My Project

Contents

Chapter 1

GDAL - Geospatial Data Abstraction Library

Select language: [English] [Russian] [Portuguese] [French/Francais]

GDAL/OGR is a translator library for raster and vector geospatial data formats that is released under an X/MIT style Open Source license by the Open Source Geospatial Foundation. As a library, it presents a single raster abstract data model and single vector abstract data model to the calling application for all supported formats. It also comes with a variety of useful command line utilities for data translation and processing. The NEWS page describes the January 2017 GDAL/OGR 2.1.3 release.	

Traditionally GDAL used to design the raster part of the library, and OGR the vector part for Simple Features. Starting with GDAL 2.0, both sides have been more tightly integrated. You can still refer to the documentation of GDAL 1.X if needed.

Master: http://www.gdal.org

Download: http at download.osgeo.org

1.1 User Oriented Documentation

- Wiki - Various user and developer contributed documentation and hints
- Downloads - Ready to use binaries (executables)
- Supported raster formats (142 drivers) : GeoTIFF, Erdas Imagine, ECW, MrSID, JPEG2000, DTED, NITF, GeoPackage, ...
- Supported vector formats (84 drivers): ESRI Shapefile, ESRI ArcSDE, ESRI FileGDB, MapInfo (tab and mid/mif), GML, KML, PostGIS, Oracle Spatial, GeoPackage, ...
- Raster utility programs : gdalinfo, gdal_translate, gdaladdo, gdalwarp, ...
- Vector utility programs : ogrinfo, ogr2ogr, ogrtindex, ...
- Geographic Network utility programs : gnmmanage, gnmanalyse, ...
- GDAL FAQ
- Raster and Vector data models and architecture

- GDAL/OGR Governance and Community Participation
- GDAL Service Provider Listings (not vetted)
- Acknowledgements and Credits
- Software Using GDAL

1.2 Developer Oriented Documentation

- Building GDAL From Source
- Downloads - source code

1.2.1 Tutorials

- Raster topics:
  - Raster API tutorial
  - Raster driver implementation
  - Raster Virtual format (VRT)
  - GDAL Warp API tutorial (Reprojection, ...)
- Vector topics:
  - Vector API tutorial
  - Vector driver implementation
  - Vector Virtual format (VRT)
  - OGR - Feature Style Specification
- Geographic Network Model topics:
  - GNM basics
  - GNM API tutorial
- Projections and Spatial Reference Systems tutorial (OSR - OGRSpatialReference)

1.2.2 API documentation

- API Reference
- gdal.h: Raster C API
- ogr_api.h: Vector C API
- gdal_alg.h: GDAL Algorithms C API
- ogr_srs_api.h: Spatial Reference System C API
- gdal_utils.h: GDAL Utilities C API
- GDALDataset C++ API
- GDALRasterBand C++ API
- OGRLayer C++ API
- OGR SQL dialect and SQLITE SQL dialect

1.3 Conference

1.4 Mailing List

A gdal-announce mailing list `subscription` is a low volume way of keeping track of major developments with the GDAL/OGR project.

The `gdal-dev@lists.osgeo.org` mailing list can be used for discussion of development and user issues related to GDAL and related technologies. Subscriptions can be done, and archives reviewed `on the web`. The mailing list is also available in read-only format by NNTP at `news://news.gmane.org/gmane.comp.-gis.gdal.devel`.

Archives since 2005 are searchable on `Nabble`.

Some GDAL/OGR users and developers can also often be found in the `#gdal` IRC channel on irc.freenode.net.

1.5 Bug Reporting

GDAL bugs `can be reported`, and `can be listed` using Trac.

1.6 GDAL In Other Languages

The list of bindings of GDAL in other languages is maintained on the `GDAL Wiki`.

Chapter 2

Acknowledgements and Credits

There are too many people who have helped since GDAL/OGR was launched in late 1998 for me to thank them all. I have received moral support, financial support, code contributions, sample datasets, and bug reports from literally hundreds of people. However, below I would like to single out a few people and organizations who have supported GDAL over the years. Forgive me for all those I left out.

Frank Warmerdam

2.1 Personal

- **Andrey Kiselev**: my right hand man on GDAL for several years. He is primarily responsible for the HDF, MrSID, L1B, and PCIDSK drivers. He has also relieved me of most libtiff maintenance work.
- **Daniel Morissette**: for his key contributions to CPL library, and development of the Mapinfo TAB translator.
- **Howard Butler**: for substantial improvements to the python bindings.
- **Ken Shih**: for the bulk of the implementation of the OLE DB provider.
- **Markus Neteler**: for various contributions to GDAL documentation and general supportiveness.
- **Silke Reimer**: for work on Debian, and RPM packaging as well as the GDAL man pages.
- **Alessandro Amici**: for work on configuration and build system, and for the initial Debian packaging.
- **Stephane Villeneuve**: for development of the Mapinfo MIF translator.
- **Marin Byrne**: for producing the current GDAL icon set (based on the earlier version by Martin Daly).
- Darek Krawczyk: for producing design of the GDAL Team Member t-shirt (based on Marin's and Martin's graphics).

2.2 Corporate

- Applied Coherent Technologies: Supported implementation of the GDAL contour generator, as well as various improvements to HDF drivers. Has been a Silver sponsor of GDAL.
- Atlantis Scientific: Supported the development of the CEOS, and a variety of other radar oriented format drivers as well as development of OpenEV, my day-to-day GDAL image viewer.
- A.U.G. Signals: Supported work on the HDF, NITF and ODBC drivers.
- Avenza Systems: Supported development of dgnlib, the basis of OGR dgn support, as well as preliminary work on image warping in GDAL.

- Cadcorp: Supported development of the Virtual Warped Raster capability. Has been a silver sponsor of GDAL.
- DM Solutions Group: Supported the development of the DGN driver, the OGR Arc/Info Binary Coverage driver, OGR WCTS (Web Coordinate Transformation Server), OGR VRT driver, ODBC driver, MySQL driver, SQLite driver, OGR JOIN and OGR C API.
- ERMapper: provided primary sponsorship for GDAL from February 2005 to November 2006 to support work on GDAL improvement efforts not focused on any particular client project.
- Geological Survey of Canada, Natural Resources Canada: Supported the initial development of the ArcSDE raster driver.
- OSGIS and the Geo-Information and ICT Department of the Ministry of Transport, Public Works and Water Management: Funded the DWG/DXF writing driver in OGR.
- Geosoft: Supported improvements to libtiff (RGBA Strip/Tile access), and the Arc/Info Binary Grid driver.
- Geospace Inc, Supported the development of write functionality for the OGR ArcSDE driver.
- GeoTango: Supported OGR Memory driver, Virtual Raster Filtering, and NITF RPC capabilities.
- i-cubed: Supported the MrSID driver. Has been a Silver sponsor of GDAL.
- Ingres Corporation : is the premier provider of open source information management services to the enterprise with operations in 58 countries. Has been a Silver sponsor of GDAL.
- Intergraph: Supported development of the Erdas Imagine driver.
- Keyhole: Supported development of Erdas Imagine driver, and the GDAL Warp API.
- MicroImages Inc. : Has been a Silver sponsor of GDAL.
- OPeNDAP: Supported development of the OGR OPeNDAP Driver.
- PCI Geomatics: Supported development of the JPEG2000 (JP2KAK) driver.
- Pixia: Supported NITF/JPEG2000 read support.
- UN FAO: Supported development of the IDA (WinDisp) driver, and GDAL VB6 bindings.
- SoftMap: Supported initial development of OGR as well as the OGR MapInfo integration.
- SRC: Supported development of the OGR OCI (Oracle Spatial) driver. Has been a Silver sponsor of GDAL.
- Safe Software: Supported development of the OGR OLE DB provider, TIGER/Line driver, S-57 driver, DTED driver, FMEObjects driver, SDTS driver and NTF driver. Has been a Silver sponsor of GDAL.
- Yukon Department of the Environment: Supported development of CDED / USGS DEM Writer.
- Waypoint: Panorama(TM) mapping platform delivers powerful on-line mapping services for business and government organizations, including spatial analysis, mobile asset tracking, web publishing, and Web Mapping Services. Has been a Silver sponsor of GDAL.

Chapter 3

GDAL Downloads

This page has been moved to the Wiki with a topic on downloading `binaries (pre-built executables` and a topic on downloading `source`.

Chapter 4

Simple C Example: gdalinfo_lib.cpp

Chapter 5

Standard Driver Registration: gdalallregister.cpp

Chapter 6

Sample Driver: jdemdataset.cpp

Chapter 7

Building GDAL From Source

This topic now lives in the Wiki at: `http://trac.osgeo.org/gdal/wiki/BuildHints`

Chapter 8

GDAL Data Model

This document attempts to describe the GDAL data model. That is the types of information that a GDAL data store can contain, and their semantics.

8.1 Dataset

A dataset (represented by the GDALDataset class) is an assembly of related raster bands and some information common to them all. In particular the dataset has a concept of the raster size (in pixels and lines) that applies to all the bands. The dataset is also responsible for the georeferencing transform and coordinate system definition of all bands. The dataset itself can also have associated metadata, a list of name/value pairs in string form.

Note that the GDAL dataset, and raster band data model is loosely based on the OpenGIS Grid Coverages specification.

8.1.1 Coordinate System

Dataset coordinate systems are represented as OpenGIS Well Known Text strings. This can contain:

- An overall coordinate system name.
- A geographic coordinate system name.
- A datum identifier.
- An ellipsoid name, semi-major axis, and inverse flattening.
- A prime meridian name and offset from Greenwich.
- A projection method type (i.e. Transverse Mercator).
- A list of projection parameters (i.e. central_meridian).
- A units name, and conversion factor to meters or radians.
- Names and ordering for the axes.
- Codes for most of the above in terms of predefined coordinate systems from authorities such as EPSG.

For more information on OpenGIS WKT coordinate system definitions, and mechanisms to manipulate them, refer to the `osr_tutorial` document and/or the OGRSpatialReference class documentation.

The coordinate system returned by GDALDataset::GetProjectionRef() describes the georeferenced coordinates implied by the affine georeferencing transform returned by GDALDataset::GetGeoTransform(). The coordinate system returned by GDALDataset::GetGCPProjection() describes the georeferenced coordinates of the GCPs returned by GDALDataset::GetGCPs().

Note that a returned coordinate system strings of "" indicates nothing is known about the georeferencing coordinate system.

8.1.2 Affine GeoTransform

GDAL datasets have two ways of describing the relationship between raster positions (in pixel/line coordinates) and georeferenced coordinates. The first, and most commonly used is the affine transform (the other is GCPs).

The affine transform consists of six coefficients returned by GDALDataset::GetGeoTransform() which map pixel/line coordinates into georeferenced space using the following relationship:

```
    Xgeo = GT(0) + Xpixel*GT(1) + Yline*GT(2)
    Ygeo = GT(3) + Xpixel*GT(4) + Yline*GT(5)
```

In case of north up images, the GT(2) and GT(4) coefficients are zero, and the GT(1) is pixel width, and GT(5) is pixel height. The (GT(0),GT(3)) position is the top left corner of the top left pixel of the raster.

Note that the pixel/line coordinates in the above are from (0.0,0.0) at the top left corner of the top left pixel to (width-_in_pixels,height_in_pixels) at the bottom right corner of the bottom right pixel. The pixel/line location of the center of the top left pixel would therefore be (0.5,0.5).

8.1.3 GCPs

A dataset can have a set of control points relating one or more positions on the raster to georeferenced coordinates. All GCPs share a georeferencing coordinate system (returned by GDALDataset::GetGCPProjection()). Each GCP (represented as the GDAL_GCP class) contains the following:

```
typedef struct
{
    char    *pszId;
    char    *pszInfo;
    double  dfGCPPixel;
    double  dfGCPLine;
    double  dfGCPX;
    double  dfGCPY;
    double  dfGCPZ;
} GDAL_GCP;
```

The pszId string is intended to be a unique (and often, but not always numerical) identifier for the GCP within the set of GCPs on this dataset. The pszInfo is usually an empty string, but can contain any user defined text associated with the GCP. Potentially this can also contain machine parsable information on GCP status though that isn't done at this time.

The (Pixel,Line) position is the GCP location on the raster. The (X,Y,Z) position is the associated georeferenced location with the Z often being zero.

The GDAL data model does not imply a transformation mechanism that must be generated from the GCPs ... this is left to the application. However 1st to 5th order polynomials are common.

Normally a dataset will contain either an affine geotransform, GCPs or neither. It is uncommon to have both, and it is undefined which is authoritative.

8.1.4 Metadata

GDAL metadata is auxiliary format and application specific textual data kept as a list of name/value pairs. The names are required to be well behaved tokens (no spaces, or odd characters). The values can be of any length, and contain anything except an embedded null (ASCII zero).

The metadata handling system is not well tuned to handling very large bodies of metadata. Handling of more than 100K of metadata for a dataset is likely to lead to performance degradation.

Some formats will support generic (user defined) metadata, while other format drivers will map specific format fields to metadata names. For instance the TIFF driver returns a few information tags as metadata including the date/time field which is returned as:

```
TIFFTAG_DATETIME=1999:05:11 11:29:56
```

Metadata is split into named groups called domains, with the default domain having no name (NULL or ""). Some specific domains exist for special purposes. Note that currently there is no way to enumerate all the domains available for a given object, but applications can "test" for any domains they know how to interpret.

The following metadata items have well defined semantics in the default domain:

- AREA_OR_POINT: May be either "Area" (the default) or "Point". Indicates whether a pixel value should be assumed to represent a sampling over the region of the pixel or a point sample at the center of the pixel. This is not intended to influence interpretation of georeferencing which remains area oriented.
- NODATA_VALUES: The value is a list of space separated pixel values matching the number of bands in the dataset that can be collectively used to identify pixels that are nodata in the dataset. With this style of nodata a pixel is considered nodata in all bands if and only if all bands match the corresponding value in the NODATA_VALUES tuple. This metadata is not widely honoured by GDAL drivers, algorithms or utilities at this time.
- MATRIX_REPRESENTATION: This value, used for Polarimetric SAR datasets, contains the matrix representation that this data is provided in. The following are acceptable values:
  - SCATTERING
  - SYMMETRIZED_SCATTERING
  - COVARIANCE
  - SYMMETRIZED_COVARIANCE
  - COHERENCY
  - SYMMETRIZED_COHERENCY
  - KENNAUGH
  - SYMMETRIZED_KENNAUGH
- POLARIMETRIC_INTERP: This metadata item is defined for Raster Bands for polarimetric SAR data. This indicates which entry in the specified matrix representation of the data this band represents. For a dataset provided as a scattering matrix, for example, acceptable values for this metadata item are HH, HV, VH, V-V. When the dataset is a covariance matrix, for example, this metadata item will be one of Covariance_11, Covariance_22, Covariance_33, Covariance_12, Covariance_13, Covariance_23 (since the matrix itself is a hermitian matrix, that is all the data that is required to describe the matrix).
- METADATATYPE: If IMAGERY Domain present, the item consist the reader which processed the metadata. Now present such readers:
  - DG: DigitalGlobe imagery metadata
  - GE: GeoEye (or formally SpaceImaging) imagery metadata
  - OV: OrbView imagery metadata
  - DIMAP: Pleiades imagery metadata
  - MSP: Resurs DK-1 imagery metadata
  - ODL: Landsat imagery metadata

8.1.4.1 SUBDATASETS Domain

The SUBDATASETS domain holds a list of child datasets. Normally this is used to provide pointers to a list of images stored within a single multi image file.

For example, an NITF with two images might have the following subdataset list.

```
SUBDATASET_1_NAME=NITF_IM:0:multi_1b.ntf
SUBDATASET_1_DESC=Image 1 of multi_1b.ntf
SUBDATASET_2_NAME=NITF_IM:1:multi_1b.ntf
SUBDATASET_2_DESC=Image 2 of multi_1b.ntf
```

The value of the _NAME is the string that can be passed to GDALOpen() to access the file. The _DESC value is intended to be a more user friendly string that can be displayed to the user in a selector.

Drivers which support subdatasets advertize the DMD_SUBDATASETS capability. This information is reported when the --format and --formats options are passed to the command line utilities.

Currently, drivers which support subdatasets are: ADRG, ECRGTOC, GEORASTER, GTiff, HDF4, HDF5, netCDF, NITF, NTv2, OGDI, PDF, PostGISRaster, Rasterlite, RPFTOC, RS2, WCS, and WMS.

8.1.4.2 IMAGE_STRUCTURE Domain

Metadata in the default domain is intended to be related to the image, and not particularly related to the way the image is stored on disk. That is, it is suitable for copying with the dataset when it is copied to a new format. Some information of interest is closely tied to a particular file format and storage mechanism. In order to prevent this getting copied along with datasets it is placed in a special domain called IMAGE_STRUCTURE that should not normally be copied to new formats.

Currently the following items are defined by RFC 14 as having specific semantics in the IMAGE_STRUCTURE domain.

- COMPRESSION: The compression type used for this dataset or band. There is no fixed catalog of compression type names, but where a given format includes a COMPRESSION creation option, the same list of values should be used here as there.
- NBITS: The actual number of bits used for this band, or the bands of this dataset. Normally only present when the number of bits is non-standard for the datatype, such as when a 1 bit TIFF is represented through GDAL as GDT_Byte.
- INTERLEAVE: This only applies on datasets, and the value should be one of PIXEL, LINE or BAND. It can be used as a data access hint.
- PIXELTYPE: This may appear on a GDT_Byte band (or the corresponding dataset) and have the value SIGNEDBYTE to indicate the unsigned byte values between 128 and 255 should be interpreted as being values between -128 and -1 for applications that recognise the SIGNEDBYTE type.

8.1.4.3 RPC Domain

The RPC metadata domain holds metadata describing the Rational Polynomial Coefficient geometry model for the image if present. This geometry model can be used to transform between pixel/line and georeferenced locations. The items defining the model are:

- ERR_BIAS: Error - Bias. The RMS bias error in meters per horizontal axis of all points in the image (-1.0 if unknown)
- ERR_RAND: Error - Random. RMS random error in meters per horizontal axis of each point in the image (-1.0 if unknown)
- LINE_OFF: Line Offset

- SAMP_OFF: Sample Offset
- LAT_OFF: Geodetic Latitude Offset
- LONG_OFF: Geodetic Longitude Offset
- HEIGHT_OFF: Geodetic Height Offset
- LINE_SCALE: Line Scale
- SAMP_SCALE: Sample Scale
- LAT_SCALE: Geodetic Latitude Scale
- LONG_SCALE: Geodetic Longitude Scale
- HEIGHT_SCALE: Geodetic Height Scale
- LINE_NUM_COEFF (1-20): Line Numerator Coefficients. Twenty coefficients for the polynomial in the Numerator of the rn equation. (space separated)
- LINE_DEN_COEFF (1-20): Line Denominator Coefficients. Twenty coefficients for the polynomial in the Denominator of the rn equation. (space separated)
- SAMP_NUM_COEFF (1-20): Sample Numerator Coefficients. Twenty coefficients for the polynomial in the Numerator of the cn equation. (space separated)
- SAMP_DEN_COEFF (1-20): Sample Denominator Coefficients. Twenty coefficients for the polynomial in the Denominator of the cn equation. (space separated)

These fields are directly derived from the document prospective GeoTIFF RPC document (`http://geotiff.maptools.org/rpc_prop.html`) which in turn is closely modeled on the NITF RPC00B definition.

The line and pixel offset expressed with LINE_OFF and SAMP_OFF are with respect to the center of the pixel.

8.1.4.4 IMAGERY Domain (remote sensing)

For satellite or aerial imagery the IMAGERY Domain may be present. It depends on exist special metadata files near the image file. The files at the same directory with image file tested by the set of metadata readers, if files can be processed by the metadata reader, it fill the IMAGERY Domain with the following items:

- SATELLITEID: A satellite or scanner name
- CLOUDCOVER: Cloud coverage. The value between 0 - 100 or 999 if not available
- ACQUISITIONDATETIME: The image acquisition date time in UTC

8.1.4.5 xml: Domains

Any domain name prefixed with "xml:" is not normal name/value metadata. It is a single XML document stored in one big string.

8.2 Raster Band

A raster band is represented in GDAL with the GDALRasterBand class. It represents a single raster band/channel/layer. It does not necessarily represent a whole image. For instance, a 24bit RGB image would normally be represented as a dataset with three bands, one for red, one for green and one for blue.

A raster band has the following properties:

- A width and height in pixels and lines. This is the same as that defined for the dataset, if this is a full resolution band.
- A datatype (GDALDataType). One of Byte, UInt16, Int16, UInt32, Int32, Float32, Float64, and the complex types CInt16, CInt32, CFloat32, and CFloat64.
- A block size. This is a preferred (efficient) access chunk size. For tiled images this will be one tile. For scanline oriented images this will normally be one scanline.
- A list of name/value pair metadata in the same format as the dataset, but of information that is potentially specific to this band.
- An optional description string.
- An optional single nodata pixel value (see also NODATA_VALUES metadata on the dataset for multi-band style nodata values).
- An optional nodata mask band marking pixels as nodata or in some cases transparency as discussed in `RFC 15: Band Masks` and documented in GDALRasterBand::GetMaskBand().
- An optional list of category names (effectively class names in a thematic image).
- An optional minimum and maximum value.
- An optional offset and scale for transforming raster values into meaning full values (i.e. translate height to meters).
- An optional raster unit name. For instance, this might indicate linear units for elevation data.
- A color interpretation for the band. This is one of:
  - GCI_Undefined: the default, nothing is known.
  - GCI_GrayIndex: this is an independent gray-scale image
  - GCI_PaletteIndex: this raster acts as an index into a color table
  - GCI_RedBand: this raster is the red portion of an RGB or RGBA image
  - GCI_GreenBand: this raster is the green portion of an RGB or RGBA image
  - GCI_BlueBand: this raster is the blue portion of an RGB or RGBA image
  - GCI_AlphaBand: this raster is the alpha portion of an RGBA image
  - GCI_HueBand: this raster is the hue of an HLS image
  - GCI_SaturationBand: this raster is the saturation of an HLS image
  - GCI_LightnessBand: this raster is the hue of an HLS image
  - GCI_CyanBand: this band is the cyan portion of a CMY or CMYK image
  - GCI_MagentaBand: this band is the magenta portion of a CMY or CMYK image
  - GCI_YellowBand: this band is the yellow portion of a CMY or CMYK image
  - GCI_BlackBand: this band is the black portion of a CMYK image.
- A color table, described in more detail later.
- Knowledge of reduced resolution overviews (pyramids) if available.

8.3 Color Table

A color table consists of zero or more color entries described in C by the following structure:

```
typedef struct
{
    /- gray, red, cyan or hue -/
    short      c1;
    /- green, magenta, or lightness -/
    short      c2;

    /- blue, yellow, or saturation -/
    short      c3;

    /- alpha or black band -/
    short      c4;
} GDALColorEntry;
```

The color table also has a palette interpretation value (GDALPaletteInterp) which is one of the following values, and indicates how the c1/c2/c3/c4 values of a color entry should be interpreted.

- GPI_Gray: Use c1 as gray scale value.
- GPI_RGB: Use c1 as red, c2 as green, c3 as blue and c4 as alpha.
- GPI_CMYK: Use c1 as cyan, c2 as magenta, c3 as yellow and c4 as black.
- GPI_HLS: Use c1 as hue, c2 as lightness, and c3 as saturation.

To associate a color with a raster pixel, the pixel value is used as a subscript into the color table. That means that the colors are always applied starting at zero and ascending. There is no provision for indicating a pre-scaling mechanism before looking up in the color table.

8.4 Overviews

A band may have zero or more overviews. Each overview is represented as a "free standing" GDALRasterBand. The size (in pixels and lines) of the overview will be different than the underlying raster, but the geographic region covered by overviews is the same as the full resolution band.

The overviews are used to display reduced resolution overviews more quickly than could be done by reading all the full resolution data and downsampling.

Bands also have a HasArbitraryOverviews property which is TRUE if the raster can be read at any resolution efficiently but with no distinct overview levels. This applies to some FFT encoded images, or images pulled through gateways (like OGDI) where downsampling can be done efficiently at the remote point.

Chapter 9

GDAL Driver Implementation Tutorial

9.1 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.

9.2 Contents

1. Implementing the Dataset
2. Implementing the RasterBand
3. The Driver
4. Adding Driver to GDAL Tree
5. Adding Georeferencing
6. Overviews
7. File Creation
8. RawDataset/RawRasterBand Helper Classes
9. Metadata, and Other Exotic Extensions

9.3 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;
}
```

9.4 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.

9.5 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 G-DALDriverManager. 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)

9.6 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.

9.7 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.

9.8 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.

9.9 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.

9.9.1 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));
}
```

9.9.2 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 AuxilaryTarget 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.

9.10 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.

9.11 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 GetCategory-Names() method though no formats currently implement this.

Chapter 10

GDAL API Tutorial

10.1 Opening the File

Before opening a GDAL supported raster datastore it is necessary to register drivers. There is a driver for each supported format. Normally this is accomplished with the GDALAllRegister() function which attempts to register all known drivers, including those auto-loaded from .so files using GDALDriverManager::AutoLoadDrivers(). If for some applications it is necessary to limit the set of drivers it may be helpful to review the code from `gdalallregister.cpp`. Python automatically calls GDALAllRegister() when the gdal module is imported.

Once the drivers are registered, the application should call the free standing GDALOpen() function to open a dataset, passing the name of the dataset and the access desired (GA_ReadOnly or GA_Update).

In C++:

```
#include "gdal_priv.h"
#include "cpl_conv.h" // for CPLMalloc()

int main()
{
    GDALDataset  *poDataset;

    GDALAllRegister();

    poDataset = (GDALDataset *) GDALOpen( pszFilename, GA_ReadOnly );
    if( poDataset == NULL )
    {
    ...;
    }
```

In C:

```
#include "gdal.h"
#include "cpl_conv.h" /* for CPLMalloc() */

int main()
{
    GDALDatasetH  hDataset;

    GDALAllRegister();

    hDataset = GDALOpen( pszFilename, GA_ReadOnly );
    if( hDataset == NULL )
    {
    ...;
    }
```

In Python:

```
import gdal
from gdalconst import *

dataset = gdal.Open( filename, GA_ReadOnly )
if dataset is None:
    ...
```

Note that if GDALOpen() returns NULL it means the open failed, and that an error messages will already have been emitted via CPLError(). If you want to control how errors are reported to the user review the CPLError() documentation. Generally speaking all of GDAL uses CPLError() for error reporting. Also, note that pszFilename need not actually be the name of a physical file (though it usually is). It's interpretation is driver dependent, and it might be an URL, a filename with additional parameters added at the end controlling the open or almost anything. Please try not to limit GDAL file selection dialogs to only selecting physical files.

10.2 Getting Dataset Information

As described in the GDAL Data Model, a GDALDataset contains a list of raster bands, all pertaining to the same area, and having the same resolution. It also has metadata, a coordinate system, a georeferencing transform, size of raster and various other information.

In the particular, but common, case of a "north up" image without any rotation or shearing, the georeferencing transform takes the following form :

```
adfGeoTransform[0] /* top left x */
adfGeoTransform[1] /* w-e pixel resolution */
adfGeoTransform[2] /* 0 */
adfGeoTransform[3] /* top left y */
adfGeoTransform[4] /* 0 */
adfGeoTransform[5] /* n-s pixel resolution (negative value) */
```

In the general case, this is an affine transform.

If we wanted to print some general information about the dataset we might do the following:

In C++:

```
double        adfGeoTransform[6];

printf( "Driver: %s/%s\n",
        poDataset->GetDriver()->GetDescription(),
        poDataset->GetDriver()->GetMetadataItem( GDAL_DMD_LONGNAME ) );

printf( "Size is %dx%dx%d\n",
        poDataset->GetRasterXSize(), poDataset->GetRasterYSize(),
        poDataset->GetRasterCount() );

if( poDataset->GetProjectionRef()  != NULL )
    printf( "Projection is `%s'\n", poDataset->GetProjectionRef() );

if( poDataset->GetGeoTransform( adfGeoTransform ) == CE_None )
{
    printf( "Origin = (%.6f,%.6f)\n",
            adfGeoTransform[0], adfGeoTransform[3] );

    printf( "Pixel Size = (%.6f,%.6f)\n",
            adfGeoTransform[1], adfGeoTransform[5] );
}
```

In C:

```
GDALDriverH   hDriver;
double        adfGeoTransform[6];

hDriver = GDALGetDatasetDriver( hDataset );
printf( "Driver: %s/%s\n",
        GDALGetDriverShortName( hDriver ),
        GDALGetDriverLongName( hDriver ) );

printf( "Size is %dx%dx%d\n",
    GDALGetRasterXSize( hDataset ),
        GDALGetRasterYSize( hDataset ),
        GDALGetRasterCount( hDataset ) );

if( GDALGetProjectionRef( hDataset ) != NULL )
    printf( "Projection is `%s'\n", GDALGetProjectionRef( hDataset ) );

if( GDALGetGeoTransform( hDataset, adfGeoTransform ) == CE_None )
{
    printf( "Origin = (%.6f,%.6f)\n",
            adfGeoTransform[0], adfGeoTransform[3] );
    printf( "Pixel Size = (%.6f,%.6f)\n",
            adfGeoTransform[1], adfGeoTransform[5] );
}
```

In Python:

```
print 'Driver: ', dataset.GetDriver().ShortName,'/', \
      dataset.GetDriver().LongName
print 'Size is ',dataset.RasterXSize,'x',dataset.RasterYSize, \
      'x',dataset.RasterCount
print 'Projection is ',dataset.GetProjection()

geotransform = dataset.GetGeoTransform()
if not geotransform is None:
print 'Origin = (',geotransform[0], ',',geotransform[3],')'
print 'Pixel Size = (',geotransform[1], ',',geotransform[5],')'
```

10.3 Fetching a Raster Band

At this time access to raster data via GDAL is done one band at a time. Also, there is metadata, block sizes, color tables, and various other information available on a band by band basis. The following codes fetches a GDAL-RasterBand object from the dataset (numbered 1 through GetRasterCount()) and displays a little information about it.

In C++:

```
GDALRasterBand  *poBand;
    int             nBlockXSize, nBlockYSize;
    int             bGotMin, bGotMax;
    double          adfMinMax[2];

    poBand = poDataset->GetRasterBand( 1 );
poBand->GetBlockSize( &nBlockXSize, &nBlockYSize );
    printf( "Block=%dx%d Type=%s, ColorInterp=%s\n",
            nBlockXSize, nBlockYSize,
            GDALGetDataTypeName(poBand->GetRasterDataType()),
            GDALGetColorInterpretationName(
                poBand->GetColorInterpretation()) );

    adfMinMax[0] = poBand->GetMinimum( &bGotMin );
    adfMinMax[1] = poBand->GetMaximum( &bGotMax );
    if( ! (bGotMin && bGotMax) )
        GDALComputeRasterMinMax((GDALRasterBandH)poBand, TRUE, adfMinMax);

    printf( "Min=%.3fd, Max=%.3f\n", adfMinMax[0], adfMinMax[1] );

    if( poBand->GetOverviewCount() > 0 )
        printf( "Band has %d overviews.\n", poBand->GetOverviewCount() );

    if( poBand->GetColorTable() != NULL )
        printf( "Band has a color table with %d entries.\n",
                 poBand->GetColorTable()->GetColorEntryCount() );
```

In C:

```
GDALRasterBandH hBand;
    int             nBlockXSize, nBlockYSize;
    int             bGotMin, bGotMax;
    double          adfMinMax[2];

    hBand = GDALGetRasterBand( hDataset, 1 );
    GDALGetBlockSize( hBand, &nBlockXSize, &nBlockYSize );
    printf( "Block=%dx%d Type=%s, ColorInterp=%s\n",
            nBlockXSize, nBlockYSize,
            GDALGetDataTypeName(GDALGetRasterDataType(hBand)),
            GDALGetColorInterpretationName(
                GDALGetRasterColorInterpretation(hBand)) );

    adfMinMax[0] = GDALGetRasterMinimum( hBand, &bGotMin );
    adfMinMax[1] = GDALGetRasterMaximum( hBand, &bGotMax );
    if( ! (bGotMin && bGotMax) )
        GDALComputeRasterMinMax( hBand, TRUE, adfMinMax );

    printf( "Min=%.3fd, Max=%.3f\n", adfMinMax[0], adfMinMax[1] );
    if( GDALGetOverviewCount(hBand) > 0 )
        printf( "Band has %d overviews.\n", GDALGetOverviewCount(hBand));

    if( GDALGetRasterColorTable( hBand ) != NULL )
        printf( "Band has a color table with %d entries.\n",
                 GDALGetColorEntryCount(
                     GDALGetRasterColorTable( hBand ) ) );
```

In Python (note several bindings are missing):

```
band = dataset.GetRasterBand(1)

print 'Band Type=',gdal.GetDataTypeName(band.DataType)

min = band.GetMinimum()
max = band.GetMaximum()
if min is None or max is None:
    (min,max) = band.ComputeRasterMinMax(1)
print 'Min=%.3f, Max=%.3f' % (min,max)

if band.GetOverviewCount() > 0:
    print 'Band has ', band.GetOverviewCount(), ' overviews.'

    if not band.GetRasterColorTable() is None:
    print 'Band has a color table with ', \
    band.GetRasterColorTable().GetCount(), ' entries.'
```

10.4 Reading Raster Data

There are a few ways to read raster data, but the most common is via the GDALRasterBand::RasterIO() method. This method will automatically take care of data type conversion, up/down sampling and windowing. The following code will read the first scanline of data into a similarly sized buffer, converting it to floating point as part of the operation.

In C++:

```
float *pafScanline;
    int   nXSize = poBand->GetXSize();

    pafScanline = (float *) CPLMalloc(sizeof(float)*nXSize);
    poBand->RasterIO( GF_Read, 0, 0, nXSize, 1,
                      pafScanline, nXSize, 1, GDT_Float32,
                      0, 0 );
```

The pafScanline buffer should be freed with CPLFree() when it is no longer used.

In C:

```
float *pafScanline;
    int   nXSize = GDALGetRasterBandXSize( hBand );

    pafScanline = (float *) CPLMalloc(sizeof(float)*nXSize);
GDALRasterIO( hBand, GF_Read, 0, 0, nXSize, 1,
                  pafScanline, nXSize, 1, GDT_Float32,
                  0, 0 );
```

The pafScanline buffer should be freed with CPLFree() when it is no longer used.

In Python:

```
scanline = band.ReadRaster( 0, 0, band.XSize, 1, \
                                 band.XSize, 1, GDT_Float32 )
```

Note that the returned scanline is of type string, and contains xsize$*$4 bytes of raw binary floating point data. This can be converted to Python values using the **struct** module from the standard library:

```
import struct

tuple_of_floats = struct.unpack('f' * b2.XSize, scanline)
```

The RasterIO call takes the following arguments.

```
CPLErr GDALRasterBand::RasterIO( GDALRWFlag eRWFlag,
                                 int nXOff, int nYOff, int nXSize, int nYSize,
                                 void * pData, int nBufXSize, int nBufYSize,
                                 GDALDataType eBufType,
                                 int nPixelSpace,
                                 int nLineSpace )
```

Note that the same RasterIO() call is used to read, or write based on the setting of eRWFlag (either GF_Read or GF_Write). The nXOff, nYOff, nXSize, nYSize argument describe the window of raster data on disk to read (or write). It doesn't have to fall on tile boundaries though access may be more efficient if it does.

The pData is the memory buffer the data is read into, or written from. It's real type must be whatever is passed as eBufType, such as GDT_Float32, or GDT_Byte. The RasterIO() call will take care of converting between the buffer's data type and the data type of the band. Note that when converting floating point data to integer RasterIO() rounds down, and when converting source values outside the legal range of the output the nearest legal value is used. This implies, for instance, that 16bit data read into a GDT_Byte buffer will map all values greater than 255 to 255, **the data is not scaled!**

The nBufXSize and nBufYSize values describe the size of the buffer. When loading data at full resolution this would be the same as the window size. However, to load a reduced resolution overview this could be set to smaller than the window on disk. In this case the RasterIO() will utilize overviews to do the IO more efficiently if the overviews are suitable.

The nPixelSpace, and nLineSpace are normally zero indicating that default values should be used. However, they can be used to control access to the memory data buffer, allowing reading into a buffer containing other pixel interleaved data for instance.

10.5 Closing the Dataset

Please keep in mind that GDALRasterBand objects are *owned* by their dataset, and they should never be destroyed with the C++ delete operator. GDALDataset's can be closed by calling GDALClose() (it is NOT recommended to use the delete operator on a GDALDataset for Windows users because of known issues when allocating and freeing memory across module boundaries. See the relevant `topic` on the FAQ). Calling GDALClose will result in proper cleanup, and flushing of any pending writes. Forgetting to call GDALClose on a dataset opened in update mode in a popular format like GTiff will likely result in being unable to open it afterwards.

10.6 Techniques for Creating Files

New files in GDAL supported formats may be created if the format driver supports creation. There are two general techniques for creating files, using CreateCopy() and Create(). The CreateCopy method involves calling the Create-Copy() method on the format driver, and passing in a source dataset that should be copied. The Create method involves calling the Create() method on the driver, and then explicitly writing all the metadata, and raster data with separate calls. All drivers that support creating new files support the CreateCopy() method, but only a few support the Create() method.

To determine if a particular format supports Create or CreateCopy it is possible to check the DCAP_CREATE and DCAP_CREATECOPY metadata on the format driver object. Ensure that GDALAllRegister() has been called before calling GetDriverByName(). In this example we fetch a driver, and determine whether it supports Create() and/or CreateCopy().

In C++:

```
#include "cpl_string.h"
...
    const char *pszFormat = "GTiff";
    GDALDriver *poDriver;
    char **papszMetadata;

    poDriver = GetGDALDriverManager()->GetDriverByName(pszFormat);
    if( poDriver == NULL )
        exit( 1 );

    papszMetadata = poDriver->GetMetadata();
    if( CSLFetchBoolean( papszMetadata, GDAL_DCAP_CREATE, FALSE ) )
        printf( "Driver %s supports Create() method.\n", pszFormat );
    if( CSLFetchBoolean( papszMetadata, GDAL_DCAP_CREATECOPY, FALSE ) )
        printf( "Driver %s supports CreateCopy() method.\n", pszFormat );
```

In C:

```
#include "cpl_string.h"
...
    const char *pszFormat = "GTiff";
    GDALDriverH hDriver = GDALGetDriverByName( pszFormat );
    char **papszMetadata;

    if( hDriver == NULL )
        exit( 1 );

    papszMetadata = GDALGetMetadata( hDriver, NULL );
    if( CSLFetchBoolean( papszMetadata, GDAL_DCAP_CREATE, FALSE ) )
        printf( "Driver %s supports Create() method.\n", pszFormat );
    if( CSLFetchBoolean( papszMetadata, GDAL_DCAP_CREATECOPY, FALSE ) )
        printf( "Driver %s supports CreateCopy() method.\n", pszFormat );
```

In Python:

```
format = "GTiff"
driver = gdal.GetDriverByName( format )
metadata = driver.GetMetadata()
if metadata.has_key(gdal.DCAP_CREATE) \
   and metadata[gdal.DCAP_CREATE] == 'YES':
    print 'Driver %s supports Create() method.' % format
if metadata.has_key(gdal.DCAP_CREATECOPY) \
   and metadata[gdal.DCAP_CREATECOPY] == 'YES':
    print 'Driver %s supports CreateCopy() method.' % format
```

Note that a number of drivers are read-only and won't support Create() or CreateCopy().

10.7 Using CreateCopy()

The GDALDriver::CreateCopy() method can be used fairly simply as most information is collected from the source dataset. However, it includes options for passing format specific creation options, and for reporting progress to the user as a long dataset copy takes place. A simple copy from the a file named pszSrcFilename, to a new file named pszDstFilename using default options on a format whose driver was previously fetched might look like this:

In C++:

```
GDALDataset *poSrcDS =
   (GDALDataset *) GDALOpen( pszSrcFilename, GA_ReadOnly );
GDALDataset *poDstDS;

poDstDS = poDriver->CreateCopy( pszDstFilename, poSrcDS, FALSE,
                                NULL, NULL, NULL );

/* Once we're done, close properly the dataset */
if( poDstDS != NULL )
    GDALClose( (GDALDatasetH) poDstDS );
GDALClose( (GDALDatasetH) poSrcDS );
```

In C:

```
GDALDatasetH hSrcDS = GDALOpen( pszSrcFilename, GA_ReadOnly );
GDALDatasetH hDstDS;

hDstDS = GDALCreateCopy( hDriver, pszDstFilename, hSrcDS, FALSE,
                         NULL, NULL, NULL );

/* Once we're done, close properly the dataset */
if( hDstDS != NULL )
    GDALClose( hDstDS );
GDALClose(hSrcDS);
```

In Python:

```
src_ds = gdal.Open( src_filename )
dst_ds = driver.CreateCopy( dst_filename, src_ds, 0 )

# Once we're done, close properly the dataset
dst_ds = None
src_ds = None
```

Note that the CreateCopy() method returns a writable dataset, and that it must be closed properly to complete writing and flushing the dataset to disk. In the Python case this occurs automatically when "dst_ds" goes out of scope. The FALSE (or 0) value used for the bStrict option just after the destination filename in the CreateCopy() call indicates that the CreateCopy() call should proceed without a fatal error even if the destination dataset cannot be created to exactly match the input dataset. This might be because the output format does not support the pixel datatype of the input dataset, or because the destination cannot support writing georeferencing for instance.

A more complex case might involve passing creation options, and using a predefined progress monitor like this:

In C++:

```
#include "cpl_string.h"
...
    char **papszOptions = NULL;

    papszOptions = CSLSetNameValue( papszOptions, "TILED", "YES" );
    papszOptions = CSLSetNameValue( papszOptions, "COMPRESS", "PACKBITS" );
    poDstDS = poDriver->CreateCopy( pszDstFilename, poSrcDS, FALSE,
                                    papszOptions, GDALTermProgress, NULL );

    /* Once we're done, close properly the dataset */
    if( poDstDS != NULL )
        GDALClose( (GDALDatasetH) poDstDS );
    CSLDestroy( papszOptions );
```

In C:

```
#include "cpl_string.h"
...
    char **papszOptions = NULL;

    papszOptions = CSLSetNameValue( papszOptions, "TILED", "YES" );
    papszOptions = CSLSetNameValue( papszOptions, "COMPRESS", "PACKBITS" );
    hDstDS = GDALCreateCopy( hDriver, pszDstFilename, hSrcDS, FALSE,
                             papszOptions, GDALTermProgres, NULL );

    /* Once we're done, close properly the dataset */
    if( hDstDS != NULL )
        GDALClose( hDstDS );
    CSLDestroy( papszOptions );
```

In Python:

```
src_ds = gdal.Open( src_filename )
dst_ds = driver.CreateCopy( dst_filename, src_ds, 0,
                            [ 'TILED=YES', 'COMPRESS=PACKBITS' ] )

# Once we're done, close properly the dataset
dst_ds = None
src_ds = None
```

10.8 Using Create()

For situations in which you are not just exporting an existing file to a new file, it is generally necessary to use the G-DALDriver::Create() method (though some interesting options are possible through use of virtual files or in-memory files). The Create() method takes an options list much like CreateCopy(), but the image size, number of bands and band type must be provided explicitly.

In C++:

```
GDALDataset *poDstDS;
char **papszOptions = NULL;

poDstDS = poDriver->Create( pszDstFilename, 512, 512, 1, GDT_Byte,
                            papszOptions );
```

In C:

```
GDALDatasetH hDstDS;
char **papszOptions = NULL;

hDstDS = GDALCreate( hDriver, pszDstFilename, 512, 512, 1, GDT_Byte,
                     papszOptions );
```

In Python:

```
dst_ds = driver.Create( dst_filename, 512, 512, 1, gdal.GDT_Byte )
```

Once the dataset is successfully created, all appropriate metadata and raster data must be written to the file. What this is will vary according to usage, but a simple case with a projection, geotransform and raster data is covered here.

In C++:

```
double adfGeoTransform[6] = { 444720, 30, 0, 3751320, 0, -30 };
OGRSpatialReference oSRS;
char *pszSRS_WKT = NULL;
GDALRasterBand *poBand;
GByte abyRaster[512*512];

poDstDS->SetGeoTransform( adfGeoTransform );

oSRS.SetUTM( 11, TRUE );
oSRS.SetWellKnownGeogCS( "NAD27" );
oSRS.exportToWkt( &pszSRS_WKT );
poDstDS->SetProjection( pszSRS_WKT );
CPLFree( pszSRS_WKT );

poBand = poDstDS->GetRasterBand(1);
poBand->RasterIO( GF_Write, 0, 0, 512, 512,
                  abyRaster, 512, 512, GDT_Byte, 0, 0 );

/* Once we're done, close properly the dataset */
GDALClose( (GDALDatasetH) poDstDS );
```

In C:

```
double adfGeoTransform[6] = { 444720, 30, 0, 3751320, 0, -30 };
OGRSpatialReferenceH hSRS;
char *pszSRS_WKT = NULL;
GDALRasterBandH hBand;
GByte abyRaster[512*512];

GDALSetGeoTransform( hDstDS, adfGeoTransform );

hSRS = OSRNewSpatialReference( NULL );
OSRSetUTM( hSRS, 11, TRUE );
OSRSetWellKnownGeogCS( hSRS, "NAD27" );
OSRExportToWkt( hSRS, &pszSRS_WKT );
OSRDestroySpatialReference( hSRS );

GDALSetProjection( hDstDS, pszSRS_WKT );
CPLFree( pszSRS_WKT );

hBand = GDALGetRasterBand( hDstDS, 1 );
GDALRasterIO( hBand, GF_Write, 0, 0, 512, 512,
              abyRaster, 512, 512, GDT_Byte, 0, 0 );

/* Once we're done, close properly the dataset */
GDALClose( hDstDS );
```

In Python:

```
import osr
import numpy
```

```
dst_ds.SetGeoTransform( [ 444720, 30, 0, 3751320, 0, -30 ] )

srs = osr.SpatialReference()
srs.SetUTM( 11, 1 )
srs.SetWellKnownGeogCS( 'NAD27' )
dst_ds.SetProjection( srs.ExportToWkt() )

raster = numpy.zeros( (512, 512), dtype=numpy.uint8 )
dst_ds.GetRasterBand(1).WriteArray( raster )

# Once we're done, close properly the dataset
dst_ds = None
```

Chapter 11

GDAL Grid Tutorial

11.1 Introduction to Gridding

Gridding is a process of creating a regular grid (or call it a raster image) from the scattered data. Typically you have a set of arbitrary scattered over the region of survey measurements and you would like to convert them into the regular grid for further processing and combining with other grids.

Figure 11.1: Scattered data gridding

This problem can be solved using data interpolation or approximation algorithms. But you are not limited by interpolation here. Sometimes you don't need to interpolate your data but rather compute some statistics or data metrics over the region. Statistics is valuable itself or could be used for better choosing the interpolation algorithm and parameters.

That is what GDAL Grid API is about. It helps you to interpolate your data (see Interpolation of the Scattered Data) or compute data metrics (see Data Metrics Computation).

There are two ways of using this interface. Programmatically it is available through the GDALGridCreate C function; for end users there is a gdal_grid utility. The rest of this document discusses details on algorithms and their parameters implemented in GDAL Grid API.

11.2 Interpolation of the Scattered Data

11.2.1 Inverse Distance to a Power

The Inverse Distance to a Power gridding method is a weighted average interpolator. You should supply the input arrays with the scattered data values including coordinates of every data point and output grid geometry. The function will compute interpolated value for the given position in output grid.

For every grid node the resulting value Z will be calculated using formula:

$$Z = \frac{\sum_{i=1}^{n} \frac{Z_i}{r_i^p}}{\sum_{i=1}^{n} \frac{1}{r_i^p}}$$

where

- Z_i is a known value at point i,
- r is a distance from the grid node to point i,

- p is a weighting power,
- n is a number of points in search ellipse".

In this method the weighting factor w is

$$w = \frac{1}{r^p}$$

See GDALGridInverseDistanceToAPowerOptions for the list of GDALGridCreate parameters and gdal_grid_-algorithms_invdist for the list of gdal_grid options.

11.2.2 Moving Average

The Moving Average is a simple data averaging algorithm. It uses a moving window of elliptic form to search values and averages all data points within the window. Search ellipse can be rotated by specified angle, the center of ellipse located at the grid node. Also the minimum number of data points to average can be set, if there are not enough points in window, the grid node considered empty and will be filled with specified NODATA value.

Mathematically it can be expressed with the formula:

$$Z = \frac{\sum_{i=1}^{n} Z_i}{n}$$

where

- Z is a resulting value at the grid node,
- Z_i is a known value at point i,
- n is a number of points in search search ellipse.

See GDALGridMovingAverageOptions for the list of GDALGridCreate parameters and gdal_grid_algorithms_-average for the list of gdal_grid options.

11.2.3 Nearest Neighbor

The Nearest Neighbor method doesn't perform any interpolation or smoothing, it just takes the value of nearest point found in grid node search ellipse and returns it as a result. If there are no points found, the specified NODATA value will be returned.

See GDALGridNearestNeighborOptions for the list of GDALGridCreate parameters and gdal_grid_algorithms_-nearest for the list of gdal_grid options.

11.3 Data Metrics Computation

All the metrics have the same set controlling options. See the GDALGridDataMetricsOptions.

11.3.1 Minimum Data Value

Minimum value found in grid node search ellipse. If there are no points found, the specified NODATA value will be returned.

$$Z = \min(Z_1, Z_2, \ldots, Z_n)$$

where

- Z is a resulting value at the grid node,
- Z_i is a known value at point i,
- n is a number of points in search ellipse".

11.3.2 Maximum Data Value

Maximum value found in grid node search ellipse. If there are no points found, the specified NODATA value will be returned.

$$Z = \max(Z_1, Z_2, \ldots, Z_n)$$

where

- Z is a resulting value at the grid node,
- Z_i is a known value at point i,
- n is a number of points in search ellipse".

11.3.3 Data Range

A difference between the minimum and maximum values found in grid node search ellipse. If there are no points found, the specified NODATA value will be returned.

$$Z = \max(Z_1, Z_2, \ldots, Z_n) - \min(Z_1, Z_2, \ldots, Z_n)$$

where

- Z is a resulting value at the grid node,
- Z_i is a known value at point i,
- n is a number of points in search ellipse".

11.4 Search Ellipse

Search window in gridding algorithms specified in the form of rotated ellipse. It is described by the three parameters:

- $radius_1$ is the first radius (x axis if rotation angle is 0),
- $radius_2$ is the second radius (y axis if rotation angle is 0),
- $angle$ is a search ellipse rotation angle (rotated counter clockwise).

Figure 11.2: Search ellipse

Only points located inside the search ellipse (including its border line) will be used for computation.

Chapter 12

Sponsoring GDAL/OGR

Development and maintenance of GDAL/OGR is supported by organizations contracting developers, organizations contributing improvements, users contributing improvements, and volunteers. Generally speaking this works well, and GDAL/OGR has improved substantially over the years.

However, there are still many tasks which do not receive the attention they should. Processing bug reports, writing documentation, writing test scripts, evaluating test script failures and user support often receive less attention than would be desired. Some new features of broad interest are not implemented because they aren't important enough to any one person or organization.

In order to provide sustained funding to support the maintenance, improvement and promotion of the GDAL/OGR project, the project seeks project sponsors to provide financial support. Sponsorship would be accomplished via the `OSGeo Project Sponsorship` program. Funds are held by OSGeo for disposition on behalf of the project, and dispersed at the discretion of the GDAL/OGR Project Steering Committee.

12.1 Sponsorship Uses

The primary intended use of the sponsorship funds is to hire a maintainer on a contract basis. The responsibilities would include:

- Addressing bug reports - reproducing then fixing or passing on to another developer.
- Extending, and running the test suite.
- Improving documentation.
- Other improvements to the software.
- General user support on the mailing list.

Sponsorship funds may also be used to contract for specific improvements to GDAL, provision of resources such as web hosting, funding code sprints, or funding project promotion. Decisions on spending of sponsorship funds will be made by the GDAL/OGR Project Steering Committee.

12.2 Sponsorship Benefits

Sponsoring GDAL/OGR provides the following benefits:

1. Ensures the sustainability and health of the GDAL/OGR project.
2. All sponsors will be listed on the project `Credits` page, ordered by contribution class (Platinum, Gold, Silver) with a link back to the sponsor. Silver sponsors and above may include a logo. Platinum sponsors may also have a logo appearing on the OSGeo main page.

3. Sponsors will be permitted to indicate they are project sponsors in web and other promotional materials, and use the GDAL/OGR logo.
4. Sponsor input on project focus and direction will be solicited via a survey.
5. Sponsors will received a degree of priority in processing of bug reports by any maintainer hired with sponsorship funds.
6. Sponsors will receive a detailed report annually on the use of sponsorship funds.

12.3 Sponsorship Process

Sponsors can sponsor GDAL for any amount of money of at least $500 USD. At or above the following levels a sponsor will be designated as being one of the following class:

1. $27000+ USD: Platinum Sponsor
2. $9000+ USD: Gold Sponsor
3. $3000+ USD: Silver Sponsor

Sponsorships last one year, after which they may be continuing with a new payment, or allowed to lapse. OSGeo is planning to be US 501(c)3 charity and sponsorships will be eligible as a charitable contribution for US taxpayers. Appropriate receipts can be issued when needed.

Organizations or individuals interested in sponsoring the GDAL/OGR project should contact Frank Warmerdam (warmerdam@pobox.com, +1 650 701-7823) with questions, or to make arrangements.

Chapter 13

GDAL Warp API Tutorial

13.1 Overview

The GDAL Warp API (declared in gdalwarper.h) provides services for high performance image warping using application provided geometric transformation functions (GDALTransformerFunc), a variety of resampling kernels, and various masking options. Files much larger than can be held in memory can be warped.

This tutorial demonstrates how to implement an application using the Warp API. It assumes implementation in C++ as C and Python bindings are incomplete for the Warp API. It also assumes familiarity with the `GDAL Data Model`, and the general GDAL API.

Applications normally perform a warp by initializing a GDALWarpOptions structure with the options to be utilized, instantiating a GDALWarpOperation based on these options, and then invoking the GDALWarpOperation::ChunkAndWarpImage() method to perform the warp options internally using the GDALWarpKernel class.

13.2 A Simple Reprojection Case

First we will construct a relatively simple example for reprojecting an image, assuming an appropriate output file already exists, and with minimal error checking.

```
#include "gdalwarper.h"

int main()
{
    GDALDatasetH  hSrcDS, hDstDS;

    // Open input and output files.

    GDALAllRegister();

    hSrcDS = GDALOpen( "in.tif", GA_ReadOnly );
    hDstDS = GDALOpen( "out.tif", GA_Update );

    // Setup warp options.

    GDALWarpOptions *psWarpOptions = GDALCreateWarpOptions();

    psWarpOptions->hSrcDS = hSrcDS;
    psWarpOptions->hDstDS = hDstDS;

    psWarpOptions->nBandCount = 1;
    psWarpOptions->panSrcBands =
    (int *) CPLMalloc(sizeof(int) * psWarpOptions->nBandCount );
    psWarpOptions->panSrcBands[0] = 1;
    psWarpOptions->panDstBands =
    (int *) CPLMalloc(sizeof(int) * psWarpOptions->nBandCount );
    psWarpOptions->panDstBands[0] = 1;

    psWarpOptions->pfnProgress = GDALTermProgress;

    // Establish reprojection transformer.
    psWarpOptions->pTransformerArg =
        GDALCreateGenImgProjTransformer( hSrcDS,
                                         GDALGetProjectionRef(hSrcDS),
                                         hDstDS,
                                         GDALGetProjectionRef(hDstDS),
                                         FALSE, 0.0, 1 );
    psWarpOptions->pfnTransformer = GDALGenImgProjTransform;

    // Initialize and execute the warp operation.

    GDALWarpOperation oOperation;

    oOperation.Initialize( psWarpOptions );
    oOperation.ChunkAndWarpImage( 0, 0,
                          GDALGetRasterXSize( hDstDS ),
                      GDALGetRasterYSize( hDstDS ) );

    GDALDestroyGenImgProjTransformer( psWarpOptions->pTransformerArg );
    GDALDestroyWarpOptions( psWarpOptions );

    GDALClose( hDstDS );
    GDALClose( hSrcDS );

    return 0;
}
```

This example opens the existing input and output files (in.tif and out.tif). A GDALWarpOptions structure is allocated (GDALCreateWarpOptions() sets lots of sensible defaults for stuff, always use it for defaulting things), and the input and output file handles, and band lists are set. The panSrcBands and panDstBands lists are dynamically allocated here and will be free automatically by GDALDestroyWarpOptions(). The simple terminal output progress monitor (GDALTermProgress) is installed for reporting completion progress to the user.

GDALCreateGenImgProjTransformer() is used to initialize the reprojection transformation between the source and destination images. We assume that they already have reasonable bounds and coordinate systems set. Use of GCPs is disabled.

Once the options structure is ready, a GDALWarpOperation is instantiated using them, and the warp actually performed with GDALWarpOperation::ChunkAndWarpImage(). Then the transformer, warp options and datasets are cleaned up.

Normally error check would be needed after opening files, setting up the reprojection transformer (returns NULL on failure), and initializing the warp.

13.3 Other Warping Options

The GDALWarpOptions structures contains a number of items that can be set to control warping behavior. A few of particular interest are:

1. GDALWarpOptions::dfWarpMemoryLimit - Set the maximum amount of memory to be used by the GDAL-WarpOperation when selecting a size of image chunk to operate on. The value is in bytes, and the default is likely to be conservative (small). Increasing the chunk size can help substantially in some situations but care should be taken to ensure that this size, plus the GDAL cache size plus the working set of GDAL, your application and the operating system are less than the size of RAM or else excessive swapping is likely to interfere with performance. On a system with 256MB of RAM, a value of at least 64MB (roughly 64000000 bytes) is reasonable. Note that this value does **not** include the memory used by GDAL for low level block caching.
2. GDALWarpOpations::eResampleAlg - One of GRA_NearestNeighbour (the default, and fastest), GRA_-Bilinear (2x2 bilinear resampling) or GRA_Cubic. The GRA_NearestNeighbour type should generally be used for thematic or color mapped images. The other resampling types may give better results for thematic images, especially when substantially changing resolution.
3. GDALWarpOptions::padfSrcNoDataReal - This array (one entry per band being processed) may be setup with a "nodata" value for each band if you wish to avoid having pixels of some background value copied to the destination image.
4. GDALWarpOptions::papszWarpOptions - This is a string list of NAME=VALUE options passed to the warper. See the GDALWarpOptions::papszWarpOptions docs for all options. Supported values include:

- INIT_DEST=[value] or INIT_DEST=NO_DATA: This option forces the destination image to be initialized to the indicated value (for all bands) or indicates that it should be initialized to the NO_DATA value in padfDstNoDataReal/padfDstNoDataImag. If this value isn't set the destination image will be read and the source warp overlaid on it.
- WRITE_FLUSH=YES/NO: This option forces a flush to disk of data after each chunk is processed. In some cases this helps ensure a serial writing of the output data otherwise a block of data may be written to disk each time a block of data is read for the input buffer resulting in a lot of extra seeking around the disk, and reduced IO throughput. The default at this time is NO.

13.4 Creating the Output File

In the previous case an appropriate output file was already assumed to exist. Now we will go through a case where a new file with appropriate bounds in a new coordinate system is created. This operation doesn't relate specifically to the warp API. It is just using the transformation API.

```
#include "gdalwarper.h"
#include "ogr_spatialref.h"

...

    GDALDriverH hDriver;
    GDALDataType eDT;
    GDALDatasetH hDstDS;
    GDALDatasetH hSrcDS;

    // Open the source file.

    hSrcDS = GDALOpen( "in.tif", GA_ReadOnly );
    CPLAssert( hSrcDS != NULL );

    // Create output with same datatype as first input band.

    eDT = GDALGetRasterDataType(GDALGetRasterBand(hSrcDS,1));

    // Get output driver (GeoTIFF format)

    hDriver = GDALGetDriverByName( "GTiff" );
    CPLAssert( hDriver != NULL );

    // Get Source coordinate system.

    const char *pszSrcWKT, *pszDstWKT = NULL;

    pszSrcWKT = GDALGetProjectionRef( hSrcDS );
    CPLAssert( pszSrcWKT != NULL && strlen(pszSrcWKT) > 0 );

    // Setup output coordinate system that is UTM 11 WGS84.

    OGRSpatialReference oSRS;

    oSRS.SetUTM( 11, TRUE );
    oSRS.SetWellKnownGeogCS( "WGS84" );

    oSRS.exportToWkt( &pszDstWKT );

    // Create a transformer that maps from source pixel/line coordinates
    // to destination georeferenced coordinates (not destination
    // pixel line).  We do that by omitting the destination dataset
    // handle (setting it to NULL).

    void *hTransformArg;

    hTransformArg =
        GDALCreateGenImgProjTransformer( hSrcDS, pszSrcWKT, NULL, pszDstWKT,
                                         FALSE, 0, 1 );
    CPLAssert( hTransformArg != NULL );

    // Get approximate output georeferenced bounds and resolution for file.

    double adfDstGeoTransform[6];
    int nPixels=0, nLines=0;
    CPLErr eErr;

    eErr = GDALSuggestedWarpOutput( hSrcDS,
                    GDALGenImgProjTransform, hTransformArg,
                                    adfDstGeoTransform, &nPixels, &nLines );
    CPLAssert( eErr == CE_None );
    GDALDestroyGenImgProjTransformer( hTransformArg );

    // Create the output file.

    hDstDS = GDALCreate( hDriver, "out.tif", nPixels, nLines,
                         GDALGetRasterCount(hSrcDS), eDT, NULL );

    CPLAssert( hDstDS != NULL );

    // Write out the projection definition.

    GDALSetProjection( hDstDS, pszDstWKT );
    GDALSetGeoTransform( hDstDS, adfDstGeoTransform );

    // Copy the color table, if required.

    GDALColorTableH hCT;

    hCT = GDALGetRasterColorTable( GDALGetRasterBand(hSrcDS,1) );
    if( hCT != NULL )
        GDALSetRasterColorTable( GDALGetRasterBand(hDstDS,1), hCT );

    ... proceed with warp as before ...
```

Some notes on this logic:

- We need to create the transformer to output coordinates such that the output of the transformer is georeferenced, not pixel line coordinates since we use the transformer to map pixels around the source image into destination georeferenced coordinates.
- The GDALSuggestedWarpOutput() function will return an adfDstGeoTransform, nPixels and nLines that describes an output image size and georeferenced extents that should hold all pixels from the source image. The resolution is intended to be comparable to the source, but the output pixels are always square regardless of the shape of input pixels.
- The warper requires an output file in a format that can be "randomly" written to. This generally limits things to uncompressed formats that have an implementation of the Create() method (as opposed to CreateCopy()). To warp to compressed formats, or CreateCopy() style formats it is necessary to produce a full temporary copy of the image in a better behaved format, and then CreateCopy() it to the desired final format.
- The Warp API copies only pixels. All color maps, georeferencing and other metadata must be copied to the destination by the application.

13.5 Performance Optimization

There are a number of things that can be done to optimize the performance of the warp API.

1. Increase the amount of memory available for the Warp API chunking so that larger chunks can be operated on at a time. This is the GDALWarpOptions::dfWarpMemoryLimit parameter. In theory the larger the chunk size operated on the more efficient the I/O strategy, and the more efficient the approximated transformation will be. However, the sum of the warp memory and the GDAL cache should be less than RAM size, likely around 2/3 of RAM size.
2. Increase the amount of memory for GDAL caching. This is especially important when working with very large input and output images that are scanline oriented. If all the input or output scanlines have to be re-read for each chunk they intersect performance may degrade greatly. Use GDALSetCacheMax() to control the amount of memory available for caching within GDAL.
3. Use an approximated transformation instead of exact reprojection for each pixel to be transformed. This code illustrates how an approximated transformation could be created based on a reprojection transformation, but with a given error threshold (dfErrorThreshold in output pixels).

```
hTransformArg =
    GDALCreateApproxTransformer( GDALGenImgProjTransform,
                                 hGenImgProjArg, dfErrorThreshold );
pfnTransformer = GDALApproxTransform;
```

4. When writing to a blank output file, use the INIT_DEST option in the GDALWarpOptions::papszWarpOptions to cause the output chunks to be initialized to a fixed value, instead of being read from the output. This can substantially reduce unnecessary IO work.

5. Use tiled input and output formats. Tiled formats allow a given chunk of source and destination imagery to be accessed without having to touch a great deal of extra image data. Large scanline oriented files can result in a great deal of wasted extra IO.

6. Process all bands in one call. This ensures the transformation calculations don't have to be performed for each band.

7. Use the GDALWarpOperation::ChunkAndWarpMulti() method instead of GDALWarpOperation::ChunkAndWarpImage(). It uses a separate thread for the IO and the actual image warp operation allowing more effective use of CPU and IO bandwidth. For this to work GDAL needs to have been built with multi-threading support (default on Win32, default on Unix since GDAL 1.8.0, for previous versions –with-threads was required in configure).

8. The resampling kernels vary is work required from nearest neighbour being least, then bilinear then cubic. Don't use a more complex resampling kernel than needed.

9. Avoid use of esoteric masking options so that special simplified logic case be used for common special cases. For instance, nearest neighbour resampling with no masking on 8bit data is highly optimized compared to the general case.

13.6 Other Masking Options

The GDALWarpOptions include a bunch of esoteric masking capabilities, for validity masks, and density masks on input and output. Some of these are not yet implemented and others are implemented but poorly tested. Other than per-band validity masks it is advised that these features be used with caution at this time.