GDAL Driver Implementation Tutorial

Overall Approach

In general new formats are added to GDAL by implementing format specific drivers as subclasses of **GDALDataset**, and band accessors as subclasses of **GDALRasterBand**. As well, a **GDALDriver** instance is created for the format, and registered with the **GDALDriverManager**, to ensure that the system *knows* about the format.

This tutorial will start with implementing a simple read-only driver (based on the JDEM driver), and then proceed to utilizing the RawRasterBand helper class, implementing creatable and updatable formats, and some esoteric issues.

It is strongly advised that the GDAL Data Model description be reviewed and understood before attempting to implement a GDAL driver.

Contents

Implementing the Dataset

We will start showing minimal implementation of a read-only driver for the Japanese DEM format (jdemdataset.cpp). First we declare a format specific dataset class, JDEMDataset in this case.

```
class JDEMDataset : public GDALPamDataset
{
    friend class JDEMRasterBand;

    FILE        *fp;
    GByte       abyHeader[1012];

  public:
                ~JDEMDataset();

    static GDALDataset *Open( GDALOpenInfo * );
    static int          Identify( GDALOpenInfo * );

    CPLErr      GetGeoTransform( double * padfTransform );
    const char *GetProjectionRef();
};
```

In general we provide capabilities for a driver, by overriding the various virtual methods on the **GDALDataset** base class. However, the Open() method is special. This is not a virtual method on the base class, and we will need a freestanding function for this operation, so we declare it static.

Implementing it as a method in the JDEMDataset class is convenient because we have privileged access to modify the contents of the database object.

The open method itself may look something like this:

```
GDALDataset *JDEMDataset::Open( GDALOpenInfo *poOpenInfo )

{
    // Confirm that the header is compatible with a JDEM dataset.
    if( !Identify(poOpenInfo) )
        return NULL;

    // Confirm the requested access is supported.
    if( poOpenInfo->eAccess == GA_Update )
    {
        CPLError(CE_Failure, CPLE_NotSupported,
                 "The JDEM driver does not support update access to existing "
                 "datasets.");
        return NULL;
    }

    // Check that the file pointer from GDALOpenInfo* is available
    if( poOpenInfo->fpL == NULL )
    {
        return NULL;
    }

    // Create a corresponding GDALDataset.
    JDEMDataset *poDS = new JDEMDataset();

    // Borrow the file pointer from GDALOpenInfo*.
    poDS->fp = poOpenInfo->fpL;
    poOpenInfo->fpL = NULL;

    // Read the header.
    VSIFReadL(poDS->abyHeader, 1, 1012, poDS->fp);

    poDS->nRasterXSize =
        JDEMGetField(reinterpret_cast<char *>(poDS->abyHeader) + 23, 3);
    poDS->nRasterYSize =
        JDEMGetField(reinterpret_cast<char *>(poDS->abyHeader) + 26, 3);
    if( poDS->nRasterXSize <= 0 || poDS->nRasterYSize <= 0 )
    {
        CPLError(CE_Failure, CPLE_AppDefined,
                 "Invalid dimensions : %d x %d",
                 poDS->nRasterXSize, poDS->nRasterYSize);
        delete poDS;
        return NULL;
    }

    // Create band information objects.
    poDS->SetBand(1, new JDEMRasterBand(poDS, 1));

    // Initialize any PAM information.
    poDS->SetDescription(poOpenInfo->pszFilename);
    poDS->TryLoadXML();

    // Initialize default overviews.
    poDS->oOvManager.Initialize(poDS, poOpenInfo->pszFilename);
    return poDS;
}
\code

The first step in any database Open function is to verify that the file
being passed is in fact of the type this driver is for.  It is important
to realize that each driver's Open function is called in turn till one
succeeds.  Drivers must quietly return NULL if the passed file is not of
their format.  They should only produce an error if the file does appear to
be of their supported format, but is for some reason unsupported or corrupt.

The information on the file to be opened is passed in contained in a
GDALOpenInfo object.  The GDALOpenInfo includes the following public
data members:

\code
    char        *pszFilename;
    char**      papszOpenOptions;
    GDALAccess  eAccess;  // GA_ReadOnly or GA_Update
    int         nOpenFlags;

    int         bStatOK;
    int         bIsDirectory;

    VSILFILE   *fpL;

    int         nHeaderBytes;
    GByte       *pabyHeader;
```

The driver can inspect these to establish if the file is supported. If the pszFilename refers to an object in the file system, the **bStatOK** flag will be set to TRUE. As well, if the file was successfully opened, the first kilobyte or so is read in, and put in **pabyHeader**, with the exact size in **nHeaderBytes**.

In this typical testing example it is verified that the file was successfully opened, that we have at least enough header information to perform our test, and that various parts of the header are as expected for this format. In this case, there are no *magic* numbers for JDEM format so we check various date fields to ensure they have reasonable century values. If the test fails, we quietly return NULL indicating this file isn't of our supported format.

The identification is in fact delegated to a Identify() static function :

```
/************************************************************************/
/*                              Identify()                              */
/************************************************************************/

int JDEMDataset::Identify( GDALOpenInfo * poOpenInfo )

{
    // Confirm that the header has what appears to be dates in the
    // expected locations.  Sadly this is a relatively weak test.
    if( poOpenInfo->nHeaderBytes < 50 )
        return FALSE;

    // Check if century values seem reasonable.
    const char *psHeader = reinterpret_cast<char *>(poOpenInfo->pabyHeader);
    if( (!EQUALN(psHeader + 11, "19", 2) &&
         !EQUALN(psHeader + 11, "20", 2)) ||
        (!EQUALN(psHeader + 15, "19", 2) &&
         !EQUALN(psHeader + 15, "20", 2)) ||
        (!EQUALN(psHeader + 19, "19", 2) &&
         !EQUALN(psHeader + 19, "20", 2)) )
    {
        return FALSE;
    }

    return TRUE;
}
\code

It is important to make the <i>is this my format</i> test as stringent as
possible.  In this particular case the test is weak, and a file that happened
to have 19s or 20s at a few locations could be erroneously recognized as
JDEM format, causing it to not be handled properly.

Once we are satisfied that the file is of our format, we can do any other
tests that are necessary to validate the file is usable, and in particular
that we can provide the level of access desired.  Since the JDEM driver does
not provide update support, error out in that case.

\code
    if( poOpenInfo->eAccess == GA_Update )
    {
        CPLError(CE_Failure, CPLE_NotSupported,
                 "The JDEM driver does not support update access to existing "
                 "datasets.");
        return NULL;
    }
```

Next we need to create an instance of the database class in which we will set various information of interest.

```
// Check that the file pointer from GDALOpenInfo* is available.
if( poOpenInfo->fpL == NULL )
{
    return NULL;
}

JDEMDataset *poDS = new JDEMDataset();

// Borrow the file pointer from GDALOpenInfo*.
poDS->fp = poOpenInfo->fpL;
poOpenInfo->fpL = NULL;
```

At this point we "borrow" the file handle that was held by GDALOpenInfo*. This file pointer uses the VSI*L GDAL API to access files on disk. This virtualized POSIX-style API allows some special capabilities like supporting large files, in-memory files and zipped files.

Next the X and Y size are extracted from the header. The nRasterXSize and nRasterYSize are data fields inherited from the **GDALDataset** base class, and must be set by the Open() method.

```
VSIFReadL(poDS->abyHeader, 1, 1012, poDS->fp);

poDS->nRasterXSize =
    JDEMGetField(reinterpret_cast<char *>(poDS->abyHeader) + 23, 3);
poDS->nRasterYSize =
    JDEMGetField(reinterpret_cast<char *>(poDS->abyHeader) + 26, 3);

if  (poDS->nRasterXSize <= 0 || poDS->nRasterYSize <= 0 )
{
    CPLError(CE_Failure, CPLE_AppDefined,
             "Invalid dimensions : %d x %d",
             poDS->nRasterXSize, poDS->nRasterYSize);
    delete poDS;
    return NULL;
}
```

All the bands related to this dataset must be created and attached using the SetBand() method. We will explore the JDEMRasterBand() class shortly.

```
// Create band information objects.
poDS->SetBand(1, new JDEMRasterBand(poDS, 1));
```

Finally we assign a name to the dataset object, and call the **GDALPamDataset** TryLoadXML() method which can initialize auxiliary information from an .aux.xml file if available. For more details on these services review the **GDALPamDataset** and related classes.

```
    // Initialize any PAM information.
    poDS->SetDescription( poOpenInfo->pszFilename );
    poDS->TryLoadXML();

    return poDS;
}
```

Implementing the RasterBand

Similar to the customized JDEMDataset class subclassed from **GDALDataset**, we also need to declare and implement a customized JDEMRasterBand derived from **GDALRasterBand** for access to the band(s) of the JDEM file. For JDEMRasterBand the declaration looks like this:

```
class JDEMRasterBand : public GDALPamRasterBand
{
  public:
    JDEMRasterBand( JDEMDataset *, int );
    virtual CPLErr IReadBlock( int, int, void * );
};
```

The constructor may have any signature, and is only called from the Open() method. Other virtual methods, such as IReadBlock() must be exactly matched to the method signature in **gdal_priv.h**.

The constructor implementation looks like this:

```
JDEMRasterBand::JDEMRasterBand( JDEMDataset *poDSIn, int nBandIn )

{
    poDS = poDSIn;
    nBand = nBandIn;

    eDataType = GDT_Float32;

    nBlockXSize = poDS->GetRasterXSize();
    nBlockYSize = 1;
}
```

The following data members are inherited from **GDALRasterBand**, and should generally be set in the band constructor.

- **poDS**: Pointer to the parent **GDALDataset**.
- **nBand**: The band number within the dataset.
- **eDataType**: The data type of pixels in this band.
- **nBlockXSize**: The width of one block in this band.
- **nBlockYSize**: The height of one block in this band.

The full set of possible GDALDataType values are declared in **gdal.h**, and include GDT_Byte, GDT_UInt16, GDT_Int16, and GDT_Float32. The block size is used to establish a *natural* or efficient block size to access the data with. For tiled datasets this will be the size of a tile, while for most other datasets it will be one scanline, as in this case.

Next we see the implementation of the code that actually reads the image data, IReadBlock().

```
CPLErr JDEMRasterBand::IReadBlock( int nBlockXOff, int nBlockYOff,
                                   void * pImage )

{
    JDEMDataset *poGDS = static_cast<JDEMDataset *>(poDS);
    int nRecordSize = nBlockXSize * 5 + 9 + 2;

    VSIFSeekL(poGDS->fp, 1011 + nRecordSize*nBlockYOff, SEEK_SET);

    char *pszRecord = static_cast<char *>(CPLMalloc(nRecordSize));
    VSIFReadL(pszRecord, 1, nRecordSize, poGDS->fp);

    if( !EQUALN(reinterpret_cast<char *>(poGDS->abyHeader), pszRecord, 6) )
    {
        CPLFree(pszRecord);

        CPLError(CE_Failure, CPLE_AppDefined,
                 "JDEM Scanline corrupt.  Perhaps file was not transferred "
                 "in binary mode?");
        return CE_Failure;
    }

    if( JDEMGetField(pszRecord + 6, 3) != nBlockYOff + 1 )
    {
        CPLFree(pszRecord);

        CPLError(CE_Failure, CPLE_AppDefined,
                 "JDEM scanline out of order, JDEM driver does not "
                 "currently support partial datasets.");
        return CE_Failure;
    }

    for( int i = 0; i < nBlockXSize; i++ )
        ((float *) pImage)[i] = JDEMGetField(pszRecord + 9 + 5 * i, 5) * 0.1;

    return CE_None;
}
```

Key items to note are:

- It is typical to cast the GDALRasterBand::poDS member to the derived type of the owning dataset. If your RasterBand class will need privileged access to the owning dataset object, ensure it is declared as a friend (omitted above for brevity).
- If an error occurs, report it with **CPLError()**, and return CE_Failure. Otherwise return CE_None.
- The pImage buffer should be filled with one block of data. The block is the size declared in nBlockXSize and nBlockYSize for the raster band. The type of the data within pImage should match the type declared in eDataType in the raster band object.
- The nBlockXOff and nBlockYOff are block offsets, so with 128x128 tiled datasets values of 1 and 1 would indicate the block going from (128,128) to (255,255) should be loaded.

The Driver

While the JDEMDataset and JDEMRasterBand are now ready to use to read image data, it still isn't clear how the GDAL system knows about the new driver. This is accomplished via the **GDALDriverManager**. To register our format we implement a registration function. The declaration goes in **gcore/gdal_frmts.h**:

```
void CPL_DLL GDALRegister_JDEM(void);
```

The definition in the driver file is:

```
void GDALRegister_JDEM()

{
    if( !GDAL_CHECK_VERSION("JDEM") )
        return;

    if( GDALGetDriverByName("JDEM") != NULL )
        return;

    GDALDriver *poDriver = new GDALDriver();

    poDriver->SetDescription("JDEM");
    poDriver->SetMetadataItem(GDAL_DCAP_RASTER, "YES");
    poDriver->SetMetadataItem(GDAL_DMD_LONGNAME,
                              "Japanese DEM (.mem)");
    poDriver->SetMetadataItem(GDAL_DMD_HELPTOPIC,
                              "frmt_various.html#JDEM");
    poDriver->SetMetadataItem(GDAL_DMD_EXTENSION, "mem");
    poDriver->SetMetadataItem(GDAL_DCAP_VIRTUALIO, "YES");

    poDriver->pfnOpen = JDEMDataset::Open;
    poDriver->pfnIdentify = JDEMDataset::Identify;

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

Note the use of GDAL_CHECK_VERSION macro (starting with GDAL 1.5.0). This is an optional macro for drivers inside GDAL tree that don't depend on external libraries, but that can be very useful if you compile your driver as a plugin (that is to say, an out-of-tree driver). As the GDAL C++ ABI may, and will, change between GDAL releases (for example from GDAL 1.5.0 to 1.6.0), it may be necessary to recompile your driver against the header files of the GDAL version with which you want to make it work. The GDAL_CHECK_VERSION macro will check that the GDAL version with which the driver was compiled and the version against which it is running are compatible.

The registration function will create an instance of a **GDALDriver** object when first called, and register it with the **GDALDriverManager**. The following fields can be set in the driver before registering it with the **GDALDriverManager()**.

- The description is the short name for the format. This is a unique name for this format, often used to identity the driver in scripts and command line programs. Normally 3-5 characters in length, and matching the prefix of the format classes. (mandatory)
- GDAL_DCAP_RASTER: set to YES to indicate that this driver handles raster data. (mandatory)
- GDAL_DMD_LONGNAME: A longer descriptive name for the file format, but still no longer than 50-60 characters. (mandatory)
- GDAL_DMD_HELPTOPIC: The name of a help topic to display for this driver, if any. In this case JDEM format is contained within the various format web page held in gdal/html. (optional)
- GDAL_DMD_EXTENSION: The extension used for files of this type. If more than one pick the primary extension, or none at all. (optional)
- GDAL_DMD_MIMETYPE: The standard mime type for this file format, such as "image/png". (optional)
- GDAL_DMD_CREATIONOPTIONLIST: There is evolving work on mechanisms to describe creation options. See the geotiff driver for an example of this. (optional)
- GDAL_DMD_CREATIONDATATYPES: A list of space separated data types supported by this create when creating new datasets. If a Create() method exists, these will be will supported. If a CreateCopy() method exists, this will be a list of types that can be losslessly exported but it may include weaker data types than the type eventually written. For instance, a format with a CreateCopy() method, and that always writes Float32 might also list Byte, Int16, and UInt16 since they can losslessly translated to Float32. An example value might be "Byte Int16 UInt16". (required - if creation supported)
- GDAL_DCAP_VIRTUALIO: set to YES to indicate that this driver can deal with files opened with the VSI*L GDAL API. Otherwise this metadata item should not be defined. (optional)
- pfnOpen: The function to call to try opening files of this format. (optional)
- pfnIdentify: The function to call to try identifying files of this format. A driver should return 1 if it recognizes the file as being of its format, 0 if it recognizes the file as being NOT of its format, or -1 if it cannot reach to a firm conclusion by just examining the header bytes. (optional)
- pfnCreate: The function to call to create new updatable datasets of this format. (optional)
- pfnCreateCopy: The function to call to create a new dataset of this format copied from another source, but not necessary updatable. (optional)
- pfnDelete: The function to call to delete a dataset of this format. (optional)

- pfnUnloadDriver: A function called only when the driver is destroyed. Could be used to cleanup data at the driver level. Rarely used. (optional)

Adding Driver to GDAL Tree

Note that the GDALRegister_JDEM() method must be called by the higher level program in order to have access to the JDEM driver. Normal practice when writing new drivers is to:

1. Add a driver directory under gdal/frmts, with the directory name the same as the short name.
2. Add a GNUmakefile and makefile.vc in that directory modeled on those from other similar directories (i.e. the jdem directory).
3. Add the module with the dataset, and rasterband implementation. Generally this is called <short_name>dataset.cpp, with all the GDAL specific code in one file, though that is not required.
4. Add the registration entry point declaration (i.e. GDALRegister_JDEM()) to **gdal/gcore/gdal_frmts.h**.
5. Add a call to the registration function to frmts/gdalallregister.cpp, protected by an appropriate #ifdef.
6. Add the format short name to the GDAL_FORMATS macro in GDALmake.opt.in (and to GDALmake.opt).
7. Add a format specific item to the EXTRAFLAGS macro in frmts/makefile.vc.

Once this is all done, it should be possible to rebuild GDAL, and have the new format available in all the utilities. The gdalinfo utility can be used to test that opening and reporting on the format is working, and the gdal_translate utility can be used to test image reading.

Adding Georeferencing

Now we will take the example a step forward, adding georeferencing support. We add the following two virtual method overrides to JDEMDataset, taking care to exactly match the signature of the method on the GDALRasterDataset base class.

```
CPLErr      GetGeoTransform( double * padfTransform );
const char *GetProjectionRef();
```

The implementation of GetGeoTransform() just copies the usual geotransform matrix into the supplied buffer. Note that GetGeoTransform() may be called a lot, so it isn't generally wise to do a lot of computation in it. In many cases the Open() will collect the geotransform, and this method will just copy it over. Also note that the geotransform return is based on an anchor point at the top left corner of the top left pixel, not the center of pixel approach used in some packages.

```
CPLErr JDEMDataset::GetGeoTransform( double * padfTransform )

{
    const char *psHeader = reinterpret_cast<char *>(abyHeader);

    const double dfLLLat = JDEMGetAngle(psHeader + 29);
    const double dfLLLong = JDEMGetAngle(psHeader + 36);
    const double dfURLat = JDEMGetAngle(psHeader + 43);
    const double dfURLong = JDEMGetAngle(psHeader + 50);
    padfTransform[0] = dfLLLong;
    padfTransform[3] = dfURLat;
    padfTransform[1] = (dfURLong - dfLLLong) / GetRasterXSize();
    padfTransform[2] = 0.0;

    padfTransform[4] = 0.0;
    padfTransform[5] = -1 * (dfURLat - dfLLLat) / GetRasterYSize();


    return CE_None;
}
```

The GetProjectionRef() method returns a pointer to an internal string containing a coordinate system definition in OGC WKT format. In this case the coordinate system is fixed for all files of this format, but in more complex cases a definition may need to be composed on the fly, in which case it may be helpful to use the **OGRSpatialReference** class to help build the definition.

```
const char *JDEMDataset::GetProjectionRef()

{
    return
        "GEOGCS[\"Tokyo\",DATUM[\"Tokyo\",SPHEROID[\"Bessel 1841\","
        "6377397.155,299.1528128,AUTHORITY[\"EPSG\",7004]],TOWGS84[-148,"
        "507,685,0,0,0,0],AUTHORITY[\"EPSG\",6301]],PRIMEM[\"Greenwich\","
        "0,AUTHORITY[\"EPSG\",8901]],UNIT[\"DMSH\",0.0174532925199433,"
        "AUTHORITY[\"EPSG\",9108]],AXIS[\"Lat\",NORTH],AXIS[\"Long\",EAST],"
        "AUTHORITY[\"EPSG\",4301]]";
}
```

This completes explanation of the features of the JDEM driver. The full source for jdemdataset.cpp can be reviewed as needed.

Overviews

GDAL allows file formats to make pre-built overviews available to applications via the **GDALRasterBand::GetOverview()** and related methods. However, implementing this is pretty involved, and goes beyond the scope of this document for now. The GeoTIFF driver (gdal/frmts/gtiff/geotiff.cpp) and related source can be reviewed for an example of a file format implementing overview reporting and creation support.

Formats can also report that they have arbitrary overviews, by overriding the HasArbitraryOverviews() method on the **GDALRasterBand**, returning TRUE. In this case the raster band object is expected to override the RasterIO() method itself, to implement efficient access to imagery with resampling. This is also involved, and there are a lot of requirements for correct implementation of the RasterIO() method. An example of this can be found in the OGDI and ECW formats.

However, by far the most common approach to implementing overviews is to use the default support in GDAL for external overviews stored in TIFF files with the same name as the dataset, but the extension .ovr appended. In order to enable reading and creation of this style of overviews it is necessary for the **GDALDataset** to initialize the oOvManager object within itself. This is typically accomplished with a call like the following near the end of the Open() method (after the PAM TryLoadXML()).

```
poDS->oOvManager.Initialize(poDS, poOpenInfo->pszFilename);
```

This will enable default implementations for reading and creating overviews for the format. It is advised that this be enabled for all simple file system based formats unless there is a custom overview mechanism to be tied into.

File Creation

There are two approaches to file creation. The first method is called the CreateCopy() method, and involves implementing a function that can write a file in the output format, pulling all imagery and other information needed from a source **GDALDataset**. The second method, the dynamic creation method, involves implementing a Create method to create the shell of the file, and then the application writes various information by calls to set methods.

The benefits of the first method are that that all the information is available at the point the output file is being created. This can be especially important when implementing file formats using external libraries which require information like color maps, and georeferencing information at the point the file is created. The other advantage of this method is that the CreateCopy() method can read some kinds of information, such as min/max, scaling, description and GCPs for which there are no equivalent set methods.

The benefits of the second method are that applications can create an empty new file, and write results to it as they become available. A complete image of the desired data does not have to be available in advance.

For very important formats both methods may be implemented, otherwise do whichever is simpler, or provides the required capabilities.

CreateCopy

The **GDALDriver::CreateCopy()** method call is passed through directly, so that method should be consulted for details of arguments. However, some things to keep in mind are:

- If the bStrict flag is FALSE the driver should try to do something reasonable when it cannot exactly represent the source dataset, transforming data types on the fly, dropping georeferencing and so forth.

- Implementing progress reporting correctly is somewhat involved. The return result of the progress function needs always to be checked for cancellation, and progress should be reported at reasonable intervals. The JPEGCreateCopy() method demonstrates good handling of the progress function.

- Special creation options should be documented in the on-line help. If the options take the format "NAME=VALUE" the papszOptions list can be manipulated with CPLFetchNameValue() as demonstrated in the handling of the QUALITY and PROGRESSIVE flags for JPEGCreateCopy().

- The returned **GDALDataset** handle can be in ReadOnly or Update mode. Return it in Update mode if practical, otherwise in ReadOnly mode is fine.

The full implementation of the CreateCopy function for JPEG (which is assigned to pfnCreateCopy in the **GDALDriver** object) is here.

```
static GDALDataset *
JPEGCreateCopy( const char * pszFilename, GDALDataset *poSrcDS,
                int bStrict, char ** papszOptions,
                GDALProgressFunc pfnProgress, void * pProgressData )

{
    const int nBands = poSrcDS->GetRasterCount();
    const int nXSize = poSrcDS->GetRasterXSize();
    const int nYSize = poSrcDS->GetRasterYSize();

    // Some some rudimentary checks
    if( nBands != 1 && nBands != 3 )
    {
        CPLError(CE_Failure, CPLE_NotSupported,
                 "JPEG driver doesn't support %d bands.  Must be 1 (grey) "
                 "or 3 (RGB) bands.", nBands);

        return NULL;
    }

    if( poSrcDS->GetRasterBand(1)->GetRasterDataType() != GDT_Byte && bStrict )
    {
        CPLError(CE_Failure, CPLE_NotSupported,
                 "JPEG driver doesn't support data type %s. "
                 "Only eight bit byte bands supported.",
                 GDALGetDataTypeName(
                     poSrcDS->GetRasterBand(1)->GetRasterDataType()));

        return NULL;
    }

    // What options has the user selected?

    int nQuality = 75;
    if( CSLFetchNameValue(papszOptions, "QUALITY") != NULL )
    {
        nQuality = atoi(CSLFetchNameValue(papszOptions, "QUALITY"));
        if( nQuality < 10 || nQuality > 100 )
        {
            CPLError(CE_Failure, CPLE_IllegalArg,
                     "QUALITY=%s is not a legal value in the range 10 - 100.",
                     CSLFetchNameValue(papszOptions, "QUALITY"));
            return NULL;
        }
    }

    bool bProgressive = false;
    if( CSLFetchNameValue(papszOptions, "PROGRESSIVE") != NULL )
    {
        bProgressive = true;
    }

    // Create the dataset.
    VSILFILE *fpImage = VSIFOpenL(pszFilename, "wb");
    if( fpImage == NULL )
    {
        CPLError(CE_Failure, CPLE_OpenFailed,
                 "Unable to create jpeg file %s.",
                 pszFilename);
        return NULL;
    }

    // Initialize JPG access to the file.
    struct jpeg_compress_struct sCInfo;
    struct jpeg_error_mgr sJErr;

    sCInfo.err = jpeg_std_error(&sJErr);
    jpeg_create_compress(&sCInfo);

    jpeg_stdio_dest(&sCInfo, fpImage);

    sCInfo.image_width = nXSize;
    sCInfo.image_height = nYSize;
    sCInfo.input_components = nBands;

    if( nBands == 1 )
    {
        sCInfo.in_color_space = JCS_GRAYSCALE;
    }
    else
    {
        sCInfo.in_color_space = JCS_RGB;
    }

    jpeg_set_defaults(&sCInfo);

    jpeg_set_quality(&sCInfo, nQuality, TRUE);
    if( bProgressive )
        jpeg_simple_progression(&sCInfo);

    jpeg_start_compress(&sCInfo, TRUE);

    // Loop over image, copying image data.
    GByte *pabyScanline = static_cast<GByte *>(CPLMalloc(nBands * nXSize));

    for( int iLine = 0; iLine < nYSize; iLine++ )
    {
        for( int iBand = 0; iBand < nBands; iBand++ )
        {
            GDALRasterBand * poBand = poSrcDS->GetRasterBand(iBand + 1);
            const CPLErr eErr =
                poBand->RasterIO(GF_Read, 0, iLine, nXSize, 1,
                                 pabyScanline + iBand, nXSize, 1, GDT_Byte,
                                 nBands, nBands * nXSize);
            // TODO: Handle error.
        }

        JSAMPLE *ppSamples = pabyScanline;
        jpeg_write_scanlines(&sCInfo, &ppSamples, 1);
    }

    CPLFree(pabyScanline);

    jpeg_finish_compress(&sCInfo);
    jpeg_destroy_compress(&sCInfo);

    VSIFCloseL(fpImage);

    return static_cast<GDALDataset *>(GDALOpen(pszFilename, GA_ReadOnly));
}
```

Dynamic Creation

In the case of dynamic creation, there is no source dataset. Instead the size, number of bands, and pixel data type of the desired file is provided but other information (such as georeferencing, and imagery data) would be supplied later via other method calls on the resulting **GDALDataset**.

The following sample implement PCI .aux labeled raw raster creation. It follows a common approach of creating a blank, but valid file using non-GDAL calls, and then calling GDALOpen(,GA_Update) at the end to return a writable file handle. This avoids having to duplicate the various setup actions in the Open() function.

```
GDALDataset *PAuxDataset::Create( const char * pszFilename,
                                  int nXSize, int nYSize, int nBands,
                                  GDALDataType eType,
                                  char ** /* papszParmList */ )

{
    // Verify input options.
    if( eType != GDT_Byte && eType != GDT_Float32 &&
        eType != GDT_UInt16 && eType != GDT_Int16 )
    {
        CPLError(
            CE_Failure, CPLE_AppDefined,
            "Attempt to create PCI .Aux labeled dataset with an illegal "
            "data type (%s).",
            GDALGetDataTypeName(eType));

        return NULL;
    }

    // Try to create the file.
    FILE *fp = VSIFOpen(pszFilename, "w");

    if( fp == NULL )
    {
        CPLError(CE_Failure, CPLE_OpenFailed,
                 "Attempt to create file `%s' failed.",
                 pszFilename);
        return NULL;
    }

    // Just write out a couple of bytes to establish the binary
    // file, and then close it.
    VSIFWrite("\0\0", 2, 1, fp);
    VSIFClose(fp);

    // Create the aux filename.
    char *pszAuxFilename = static_cast<char *>(CPLMalloc(strlen(pszFilename) + 5));
    strcpy(pszAuxFilename, pszFilename);;

    for( int i = strlen(pszAuxFilename) - 1; i > 0; i-- )
    {
        if( pszAuxFilename[i] == '.' )
        {
            pszAuxFilename[i] = '\0';
            break;
        }
    }

    strcat(pszAuxFilename, ".aux");

    // Open the file.
    fp = VSIFOpen(pszAuxFilename, "wt");
    if( fp == NULL )
    {
        CPLError(CE_Failure, CPLE_OpenFailed,
                 "Attempt to create file `%s' failed.",
                 pszAuxFilename);
        return NULL;
    }

    // We need to write out the original filename but without any
    // path components in the AuxiliaryTarget line.  Do so now.
    int iStart = strlen(pszFilename) - 1;
    while( iStart > 0 && pszFilename[iStart - 1] != '/' &&
           pszFilename[iStart - 1] != '\\' )
        iStart--;

    VSIFPrintf(fp, "AuxilaryTarget: %s\n", pszFilename + iStart);

    // Write out the raw definition for the dataset as a whole.
    VSIFPrintf(fp, "RawDefinition: %d %d %d\n",
               nXSize, nYSize, nBands);

    // Write out a definition for each band.  We always write band
    // sequential files for now as these are pretty efficiently
    // handled by GDAL.
    int nImgOffset = 0;

    for( int iBand = 0; iBand < nBands; iBand++ )
    {
        const int nPixelOffset = GDALGetDataTypeSize(eType)/8;
        const int nLineOffset = nXSize * nPixelOffset;

        const char *pszTypeName = NULL;
        if( eType == GDT_Float32 )
            pszTypeName = "32R";
        else if( eType == GDT_Int16 )
            pszTypeName = "16S";
        else if( eType == GDT_UInt16 )
            pszTypeName = "16U";
        else
            pszTypeName = "8U";

        VSIFPrintf( fp, "ChanDefinition-%d: %s %d %d %d %s\n",
                    iBand + 1, pszTypeName,
                    nImgOffset, nPixelOffset, nLineOffset,
#ifdef CPL_LSB
                    "Swapped"
#else
                    "Unswapped"
#endif
                    );

        nImgOffset += nYSize * nLineOffset;
    }

    // Cleanup.
    VSIFClose(fp);
    return static_cast<GDALDataset *>(GDALOpen(pszFilename, GA_Update));
}
```

File formats supporting dynamic creation, or even just update-in-place access also need to implement an IWriteBlock() method on the raster band class. It has semantics similar to IReadBlock(). As well, for various esoteric reasons, it is critical that a FlushCache() method be implemented in the raster band destructor. This is to ensure that any write cache blocks for the band be flushed out before the destructor is called.

RawDataset/RawRasterBand Helper Classes

Many file formats have the actual imagery data stored in a regular, binary, scanline oriented format. Rather than re-implement the access semantics for this for each formats, there are provided RawDataset and RawRasterBand classes declared in gdal/frmts/raw that can be utilized to implement efficient and convenient access.

In these cases the format specific band class may not be required, or if required it can be derived from RawRasterBand. The dataset class should be derived from RawDataset.

The Open() method for the dataset then instantiates raster bands passing all the layout information to the constructor. For instance, the PNM driver uses the following calls to create it's raster bands.

```
if( poOpenInfo->pabyHeader[1] == '5' )
{
    poDS->SetBand(
        1, new RawRasterBand(poDS, 1, poDS->fpImage,
                             iIn, 1, nWidth, GDT_Byte, TRUE));
}
else
{
    poDS->SetBand(
        1, new RawRasterBand(poDS, 1, poDS->fpImage,
                             iIn, 3, nWidth*3, GDT_Byte, TRUE));
    poDS->SetBand(
        2, new RawRasterBand(poDS, 2, poDS->fpImage,
                             iIn+1, 3, nWidth*3, GDT_Byte, TRUE));
    poDS->SetBand(
        3, new RawRasterBand(poDS, 3, poDS->fpImage,
                             iIn+2, 3, nWidth*3, GDT_Byte, TRUE));
}
```

The RawRasterBand takes the following arguments.

- **poDS**: The **GDALDataset** this band will be a child of. This dataset must be of a class derived from RawRasterDataset.
- **nBand**: The band it is on that dataset, 1 based.
- **fpRaw**: The FILE * handle to the file containing the raster data.
- **nImgOffset**: The byte offset to the first pixel of raster data for the first scanline.
- **nPixelOffset**: The byte offset from the start of one pixel to the start of the next within the scanline.
- **nLineOffset**: The byte offset from the start of one scanline to the start of the next.
- **eDataType**: The GDALDataType code for the type of the data on disk.
- **bNativeOrder**: FALSE if the data is not in the same endianness as the machine GDAL is running on. The data will be automatically byte swapped.

Simple file formats utilizing the Raw services are normally placed all within one file in the gdal/frmts/raw directory. There are numerous examples there of format implementation.

Metadata, and Other Exotic Extensions

There are various other items in the GDAL data model, for which virtual methods exist on the **GDALDataset** and **GDALRasterBand**. They include:

- **Metadata**: Name/value text values about a dataset or band. The **GDALMajorObject** (base class for **GDALRasterBand** and **GDALDataset**) has built-in support for holding metadata, so for read access it only needs to be set with calls to SetMetadataItem() during the Open(). The SAR_CEOS (frmts/ceos2/sar_ceosdataset.cpp) and GeoTIFF drivers are examples of drivers implementing readable metadata.
- **ColorTables**: GDT_Byte raster bands can have color tables associated with them. The frmts/png/pngdataset.cpp driver contains an example of a format that supports colortables.
- **ColorInterpretation**: The PNG driver contains an example of a driver that returns an indication of whether a band should be treated as a Red, Green, Blue, Alpha or Greyscale band.
- **GCPs**: GDALDatasets can have a set of ground control points associated with them (as opposed to an explicit affine transform returned by GetGeotransform()) relating the raster to georeferenced coordinates. The MFF2 (gdal/frmts/raw/hkvdataset.cpp) format is a simple example of a format supporting GCPs.
- **NoDataValue**: Bands with known "nodata" values can implement the GetNoDataValue() method. See the PAux (frmts/raw/pauxdataset.cpp) for an example of this.
- **Category Names**: Classified images with names for each class can return them using the GetCategoryNames() method though no formats currently implement this.

Id