-dependent, variable length) representing a Routing Parcel ID, in the form of an <i>index</i> to this parcel's Parcel ID Array.
LocationIndex	(pdXroads_LocIndex) Type IntP (Parcel-dependent, variable length) representing an index to the Location Array.

Crossroads Spot Array

Each entry has the format shown below. If the length (in parentheses) contains a name, the length is in that parcel descriptor table entry.

Field	Description
SegmentEntID	(pdXroads_SegEnt) Type IntP (Parcel-dependent, variable length) representing a Segment Entity ID.
SegmentPclID	(pdXroads_PclIndex) Type IntP (Parcel-dependent, variable length) representing a Routing Parcel ID, in the form of an <i>index</i> to this parcel's Parcel ID Array.
Spot	(8 bits) A number in the range 0-255 indicating the percentage of the segment traversed (left node to right node) before the crossroads is reached.
LocationIndex	(pdXroads_LocIndex) Type IntP (Parcel-dependent, variable length) representing an index to the Location Array.

Crossroads Location Array

Each Location record contains a latitude and longitude, a count of places, and an array of Place Record indexes. If this location corresponds to:

- A single node, the location record also contains a Node ID.

Multiple nodes or no node (i.e., the crossroads is for an overpass), the crossroads record contains locus information. Each entry has the format shown below. If the length (in parentheses) contains a name, the length is in that parcel descriptor table entry.

Field	Description
Latitude	(pdXroads_Lat) Type IntP (Parcel-dependent, variable length) representing an offset from the latitude of the southwest corner of this parcel's bounding rectangle.
Longitude	(pdXroads_Long) Type IntP (Parcel-dependent, variable length) representing an offset from the longitude of the southwest corner of this parcel's bounding rectangle.
PlaceCount	(pdXroads_LocPlcCnt) Type IntP (Parcel-dependent, variable length) giving the number of Places associated with this location.
PlaceListIndex	(pdXroads_LocPlcIndex) Type IntP (Parcel-dependent, variable length) representing an Index to an array of Place indexes.

Crossroads Record Array

Each Crossroads record contains two strand record indexes and one location record index. If this crossroads represents an overpass, the record contains, for each of the two strands, either (1) a Node ID, or (2) a Segment ID and a Spot.

Each entry has the format shown below. If the length (in parentheses) contains a name, it is in that parcel descriptor table entry.

Field	Description
Strand1Index	(pdXroads_StrandIndex) Type IntP (Parcel-dependent, variable length) representing an index in the Strand Array.
Strand2Index	(pdXroads_StrandIndex) Type IntP (Parcel-dependent, variable length) representing an index in the Strand Array.
Strand1Flag	(1 bit) If set to binary 0, the Strand 1 Node field is present. If set to binary 1, the Strand 1 Spot field is present.
Attributes	(7 bits) <u>Bit</u> <u>Represents this attribute:</u>
	0x40 Sign(s) associated with this crossroads
	0x20 Exit from Strand 1 to Strand 2
	0x10 Entrance to Strand 1 from Strand 2
	0x08 Supernode
	0x04 Overpass (Strand 1 over Strand 2)
	0x02 Underpass (Strand 1 under Strand 2)
	0x01 Intersection at grade
Strand1Node	(pdXroads_NodeSpotIndex) Type IntP (Parcel-dependent, variable length) representing an index to the Node Array; present only if Strand 1 Flag set to binary 0.
Strand1Spot	(pdXroads_NodeSpotIndex) Type IntP (Parcel-dependent, variable length) representing an index to the Spot Array; present only if Strand 1 Flag set to binary 1.

Crossroads List Array

Each entry has the format shown below. If the length (in parentheses) contains a name, the length is in that parcel descriptor table entry.

Field	Description
CrossroadsIndex	(pdXroads_XrdsEnt) Type IntP (Parcel-dependent, variable length) representing a Crossroads Entity ID.

Crossroads Sign Xref Array

Each entry has the format shown below. If the length (in parentheses) contains a name, the length is in that parcel descriptor table entry.

Field (Length)	Description
CrossroadsIndex	(pdXroads_XrdsEnt) Type IntP (Parcel-dependent, variable length) representing a Crossroads Entity ID.
SignCount	(pdXroads_SignCnt) Type IntP (Parcel-dependent, variable length) giving the number of Signs associated with this Crossroads.
SignListIndex	(pdXroads_SignListIndex) Type IntP (Parcel-dependent, variable length) representing an index to an array of Sign Records.

Crossroads Sign Array

Each entry has the format shown below. If the length (in parentheses) contains a name, the length is in that parcel descriptor table entry.

Field (Length)	Description
SignEntID	(pdXroads_SignEnt) Type IntP (Parcel-dependent, variable length) representing a Sign Entity ID.
SignPclID	(pdXroads_PclIndex) Type IntP (Parcel-dependent, variable length) representing a Maneuver Parcel ID, in the form of an <i>index</i> to this parcel's Parcel ID Array.
DestinationIdx	(1 byte) The index of the destination within the sign record associated with the crossroads.

3.8.2 Crossroads Index Parcels

Crossroads Index Parcels contain the indexes used to quickly search Crossroads Parcels.

3.8.2.1 Technical Description

An index for obtaining Crossroads Parcel IDs for a given Navigable Feature Entity is dense, covering all existing crossroads records. Each index record contains a Navigable Feature Entity ID as a search key, and a Crossroads Parcel ID. A Crossroads Entity ID is not included in the index, since this entity ID is available via internal indexes in the Crossroads Parcel. Also, an index record exists only for the first Crossroads Parcel ID of each strand, because following Parcel IDs can be obtained from the data parcels. If the first and last Parcel ID in a strand are the same, there is only one index record.

The Cross-names index for obtaining Crossroads Parcel IDs for a given Navigable Feature Entity is also dense, covering all existing crossroads records. Each index record contains the Primary Navigable Feature Entity ID, Intersecting Navigable Feature Name and Entity ID, Place Parcel and Entity ID and Postal Parcel and Entity ID as search keys. Each record also has the Crossroad Parcel ID, Entity ID, and crossroad attributes.

3.8.2.2 Crossroads Index Parcel Header

Each B-tree index parcel has a header of the following structure:

```
typedef struct
```

```

{
    PclHdr_t    pclHdr;
    Ushort_t    usNumKeys;           /* number of keys in this parcel*/
    Ushort_t    usKeySize;           /* key size in bytes */
    Ushort_t    usChildParcelPosition; /*
        ** position of first child of
        ** this parcel where tree root
        ** is position 0
        */
    Uchar_t     ucTreeLevel;
        /*
        ** parcels that index records are
        ** at level 1; parcels that index
        ** level n index parcels are at
        ** level n+1
        */
    Uchar_t     ucReserved[1];       /* maintain 4-byte alignment */
}
BtreeIdxHdr_t;

```

Crossroads Index on Navigable Feature (sparse)

ucParcelType

CROSSROADS_NAV_IDX_PARCEL 248

Crossroads Index on Navigable Feature key record (sparse)

Field	Description												
ucNavFtrEntID	(3 bytes) Primary Navigable Feature Entity ID												
ucDirOnSign	(1 bytes) Direction-on-Sign indicates the official directional identifiers assigned to highways, usually “North/South” or “East/West”.												
	<table> <tr> <th><u>Bit</u></th><th><u>Direction</u></th></tr> <tr> <td>000</td><td>blank</td></tr> <tr> <td>001</td><td>N</td></tr> <tr> <td>010</td><td>E</td></tr> <tr> <td>011</td><td>S</td></tr> <tr> <td>100</td><td>W</td></tr> </table>	<u>Bit</u>	<u>Direction</u>	000	blank	001	N	010	E	011	S	100	W
<u>Bit</u>	<u>Direction</u>												
000	blank												
001	N												
010	E												
011	S												
100	W												
ucXroadPclID	(4 bytes) Crossroads Parcel ID												

Crossroads Index on Navigable Feature (dense)

CROSSROADS_NAV_PARCEL 14

Crossroads Index on Navigable Feature key record (dense)

Field	Description								
ucNavFtrEntID	(3 bytes) Primary Navigable Feature Entity ID								
ucDirOnSign	(1 bytes) Direction-on-Sign indicates the official directional identifiers assigned to highways, usually “North/South” or “East/West”.								
	<table> <tr> <th><u>Bit</u></th><th><u>Direction</u></th></tr> <tr> <td>000</td><td>blank.</td></tr> <tr> <td>001</td><td>N</td></tr> <tr> <td>010</td><td>E</td></tr> </table>	<u>Bit</u>	<u>Direction</u>	000	blank.	001	N	010	E
<u>Bit</u>	<u>Direction</u>								
000	blank.								
001	N								
010	E								

011 S
100 W

ucXroadPclID (4 bytes) Crossroads Parcel ID

Crossroad Name Index on Navigable Feature (dense)

ucParcelType

CROSSNAMES_NAV_IDX_PARCEL 225

Crossroad Name Index on Navigable Feature key record (sparse)

Field	Description
ucPrimNavFtrEntID	(3 bytes) Primary Navigable Feature Entity ID
ucXctNavFtrName	(4 bytes) Begin Intersecting Navigable Feature Name in Canonical Form.
ucXroadPclID	(4 bytes) Crossroads Parcel ID
ucXroadEntID	(4 bytes) Crossroads Entity ID
ucXctNavFtrEntID	(2 bytes) Intersecting Navigable Feature Parcel ID Index
ucXctNavFtrEntID	(3 bytes) Intersecting Navigable Feature Entity ID
ucPlacePclID	(1 byte) Place Parcel ID Index
ucPlaceEntID	(3 bytes) Place Entity ID
ucPostalPclID	(1 byte) Postal Parcel ID Index
ucPostalEntID	(3 bytes) Postal Entity ID
ucAttributes	(2 bytes) 4 Bit Flags indicate properties of the intersection, 8 bits indicate direction of travel on Primary and Intersecting Navigable Features, and 4 bits are reserved:
<u>Value</u>	<u>Property</u>
x8000	Overpass
x4000	Underpass
x2000	Exit
x1000	Entrance
Primary NavFeature Direction of Travel:	
x0200	N
x0400	E
x0600	S
x0800	W
x0000	blank
Intersecting NavFeature Direction of Travel	
x0100	N
x00c0	E
x0080	S
x0040	W
x0000	blank

Crossroad Name Index on Navigable Feature (dense)

CROSSNAMES_NAV_PARCEL

40

Crossroads Index on Navigable Feature key record (dense)

Field	Description																																		
ucPrimNavFtrEntID	(3 bytes) Primary Navigable Feature Entity ID																																		
ucXctNavFtrName	(4 bytes) Begin Intersecting Navigable Feature Name in Canonical Form.																																		
ucXroadPclID	(4 bytes) Crossroads Parcel ID																																		
ucXroadEntID	(4 bytes) Crossroads Entity ID																																		
ucXctNavFtrEntID	(2 bytes) Intersecting Navigable Feature Parcel ID Index																																		
ucXctNavFtrEntID	(3 bytes) Intersecting Navigable Feature Entity ID																																		
ucPlacePclID	(1 byte) Place Parcel ID Index																																		
ucPlaceEntID	(3 bytes) Place Entity ID																																		
ucPostalPclID	(1 byte) Postal Parcel ID Index																																		
ucPostalEntID	(3 bytes) Postal Entity ID																																		
ucAttributes	(2 bytes) 4 Bit Flags indicate properties of the intersection, 8 bits indicate direction of travel on Primary and Intersecting Navigable Features, and 4 bits are reserved:																																		
	<table> <tr> <th><u>Value</u></th><th><u>Property</u></th></tr> <tr> <td>x8000</td><td>Overpass</td></tr> <tr> <td>x4000</td><td>Underpass</td></tr> <tr> <td>x2000</td><td>Exit</td></tr> <tr> <td>x1000</td><td>Entrance</td></tr> <tr> <td colspan="2">Primary NavFeature Direction of Travel:</td></tr> <tr> <td>x0200</td><td>N</td></tr> <tr> <td>x0400</td><td>E</td></tr> <tr> <td>x0600</td><td>S</td></tr> <tr> <td>x0800</td><td>W</td></tr> <tr> <td>x0000</td><td>blank</td></tr> <tr> <td colspan="2">Intersect NavFeature Direction of Travel</td></tr> <tr> <td>x0100</td><td>N</td></tr> <tr> <td>x00c0</td><td>E</td></tr> <tr> <td>x0080</td><td>S</td></tr> <tr> <td>x0040</td><td>W</td></tr> <tr> <td>x0000</td><td>blank</td></tr> </table>	<u>Value</u>	<u>Property</u>	x8000	Overpass	x4000	Underpass	x2000	Exit	x1000	Entrance	Primary NavFeature Direction of Travel:		x0200	N	x0400	E	x0600	S	x0800	W	x0000	blank	Intersect NavFeature Direction of Travel		x0100	N	x00c0	E	x0080	S	x0040	W	x0000	blank
<u>Value</u>	<u>Property</u>																																		
x8000	Overpass																																		
x4000	Underpass																																		
x2000	Exit																																		
x1000	Entrance																																		
Primary NavFeature Direction of Travel:																																			
x0200	N																																		
x0400	E																																		
x0600	S																																		
x0800	W																																		
x0000	blank																																		
Intersect NavFeature Direction of Travel																																			
x0100	N																																		
x00c0	E																																		
x0080	S																																		
x0040	W																																		
x0000	blank																																		

Section 4 Non-Spatially Organized Data

4.1 Non-Spatial Data

Some of the data in the PSF region file is stored and accessed non-spatially. This data in the PSF includes:

- POI data
- Junctions
- Navigable Features
- Places
- Postal Codes
- RD-TMC Data

Non-spatially organized data has the following characteristics:

- The parcelization of the data occurs non-spatially, by index values. A parcel contains all of a particular entity's data within a range of b-tree indexes.
- All non-spatial parcels in the PSF region file are indexed by b-tree indexes.

4.1.1 POI Data

POI Parcels contain information about Points of Interest which are single points or polygonal areas typically known by name rather than by an address. POIs are associated with a segment based on their location. This provides a navigable destination which most directly leads to access of the POI. In addition to its location along a segment, the POI entity also contains street address and phone number information.

POIs may be associated with each other. A primary POI is called a parent. A parent POI may have many secondary POIs called children POIs. A child POI may also have many parent POIs.

4.1.2 Technical Description

POI Data is alphabetically organized and non-spatially parcelized. This localizes data for alphabetic searches by POI name. POI Entity IDs are unique across a region and are in alphabetic order. Any common initial characters of all names in a parcel up to a specified maximum length are stored at the `usCommonPrefixOffset` offset within the parcel. These common characters are excluded from the name string for each record, and the length of the common string is stored in the `ucCommonPrefixLength` field.

Internal indexes within a POI parcel search POIs by POI Name or POI Facility Type as follows:

1. By Name: Each entry of the "POI Name Prefix Index Table" contains name prefixes for every `ucRecsPerNameIndexEnt` records starting with the first record. These prefixes do not include the common prefix; the length of each prefix in this table is stored in the `ucNameIndexRecPrfxLen` field. A binary range search of this table provides a range of possible records (e.g., if the common prefix is "HI", then "Higgins" could be between the entries with prefixes "GA" and "GO"). The index within this table of the lower range entry is multiplied by `ucRecsPerNameIndexEnt` to get an index into the `POIOffsetEnt_t` offset array that provides an offset to the first record to examine. POI records are then sequentially scanned for records with the desired base name.
2. By POI Type: Each record in the "POI Type Index Table" contains a POI Facility Type, offset, and count of POI basename records of this type. The offset field specifies a parcel offset to a list of offsets to all POI basename records for a type. This table is ordered by facility type.

4.1.2.1 POI Data Parcel Header Information

The `POIParcelHeader_t` structure defines the initial, fixed portion of the parcel data.

```
/*
**POI Data Parcel Header
*/
typedef struct
{
    XrfPclHdr_t      xrfPclHdr;

    Ulong_t          ulFirstEntityID;      /* entity ID of the first POI
                                           record */
    BlkDesc2_t        poiDesc;              /* POI block descriptor */
    BlkDesc4_t        nameIndexDesc;        /* POI Name Prefix Index
                                           descriptor */
    BlkDesc3_t        dateTimeDesc;        /* Date-time block descriptor
                                           */
    BlkDesc4_t        typeDesc;            /* type index table descriptor
                                           */
    BlkDesc4_t        timeZoneIndexDesc;    /* Time Zone
                                           / DST Index Array */
    Ushort_t          usCommonPrefixOffset; /* offset to POI Name Common
                                           Prefix */
    Uchar_t           ucCommonPrefixLength; /* POI Name Common Prefix
                                           length */
    Uchar_t           ucNameIndexRecPrfxLen; /* Name Prefix Index length */
    Uchar_t           ucRecsPerNameIndexEnt; /* POI records/Name Prefix
                                           Index entry */
    Uchar_t           ucaReserved[3];      /* Maintain alignment */
}
PoiParcelHeader_t;

/*
** enumerated types for descriptor offsets array
*/
enum pcldescPOI {
    pdPOI_BaseLatitudeOffset,      /* Offset to parcel base latitude */
    pdPOI_BaseLatitudeOffsetDummy, /* next byte place holder */
    pdPOI_BaseLongitudeOffset,     /* Offset to parcel base latitude */
    pdPOI_BaseLongitudeOffsetDummy, /* next byte place holder */
    pdPOI_DescSize
};
```

Per-parcel Data Arrays

The locations in the POI parcel of the following arrays, and their sizes, are defined in the parcel header:

```
ParcelID_t          aPclId[];      /* Parcel ID Array */

Uchar_t             ucaTimeZoneIndex[]; /* Two-byte indices into TZ/DST
                                           Reference data
** 1st byte is TZ index, 2nd byte is DST Index */
```

4.1.2.2 POI Type Internal Index Table

This table is an array of POI types and the first two characters of POI names:

```
typedef struct poi_type
{
    Ushort_t  usType;          /* Index into global facility type array */
    Ushort_t  usCount;         /* Count of basename records this parcel this
                                type */
    Ushort_t  usOffset;        /* Offset to array of indices to basename
                                ** records of this type */
};
```

4.1.2.3 POI Record Offset Array

POI records are organized in POI ID order.

```
typedef Uchar_t POIOffsetEnt_t[6]; /* First bit:1 = multiword index
                                    record;
                                    ** 0 - normal record

    ** Next 23 bits: POI Entity ID
    ** Next two bytes: Record Offset.
    ** Next byte: Language Bitmask.
    */
POIOffsetEnt_t aPOIOffset[]; /* POI Record Offset Array */
```

4.1.2.4 Base Name POI Record

The Base Name POI record is a character array of variable length.

Field (Length)	Description
POI Attributes	(2 bytes) The following values are defined:
	0x4000 Partial long haul
	0x2000 Long haul
	0x1000 POI is on the upper side of the segment
	0x0800 POI is on the lower side of the segment
	0x0400 Synonyms/Exonyms present
	0x0200 POI has attribute modifiers
	0x0100 Chain ID present
	0x0080 Postal Code Xref present
	0x0040 TPDs present
	0x0020 DTMs present
	0x0010 POI has children
	0x0008 POI has parents
	0x0004 In Vicinity
	0x0001 POI is nationally important
Spot	(1 byte) A value from 0 to 255 indicating the scaled relative location of the POI along the associated segment. Zero corresponds to the left end of the segment; 255 corresponds to the right end of the segment.
Access Type	(1 byte) Specifies the type of access permitted to this entrance to the POI:

	<u>Value</u>	<u>Description</u>
	1	Main entrance
	2	Taxi entrance
	3	Loading dock
Num Places	(1 byte)	The count of Place IDs in the Place ID Array below.
Addr Length	(1 byte)	The length of the Address field.
Phone Length	(1 byte)	The length of the Phone Num field.
Lang. Code	(1 byte)	The language code index.
Name Length	(1 byte)	The length of the Name field.
Name	(variable)	Length is the number of bytes specified in the Name Length field; the POI name.
Address	(variable)	Length is the number of bytes specified in the Addr Length field; the POI street address.
Phone Num	(variable)	Length is the number of bytes specified in the Phone Length field; the POI phone number.
Latitude	(variable)	Length is the length of this value is specified in the IntBaseP type field at the offset given by the parcel descriptor index <i>pdPOI_BaseLatitudeOffset</i> . Refer to the <i>P2L Record Description Framework</i> document for details on the IntBaseP type. This relative value, when added to the parcel's base latitude, gives the latitude of this POI.
Longitude	(variable)	Length is the length of this value is specified in the IntBaseP type field at the offset given by the parcel descriptor index <i>pdPOI_BaseLongitudeOffset</i> . Refer to the "P2L Record Description Framework" document for details on the IntBaseP type. This relative value, when added to the parcel's base latitude, gives the longitude of this POI.
Facility Type	(1–3 bytes)	Type 1 Variable Length Unsigned Value.
Chain ID	(1–3 bytes)	Type 1 Variable Length Unsigned Value; present only if the "Chain ID present" bit of POI Attributes is set.
Segment ID	(4-5 bytes)	Consists of a three-byte segment Entity ID and a 1-2 byte Type 1 Variable Length Unsigned Value constituting an index to this parcel's Parcel ID Array for the segment Parcel ID..
Nav Feat ID	(4-5 bytes)	A three-byte Nav Feature Entity ID, followed by a 1-2 byte Type 1 Variable Length Unsigned Value representing the index to the Parcel ID Table for the Nav Feature Parcel ID. Set the Nav feature entity ID to 0xFFFFFFFF and the parcel ID index to 0x00 to indicate the absence of a Nav Feature Cross Reference.
Postal Code ID	(3-4 bytes)	A two-byte Postal Code Entity ID, followed by a Type 1 Variable Length Unsigned Value representing the index to the Parcel ID Table for the Postal Code Parcel ID. Set the "Postal Code Xref Present" bit of POI Attributes to 1 when present.
Place ID Array	(variable)	The value in the NumPlaces field. Each entry contains the following fields: 1. a two-byte Place Entity ID

	2. a 1-2 byte Type 1 Variable Length Unsigned Value representing the index to the parcel ID table of the Place Parcel ID						
	3. the one-byte Place Type:						
	<table> <tr> <th><u>Value</u></th><th><u>Description</u></th></tr> <tr> <td>1</td><td>Administrative area</td></tr> <tr> <td>2</td><td>Postal zone</td></tr> </table>	<u>Value</u>	<u>Description</u>	1	Administrative area	2	Postal zone
<u>Value</u>	<u>Description</u>						
1	Administrative area						
2	Postal zone						
DtTm Count	(1 byte) The number of date-time records attached to this POI. If present, set bit 0x0020 of POI Attributes to 1.						
DtTm Array	Length is two bytes multiplied by the value in DtTm Count. Each entry here is the offset of a date-time record. If present, set bit 0x0020 of POI Attributes to 1.						
Syn/Ex Cnt	(1 byte) The number of entries in the Syn/Ex Array (Synonym/Exonym Array). If present, set bit 0x0400 of POI Attributes to 1.						
Syn/Ex Array	The number of entries in this array is given by Syn/Ex Cnt. Each entry contains a Type 1 Variable Length Unsigned Value representing the index to the POI record offset array of a synonym or exonym of this base name. If present, set bit 0x0400 of POI Attributes to 1.						
Parent Count	(1 byte) The number of parent POIs. If present, set bit 0x0008 of POI Attributes to 1.						
Parent Array	(variable) If present, set bit 0x0008 of POI Attributes to 1. Each entry here is a three to five byte POI ID: The first three bytes are the entity ID of the parent POI. If the high order bit of the first byte is 1, a fourth byte exists and is the beginning of an index to this parcel's parcel ID array. This index, if it exists, is a Type 1 Variable Length Unsigned Value.						
Child Count	(1 byte) If present, set bit 0x0010 of POI Attributes to 1; the number of child POIs.						
Child Array	(variable) If present, set bit 0x0010 of the POI Attributes field to 1. Each entry here is a three to five byte POI ID: The first three bytes are the entity ID of the child POI. If the high order bit of the first byte is 1, a fourth byte exists and is the beginning of an index to this parcel's Parcel ID array. This index, if it exists, is a Type 1 Variable Length Unsigned Value.						
TPD Count	(1 byte) If this field is present, set bit 0x0040 of POI Attributes to 1; the number of TPD records in the TPD Array.						
TPD Array	(variable) Length is eight bytes multiplied by the value in TPD Count. If present, set bit 0x0040 of POI Attributes to 1. Each entry here contains the following: <table> <tr> <td>Vendor ID</td><td>(2 bytes)</td></tr> <tr> <td>TPD ID</td><td>(6 bytes)</td></tr> </table>	Vendor ID	(2 bytes)	TPD ID	(6 bytes)		
Vendor ID	(2 bytes)						
TPD ID	(6 bytes)						
Attr/Mod Cnt	(1 byte) This contains the number of entries in the Attr/Mod Array. If present, set bit 0x0200 of POI Attributes to 1.						
Attr/Mod Array	(variable) The number of entries is given by the Attr/Mod Cnt field. If present, set bit 0x0200 of POI Attributes to 1. Each entry here contains the following:						

Mod Length (2 bits) This value plus one contains the length of the Modifier field in bytes.

Attr Type (6 bits) Current Attribute Type values are:

<u>Value</u>	<u>Description</u>
000001	Cuisine Type
000010	Population
000011	Capital Indicator
Modifier	(variable) The length of this in bytes is one greater than the Mod Length field.

4.1.3 Synonyms and Exonyms

A PSF region can contain multiple countries, each of which has a single default language. Each POI has:

- Basename in its country's default language (designated "default" name)
- Multiple synonyms in its default language, as appropriate
- Exonym in a different language (one exonym per language maximum)

4.1.3.1 Synonym/Exonym POI Record

The POI record is a character array of variable length.

Field (Length)	Description
POI Attributes	(1 byte) The following values are defined: 0x80: Synonym-exonym record 0x40: Is partial long haul 0x20: Is long haul
Name Length	(1 byte) The length of the Name field.
Name (variable)	Length is the number of bytes specified in Name Length; the POI synonym or exonym name.
Base POI Record Parcel ID Table Index Present	(1 bit) Set to 1 if a Type1 value field follows the base record entity ID.
Base POI Record Entity ID (23 bits)	The entity ID of this synonym's or exonym's base POI record.
Base POI Record Parcel ID Table Index	(1-3 bytes) Type 1 Variable Length Unsigned Value; represents the index to the Parcel ID Table for the base POI record.

4.1.3.2 Multiword Index POI Record

This record is a character array of variable length.

Field (Length)	Description
POI Attributes	(1 byte) The following values are defined: 0x40 Partial long haul 0x20 Long haul
Multiword Offset	(1 byte) The offset to the base record name string of where the multiword begins.

Name Length	(1 byte) The length of the Name field.
Base Name	(variable) Length is the number of bytes specified in the Name Length field; the POI base name string.
Base POI Record Parcel ID Table Index Present (1 bit)	Set to 1 if the Base POI Record Entity ID is present.
Base POI Record Entity ID (23 bits)	The entity ID of this synonym's or exonym's base POI record.
Base POI Record Parcel ID Table Index (1-3 bytes)	Type1 Variable Length Unsigned Value; represents the index to the Parcel ID Table for the base POI record

4.1.3.3 Date-time Record

The date-time record is a character array of variable length. All date-time records are accessed directly as offsets. The maximum length for this record is 16 bytes. See the section "Route Calculation Data, Routing Date/Time Records" for the definition of Date/Time records.

4.1.4 POI Index Parcels

Three types of external indices exist to facilitate POI searches:

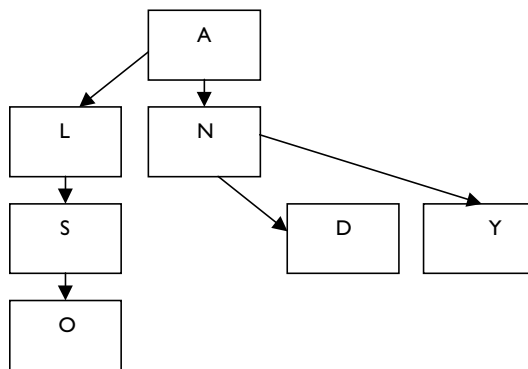
- character tree – searches strictly by POI name,
- spatial index – searches using a bounding geographic rectangle, and
- non-spatial index – searches using POI attributes such as type and place

4.1.4.1 Character Tree Index

The character tree index is used for POI searches by POI name including those necessary for next-valid character determination.

The POI name character tree extends to a depth to reach a unique parcel.

The character tree is constructed with each node containing a variable number of pointers to later nodes, one per valid character at this level. For example, to represent the words ALSO, AND, and ANY the following tree would be constructed:



A difference from other tree implementations is that only pointers to existing nodes are present. This means that nodes are variable-length.

The PSF character tree implementation continues the tree only to the point where there is a unique data parcel. The node then becomes a terminal node that points to the data parcel. This leads to an additional breakdown of nodes in the tree: internal and external. Trees are organized so nodes point to either external or internal nodes exclusively, with all external nodes contained in the same parcel, limiting external nodes to containing a single parcel index.

Node Structure

Nodes use parcel-dependent bit lengths with the number of bits specified in the parcel descriptor. There are three classes of nodes, with the distinction between them being made by the offset within the parcel. There are three offsets for a character tree parcel, which indicate the offset to the first node of each type:

```
typedef struct {
    Ushort_t      usChTreeIntNode;
    Ushort_t      usChTreeExtNode;
    Ushort_t      usChTreeTerm;
} ChTreeOffsets_t;
```

All nodes begin with the character the node represents. This is not limited to any particular range of values, so all code points are valid. The total number of child node offsets, a count of non-multiword children below this node and a count of multiword children below this node follow this character. All of these counts are parcel-dependent bit length fields.

If the tree is defined to have filter information, it is present at this point in the node. Examples of filter information are

- POI facility types
- POI chain ID's
- Place identification

External and terminal nodes have a parcel ID index that identifies the parcel with an array of parcel ID's in the parcel. This index is a parcel-dependent number of bits in length.

All non-terminal nodes have an array of offsets to child nodes. These are either simple offsets within the current parcel (for internal nodes) or offsets to nodes in a single external parcel. These offsets are 16 bits in length and are byte-aligned at the beginning of the array.

4.1.4.2 Spatial Index

POI spatial index parcels are spatially parcelized in the kd-tree format. They are used to search for POIs in a given geographical area. POIs are layered based upon the density grouping of each POI facility type. The number of layers is independent of routing or cartographic data and therefore can be varied during compile time based upon performance needs versus minimizing space. POI indexes contained in a particular density layer also exist at all lower layers. Each spatial index record contains fields for name length, full name, and latitude/longitude as well as chain ID and sub-type fields to allow filtering. An internal type index similar to that in the POI data parcel allows filtering by type. Each type's density indicator is stored along with its global type reference data.

Parcel Header

```
/*
**POI Spatial Index Parcel Header
*/
typedef struct
{
    SptlPclHdr_t      spatialHdr;

    BlkDesc2_t         poiDesc;          /* POI block descriptor */
    BlkDesc4_t         typeDesc;         /* type index table descriptor */
}
PoiSpatialIndexParcelHeader_t;
```

Per-parcel Data Arrays

The locations in the parcel of the following arrays, and their sizes, are defined in the parcel header:

```
ParcelID_taPclId[]; /* Parcel ID Array */
```

POI Type Internal Index Table

This table is an array of POI types and the first two characters of POI names:

```
typedef struct poi_type
{
    Ushort_t      usType;    /* Index into global facility type array */
    Ushort_t      usCount;   /* Count of basename recs this parcel this
                             type */
    Ushort_t      usOffset; /* Offset to array of indices to basename
                             recs of this type */
};
```

POI Spatial Index Record Offset Array

POI records are varying in length and are therefore accessed using the POI Record Offset Array.

```
Ushort_t    aPOIOffset[]; /* POI Spatial Index Record Offset Array */
```

Base Name Index Record

This record is a character array of variable length.

Field	Description
Attributes	(1 byte) The following values are defined: 0x40 Partial long haul 0x20 Long haul 0x02 Chain ID present 0x01 Subtype present
Lang. Code	(1byte) The language code index.
Name Length	(1 byte) The length of the Name field.
Name	(variable) The number of bytes specified in the Name Length field; the POI name.
Base POI Record Entity ID (1 byte) The entity ID of the base name POI record.	
Base POI Record Parcel ID Table Index (1–3 bytes) The index to the Parcel ID Table for the base name POI record.	
Latitude	(2-3 bytes) A Type 4 Variable Length Unsigned Value. To compute the value for the latitude field: subtract the minimum latitude (in NTUs) of the parcel's bounding rectangle from the latitude of the POI and right shift the result by a number of bits equal to the parcel's scale factor.
Longitude	(2-3 bytes) A Type 4 Variable Length Unsigned Value. To compute the value for the longitude field: subtract the minimum longitude (in NTUs) of the parcel's bounding rectangle from the longitude of the POI and right shift the result by a number of bits equal to the parcel's scale factor.
Facility Type	(1-3 bytes) A Type 1 Variable Length Unsigned Value; the facility type code of the POI.
Chain ID	(1-3 bytes) A Type 1 Variable Length Unsigned Value. Set the "Chain ID present" bit of Attributes to 1 if this is present; the chain ID of the POI.
Subtype	(1-3 bytes) A Type 1 Variable Length Unsigned Value. Set the "Subtype present" bit of Attributes to 1 if this is present; the subtype value of the POI.

Synonym/Exonym Index Record

This record is a character array of variable length.

Field	Description
Attributes	(1 byte) The following values are defined: 0x80 Synonym-exonym record 0x40 Partial long haul 0x20 Long haul
Lang. Code	(1 byte) The language code index.
Name Length	(1 byte) The length of the Name field.
Name	(variable) The number of bytes specified in the Name Length field; the POI name.
Base Record Index	(2 bytes) The parcel offset table index of the base name index record.
POI Record Entity ID	(1 byte) The entity ID of the POI synonym/exonym record.
POI Record Parcel ID Table Index (1-3 bytes)	Type1 Variable Length Unsigned Value; represents the index into the Parcel ID Table for the POI synonym/exonym record

4.1.4.3 Non-Spatial Index

The non-spatial index is used to search for POIs using various combinations of POI attributes such as type, sub-type, chain ID, postal code, and municipal level place ID. This index also contains information necessary for efficient searching of place names or IDs within a POI facility type.

POIs are grouped into two categories within any combination of search qualifiers: those that are sparse (full name); and those that are dense (partial name), across a database. For example, there are probably many POIs of type restaurant but relatively few POIs of type airport. A sparse/dense grouping bit indicates which type of bottom level index parcel (full or partial POI name) is referenced for POI searches. Each level above the bottom level has the parcel index and the byte offset to the parcel for the first index record as well as the number of these records for every valid qualifier combination at that level.

The bottom level records are groups of contiguous sequences of records with POI IDs and names (sorted by POI ID and thusly sorted by name) pointed to by upper level POI search qualifier combinations. Any group may span multiple bottom level parcels.

The full POI name is placed within the bottom level indexes for sparse POI groups while those for dense groups contain only part of the POI name. Determination of sparse vs. dense groupings is decided at compile time. The second and third level place name index records are sorted by place name (not by place ID) within their higher level qualifiers. This provides pre-sorted place name lists as well as faster next valid character searching.

Parcel Header

```
/*
**POI Non-Spatial Multi-level Index Parcel Header
*/
typedef struct
{
    PclHdr_t      pclHdr;
    BlkDesc4_t    indexRecordDesc;
}
PoiMultiLevelExtIndexHeader_t;
```

Topmost Root Level Index Record

This record is of fixed length. The following abbreviations are used in the field descriptions:

- IRC - Index Record Count
- IRMPI - Index Record Multi-level Parcel Index
- IRMPO - Index Record Multi-level Parcel Index

External Data Description

Field	Value
Index ID Position	17
Record Size	17
Key Size	0
Record DBID Size	0
Extraction Size	17
Filter Size	0
Name Size	0
DBID Type	RECORDDBID0_0
Number Filter Types	0

Field	Description
Flags Bitfield	(1 byte) Unused and currently present only for consistency.
Type IRC	(4 bytes) The top-level type index record count and represents the number of unique POI facility types in the database.
Type IRMPI	(2 bytes) The index of the multi-level parcel containing the first root-level type index record.
Type IRMPO	(2 bytes) The offset to the first root-level type index record within the multilevel parcel pointed to by the "Type IRMPI" field
Place ID IRC	(4 bytes) The top-level place ID index record count and represents the number of unique places in the database that have POIs
Place ID IRMPI	(2 bytes) The index of the multilevel parcel containing the first root-level place ID index record.
Place ID IRMPO	(2 bytes) The offset to the first root-level place ID index record within the multilevel parcel pointed to by the "Place ID IRMPI" field

Top-level Type Index Record**External Data Description**

Field	Value
Index ID Position	53
Record Size	51
Key Size	2
Record DBID Size	0
Extraction Size	49
Filter Size	0
Name Size	0
DBID Type	RECORDDBID0_0
Number Filter Types	0

This record is of fixed length.

Field	Description
Facility Type	(2 bytes) An index into the array of POI facility types in global data.
Flags Bitfield	(1 byte) The following values are defined: 0x80 Full name group
Place ID IRC	(4 bytes) The second level type/place ID index record count.
Place ID IRMPI	(2 bytes) The index of the multi-level parcel containing the first second level type/place ID index record.
Place ID IRMPO	(2 bytes) The offset to the first second level type/place ID index record within the multi-level parcel pointed to by the "Place ID IRMPI" field.
Place Name IRC	(4 bytes) The second level type/place name index record count.
Place Name IRMPI	(2 bytes) The index of the multi-level parcel containing the first second level type/place name index record.
Place Name IRMPO	(2 bytes) The offset to the first second level type/place name index record within the multi-level parcel pointed to by the "Place Name IRMPI" field.
Subtype IRC	(4 bytes) The second level type/subtype index record count.
Subtype IRMPI	(2 bytes) The index of the multi-level parcel containing the first second level type/subtype index record.
Subtype IRMPO	(2 bytes) The offset to the first second level type/subtype index record within the multi-level parcel pointed to by the "Subtype IRMPI" field.
Chain ID IRC	(4 bytes) The second level type/chain ID index record count.
Chain ID IRMPI	(2 bytes) The index of the multi-level parcel containing the first second level type/chain ID index record.
Chain ID IRMPO	(2 bytes) The offset to the first 2 nd -level type/chain ID index record within the multi-level parcel pointed to by the "Chain ID IRMPI" field.
Postal ID IRC	(4 bytes) The second level type/postal ID index record count.
Postal ID IRMPI	(2 bytes) The index of the multi-level parcel containing the first second level type/postal ID index record.
Postal ID IRMPO	(2 bytes) The offset to the first second level type/postal ID index record within the multi-level parcel pointed to by the "Postal ID IRMPI" field.
Bottom-level POI IRC	(4 bytes) The bottom-level POI index record count.
Bottom-level POI IRMPI	(2 bytes) The index of the full or partial name bottom-level POI index record parcel containing the first bottom-level POI index record for this group.
Bottom-level POI IRMPO	(2 bytes) The offset to the first bottom-level POI index record within the bottom-level POI index record parcel pointed to by the "Bottom-level POI IRMPI" field

Top-level Place ID Index Record**External Data Description**

Field	Value
Index ID Position	47
Record Size	13
Key Size	4

Field	Value
Record DBID Size	0
Extraction Size	9
Filter Size	0
Name Size	0
DBID Type	RECORDDBID0_0
Number Filter Types	0

This is also the record format for the second level Type/Place ID, third level Type/SubType/Place ID, and third level Type/Chain/Place ID index record. This record is of fixed length.

Field	Description
Place Parcel ID Index	(2 bytes) A place ID parcel index.
Place Entity ID	(2 bytes) A place entity ID.
Flags Bitfield	(1 byte) The following values are defined: 0x80 Full name group
Bottom-level POI IRC	(4 bytes) The bottom-level POI index record count.
Bottom-level POI IRMPI	(2 bytes) The index of the full or partial name bottom-level POI index record parcel containing the first bottom-level POI index record for this group.
Bottom-level POI IRMPO	(2 bytes) The offset to the first bottom-level POI index record within the bottom-level POI index record parcel pointed to by the Bottom-level POI IRMPI field.

Second Level Type/Subtype Index Record

External Data Description

Field	Value
Index ID Position	49
Record Size	35
Key Size	2
Record DBID Size	0
Extraction Size	33
Filter Size	0
Name Size	0
DBID Type	RECORDDBID0_0
Number Filter Types	0

This record is of fixed length.

Field	Description
Subtype Index	(2 bytes) An index to the array of POI subtypes in global data.
Flags Bitfield	(1 byte) The following values are defined: 0x80 Full name group
Bottom-level POI IRC	(4 bytes) The bottom-level POI index record count.
Bottom-level POI IRMPI	(2 bytes) The index of the full or partial name bottom-level POI index record parcel containing the first bottom-level POI index record for this group.

Bottom-level POI IRMPO	(2 bytes) The offset to the first bottom-level POI index record within the bottom-level POI index record parcel pointed to by the “Bottom-level POI IRMPI” field.
Place ID IRC	(4 bytes) The third level type/subtype/place ID index record count.
Place ID IRMPI	(2 bytes) The index of the multi-level parcel containing the first third level type/subtype/place ID index record.
Place ID IRMPO	(2 bytes) The offset to the first third level type/subtype/place ID index record within the multi-level parcel pointed to by the “Place ID IRMPI” field.
Place Name IRC	(4 bytes) The third level type/subtype/place name index record count.
Place Name IRMPI	(2 bytes) The index of the multi-level parcel containing the first third level type/subtype/place name index record.
Place Name IRMPO	(2 bytes) The offset to the first third level type/subtype/place name index record within the multi-level parcel pointed to by the “Place Name IRMPI” field.
Postal ID IRC	(4 bytes) The third level type/subtype/postal ID index record count.
Postal ID IRMPI	(2 bytes) The index of the multi-level parcel containing the first third level type/subtype/postal ID index record.
Postal ID IRMPO	(2 bytes) The offset to the first third level type/subtype/postal ID index record within the multi-level parcel pointed to by the “Postal ID IRMPI” field.

Second Level Type/Chain ID Index Record

External Data Description

Field	Value
Index ID Position	50
Record Size	35
Key Size	2
Record DBID Size	0
Extraction Size	33
Filter Size	0
Name Size	0
DBID Type	RECORDDBID0_0
Number Filter Types	0

This record is of fixed length.

Field	Description
Chain ID Index	Length: 2 bytes; An index to the array of POI chain IDs in global data.
Flags Bitfield	(1 byte) The following values are defined: 0x80 Full name group
Bottom-level POI IRC	(4 bytes) The bottom-level POI index record count.
Bottom-level POI IRMPI	(2 bytes) The index of the full or partial name bottom-level POI index record parcel containing the first bottom-level POI index record for this group.

Bottom-level POI IRMPO (2 bytes)	The offset to the first bottom-level POI index record within the bottom-level POI index record parcel pointed to by the “Bottom-level POI IRMPI” field.
Place ID IRC (4 bytes)	The third level type/chain/place ID index record count.
Place ID IRMPI (2 bytes)	The index of the multi-level parcel containing the first third level type/chain/place ID index record.
Place ID IRMPO (2 bytes)	The offset to the first third level type/chain/place ID index record within the multi-level parcel pointed to by the “Place ID IRMPI” field.
Place Name IRC (4 bytes)	The third level type/chain/place name index record count.
Place Name IRMPI (2 bytes)	The index of the multi-level parcel containing the first third level type/chain/place name index record.
Place Name IRMPO (2 bytes)	The offset to the first third level type/chain/place name index record within the multi-level parcel pointed to by the “Place Name IRMPI” field.
Postal ID IRC (4 bytes)	The third level type/chain/postal ID index record count.
Postal ID IRMPI (2 bytes)	The index of the multi-level parcel containing the first third level type/chain/postal ID index record.
Postal ID IRMPO (2 bytes)	The offset to the first third level type/chain/postal ID index record within the multi-level parcel pointed to by the “Postal ID IRMPI” field.

Second Level Type/Place Name Index Record

External Data Description

Field	Value
Index ID Position	48
Record Size	42
Key Size	36
Record DBID Size	4
Extraction Size	0
Filter Size	2
Name Size	35
DBID Type	RECORDDBID2_2
Number Filter Types	3
Filter 0 Type Size	FILTERPART_LANGUAGE 8
Filter 1 Type Size	FILTERPART_MULTIWORD 1
Filter 1 Type Size	FILTERPART_LONG_HAUL 1

This is also the record format for the third level Type/SubType/Place Name and third level Type/Chain/Place Name index record. This record is of fixed length.

Field	Description
multiword Offset	(1 byte) The offset into the place name string of where the multiword reference begins. It is zero when the index record references a non-multiword record.
Place Name	(35 bytes) The full place name string and is not null-terminated.
Place Parcel ID Index	(2 bytes) A place ID parcel index.
Place Entity ID	(2 bytes) A place entity ID.
Flags Bitfield	(1 byte) The following values are defined: 0x80 Full name group
Place Name Lang Code	(1 byte) The language code index for the place name.
Filters	(1 byte) The following values are defined: 0x80 Place is multiword. 0x40 Place is long haul.

Bottom-level POI ID/Full Name Index Record

External Data Description

Field	Value
Index ID Position	18
Record Size	8 + Longest string length
Key Size	1 + longest string length
Record DBID Size	5
Extraction Size	0
Filter Size	2
Name Size	Longest name length
DBID Type	RECORDDBID2_3
Number Filter Types	3
Filter 0 Type Size	FILTERPART_LANGUAGE 8
Filter 1 Type Size	FILTERPART_MULTIWORD 1
Filter 2 Type Size	FILTERPART_LONG_HAUL 1

This record is of fixed length.

Field	Description
Multiword Offset	(1 byte) Offset into the POI name string to the start of the multiword. This is zero for index records referencing non-multiword POI records..
POI Name	(Length is maximum length of any POI name in canonical form across the database). This field contains the full POI name in non-canonical form.
POI Parcel ID Index	(2 bytes) This field contains the index into the parcel ID table for the POI record parcel ID.
POI Entity	(3 bytes) This field contains the POI record entity ID.
Language Bitmask Index	(1 byte) This field contains an index into the list of language bitmask values in global data for the language bitmask of this POI.

Filters (1 byte) These bits are defined:

0x80 Multiword record

0x40 POI is long haul

Bottom-level POI ID/Partial Name Index Record

External Data Description

Field	Value
Index ID Position	19
Record Size	11
Key Size	4
Record DBID Size	5
Extraction Size	0
Filter Size	2
Name Size	4
DBID Type	RECORDDBID_2_3
Number Filter Types	3
Filter 0 Type Size	FILTERPART_LANGUAGE 8
Filter 1 Type Size	FILTERPART_MULTIWORD 1
Filter 1 Type Size	FILTERPART_LONG_HAUL 1

This is a fixed length record.

Field	Description
POI Name	(Length is maximum length of any POI name in canonical form across the database). This field contains the full POI name in non-canonical form.
POI Parcel ID Index	(2 bytes) This field contains the index into the parcel ID table for the POI record parcel ID.
POI Entity	(3 bytes) This field contains the POI record entity ID.
Language Bitmask Index	(1 byte) This field contains an index into the list of language bitmask values in global data for the language bitmask of this POI.
Filters	(1 byte) These bits are defined: 0x80 Multiword record 0x40 POI is long haul

4.2 Junction Data

Junction data is used primarily to determine way-points, as well as starting or ending points, in route calculation.

A SDAL junction is a named node (usually a supernode) in the database and typically identifies the location of a complex intersection or named roundabout.

4.2.0.1 Technical Description

SDAL Junctions are nodes with names associated with them, similar to the way segments have names. While a regular node (the intersection of multiple segments at grade) may have a name associated with it, the more usual case is for complex intersections of numerous segments and nodes to be named.

For this more usual case, a supernode is created to generalize the set of segments and nodes comprising this junction, and a single node is created in the database to represent this set. This node is then named.

4.2.1 Junction Parcel

The Junction Parcel contains Junction Records. The Junction Record:

- Has a logical relationship among a node, a navigable feature name, and a place.
- Represents a named junction, which occurs when a feature name is associated with a single intersection node or a node generalizing a complex intersection.

4.2.1.1 Technical Description

Junction records are grouped by Place Identifier. Within Place Identifier, junction data are organized by Junction Name (Nav Feature ID).

4.2.1.2 Referenced By

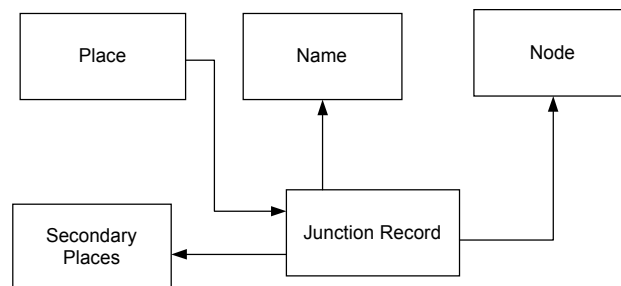


Figure 4-1: Junction Data References

4.2.1.3 Data Structures

```
typedef struct
{
    Ushort_t  usPlacePcl;    /* offset of ParcelID table Entry for this
                             place */
    Ushort_t  usEntId;       /* this Place's entity ID */
    Sshort_t  sFstJunct;     /* ID of 1st junction record, this place ID
                             */
    Sshort_t  sLstJunct;     /* ID of last junction record, this place ID
                             */
    DBRect_t  boundingRect;  /* Bounding Rectangle of this Place */
}
XRPlaceEntry_t;

typedef struct
{
    Ushort_t  usPlacePcl;    /* ParcelID table Entry offset for this
                             place */
    Ushort_t  usEntId;       /* this Place's entity ID */
}
XRSecondaryPlaceEntry_t;
```

4.2.1.4 Parcel Header

The following structure defines the initial, fixed portion of the parcel.

```
typedef struct
{
    XrfPclHdr_t  xrfPclHdr;    /* Common parcel header */
    BlkDesc_t    junctionDesc; /* Block offset array for junction
                             records */
    BlkDesc4_t   placeDesc;    /* Place Entry Array */
    BlkDesc4_t   scndPlaceDesc; /* Secondary Place Entry array */
    BlkDesc4_t   jctNavFeatIDDesc; /* Junction Nav Feature ID array */
}
JunctionHeader_t;
```

Per-parcel Data Arrays

```
ParcelID_t          aPclId[];    /* Parcel ID array      */
XRPlaceEntry_t      aPlaceEntry[]; /* Place entry array    */
XRSecondaryPlaceEntry_t aSecndPlaceEntry[]; /* secondary place level
                             array*/
```

Junction Nav Feature ID Array

Each entry in this array has the following format:

Field	Description
Nav Feat PclID	(2 bytes) Navigable Feature Parcel ID of the junction, in the form of an <i>index</i> into this parcel's Parcel ID Array.
Nav Feat EntID	(3 bytes) Navigable Feature Entity ID of the junction.

Junction Block Offset Array

Each entry of Junction Block Offset Array contains a block offset pointing to the beginning of a block of $2^K + 1$ Length fields followed by 2^K Junction records. The entity identifier is equal to the index of the record (within the parcel).

The length fields are all Type 1 Variable Length Unsigned Values. The first length field is the offset from the beginning of the block to the first Junction Record in the block, and the following 2^K Length fields are the lengths of each of the 2^K Junction records.

```
BlkOffset_t  aJunctionBlk[];    /* Junction block offset array */
```

4.2.1.5 Junction Record

Field	Description
Place ID	(1-3 bytes) (Type 1 Variable Length Unsigned Value) An index into the Place Entry Array.
Place Type	(1 byte) Administrative area or postal zone. 0x1 Administrative area 0x2 Postal zone
Nav Feat ID	(1-3 bytes) (Type 1 Variable Length Unsigned Value) The Navigable Feature ID for the junction; an index into the Junction Nav Feature ID Array.

The following two fields (2nday Places and Unused) occupy one byte:

2nday Places	(1 bit) If set to 1, this junction is in more than one low-level Place.
Unused	(7 bits)
Node ID	(4-6 bytes) The first three bytes are the node's entity ID. The fourth byte is the beginning of a Type 1 Variable Length Unsigned Value constituting an index into the Parcel ID Array. This parcel ID is for the <i>bottom layer</i> route calculation parcel containing this node
Num Places	(1 byte) Number of additional low-level place IDs associated with this junction; present only if the secondary Places bit is set.
Place ID Array	(variable) (variable length) The number of entries in this array is the value in the Num Places field. Each entry is made up of the following: (1) Type 1 Variable Length Unsigned Value representing an index into the Secondary Place Entry Array, and (1 byte) Place Type.

4.2.2 Junction Index Files

Junction Index files contain the indexes used to quickly search for the Junction record within Junction Parcels.

4.2.2.1 Technical Description

The Junction Index files use two keys: (1) Junction navigable feature ID and (2) Junction navigable feature ID and Place ID. Junction indexes are organized as B-trees of minimal height. Junction data are indexed as follows:

- Index on Junction Nav Feature ID
- Index On Place ID plus Junction Nav Feature ID

Index on Junction Nav Feature ID is composed of the two B-tree Parcels:

- Nav Feature Entity ID Ordered Junction Index (sparse) which is the Index to Nav Feature Entity ID Ordered Junction Index (dense).
- Nav Feature Entity ID Ordered Junction Index (dense) which is the Index to Junction Parcels.

Index on Place ID plus Junction Nav Feature ID is composed of the following B-tree Parcel:

- Place Ordered Nav Feature Entity ID Cross-Reference Junction Index (sparse) which is the Index to Junction Parcels.

4.2.2.2 Referenced By

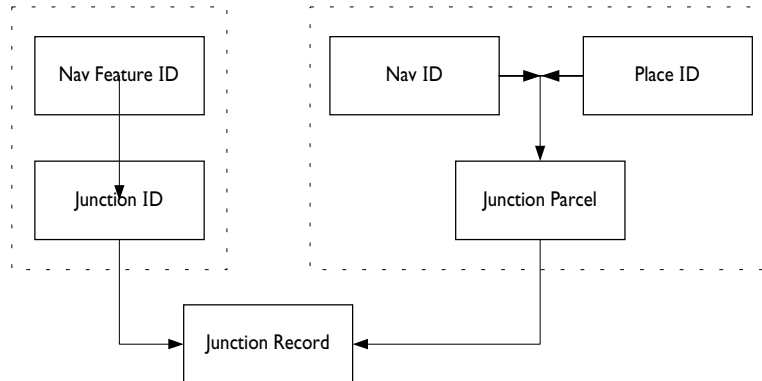


Figure 4-2: Junction Record

4.2.2.3 Junction Index Parcel Header

Each B-tree Index parcel has a header describing the Parcelization section.

```

typedef struct
{
    PclHdr_t    pclHdr;
    Ushort_t    usNumKeys; /* number of keys in this parcel */
    Ushort_t    usKeySize; /* key size in bytes */
    Ushort_t    usChildParcelPosition; /*
        ** position of first child of this
        ** parcel where tree root is
        ** position 0
        */
    Uchar_t     ucTreeLevel; /*
        ** parcels that index records are
        ** at level 1; parcels that index
        ** level n index parcels are at
        ** level n+1
        */
    Uchar_t     ucReserved[1]; /* maintain 4-byte alignment */
}
BtreeIdxHdr_t;
  
```

4.2.2.4 Nav Feature Entity ID Ordered Junction Index (sparse)

Nav Feature Entity ID Ordered Junction Index (sparse) is the Index to Nav Feature Entity ID Ordered Junction Index (dense).

Nav Feature Entity ID Ordered Junction Index consists of the Junction Index Parcel Header and usNumKeys field Nav Feature Entity ID ordered Junction Index key records. The fields in the parcel header have the following values:

Parcel Type	247
ucKeySize	3
ucTreeLevel	1

4.2.2.5 Nav Feature Entity ID Ordered Junction Index Key Record (sparse)

Nav Feature Entity ID Ordered Junction Index (dense) is the Index to Junction Parcels.

Field	Description
NavEntID	(3 bytes) Nav Feature Entity ID

4.2.2.6 Nav Feature Entity ID Ordered Junction Index (dense)

Nav Feature Entity ID ordered Junction Index (dense) consists of the Junction Index Parcel Header and usNumKeys Nav Feature Entity ID ordered Junction Index key records. The fields in the parcel header have the following values:

Parcel Type	15
ucTreeLevel	0

External Data Description

Field	Value
Index ID Position	16
Record Size	10
Key Size	3
Record DBID Size	6
Extraction Size	0
Filter Size	1
Name Size	0
DBID Type	RECORDDBID_4_2
Number Filter Types	1
Filter 0	
Type	FILTERPART_DUPLICATE
Size	1

4.2.2.7 Nav Feature Entity ID Ordered Junction Index Key Record (dense)

Field	Description
NavEntID	(3 bytes) Nav Feature Entity ID
JctPclID	(4 bytes) Junction Parcel ID
JctEntID	(2 bytes) Junction Entity ID
Filter	(1 byte) 1st bit indicates: 0x0 = Single Junction 0x1 = Duplicate Junction

4.2.2.8 Place Ordered Nav Feature Entity ID Cross-Reference Junction Index (sparse)

Place Ordered Nav Feature Entity ID Cross-Reference Junction Index (sparse) is the Index to Junction Parcels. It consists of the Junction Index Parcel Header and usNumKeys Place Ordered Nav

Feature Entity ID Cross-Reference Junction Index key records. The fields in this parcel header have the following values:

```
Parcel Type      239
ucTreeLevel     1
```

External Data Description

Field	Value
Index ID Position	23
Record Size	9
Key Size	9
Record DBID Size	0
Extraction Size	0
Filter Size	0
Name Size	0
DBID Type	RECORDDBID_0_0
Number Filter Types	0

4.2.2.9 Place Ordered Nav Feature Entity ID Cross-Reference Junction Index Key Record (sparse)

Field	Description
PlacePclID	(4 bytes) Place Parcel ID Index
PlaceEntID	(2 bytes) Place Entity ID
NavEntID	(3 bytes) Nav Feature Entity ID

4.3 Navigable Feature Data

Navigable Feature Data is ordered into two separate parcel types: alphabetical by base name and alphabetical by base name within Place.

Each of the two parcel types has an associated set of external indexes organized as a B-tree of minimal height. The alphabetical by base name within Place parcels have indexes that allow accessing the data by (1) Place Identifier plus Navigable Feature Base Name, (2) Postal Code Identifier plus Navigable Feature Base Name, and (3) Navigable Feature ID.

Navigable Feature Entity Identifiers are assigned to the records after they are sorted alphabetically by base name.

Street Type Data contains the list of valid street types in the database along with all aliases for these street types. The data is sorted alphabetically and can be searched independently of Navigable Features to provide support for recognizing street types within a given string, or for providing lists of valid street types and their aliases and abbreviations.

4.3.1 Navigable Feature Alphabetically Ordered Parcels

Navigable Feature Alphabetically Ordered Parcels contain Navigable Feature Records in basename alphabetical order.

4.3.1.1 Technical Description

Navigable Feature Alphabetically Ordered Parcels contain Navigable Feature Records, all of which are individually compressed.

4.3.1.2 Data Reference

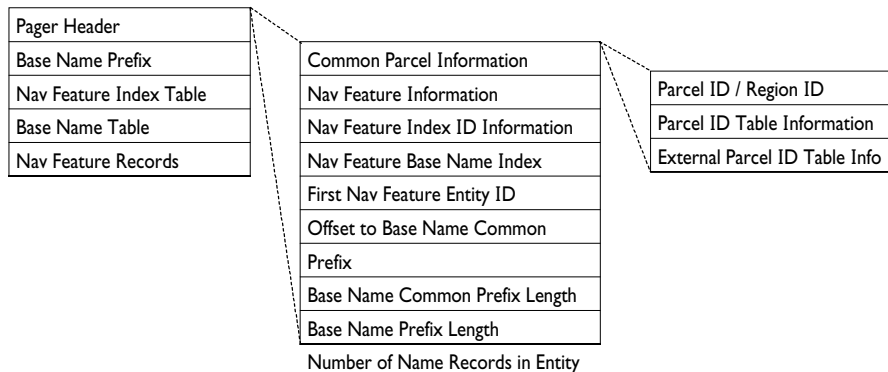


Figure 4-3: Navigable Feature Alphabetically Ordered Parcel Header Information

Parcel Header

The following structure defines the initial, fixed portion of the alphabetically ordered parcel.

```

struct
{
    XrfPclHdr_t xrfPclHdr;           /* Common Parcel Header */
    BlkDesc3_t  navFeatureDesc;      /* Nav Feature record descriptor */
    BlkDesc4_t  idIndexDesc;         /* nav feature ID index descriptor
                                     */
    BlkDesc4_t  baseNameDesc;        /* Nav Feature Base Name Index */
    Ulong_t     ulFirstNavFeature;    /* 1st Nav Feature entity ID in par-
                                     cel */
    Ushort_t    usBaseNamePrefixOffset; /* Offset to Nav Feature Base Name
                                     Common Prefix */
    Uchar_t     ucBaseNamePrefixLength; /* Base Name Common Prefix Length
                                     */
    Uchar_t     ucNavFtrRecPrfxLength; /* Length of the Base Name Prefix */
    Uchar_t     ucBaseNameRecsPerEnt; /* Nav Feature Base Name Recs per
                                     Entry */
}
NavFeatureParcelHeaderA_t;
    
```

Per-Parcel Data Arrays

The following array follows the parcel header. Its location and size are defined in the parcel header. The BaseNamePrefixArray is referenced by the usBaseNamePrefixOffset in the Alphabetically Ordered Navigable Feature Parcel Header.

```

Uchar_t    aBaseNamePrefixArray[]; /* Variable length Base Name Prefix
                                     array with a maximum size of 35
                                     bytes, even aligned */
    
```

Navigable Feature ID Index Table

The following array is indexed by the Navigable Feature ID minus First Navigable Feature ID in parcel.

```

Ushort_t    usaFeatureIdIndex[];    /* Array of Record Offsets */
    
```

Navigable Feature ID Index Record

Field	Description
Record Type	(1 bit) Indicates whether the record is a normal record or multiword index record.
	0x0 normal record
	0x1 multiword index record
Offset	(15 bit) Record Offset

Navigable Feature Base Name Index Table

This Navigable Feature Base Name Index Table is an array of two-character canonical prefixes:

```
typedef struct
{
    Uchar_t aPrefix[2]; /* Base Name Canonical Prefix (two char) */
}
BaseNameTable_t [];

BaseNameTable_t aBaseNameTable[];
```

Uncompressed Navigable Feature Record Format

The uncompressed format of the navigable feature record is a variable length character array as detailed below.

FieldDescription

Feature Type	(1-3 bytes) A Type 1 Variable Length unsigned value representing an index to a global array of Feature Types.
Language Code	(1 byte) Index to a global array of Language Codes.

The following two fields occupy one byte:

Attributes	(4 bits)
	0x0 = Street type following/unattached
	0x1 = Street type preceding/unattached
	0x2 = Street type following/attached
	0x3 = Street type preceding/attached

Route Type	(4 bits) The numeric route type the feature represents
Name Length	(1 byte) Contains the length of the uncompressed Name field.

The following two fields occupy one byte:

Long Haul Attribute	(2 bits) Long haul attribute values are:
	0x0 Not Long Haul
	0x1 Partial Long Haul
	0x2 Long Haul

Type Length	(6 bits) The length of the Street Type field.
-------------	---

The following two fields occupy one byte:

Prefix	(4 bits) An index into a global array of prefix strings.
Suffix	(4 bits) An index into a global array of suffix strings.

Name	(Name Length bytes) A variable-length character array containing the navigable feature basename. The size of this field is given by the Name Length field.
Street Type	(Type Length bytes) A variable-length character array containing the navigable feature street type. The size of this field is given by the Type Length field.

Navigable Feature multiword Index Record

This record is a variable-length character array.

Field	Description
-------	-------------

The first two fields that follow occupy three bytes:

PclID Flag	(1 bit) Set to 1, if the Ref PclID is present in this record. Identifies the parcel ID of the Navigable Feature Record referred to by this index record.
Ref EntID	(23 bits) The entity ID of the Navigable Feature Record referred to by this index record.

The following field is present only if the PclID Flag is set to 1.

Ref PclID	(2 bytes) Identifies the parcel ID of the Navigable Feature Record referenced by Ref EntID, as follows: The zero-based index of the Parcel containing the record, where the parcel with index zero is the first Navigable Feature b-tree index parcel.
Word Start	(1 byte) The zero-based index of the first character of the word (in the Name field of the Navigable Feature Record referenced by this record's Ref Entid field) for which this record was created.
Base Name Length	(2 bytes) The length of the Base Name field succeeding it.
Base Name	(Base Name Length) A variable-length character array containing the Navigable Feature base name. The size of this field is given by the Base Name Length field.

4.3.2 Navigable Feature Place-Ordered Cross-Reference Parcels

Navigable Feature Place-Ordered Cross-Reference Parcels contain a separate set of cross-reference records, ordered alphabetically by base name within Place.

4.3.2.1 Technical Description

Navigable Feature Alpha Place parcels are sorted by Navigable Feature within Place ID.

4.3.2.2 Data Reference

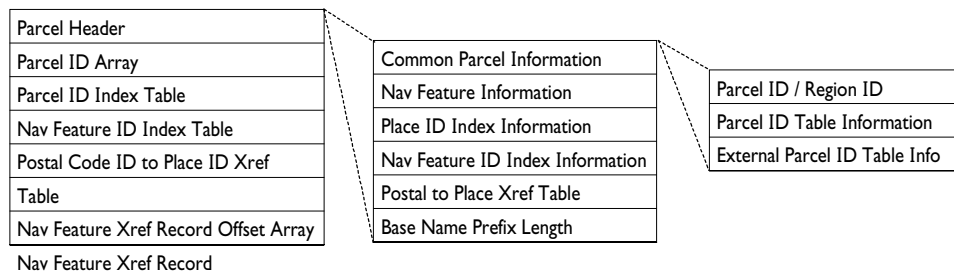


Figure 4-4: Navigable Feature Place Ordered Parcel Header Information

4.3.2.3 Navigable Feature Place-Ordered Cross-Reference Parcels

Parcel Header

The following structure defines the initial, fixed portion of the place-ordered cross-reference parcel.

```
typedef struct
{
    XrfPclHdr_t    xrfPclHdr;           /* Common Parcel Header */
    BlkDesc2_t     navFeatureDesc;      /* Nav Feature Xref record
                                         descriptor */
    BlkDesc4_t     placeIndexDesc;      /* Place ID index descriptor */
    BlkDesc4_t     featureIndexDesc;    /* Nav Feature ID index descrip-
                                         tor */
    BlkDesc4_t     postalPlaceXrefDesc; /* Postal to Place Xref Table */
    Uchar_t        ucNavFtrPrfxLength, /* Length of the Base Name Prefix
                                         */
}
NavFeatureParcelHeaderP_t;
```

Per-Parcel Data Arrays

The following arrays are at the bottom of the parcel. Their locations in the parcel and their sizes are defined in the parcel header. The Parcel ID Array and the External Parcel Array are referenced by the XrfPclHdr_t structure of the common parcel header.

```
ParcelID_t        aPclId[]; /* Parcel ID Array */
ExternalPcl_t     aExtPcl[]; /* External parcel Array - generally not
                               present */
```

Place ID Index Table

```
PlaceIDIndexTable_t  PlaceIDIndexTable[];
```

This table is an array of Place DBIDs; each entry points to an entry in the Navigable Feature Base Name Index Table.

Field	Description
Parcel Index	(2 bytes) The index into the Parcel ID array.
Place Entity ID	(2 bytes) The Place Entity ID.
NavFeature Count	(2 bytes) A count of the Navigable Features of this place, in the parcel.

OffsetFirstNavFeat (2 bytes) The offset of the first Navigable Feature in the parcel.

These entries are ordered by Place Parcel ID Index and Place Entity ID. Each entry contains an offset into the Navigable Feature ID Index table.

Navigable Feature ID Index Table

```
NavFeatureIDIndexTable_t NavFeatureIDIndexTable[];
```

This table is an array of Navigable Feature Entity IDs plus offsets to the corresponding Navigable Feature records.

Field	Description
Offset to Index	(2 bytes) The offset to the index record in the NavFeature Record Offset array.
Name Prefix	(2 bytes) The two-character Navigable Feature Name prefix.

Postal Code ID to Place ID Cross-Reference Table

```
PostalPlaceXrefTable_t PostalPlaceXrefTable[];
```

This table is an array of Postal Code ID and Place ID combinations. If a Postal Code contains multiple places, that postal code appears multiple times in this table. The “offset to place table” field is set to the special value 0xFFFF, if a postal code appears only on the boundary of a place included in this parcel; it is present in this table only as a place where the full Postal Code ID can be looked up.

Field	Description
Place Entity ID Index	(2 bytes) The index for the Place Entity ID in the Postal Code ID to Place ID Cross Reference Table.
Postal Code Entity ID	(2 bytes) The Postal Code Entity ID.
Offset to Place Table	(2 bytes) An offset to the PlaceIDIndexTable for this Place Entity ID.

Navigable Feature Cross-Reference Record Offset Array

```
NavFeatureXrefRecOffsetArray_t aNavFeatureXrefRecOffsetArray[];
```

Nav Feature Offset	(2 bytes) The offset to the Nav Feature Record.
Field	Description.
Nav Feature Parcel ID Index	(2 bytes) The index to the Navigable Feature Place Ordered Cross Reference Parcel's Parcel ID array.
Nav Feature EntityID	(3 bytes) The Navigable Feature Entity ID.

Navigable Feature Cross-Reference Record

The Navigable Feature Cross-Reference Record is a variable-length character array.

Field (Length)	Description
Number of Postal Code IDs (1 byte)	The number of Postal Code identifiers in the Postal ID array.
Number of Maneuver Parcel IDs (1 byte)	The number of Maneuver Parcel identifiers in the Maneuver Parcel ID array.
Postal ID Array	(2 byte entry) An index to the Postal Code ID to Place ID Cross-Reference Table. The full Postal Code Parcel ID and Entity ID can be obtained from this table.
Maneuver Parcel ID Array (Number of Maneuver Parcel IDs)	A variable length array. Each entry in this array is a Type 1 Variable Length unsigned value representing an index to the Maneuver Parcel ID array.

4.3.3 Navigable Features Data Index Files

Navigable Features Data Index Files are the indexes of Navigable Features Data Parcels. The following index files are B-trees of minimum height containing key values and file offsets of Navigable Features Data Parcels for those key values.

- Navigable Feature Alphabetically Ordered Index
- Navigable Feature Place Ordered Cross-Reference Index
- Navigable Feature Full Name Ordered Index
- Navigable Feature Next-Char-Alpha Index
- Navigable Feature Alpha-Place Index
- Navigable Feature Highway Index
- Navigable Feature by Postal Code Index
- Navigable Feature Street Type Index
- Navigable Feature by Country Index

4.3.3.1 Technical Description

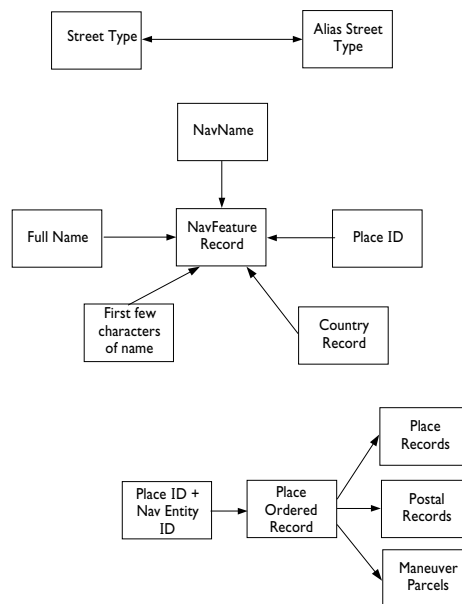
Navigable Features Data Index Files are B-tree Index Parcels of minimum height, of the same single fixed size as the record parcels that they index. The index files contain key values and file offsets of Navigable Feature parcels for those key values. They are stored contiguously in breadth-first tree order and precede the record parcels in a single contiguous storage sequence of index parcels followed by record parcels.

Databases containing more than one country may contain Navigable Feature by Country Index and Data parcels. If there are countries in a multiple country database containing less than a certain threshold of navigable features, Navigable Feature by Country Index and Data parcels are present for these countries. For example, Benelux contains Navigable Feature by Country Index and Data parcels for Luxembourg.

- Navigable Feature Alphabetically Ordered Index constructed of: Navigable Feature Alphabetically Ordered Index Parcels (sparse) — the index to Navigable Feature Alphabetically Ordered Data Parcels.
- Navigable Feature Place Ordered Cross-Reference Index constructed of Navigable Feature Place Ordered Cross-Reference Index Parcels (sparse) — the index to Navigable Feature Place Ordered Cross-Reference Data Parcels.
- Navigable Feature Full Name Ordered Index constructed of
 - (1) Navigable Feature Full Name Ordered Index Parcel (sparse) — the index to Navigable Feature Full Name Ordered Index Parcels (dense);
 - (2) Navigable Feature Full Name Ordered Index Parcels (dense) — the index to Navigable Feature Alphabetically Ordered Data Parcels.
- Navigable Feature Next-Char-Alpha Index constructed of
 - (1) Navigable Feature Next-Char-Alpha Index Parcel (sparse) — the index to Navigable Feature Next-Char-Alpha Index Parcels (dense).
 - (2) Navigable Feature Next-Char-Alpha Index Parcels (dense) — the index to Navigable Feature Alphabetically Ordered Data Parcels.
- Navigable Feature Alpha-Place Index constructed of
 - (1) Navigable Feature Alpha-Place Index Parcel (sparse) — the index to Navigable Feature Alpha-Place Index Parcels (sparse).
 - (2) Navigable Feature Alpha-Place Index Parcels (dense) — the index to Navigable Feature Alphabetically Ordered Data Parcels.
- Navigable Feature Highway Index constructed of Navigable Feature Highway Index Parcels (dense) — the index to Navigable Feature Alphabetically Ordered Data Parcels.

- Navigable Feature Street Type Index constructed of
 - (1) Navigable Feature Street Type Index Parcel (sparse) — the index to Navigable Feature Street Type Data Parcels (dense).
 - (2) Navigable Feature Street Type Index Parcels (dense) — the index to Navigable Feature Alphabetically Ordered Data Parcels.
- Navigable Feature by Postal Code Index constructed of
 - (1) Navigable Feature by Postal Code Index Parcel (sparse) — the index to Navigable Feature Postal Code Data Parcels (dense).
 - (2) Navigable Feature Postal Code Index Parcels (dense) — the index to Navigable Feature Alphabetically Ordered Data Parcels.
- Navigable Feature by Country Index constructed of
 - (1) Navigable Feature by Country Index Parcel (sparse) — the index to Navigable Feature by Country Index Parcels (dense).
 - (2) Navigable Feature by Country Index Parcels (dense) — the index to Navigable Feature Alphabetically Ordered Data Parcels.

4.3.3.2 Referenced By



Navigable Feature Data Index Parcel Header

Each B-tree index parcel contain a header of the following structure:

```

typedef struct
{
    PclHdr_t    pclHdr;
    Ushort_t    usNumKeys; /* number of keys in this parcel */
    Ushort_t    usKeySize; /* key size in bytes */
    Ushort_t    usChildParcelPosition; /*
        ** position of first child of this
        ** parcel where tree root is
        ** position 0
        */
}
  
```

```

    Ushort_t  usExtensionOffset; /* if non-zero, offset to
                                   Extension of header */

    Uchar_t   ucTreeLevel; /*
                            ** parcels that index records are
                            ** at level 1; parcels that index
                            ** level n index parcels are at
                            ** level n+1
                            */
    Uchar_t   ucReserved[3]; /* maintain 4-byte alignment */
  }
  BTreeIdxHdr_t;

```

Navigable Feature Indexes

Navigable Feature Alphabetically Ordered Index Parcels (sparse)

Navigable Feature Alphabetically Ordered Index contains the:

- B-Tree Index Parcel Header
- Number of *usNumKeys* field Basename records.

The following fields in this parcel header have the following values:

```

Parcel Type      254
ucKeysize        10
ucTreeLevel      1

```

External Data Description

Field	Value
Index ID Position	9
Record Size	10
Key Size	10
Record DBID Size	0
Extraction Size00	0
Filter Size	0
Name Size	0
DBID Type	RECORDDBID_0_0
Number Filter Types	0

Navigable Feature Alphabetically Ordered Index Key Record

Field	Description
caBaseName	(10 bytes) First 10 bytes of Basename

Navigable Feature Place Ordered Cross-Reference Index Parcels (sparse)

Navigable Feature Place Ordered Cross-Reference Index contains the:

- B-Tree Index Parcel Header
- Number of *usNumKeys* field Place ID records

The following fields in this parcel header have the following values:

```

Parcel Type      253
ucKeysize        8
ucTreeLevel      1

```

External Data Description

Field	Value
Index ID Position	10
Record Size	8
Key Size	8
Record DBID Size	0
Extraction Size	0
Filter Size	0
Name Size	2
DBID Type	RECORDDBID_0_0
Number Filter Types	0

Navigable Feature Place Ordered Cross-Reference Index Key Record (sparse)

Field	Description
Place Parcel ID	(4 bytes) Place Parcel ID
Place Entity ID	(2 bytes) Place Entity ID
Nav Feature BaseName	(2 bytes) First 2 bytes of Navigable Feature Base Name

Navigable Feature Full Name Ordered Index Parcels (dense)

Navigable Feature Full Name Ordered Index contains the:

- B-Tree Index Parcel Header
- Number of *usNumKeys* field Full Name records

The following fields in this parcel header have the following values:

Parcel Type	227
ucKeysize	20
ucTreeLevel	1

Navigable Feature Full Name Ordered Index Key Record (dense)

Field	Description
caBaseName	(20 bytes) First 20 bytes of Base Name

Navigable Feature Full Name Ordered Index Parcel (dense)

Navigable Feature Full Name Ordered Index contains the:

- B-Tree Index Parcel Header
- Number of *usNumKeys* field Base Name DBID records

The following fields in this parcel header have the following values:

Parcel Type	38
ucTreeLevel	0

External Data Description

Field	Value
Index ID Position	36
Record Size	25
Key Size	20
Record DBID Size	5
Extraction Size	0
Filter Size	0
Name Size	0
DBID Type	RECORDDBID_2_3

Field	Value
Number Filter Types	1
Filter 0 Type Size	FILTERPART_PERMUTEDNAME 1

Navigable Feature Full Name Ordered Index Key Record (dense)

Field	Description
caBaseName	(20 bytes) First 20 bytes of Base Name
caNavFeatPrIID	(2 bytes) Navigable Feature Parcel ID
caNavFeatEntID	(3 bytes) Navigable Feature Entity ID

Navigable Feature Next-Char-Alpha Index Parcel (dense)

Navigable Feature Next-Char-Alpha index Parcel (sparse) contains the:

- B-Tree Index Parcel Header
- Number of usNumKeys field Base Name characters.

The following fields in this parcel header have the following values:

Parcel Type	229
ucKeysize	4
ucTreeLevel	1

Navigable Feature Next-Char-Alpha Index Record (dense)

Field	Description
caBaseName	(3 bytes) First 3 bytes of Base Name
ucFilter	(1 bit) Record type
	0x0 Normal record
	0x1 multiword record
unused	(7 bit)

Navigable Feature Next-Char-Alpha Index Parcel (dense)

Navigable Feature Next-Char-Alpha index Parcel (dense) contains the:

- B-Tree Index Parcel Header
- Number of usNumKeys field Base Name characters.

The following fields in this parcel header have the following values:

Parcel Type	36
ucTreeLevel	0

External Data Description

Field	Value
Index ID Position	34
Record Size	4
Key Size	3
Record DBID Size	0
Extraction Size	0
Filter Size	1
Name Size	3
DBID Type	RECORDDBID_0_0
Number Filter Types	1

Field	Value
Filter 0	
Type	FILTERPART_MULTIWORD
Size	1

Navigable Feature Next-Char-Alpha Index Key Record (dense)

Field	Description
caBaseName	(3 bytes) First 3 bytes of Base Name
ucFilter	(1 bit) Record type
	0x0 Normal record
	0x1 multiword record
unused	(7 bit)

Navigable Feature Alpha-Place Index Parcel (sparse)

Navigable Feature Alpha-Place index Parcel (sparse) contains the:

- B-Tree Index Parcel Header
- Number of usNumKeys field Place DBID records

The following fields in this parcel header have the following values:

Parcel Type	252
ucKeysize	9
ucTreeLevel	1

Navigable Feature Alpha-Place Index Key Record (sparse)

Field	Description
EntID	(3 bytes) Navigable Feature Entity ID
PlaceID	(4 bytes) Place Parcel ID
PlaceEntID	(2 bytes) Place Entity ID

Navigable Feature Alpha-Place Index Parcel (dense)

Navigable Feature Alpha-Place Index Parcel (dense) contains the:

- B-Tree Index Parcel Header
- Number of usNumKeys Alpha-Place index records

The following fields in this parcel header have the following values:

Parcel Type	11
ucTreeLevel	0

External Data Description

Field	Value
Index ID Position	12
Record Size	14
Key Size	3
Record DBID Size	6
Extraction Size	5
Filter Size	0
Name Size	0
DBID Type	RECORDDBID_4_2
Number Filter Types	0

Navigable Feature Alpha-Place Index Key Record (dense)

Field	Description
Entity ID	(3 bytes) Nav Feature Entity ID
Place Pcl ID	(4 bytes) Place Parcel Parcel ID
Place Entity	(2 bytes) Place Entity ID
Place Ordered Pcl	(4 byte) Nav Feature Place Parcel Index
Relationship	(1 bit) Administrative/Zone Relationship
	0x0 Administrative
	0x1 Zone
unused	(7 bit)

Navigable Feature By Postal Code Index Parcel (sparse)

Navigable Feature By Postal Code index Parcel (sparse) contains the:

- B-Tree Index Parcel Header
- Number of usNumKeys field Postal ID Base Name Records

The following fields in this parcel header have the following values:

Parcel Type	223
ucKeysize	7
ucTreeLevel	1

Navigable Feature By Postal Code Index Key Record (sparse)

Field	Description
EntID	(1 bytes) Navigable Feature Entity ID
PostalcodeEntID	(4 bytes) Postal Code Entity ID
caBasename	(4 bytes) First four characters from Nav Feature basename.

Navigable Feature By Postal Code Index Parcel (dense)

Navigable Feature By Postal Code Index Parcel (dense) contains the:

- B-Tree Index Parcel Header
- Number of usNumKeys postal to Navigable Feature Ids.

The following fields in this parcel header have the following values:

Parcel Type	42
ucTreeLevel	0

External Data Description

Field	Value
Index ID Position	42
Record Size	13
Key Size	7
Record DBID Size	5
Extraction Size	0
Filter Size	1
Name Size	4
DBID Type	RECORDDBID_2_3
Number Filter Types	1

Field	Value
Filter 0	
Type	FILTERPART_MULTIWORD
Size	1

Navigable Feature By Postal Code Index Key Record (dense)

Field	Description
Postalcode Pcl ID	(1 bytes) Postal Code Parcel ID
Postalcode Ent ID	(2 bytes) Postal Code Entity ID
caBasename	(4 bytes) First four characters from Navfeature basename.
Navfeature Pcl ID	(2 bytes) Nav Feature Parcel ID
Navfeature Entity ID	(3 bytes) Navfeature Entity ID
Attributes	(1 bit) Multiword Flag
	0x0 Normal record
	0x1 multiword record
unused	(7 bit)

Navigable Feature Street Type Index Parcel (sparse)

Navigable Feature Street Type index Parcel (sparse) contains the:

- B-Tree Index Parcel Header
- Number of usNumKeys field Street Type Index records

The following fields in this parcel header have the following values:

Parcel Type	226
ucKeySize	Identical to Dense Record
ucTreeLevel	1

Navigable Feature Street Type Index Parcel (dense)

Navigable Feature Street Type Index contains the:

- B-Tree Index Parcel Header
- Number of usNumKeys field Street Type Index records

The following fields in this parcel header have the following values:

Parcel Type	39
ucTreeLevel	0

External Data Description

Field	Value
Index ID Position	37
Record Size	Sum of following
Key Size	Maximum type length
Record DBID Size	0
Extraction Size	6+(3 x MaxAliases)
Filter Size	1
Name Size	Maximum type length
DBID Type	RECORDDBID_INDEX
Number Filter Types	1
Filter 0	
Type	FILTERPART_MULTIWORD
Size	1

Navigable Feature Street Type Index Key Record (dense)

Field	Description
Street Type Name	(n bytes) The null terminated street type string; as long as the longest non-null terminated street type name in the database.
Parcel ID Base Type	(1 byte) Street Type parcel index of base street type; if this is an alias, or self if base type; for aliases and multiwords, the ID of the base type.
Entity ID Base Type	(2 bytes) Street Type Entity ID of base street type, if this is an alias, or self if base type. For aliases and multiwords, this is the ID of the base type.
Attributes	(1 byte) Bit Mask of valid uses for this street type
	0x01 Following Attached
	0x02 Following Unattached
	0x04 Before Attached
	0x08 Before Unattached
	0x10 In Database
	0x20 Preferred
Language	(1 byte) Index to global data table of language codes.
Number Aliases	(1 byte) Number of valid aliases for a street type.
Alias IDs	(3 * m bytes) <i>m</i> is the maximum number of aliases for any street type in the database. Each entry is : 1 byte Parcel Index of alias 2 byte Entity ID of alias

Navigable Feature By Country Index Parcel (sparse)

Navigable Feature By Country index contains the:

- B-Tree Index Parcel Header
- Number of *usNumKeys* field Navigable Feature by Country Index Records

The following fields in this parcel header have the following values:

Parcel Type	222
ucKeysize	6
ucTreeLevel	1

Navigable Feature By Country Index Key Record (sparse)

Field	Description
uiCountryPID	(1 byte) Country Place Parcel ID offset
uiCountryEID	(2 bytes) Country Place Entity ID
caNavName	(3 bytes) First three bytes of Nav Feature name in canonical form.

Navigable Feature By Country Index Parcel (dense)

Navigable Feature By Country Index contains the:

- B-Tree Index Parcel Header
- Number of *usNumKeys* field Navigable Feature by Country Index records.

The following fields in this parcel header have the following values:

Parcel Type	45
-------------	----

ucTreeLevel 0

External Data Description

Field	Value
Index ID Position	45
Record Size	12
Key Size	6
Record DBID Size	5
Extraction Size	0
Filter Size	1
Name Size	3
DBID Type	RECORDDBID_2_3
Number Filter Types	1
Filter 0	
Type	FILTERPART_MULTIWORD
Size	1

Navigable Feature By Country Index Key Record (dense)

Field	Description
uiCountryPID	(1 byte) Place Parcel ID offset
uiCountryEID	(2 bytes) Place Entity ID
caNavName	(3 bytes) First three bytes of Navfeature name is in canonical form
uiNavFtrPID	(2 bytes) Navigable Feature Parcel ID offset
uiNavFtrEID	(3 bytes) Navigable Feature Entity ID
ucFilter	(1 bit) Record type:
	0x0 Normal record
	0x1 Multiword record
unused	(7 bits)

4.3.4 Navigable Feature Character Tree Parcels

Navigable feature data alphabetical by base name is used as input to generate Navfeature character tree parcels. The navigable feature character tree only extends to a depth to reach a unique parcel. This optimizes the access to these parcels are used to accomplish next-valid character searches by accessing minimum number of parcels. Each node in the character tree contains a variable number of pointers to later nodes (child nodes), one per valid character at this level. Character tree is unique from other tree implementations because only pointers to nodes that exist are present. This implies that character tree nodes are variable-length. The tree is continued to the point where there is a unique navfeature data parcel. The leaf node then becomes a terminal node that points to the data parcel. The remaining nodes in the tree are classified into either internal or external nodes. A node where all the child nodes are in the same parcel as the parent node is defined as an internal node. On the contrary, if the parent node is in a different parcel than its children then the node is defined as an external node. However, the basic premise in character tree parcelization is that all child nodes of a given parent node lie in one parcel as internal/external/terminal nodes. Unlike other parcels character tree parcels are not fixed size parcels.

The distinction between the three classes of nodes is made by the offset with the character parcel. There are three offsets for a character tree parcel in the character tree parcel header corresponding to the three types of nodes (internal, terminal and external). These offsets indicate the offset of

the first node of each type in the given character tree parcel. The parcel header provides the offset and the parcel size information for each character tree.

The character tree parcels contain two major sets of data:

- Parcel Header
- Node Data (includes internal, terminal and external nodes)

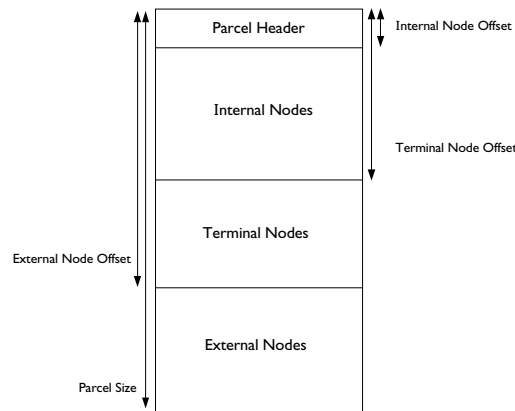


Figure 4-5: Character tree parcel format

Parcel Header

The following structure defines the initial, fixed portion of the character tree parcel.

```
typedef struct CharTreeHdr_t {
    PclHdr_t      pclHdr; /* Common Parcel Header */
    ChTreeOffset_t Offset; /* Offset to first node of each type */
} CharTreeHdr_t;

typedef struct ChTreeOffset_t {
    Ushort_t usIntNode; /* Offset to the first internal node */
    Ushort_t usExtNode; /* Offset to the first external node */
    Ushort_t usTermNode; /* Offset to the first terminal node */
    Ushort_t usPclSize; /* Size of the character tree parcel */
} ChTreeOffset_t;
```

Node Data

The following structure represents a character tree node in PSF.

```
typedef struct PSFCharTreeNode_t {
    Ushort_t usNumChildren; /* Number of child nodes */
    Ulong_t  ulNonMWCount; /* count of non-multiword navfeature
                           records below this node */
    Ulong_t  ulMWCount; /* count of multiword navfeature
                        records below this node */

    ParcelID_t ParcelID;
    char      cNextValidChar; /* Character representing the node */
    Sint_t    (*Filter)(void *, void *); /* Callback function can be used
                                         to process during tree traversals */
    Ushort_t pChildOffset; /* Offsets to each of the child nodes */
} PSFCharTreeNode_t;
```

The same generic node structure is used for all the three types of nodes. Number of child nodes for a terminal node is always zero and hence there are no child offsets for terminal nodes. Furthermore, the Parcel ID for a terminal node is the navfeature data parcel index containing navigable feature base names, matching the string from the root node character until the terminal node character. Internal and external nodes have one or more child nodes and hence corresponding number of child offsets. However, for internal nodes the child offsets are simple offsets within the current character parcel. For external nodes the child offsets are offsets in another character parcel whose parcel index is given by the parcelID field of the node.

Note: The generic structure of the node makes the parcelID field redundant for internal nodes.

4.4 Place Data (Administrative Areas, Zones)

The Place record stores information about a named place. Named places can be categorized as administrative areas or zones. Administrative areas are hierarchical governmental entities such as Country, State, and City in the United States or Country, Bundesland, Kreis, Gemeinde, and Settlement in Germany. Zones are named places that fall outside of the governmental hierarchy such as townships or neighborhoods. The Place record contains a unique identifier for the place, the name of the place, the names of all the parent administrative levels of the place, and the minimum bounding rectangle for the entities within a place.

Zones may be combined with related admin areas of the same name to produce a single Place ID, referenced by all entities within these areas. However, the relationship between every entity and the zone or admin area is retained by using a place type relationship for each entity.

Places are treated as separate entities in the PSF rather than being treated as a segment attribute because a segment can be associated with a variable number of places. Also, the Place entity allows POIs and navigable features to be directly associated with a place rather than indirectly associated through a segment. This aids the ability to query POIs and navigable features by place.

4.4.1 Place Data Parcel

The Place Data Parcel stores information about a named place and its alternate names (e.g., abbreviations). Named places can be categorized as administrative areas or zones.

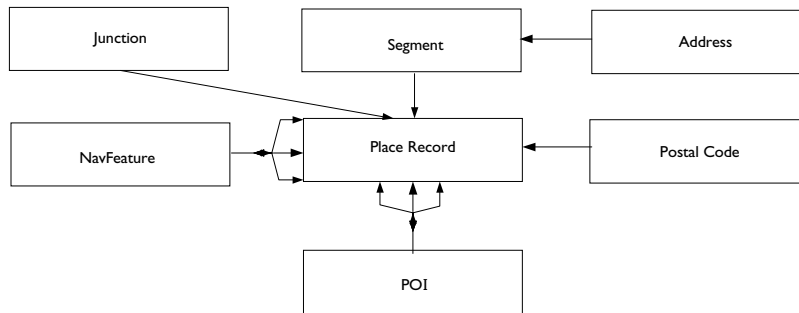
4.4.1.1 Technical Description

The Place Data Parcel stores information about a named place and its alternate names. Named places can be categorized as administrative areas or zones.

The Place Data Parcel contains a unique identifier for the place, the name of the place, the unique identifiers of all the parent administrative levels of the place, and the minimum bounding rectangle for the place.

A Bypass level is not needed for precise address resolution; e.g., County level within United States. Bypass levels vary between countries, and there may be any number of them. The Country and lowest level places are Non-Bypass.

4.4.1.2 Data Reference



4.4.1.3 Data Structures

The Place Data Parcel structure includes:

- Place Parent Entry Structure for Type 1 (bypassed parcel) and Type 2 (non-bypassed parcel)
- Parcel Header
- Per-parcel Data Arrays
- Place ID Index Table
- Full Format Place Record
- Synonym/Exonym Place Record
- multiword Indexed Place Record

Place Parent Entry Structure for Type_1 and Type_2

```

typedef struct
{
    Ushort_t    usPclIdx;          /* Index to parcel table and */
                                   /* 1st bit is set to 0 */
    Ushort_t    usEntityId;        /* Place entity ID */
    Ushort_t    usParent;          /* Offset to parent place entry */
    Uchar_t     ucReserve[2];
}
PlaceEntryType_t;

typedef struct
{
    Ushort_t    usPclIdx;          /* Index to parcel table and */
                                   /* 1st bit is set to 1 */
    Ushort_t    usEntityId;        /* Place entity ID */
    Ushort_t    usParent;          /* Offset to parent place entry */
    Ushort_t    usChildPclIdx;    /* index to Pcl ID of first
                                   ** subordinate child at first
                                   ** subordinate level with Bypass
                                   ** Flag off */
    Ushort_t    usChildEntityId;  /* Entity ID for first
                                   ** subordinate child */
    Ushort_t    usLastChildPclIdx; /* index to Pcl ID of last
                                   ** subordinate child at first
                                   ** subordinate level with Bypass
                                   ** Flag off */
    Ushort_t    usLastChildEntityId; /* Entity ID for last
                                   ** subordinate child */
    Uchar_t     ucReserve[2];
}

```

```

PlaceEntryType2_t;

typedef struct
{
    Ushort_t    usStartEntityId;    /* Place starting entity ID      */
    Ushort_t    usEndEntityId;      /* Place ending entity ID        */
    Ushort_t    usEntityLevel;      /* Level to which entities ** belong */
    Uchar_t     ucReserve[2];
}
PlaceEntityRange_t;

```

Parcel Header

This structure defines the initial, fixed portion of the parcel.

```

typedef struct {
    XrfPclHdr_t xrfPclHdr;          /* Common Parcel Header */
    BlkDesc3_t  placeDesc;          /* Place record descriptor */
    BlkDesc4_t  placeIndexDesc;     /* PlaceID Idx descriptor */
    BlkDesc3_t  parentPlaceDesc;    /* Parent Place descriptor*/
    BlkDesc4_t  languageDesc;       /* Language Code Descriptor
                                     ** (points to array of one-
                                     ** byte indices into global
                                     ** data */
    BlkDesc4_t  soundexDesc;         /* Unused */
    BlkDesc4_t  entityRangeDesc;    /* Entity Range Descriptor */
}
PlaceParcelHeader_t;

```

Per-parcel Data Arrays

```

ParcelID_t aPclId[]; /* Parcel ID Array */

Uchar_tucaLangCode[]; /* Each entry is an index to a
                       ** language code
                       ** used in this
                       ** parcel */

Uchar_tparentPlace[]; /* This is actually an array of
                       ** mixed PlaceEntry_t and
                       ** PlaceEntryType2_t structures.
                       ** They can be mixed because they
                       ** are all accessed via offset */

PlaceEntityRange_t entityRange; /* Range of Place entities
                                ** in this parcel */

```

Place ID Index Table

This structure is an array of record offsets with each entry containing a multiword indicator and language information.

```

typedef Uchar_tPlaceIdIndex_t[4];
/* Record Offset: 16 bits.
** Full/Abbr Flag: 1 bit. If 0, this
** record is in full format; if 1, this
** record is in synonym/exonym format.
** multiword Index Flag: 1 bit. If 1 this
** is a multiword index record. If 0 not.
** Unused: 6 bits.
** Language Bitmask index: 8 bits. */
PlaceIdIndex_tPlaceIdIndex[];

```

Full Format Place Record

This record is a variable-length character array.

Field	Description
Govt Code	(4 bytes) Country Government Code; unique for each place within a level.

The following four fields (Next Bypass, Bypass Flag, Parent Level, Level) occupy one byte.

Next Bypass	(1 bit) Bypass flag for the next lower administrative level.
Bypass Flag	(1 bit) Bypass flag; meaning this level is not needed to create address uniqueness for a country. For the US, Bypass level is county.

0x0 Needed in addressing

0x1 Not needed in addressing

Parent Level	(3 bits) Administrative level for the upper administrative area.
--------------	--

Level	(3 bits) Indicates the Administrative level of this place.
-------	--

0x0 invalid

0x1 highest level

:

0x7 lowest level

Place Type	(1 byte) Indicates Administrative area or Postal zone.
------------	--

0x1 Administrative area

0x2 Postal zone

The following three fields (Inc-Descend, Unused, Long Haul Attr) occupy one byte.

Inc-Descend	(1 bit) If 1, during a municipal level search for entities in this place, also search for entities within this place's child places, to get a complete result set.
-------------	--

Unused	(1 bit)
--------	---------

Lng Haul Attr	(2 bits) 0x00 Not Long Haul
---------------	-----------------------------

0x01 Partial Long Haul

0x10 Long Haul

Lang Index	(4 bits) An index to this parcel's Language Code Array where each entry is itself a one-byte index to global language data.
------------	---

Min Longitude	(4 bytes) Longitude for lower left corner of minimum rectangle including all segments, and the nodes and shape points, in this administrative area.
---------------	---

Min Latitude	(4 bytes) Latitude for lower left corner of minimum rectangle including all segments, and the nodes and shape points, in this administrative area.
--------------	--

Max Longitude	(4 bytes) Longitude for upper right corner of minimum rectangle including all segments, and the nodes and shape points, in this administrative area.
---------------	--

Max Latitude (4 bytes)	Latitude for upper right corner of minimum rectangle including all segments, and the nodes and shape points, in this administrative area.
------------------------	---

The following three fields (Base/Syn Flg, Lowest Level, Syn/Ex Cnt) occupy one byte.

Base/Syn Flg	(1 bit) If set to 0, this represents a base name. If set to 1, this is not a base name; it is a synonym or exonym of a place base name. A full format Syn/Exon record is used when the information here can only be found by a secondary parcel read. In this case, the Syn/Exonym Array contains a single entry and represents the base name DBID.
Lowest Level	(1 bit) If set, this represents a place at the lowest administrative level of its country.
Syn/Ex Cnt	(6 bits) Number of entries in the Syn/Ex Array (Synonym/Exonym Array).
Name Length	(1 byte) Length of the Place Name field.
Place Name	(Name Length) Length: the number of bytes specified in the Name Length field.
Parent Place	(2 bytes) An offset to the parent Place Entry for this place; present if Level is greater than 1.

The following two fields are present (in any level above the lowest level) *either* if Bypass Flag is *not* set to 1 *or* if Next Bypass is set to 1.

First Child (3-5 bytes) The first two bytes are the Entity ID of this Place's first subordinate Place record (in the level immediately below this record's level); value 0xFFFF is a dummy value indicating there are no subordinate Places.

The next byte is the beginning of a Type 1 Variable Length Unsigned Value representing that Place record's Parcel ID; an index to this parcel's Parcel ID table.

Last Child (3-5 bytes) The first two bytes are the Entity ID of this Place's last subordinate Place record (in the level immediately below this record's level); value 0xFFFF is a dummy value indicating there are no subordinate Places.

The next byte is the beginning of a Type 1 Variable Length Unsigned Value representing that Place record's Parcel ID; an index to this parcel's Parcel ID table.

When the First/Last Child fields are not present, the child Place records must be obtained via the Parent Place record. The first non-bypassed parent record found has a Place Entry Type2-t structure. The usChildPclIdx and usChildEntityID fields of this structure identify the place to begin scanning Place records for children, however they do not all appear consecutively.

The following two fields are present (in any level above the lowest level) if Bypass Flag is not set to 1 and Next Bypass is set to 1.

First NB Child (3-5 bytes) The first two bytes are the Entity ID of the first subordinate (to this Place) Place record of the next succeeding level for which the Bypass Flag is *not* set; value 0xFFFF is a dummy value indicating there are no subordinate Places.

The next byte is the beginning of a Type 1 Variable Length Unsigned Value representing that Place record's Parcel ID; an index to this parcel's Parcel ID table. Example: If Bypass Flag is not set for level 2, is set for level 3, and is not set for level 4; this entry is present in the level 2 Place record, and points to the first level 4 Place record subordinate to it.

Last NB Child	(3-5 bytes) The first two bytes are the Entity ID of the last subordinate (to this Place) Place record of the next level for which the Bypass Flag is <i>not</i> set; value 0xFFFF is a dummy value indicating there are no subordinate Places. The next byte is the beginning of a Type 1 Variable Length Unsigned Value representing that Place record's Parcel ID; an index to this parcel's Parcel ID table. Example: If Bypass Flag is not set for level 2, is set for level 3, and is not set for level 4, this entry is present in the level 2 Place record and points to the last level 4 Place record subordinate to it.
Syn/Ex Array	(variable) Length: variable; the number of entries is given by Syn/Ex Cnt. If the Base/Syn Flag is set to 0, each entry refers to the DBID of a Place, a synonym or exonym of this base name record. Each entry is made of (1) a Type 1 Variable Length Unsigned Value representing an index into the Parcel ID table, and (2) a two-byte Place Entity ID. If the Base/Syn Flag is set to 1, this record is a synonym/exonym, even though it is in full format, and the single entry in this array refers to the base name Place record DBID.
The remaining fields of the Place record are present only at Place level 1 (country). Four fields (Display Unit, House Num, Driving Side, Num Levels) occupy one byte.	
Display Unit	(1 bit) Indicates units of measure format for a country. 0x0 Metric 0x1 English (Imperial)
House Num	(1 bit) Indicates whether the house numbers are typically represented before or after the street name. 0x0 Before the street name 0x1 After the street name
Driving Side	(1 bit) Indicates the legal driving side of the road for a country. 0x0 Driving side is left 0x1 Driving side is right
Num Levels	(3 bits) The number of Administrative level.
Unused	(2 bits)
Currency Prec	(1 byte) Number of decimal places in which currency is expressed.
Currency Type	(1 byte) Unused
Bypass Map	(1 byte) Starting at the lowest bit, Num Levels bits are used. Each bit indicates: 0x0 This level is bypass 0x1 This level is not bypass
Phone Code	(1 byte) Telephone country code.
Level Array	(10 bytes per level) Length: 10 bytes multiplied by the value in the Num Levels field; each entry is the generic name for a Place level (e.g., "Country", "Province", etc.).

4.4.1.4 Synonyms and Exonyms

Each Place has a basename (a designated “default” name) in a default language. It may have multiple synonyms in its default language, and at most one exonym in a different language. Place name abbreviations are treated as synonyms.

SDAL database cross-references from and to placeIDs, use basename placeIDs only. Synonyms and exonyms for Places are represented only in the Place data parcels (and their B-Tree). The PSF cross-references from POI to Place in POI records, segment to Place in maneuver parcels, and Navigable Feature to Place in the place-ordered navfeature records, include entries for place basename records only.

4.4.1.5 Abbreviated Synonym/Exonym Place Record

This record is a variable-length character array.

Field	Description
-------	-------------

The following two fields (PclID Flag, Base EntID) occupy two bytes:

PclID Flag	(1 bit) If set to 1, the Base PclID is present in this record, and contains the Parcel ID of the Place record for the basename for this synonym/exonym.
Base EntID	(15 bits) Entity ID of the Place record for the full format place record corresponding to this synonym/exonym. If PclID Flag is 1, this is the entity ID of the base name; If it is zero, it may refer to the basename or to another synonym/exonym record in the current parcel.
Name Length	(1 byte) Length of the Place Name field.
Place Name	(Name Length bytes) Length: the number of bytes specified in the Name Length field.

The following field is present only if the PclID Flag is set to 1.

Base PclID	(1-2 bytes) Refers to the Parcel ID of the Place record for the basename for this synonym/exonym.
------------	---

4.4.1.6 Multiwords

Multiword records are created for each significant word contained within a place name (both basename or synonym/exonym) e.g., ‘States’ within ‘United States’.

4.4.1.7 Multiword Index Place Record

This record is a variable-length character array.

Field	Description
-------	-------------

The following two fields (PclID Flag, Ref EntID) occupy two bytes:

PclID Flag	(1 bit) If set to 1, the Ref PclID is present in this record, and identifies the Parcel ID of the Place record for which this record was created.
Ref EntID	(15 bits) Entity ID of the Place record for the place record (full format or synonym/exonym) for which this record was created.

The following field is present only if the PclID Flag is set to 1.

Ref PclID	(2 bytes) Identifies the Parcel ID of the Place record referenced by Ref EntID, as follows: The zero-based index of the Parcel containing the record, where the parcel with index zero is the first Place B-tree index parcel.
-----------	--

Word Start	(1 byte) Zero-based index of the first character of the word (in the Name field of the Place record referenced by this record's Ref EntID field) for which this record was created.
Name Length	(1 byte) Length of the following Place Name field.
Place Name	(Name Length bytes) Holds the full place name containing this multi-word.

4.4.2 Place Data Index Files

Place Data Index Files contain the indexes needed to quickly search for place names.

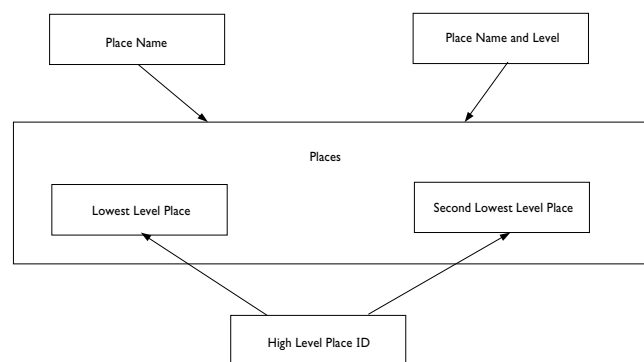
4.4.2.1 Technical Description

Place Data Index Files are B-tree parcels of minimum height containing the key values and file off-sets of the Place parcels for those key values.

Places are assigned a level of 1 to 7 inclusive. In the following, level M (Municipal Level) contains all the places in the lowest populated admin level (e.g. 4 in the United States and 5 in Germany) plus all places in the levels above this, up to (but not including) the lowest Bypassed level. In the United States, level M contains only the places at City level (level 4). In Germany, level M contains all of the places in the Gemeindes and Settlements (levels 4 and 5).

- **Place ID Ordered Place Data Index** constructed of: Place ID Ordered Place Data Index Parcel (sparse); the index to the Place Data Parcels.
- **Place Level n plus Place Name Index** $\{0 < n < 8 \mid n = M\}$ constructed of:
 - (1) Index on Place Level n plus Place Name Parcel (sparse); the index to Index on Place Level n plus Place Name Parcels (dense);
 - (2) Index on Place Level n plus Place Name Parcel (dense); the index to Place Data Parcels.
- **Place-High-Place Index** constructed of:
 - (1) Place-High-Place Index Parcel (sparse); the index to Place-High-Place Index Parcels (dense);
 - (2) Place-High-Place Index Parcel (dense); the index to Place Data Parcel.

4.4.2.2 Referenced By



A Place name can locate a place record.

A Place name and level number can locate a place record with that name at that level.

A Place ID for a high level place (above the lowest two levels) can reference the range of lowest, and second lowest level places below that place.

4.4.2.3 Index Structures

Place Data Index Parcel Header

Each B-tree index parcel has a header of the following structure:

```
typedef struct
{
    PclHdr_t    pclHdr;
    Ushort_t    usNumKeys; /* number of keys in this parcel */
    Ushort_t    usKeySize; /* key size in bytes */
    Ushort_t    usChildParcelPosition;
                    /*
                     ** position of first child of this
                     ** parcel where tree root is
                     ** position 0
                     */
    Uchar_t     ucTreeLevel;
                    /*
                     ** parcels that index records are
                     ** at level 1; parcels that index
                     ** level n index parcels are at
                     ** level n+1
                     */
    Uchar_t     ucReserved[1];
                    /* maintain 4-byte alignment */
}
BtreeIdxHdr_t;
```

Place ID Ordered Place Data Index

Parcel ID Ordered Place Data Index constructed of Place ID Ordered Place Data Index Parcel (sparse); the index to the Place Data Parcels.

Place ID Ordered Place Data Index (sparse)

```
Parcel Type      251
ucTreeLevel      1
```

External Data Description

Field	Value
Index ID Position	11
Record Size	14
Key Size	14
Record DBID Size	0
Extraction Size	0
Filter Size	0
Name Size	10
DBID Type	RECORDDBID_0_0
Number Filter Types	0

Parcel ID Ordered Place Data Index Key Record (dense)

Field	Description
Place Parcel ID	(4 bytes) Place Parcel ID
Place Name	(10 bytes) Place Name

Place Level n Plus Place Name Index

Place Level n plus Place Name Index $\{0 < n < 8 \mid n = M\}$ constructed of:

1. Index on Place Level n plus Place Name Parcel (sparse); the index to Index on Place Level n plus Place Name Parcels (dense);
2. Index on Place Level n plus Place Name Parcel (dense); the index to Place Data Parcels.

Place Level n Plus Place Data Index (sparse)

Parcel types:

LEVEL1_PLACE_ALPHA_IDX_PARCEL	238
LEVEL2_PLACE_ALPHA_IDX_PARCEL	237
LEVEL3_PLACE_ALPHA_IDX_PARCEL	236
LEVEL4_PLACE_ALPHA_IDX_PARCEL	235
LEVEL5_PLACE_ALPHA_IDX_PARCEL	234
LEVEL6_PLACE_ALPHA_IDX_PARCEL	233
LEVEL7_PLACE_ALPHA_IDX_PARCEL	232
LEVELM_PLACE_ALPHA_IDX_PARCEL	230

ucKeysize 6
ucTreeLevel 1

Place Level n Plus Place Data Index Key Record (sparse)

Field	Description
Place Name	(6 bytes) Place Name

Place Level n Plus Place Data Index (dense)

Parcel type

LEVEL1_PLACE_ALPHA_PARCEL	24
LEVEL2_PLACE_ALPHA_PARCEL	25
LEVEL3_PLACE_ALPHA_PARCEL	26
LEVEL4_PLACE_ALPHA_PARCEL	27
LEVEL5_PLACE_ALPHA_PARCEL	28
LEVEL6_PLACE_ALPHA_PARCEL	29
LEVEL7_PLACE_ALPHA_PARCEL	30
LEVELM_PLACE_ALPHA_PARCEL	33

ucTreeLevel 0

External Data Description

Field	Value
Index ID Position	24, 25, 26 ,27, 28, 29, 30, 33
Record Size	12
Key Size	6
Record DBID Size	4
Extraction Size	0
Filter Size	2
Name Size	6
DBID Type	RECORDDBID_2_2
Number Filter Types	2
Filter 0 Type Size	FILTERPART_LANGUAGE 8
Filter 1 Type Size	FILTERPART_MULTIWORD 1

Place Level n Plus Place Data Index key record (dense)

Field	Description
Place Name	(6 bytes) Place Name
Place Parcel ID	(2 bytes) Index in the Place Parcel ID array.
Place Entity ID	(2 bytes) Place Entity ID
Lang Mask Index	(1 Byte) Index to global data language bitmask.
Filters	(1 byte) The following values are defined: 0x0 Normal record. 0x1 Multiword record.

Place-High-Place Index

Place-High-Place Index constructed of:

1. Place-High-Place Index Parcel (sparse); the index to Place-High-Place Index Parcels (dense);
2. Place-High-Place Index Parcel (dense); the index to Place Data Parcel.

Place-High-Place Index Parcel (sparse)

Place-High-Place Index (sparse) Parcel has Place Index Header and the number of usNumKeys field Place-High-Place Index Parcel key records. The following fields in this header have these values.

```
Parcel Type=      228
ucKeysize  =      4
ucTreeLevel=     1
```

Place-High-Place Index Parcel key record (sparse)

Field	Description
ParPlaceParcelID	(2 bytes) An index of Parent Place Parcel ID
ParPlaceEntID	(2 bytes) Parent Place Parcel Entity ID

Place-High-Place Index Parcels (dense)

Place-High-Place Index (sparse) Parcel has Place Index Header and the number of usNumKeys field Place-High-Place Index Parcel key records. The fields in this header have these values.

```
Parcel Type=      37
ucTreeLevel=     0
```

External Data Description

Field	Value
Index ID Position	35
Record Size	20
Key Size	4
Record DBID Size	0
Extraction Size	16
Filter Size	0
Name Size	0
DBID Type	RECORDDBID_0_0
Number Filter Types	0

Place-High-Place Index Parcels key record (dense)

Field	Description
ParPlaceParcelID	(2 bytes) An index for Parent Place Parcel ID
ParPlaceEntID	(2 bytes) Parent Place Parcel Entity ID
LowLvlPlace_First PrlID	(2 bytes) An index for First children Parcel ID of Lowest Level Place
LowLvlPlace_FirstEntID	(2 bytes) First children Entity ID of Lowest Level
LowLvlPlace_last PrlID	(2 bytes) An index for last children Parcel ID of Lowest Level Place
LowLvlPlace_lastEntID	(2 bytes) Last children Entity ID of Lowest Level
2LowLvlPlace_First PrlID	(2 bytes) An index for First children Parcel ID of second Lowest Level Place
2LowLvlPlace_FirstEntID	(2 bytes) First children Entity ID of second Lowest Level
2LowLvlPlace_last PrlID	(2 bytes) An index for last children Parcel ID of second Lowest Level Place
2LowLvlPlace_lastEntID	(2 bytes) Last children Entity ID of second Lowest Level

4.4.2.4 Data Compilation Guidelines

Place records are ordered alphabetically within children of a common non-bypassed parent as follows:

```

country1
country2
...
countryn
country1 state1
country1 state2
...
country1 staten
country2 state1
...
country2 staten
...
countryn staten
country1 state1 county1
country1 state1 county2
...
country1 state1 city1 (county level is bypassed)
country1 state1 city2
...

```

Place records are parcelized into fixed size parcels; a parcel may contain records of different levels.

There are LEVEL_n_PLACE_ALPHA_PARCEL and LEVEL_n_PLACE_ALPHA_IDX_PARCEL structures for each administrative level *n* (in the database region), up to a maximum of *n*==7, as well as the combined munipale level *M*. There is one root per level, and they are stored in the region global data index parcelID table.

4.5 Postal Code Data

The postal code entity stores information about a postal code. Postal code data includes:

- Unique identifier for the postal code
- Place ID for the country associated with the postal code
- Minimum bounding rectangle for the postal code.

4.5.0.1 Technical Description

Postal code data parcels are non-spatial parcels, organized in postal code sequence, and in postal code ID sequence within postal code.

Postal code data within a postal code parcel is indexed by:

Postal Code If all postal codes in the parcel start with the same initial characters, those characters are stored in the Postal Code Common Prefix field up to a specified maximum length. The length of this string is stored in the Postal Code Common Prefix Length field. The Postal Code Index Table is organized sequentially by the remaining characters of the postal code after the common prefix. E.g., the common prefix is 600; 60010 is between the entries 09 and 30.

Postal Code Entity ID The Entity ID part of the postal code ID accesses the record via the Postal Code Block Offset Array.

Each index used to access the postal code data is organized as a B-tree of minimal height. The index page size is limited so any postal code record can be found with a minimal number of reads.

4.5.1 Postal Code Data Parcels

4.5.1.1 Data Reference

Figure 4-6 illustrates the data contained in the postal data parcel.

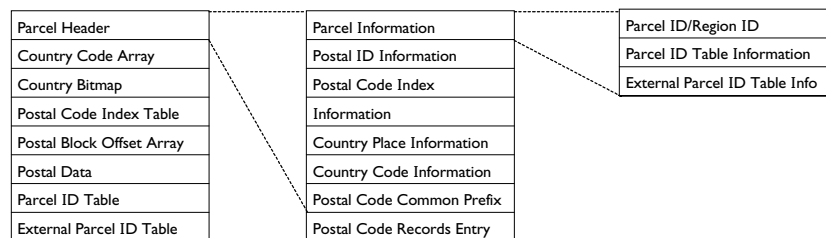


Figure 4-6: Postal Data Parcel Contents

Parcel Header

The following structure defines the initial, fixed portion of the postal data parcel.

```

/*****
**Postal Parcel Header
*****/
/
typedef struct
{
    XrfPclHdr_t    xrfPclHdr;                /* Common Parcel Header */

    BlkDesc_t      postalBlkDesc;            /* Postal Code Block Descriptor */
    BlkDesc4_t     codeIndexDesc;           /* Postal Code Index Descriptor */
    BlkDesc4_t     cntryArrayDesc;          /* Array of Country Place IDs */
    BlkDesc4_t     cntryBitmapDesc;         /* Country code bitmap */

```

```

Xchar_t    aPostalCodePrefix[5];    /* Postal Code Common Prefix */
Uchar_t    ucPostalCodePrefixLength; /* Postal Code Common Prefix
                                     Length */
Uchar_t    ucPostalCodeRecsPerEnt;  /* Postal Code Records per Entry
                                     */
}
PostalCodeHeader_t;

```

Per-Parcel Data

The locations and sizes of the following data are contained in the parcel header:

```

Uchar_tucaCntryCodeArray[ ]; /* Each entry in this array is four bytes
                               in size:
                               ** A 2 byte index into this parcel's Parcel ID table,
                               ** referencing a place parcel ID, and a 2 byte
                               ** Place entity ID. This is the Place DBID for a
                               ** country. */
Uchar_t ucaCntryBitmap[ ] /* Each entry is a 0-based index into
                           ucaCntryCodeArray.
                           ** Each entry's length in bits is given by the value in
                           ** the parcel descriptor entry with index equal to
                           ** pdPostal_CntryIdxBitLen. */

```

Postal Code Index Table

This table is an array of two-character prefixes:

```

typedef struct {
    Xchar_t prefix[2];    /** Postal Code Prefix (two character) */
} PostalPrefixTable_t;
PostalPrefixTable_t aPostalPrefixTable[ ];

```

Postal Code Block Offset Array

Postal Code records are accessed via the Postal Code Block Offset Array. Each entry contains a block offset that points to the beginning of a block of $2^K + 1$ Length fields followed by 2^K Postal Code records. The entity identifier is equal to the index of the record (within the parcel). The length fields are all Type 1 Variable Length Unsigned Values. The first length field is the offset from the beginning of the block to the first Postal Code record in the block. The following 2^K length fields are the lengths of each of the 2^K compressed Postal Code records.

```

typedef struct {
    Ushort_t usOffset;    /** Block Table Offset */
} PostalBlkTable_t;

```

Postal Record

The Postal Record is a variable-length character array.

Field	Description
Min Longitude	(4 bytes) Longitude (in NTUs) of the lower left corner of the minimum bounding rectangle of the postal code area.
Min Latitude	(4 bytes) Latitude (in NTUs) of the lower left corner of the minimum bounding rectangle of the postal code area.
Max Longitude	(4 bytes) Longitude (in NTUs) of the upper right corner of the minimum bounding rectangle of the postal code area.
Max Latitude	(4 bytes) Latitude (in NTUs) of the upper right corner of the minimum bounding rectangle of the postal code area.

The following two fields occupy one byte:

Lng Haul Attr (2 bits) Flags indicating the Long Haul status of the postal code area

<u>Value</u>	<u>Description</u>
--------------	--------------------

00	Not Long Haul
----	---------------

01	Partial Long Haul
----	-------------------

10	Long Haul.
----	------------

Postal Length (6 bits) The length of the Postal Code field.

Postal Code The postal code in compressed form.

4.5.2 Postal Code Data Index Files

The Postal Code index file is a B-trees of minimum height containing key values and file offsets of Postal Code parcels.

4.5.2.1 Technical Description

Postal code record parcels contain records ordered alphabetically by postal code string. The key consists of the null-terminated postal code string. The ucParcelType value for the index parcels is 250. This index facilitates search for Postal parcel IDs by postal code string.

4.5.2.2 Postal Code Index Parcel Header

```
typedef struct
{
    PclHdr_t   pclHdr;
    Ushort_t   usNumKeys;      /* number of keys in this parcel */
    Ushort_t   usKeySize;      /* key size in bytes */
    Ushort_t   usChildParcelPosition;
                                /*
                                ** position of first child of this
                                ** parcel where tree root is
                                ** position 0
                                */
    Uchar_t     ucTreeLevel;
                                /*
                                ** parcels that index records are
                                ** at level 1; parcels that index
                                ** level n index parcels are at
                                ** level n+1
                                */
    Uchar_t     ucReserved[1]; /* maintain 4-byte alignment */
}
BtreeIdxHdr_t;
```

4.5.2.3 Postal Code Data Index

The Postal Code Data Index fields in this header have the following values:

Parcel Type= 250
ucTreeLevel= 1

External Data Description

Field	Value
Index ID Position	13
Record Size	11

Field	Value
Key Size	11
Record DBID Size	0
Extraction Size	0
Filter Size	0
Name Size	11
DBID Type	RECORDDBID_0_0
Number Filter Types	0

4.6 Recorded Voice Explication Data Files

SDAL recorded explication voice audio data is included in the SDAL PSF CD-ROM. This data is recorded, using both male and female voices, in several languages: Dutch, French, English (U.K. and U.S.), German, Japanese, Italian, and Spanish.

A separate audio file is stored on the SDAL PSF CD-ROM for each language-voice pair. Each file contains all ADPCM encoded phrases for the pair, phrase count within that file, and a table of offsets for the phrases within that file.

4.6.0.1 Technical Description

For each language/voice pair a separate file is stored on the CD-ROM (e.g., "GERF.SDL" for German/female, "GERM.SDL" for German/male). These files store all ADPCM encoded phrases for the pair together with a small table of offsets for the phrases within that file. The 16 voice files are not database-specific. They consist of a male and female voice file for each of the supported languages.

File Description	File Name
Dutch, male	DUTM.SDL
Dutch, female	DUTF.SDL
English, male	ENGM.SDL
English, female	ENGFSDL
French, male	FREM.SDL
French, female	FREF.SDL
German, male	GERM.SDL
German, female	GERF.SDL
Italian, male	ITAM.SDL
Italian, female	ITAF.SDL
Japanese, male	JPNM.SDL
Japanese, female	JPNF.SDL
Spanish, male	SPAM.SDL
Spanish, female	SPAF.SDL
UK English, male	UKEM.SDL
UK English, female	UKEF.SDL

Offset Table

The offset table uses the following sequence:

- 4-byte place holder, filled at runtime with regionID or file descriptor.
- 4-byte N phrase count

- $N+1$ 4-byte offsets (byte offsets of phrases within the file), where N = number of phrases in the file. The size of the phrase K is computed by subtracting the offset to phrase K from the offset to phrase $K+1$. The last byte offset holds the offset to the end of last phrase, i.e., end of file.

Phrase Data

The phrase data follows the offset table. The data is in the form of IMA ADPCM mono 4 bits/sample 11.025Khz.

For each phrase, the data consists of a sequence of blocks. Each block is 256 bytes, except the last which may be a multiple of 4 bytes less than 256 bytes. Each block uses the following sequence:

- 2-byte initial sample value for block (big-endian)
- 1-byte initial IMA ADPCM step index (0-88)
- 1 byte with value 0 (reserved)
- Sample IMA ADPCM data (up to 252 bytes = 504 samples)

In general, the IMA format does not specify the size of the block, implying that the whole audio clip should be one ADPCM block. However, all practical applications of ADPCM data do break the audio clip into varying-size blocks.

The current format follows the Microsoft variation of the IMA ADPCM format. This variation establishes 256-byte blocks described above. The Microsoft header is not included with the phrase since the data it holds (phrase size, mono or stereo recording, sampling speed and sample size) is either constant for all phrases or is stored in a more space efficient format within the Offset Table in the file.

Data Compilation Guidelines

The `ucVoiceDataIndexOffset`, `usVoiceDataParcelCount`, and `usVoiceDataMaxIndex` fields in the Global Media Header should be set to zero.

Glossary

Aggregation	The combining of contiguous regular segments with sufficiently similar characteristics into a single aggregated segment
Arm-to-Arm Path	Within a supernode, the internal path between 2 segments (arms) connected to the supernode
Basename	For Places or POIs the designated primary name; For NavFeature names, the part of the name excluding street type and affix.
Database Identifier (DBID)	A unique identifier (within an entity type) for a database entity
Dense Index	The dense index re-orders records of a particular type to facilitate the searches of the data by some alternate key
Detailed City Inclusion	The inclusion and attribute level of segments (and their associated data) for urban or built up areas
Di-tile	A <i>di-tile</i> of dimensions $2^I \times 2^J$ is defined as a rectangle between latitudes $M \times 2^I$ NavTech Units and $(M+1) \times 2^I$ NavTech Units, and between longitudes $N \times 2^J$ NavTech Units and $(N+1) \times 2^J$ NavTech Units (a NavTech Unit is $1/100,000$ of a degree, M and N are integers, and I and J are positive integers).
Entity Identifier	A 24-bit identifier used to address a specific record of a given type within a specified parcel.
Exception Segment	A segment spanning multiple parcels. The parcel containing the segment's right node has the segment identified as an exception segment
Exonym	Alternate name for a Place or POI in a language other than that of the Place or POI base-name.
Explication Data	Words and Phrases used in the construction of driving directions
Ghost Segment	A segment passing through a parcel with neither node in that parcel
Icon	A graphical bit representation of highway shields or POI facility types
In-Process Data	Segments and their associated data which contain complete geometry but incomplete attributes, marked by an 'In Process' flag in the segment
Inter-Town Inclusion	The inclusion and attribute level of segments (and their associated data) in non-urban areas
IVC	In Vehicle Code
Layer	The filtering of route segments and their associated data based on rank to obtain multiple representations, at different levels of detail, of a geographic area
Long Haul Data	The highest ranking segments and data associated with these segments comprising the national level arterial road network (e.g., interstates). Long haul data can extend past the detailed geographic area contained in an SDAL database
Minimum Enclosing Di-tile	The minimum bounding rectangle for an area, with di-tile dimensions

NavTech Unit (NTU)	1/100,000 of a degree.
Node Cost	The relative cost (in time) for a vehicle to traverse a node
Normalized Attribute Array	An array of commonly occurring attribute combinations associated with a routing segment
Parcel	This is the SDAL physical record that contains all the map data in a rectangle. Included in a parcel are not only the records representing the map data within the rectangle, but also parcel overhead. The optimum parcel size is the preferred size of this physical record.
Parcel Identifier	A 32 bit identifier used to address a parcel within the SDAL Map file
Peano Key	A spatial key obtained by interleaving the bits of the latitude and longitude of an entity
Rank	The rating of a segment based on such characteristics as speed limit, road type (e.g., interstate, U.S. highway), traffic volume and access limitations. Rank 0 represents the lowest rank (local streets), while rank 4 represents the highest level (national arterial road network)
Rectangle	This is the set of all map data whose latitudes lie within some interval of NavTech Units, and whose longitudes lie within some other interval of NavTech units.
Sparse Index	The set of key records used to facilitate identifying the parcel (or range of parcels) containing a record with a given key
Spot	A relative position along a segment. Values are zero to 255.
Strand	A string of one or more physically connected segments with the same Navigable Feature ID and direction-on-sign. Strands are connected linearly, so that, for example, if a "Y" shaped intersection existed in which all three branches had the same Navigable Feature ID and direction-on-sign, the strands would terminate at that intersection. Whenever a segment has N different Navigable Feature ID and direction-on-sign combinations, it will be part of N strands.
Subrectangle	When a rectangle is divided by either a north-south line or an east-west line, the two rectangles created by this division line are referred to as subrectangles.
Supernode	The collapse of a complex intersection, consisting of many segments and nodes, into a single node. Examples are doubly digitized road intersections and roundabouts
Synonym	Alternate name for a Place or POI in the language of the Place or POI basename.
Tile	This is one of the set of fixed-size tiles that initially covers the region. Minimum Enclosing Di-tile: This is the smallest single di-tile that encloses a particular rectangle. It can be equal to the rectangle, or it can be larger than the rectangle.
Z-Level	An indication of an entity's altitude relative to other intersecting entities. Ranges in value from -5 (lowest altitude) to 4 (highest altitude)