OGR Driver Implementation Tutorial

Overall Approach

In general new formats are added to OGR by implementing format specific drivers with intanciating a **GDALDriver** and subclasses of GDALDatasetand **OGRLayer**. The **GDALDriver** instance is registered with the **GDALDriverManager** at runtime.

Before following this tutorial to implement an OGR driver, please review the OGR Architecture document carefully.

The tutorial will be based on implementing a simple ascii point format.

Contents

Implementing GDALDriver

The format specific driver class is implemented as a instance of **GDALDriver**. One instance of the driver will normally be created, and registered with the **GDALDriverManager**. The instantiation of the driver is normally handled by a global C callable registration function, similar to the following placed in the same file as the driver class.

```
void RegisterOGRSPF()

{
    GDALDriver  *poDriver;

    if( GDALGetDriverByName( "SPF" ) == NULL )
    {
        poDriver = new GDALDriver();

        poDriver->SetDescription( "SPF" );
        poDriver->SetMetadataItem( GDAL_DCAP_VECTOR, "YES" );
        poDriver->SetMetadataItem( GDAL_DMD_LONGNAME,
                                   "Long name for SPF driver" );
        poDriver->SetMetadataItem( GDAL_DMD_EXTENSION, "spf" );
        poDriver->SetMetadataItem( GDAL_DMD_HELPTOPIC,
                                   "drv_spf.html" );

        poDriver->pfnOpen = OGRSPFDriverOpen;
        poDriver->pfnIdentify = OGRSPFDriverIdentify;
        poDriver->pfnCreate = OGRSPFDriverCreate;

        poDriver->SetMetadataItem( GDAL_DCAP_VIRTUALIO, "YES" );

        GetGDALDriverManager()->RegisterDriver( poDriver );
    }
}
```

The SetDescription() sets the name of the driver. This name is specified on the commandline when creating datasources so it is generally good to keep it short and without any special characters or spaces.

SetMetadataItem(GDAL_DCAP_VECTOR, "YES") is specified to indicate that the driver will handle vector data.

SetMetadataItem(GDAL_DCAP_VIRTUALIO, "YES") is specified to indicate that the driver can deal with files opened with the VSI*L GDAL API. Otherwise this metadata item should not be defined.

The driver declaration generally looks something like this for a format with read or read and update access (the Open() method) and creation support (the Create() method).

```
static GDALDataset* OGRSPFDriverOpen(GDALOpenInfo* poOpenInfo);
static int          OGRSPFDriverIdentify(GDALOpenInfo* poOpenInfo);
static GDALDataset* OGRSPFDriverCreate(const char* pszName, int nXSize, int nYSize,
                                       int nBands, GDALDataType eDT, char** papszOptions);
```

The Open() method is called by **GDALOpenEx()**. It should quietly return NULL if the passed filename is not of the format supported by the driver. If it is the target format, then a new **GDALDataset** object for the dataset should be returned.

It is common for the Open() method to be delegated to an Open() method on the actual format's **GDALDataset** class.

```
static GDALDataset *OGRSPFDriverOpen( GDALOpenInfo* poOpenInfo )
{
    if( !OGRSPFDriverIdentify(poOpenInfo) )
        return NULL;

    OGRSPFDataSource   *poDS = new OGRSPFDataSource();
    if( !poDS->Open( poOpenInfo->pszFilename, poOpenInfo->eAcces == GA_Update ) )
    {
        delete poDS;
        return NULL;
    }
    else
        return poDS;
}
```

The Identify() method is implemented as such :

```
static int OGRSPFDriverIdentify( GDALOpenInfo* poOpenInfo )
{
// ----------------------------------------------------------------------
//      Does this appear to be an .spf file?
// ----------------------------------------------------------------------
    return EQUAL( CPLGetExtension(poOpenInfo->pszFilename), "spf" );
}
```

Examples of the Create() method is left for the section on creation and update.

Basic Read Only Data Source

We will start implementing a minimal read-only datasource. No attempt is made to optimize operations, and default implementations of many methods inherited from **GDALDataset** are used.

The primary responsibility of the datasource is to manage the list of layers. In the case of the SPF format a datasource is a single file representing one layer so there is at most one layer. The "name" of a datasource should generally be the name passed to the Open() method.

The Open() method below is not overriding a base class method, but we have it to implement the open operation delegated by the driver class.

For this simple case we provide a stub TestCapability() that returns FALSE for all extended capabilities. The TestCapability() method is pure virtual, so it does need to be implemented.

```
class OGRSPFDataSource : public GDALDataset
```

```
{
    OGRSPFLayer       **papoLayers;
    int                 nLayers;

  public:
                        OGRSPFDataSource();
                        ~OGRSPFDataSource();

    int                 Open( const char * pszFilename, int bUpdate );

    int                 GetLayerCount() { return nLayers; }
    OGRLayer            *GetLayer( int );

    int                 TestCapability( const char * ) { return FALSE; }
};
```

The constructor is a simple initializer to a default state. The Open() will take care of actually attaching it to a file. The destructor is responsible for orderly cleanup of layers.

```
OGRSPFDataSource::OGRSPFDataSource()

{
    papoLayers = NULL;
    nLayers = 0;
}

OGRSPFDataSource::~OGRSPFDataSource()

{
    for( int i = 0; i < nLayers; i++ )
        delete papoLayers[i];
    CPLFree( papoLayers );
}
```

The Open() method is the most important one on the datasource, though in this particular instance it passes most of it's work off to the OGRSPFLayer constructor if it believes the file is of the desired format.

Note that Open() methods should try and determine that a file isn't of the identified format as efficiently as possible, since many drivers may be invoked with files of the wrong format before the correct driver is reached. In this particular Open() we just test the file extension but this is generally a poor way of identifying a file format. If available, checking "magic header values" or something similar is preferrable.

In the case of the SPF format, update in place is not supported, so we always fail if bUpdate is FALSE.

```
int  OGRSPFDataSource::Open( const char *pszFilename, int bUpdate )

{
    if( bUpdate )
    {
        CPLError( CE_Failure, CPLE_OpenFailed,
                  "Update access not supported by the SPF driver." );
        return FALSE;
    }

// ----------------------------------------------------------------
//      Create a corresponding layer.
// ----------------------------------------------------------------
    nLayers = 1;
    papoLayers = (OGRSPFLayer **) CPLMalloc(sizeof(void*));

    papoLayers[0] = new OGRSPFLayer( pszFilename );
```

```
    pszName = CPLStrdup( pszFilename );

    return TRUE;
}
```

A GetLayer() method also needs to be implemented. Since the layer list is created in the Open() this is just a lookup with some safety testing.

```
OGRLayer *OGRSPFDataSource::GetLayer( int iLayer )

{
    if( iLayer < 0 || iLayer >= nLayers )
        return NULL;
    else
        return papoLayers[iLayer];
}
```

Read Only Layer

The OGRSPFLayer is implements layer semantics for an .spf file. It provides access to a set of feature objects in a consistent coordinate system with a particular set of attribute columns. Our class definition looks like this:

```
class OGRSPFLayer : public OGRLayer
{
    OGRFeatureDefn     *poFeatureDefn;

    FILE               *fp;

    int                 nNextFID;

  public:
    OGRSPFLayer( const char *pszFilename );
   ~OGRSPFLayer();

    void                ResetReading();
    OGRFeature *        GetNextFeature();

    OGRFeatureDefn *    GetLayerDefn() { return poFeatureDefn; }

    int                 TestCapability( const char * ) { return FALSE; }
};
```

The layer constructor is responsible for initialization. The most important initialization is setting up the **OGRFeatureDefn** for the layer. This defines the list of fields and their types, the geometry type and the coordinate system for the layer. In the SPF format the set of fields is fixed - a single string field and we have no coordinate system info to set.

Pay particular attention to the reference counting of the **OGRFeatureDefn**. As OGRFeature's for this layer will also take a reference to this definition it is important that we also establish a reference on behalf of the layer itself.

```
OGRSPFLayer::OGRSPFLayer( const char *pszFilename )

{
    nNextFID = 0;

    poFeatureDefn = new OGRFeatureDefn( CPLGetBasename( pszFilename ) );
    SetDescription(poFeatureDefn->GetName());
    poFeatureDefn->Reference();
```

```
    poFeatureDefn->SetGeomType( wkbPoint );

    OGRFieldDefn oFieldTemplate( "Name", OFTString );

    poFeatureDefn->AddFieldDefn( &oFieldTemplate );

    fp = VSIFOpenL( pszFilename, "r" );
    if( fp == NULL )
        return;
}
```

Note that the destructor uses Release() on the **OGRFeatureDefn**. This will destroy the feature definition if the reference count drops to zero, but if the application is still holding onto a feature from this layer, then that feature will hold a reference to the feature definition and it will not be destroyed here (which is good!).

```
OGRSPFLayer::~OGRSPFLayer()

{
    poFeatureDefn->Release();
    if( fp != NULL )
        VSIFCloseL( fp );
}
```

The GetNextFeature() method is usually the work horse of **OGRLayer** implementations. It is responsible for reading the next feature according to the current spatial and attribute filters installed.

The while() loop is present to loop until we find a satisfactory feature. The first section of code is for parsing a single line of the SPF text file and establishing the x, y and name for the line.

```
OGRFeature *OGRSPFLayer::GetNextFeature()

{
    // ----------------------------------------------------------------
    //  Loop till we find a feature matching our requirements.
    // ----------------------------------------------------------------
    while( TRUE )
    {
        const char *pszLine;
        const char *pszName;

        pszLine = CPLReadLineL( fp );

        // Are we at end of file (out of features)?
        if( pszLine == NULL )
            return NULL;

        double dfX;
        double dfY;

        dfX = atof(pszLine);

        pszLine = strstr(pszLine,"|");
        if( pszLine == NULL )
            continue; // we should issue an error!
        else
            pszLine++;

        dfY = atof(pszLine);

        pszLine = strstr(pszLine,"|");
```

```
        if( pszLine == NULL )
            continue; // we should issue an error!
        else
            pszName = pszLine+1;
```

The next section turns the x, y and name into a feature. Also note that we assign a linearly incremented feature id. In our case we started at zero for the first feature, though some drivers start at 1.

```
        OGRFeature *poFeature = new OGRFeature( poFeatureDefn );

        poFeature->SetGeometryDirectly( new OGRPoint( dfX, dfY ) );
        poFeature->SetField( 0, pszName );
        poFeature->SetFID( nNextFID++ );
```

Next we check if the feature matches our current attribute or spatial filter if we have them. Methods on the **OGRLayer** base class support maintain filters in the **OGRLayer** member fields m_poFilterGeom (spatial filter) and m_poAttrQuery (attribute filter) so we can just use these values here if they are non-NULL. The following test is essentially "stock" and done the same in all formats. Some formats also do some spatial filtering ahead of time using a spatial index.

If the feature meets our criteria we return it. Otherwise we destroy it, and return to the top of the loop to fetch another to try.

```
        if( (m_poFilterGeom == NULL
             || FilterGeometry( poFeature->GetGeometryRef() ) )
            && (m_poAttrQuery == NULL
                || m_poAttrQuery->Evaluate( poFeature )) )
            return poFeature;

        delete poFeature;
    }
}
```

While in the middle of reading a feature set from a layer, or at any other time the application can call ResetReading() which is intended to restart reading at the beginning of the feature set. We implement this by seeking back to the beginning of the file, and resetting our feature id counter.

```
void OGRSPFLayer::ResetReading()

{
    VSIFSeekL( fp, 0, SEEK_SET );
    nNextFID = 0;
}
```

In this implementation we do not provide a custom implementation for the GetFeature() method. This means an attempt to read a particular feature by it's feature id will result in many calls to GetNextFeature() till the desired feature is found. However, in a sequential text format like spf there is little else we could do anyway.

There! We have completed a simple read-only feature file format driver.