# Tasks performed in porting GVRS Java API to C

Updated 31 December 2024

gwlucas07@gmail.com
[Pick the date]

## Introduction

These notes are intended to support developers who are considering the process of porting the GVRS API from either Java or C to an alternate programming language.

Recently, the Gridfour team ported the original Java API to the C programming language. These notes provide a brief description of the steps that were followed to port the API from its original Java implementation to C. Our hope is that this information will help developers plan their own porting efforts. By illustrating how one porting effort broke out the functional elements of the API, these notes should also help developers identify the architectural elements needed for a successful port.

The sequence of tasks that were used for the port to C reflects the habits and requirements of the development team. It is not necessarily the only, or even best, way to proceed. The description below makes notes of alternate approaches. In general, the development team preferred to operate in short sprints, with testing and design reviews at the end of each.

The bulk of the coding and API design was conducted by a single volunteer developer working part time over a period of months. So the implementation sequence described below is essentially serial in nature. Time estimates are based on notes kept during the development effort.

In the beginning, we underestimated how much different the C API would be from the Java API. The primary differences were due to object-oriented versus non-object-oriented implementations and, to a lesser degree, the use of pointers and explicit memory management. For developers working in other languages, the availability of the two existing projects should provide insights suitable to most modern programming environments.

## General approach

The GVRS API was ported to C in four phases:

1. Implement the ability to read GVRS files stored without compression.
2. Implement the ability to read GVRS files stored with compression.
3. Implement the ability to write GVRS files without compression.
4. Implement the ability to write GVRS files with compression.

It is possible to follow alternate sequences. For example, data compression could be deferred until after both the read and write operations are complete. Or, if a developer were primarily concerned with the use of GVRS as a tool for maintaining a virtual raster without persistent storage of data, then it would be feasible to implement write operations first with only a partial set of read operations and no consideration of data compression.

## Resources for the developer

The GVRS FAQ answers frequently asked questions about the GVRS API at https://github.com/gwlucastrig/gridfour/wiki/A-GVRS-FAQ.

A number of web articles describing the concepts and algorithms used in the GVRS API is available at the “Gridfour Project Notes” web page at https://gwlucastrig.github.io/GridfourDocs/notes/index.html.

The GVRS file format is specified in the document “The Gridfour Virtual Raster Store (GVRS) File Format” available at https://gwlucastrig.github.io/GridfourDocs/notes/GvrsFileFormat_1_04.pdf

One of the key components of the GVRS API, the tile-cache, is described in “Managing a Virtual Raster using a Tile-Cache Algorithm” at https://gwlucastrig.github.io/GridfourDocs/notes/VirtualRaster.html

The Java software distribution for the Gridfour project includes a number of test files under the path “core/src/test/resources/org/gridfour/gvrs/SampleFiles”. A README.TXT file describes the content of individual files in that directory.

The wiki pages for both the Java and C code distributions include “how to” articles that are intended to illustrate the use of the API in applications and programs.

A small set of test programs for exercising GVRS functions are provided in the standard distribution in the folder GvrsC/test.

## Phase 1 Implement the ability to read GVRS files

GVRS files consist of a sequence of data primitives (integers, strings, floating-point values, etc.) that are given in little-endian order as described in the GVRS File Format document cited above. The first step in porting GVRS to a C API was to implement functions to read these primitives. To support this effort, we introduced a number of test files to the Java software distribution. A file called "SampleDataPrimitives.dat" provides a sequence of each of the various types used by GVRS. The layout of the file is described in the README.TXT file included in the distribution.

Once we were able to read the data primitives successfully, we focused on reading the content of a GVRS file. We were also concerned about ensuring that we could read files in a random-access order (a key requirement in implementing the GVRS API). Each variable-length record in a GVRS file contains a short header giving metadata about the record including record-type and size in bytes. We wrote a test program to confirm that we could read the records from a file (ignoring their content) and tabulate a count for each record type found in the file.

Next, we implemented the ability to read the file header and overall structural data for a GVRS file. In order to confirm that we were reading the data from the header correctly, we implemented three main functions (with supporting modules):

1. GvrsOpen – Opens a GVRS file, reads the header (creation dates, grid size, variable types, etc.) and any internal directories. In C, GvrsOpen returns a pointer to a structure of type "Gvrs" which is the main access point a GVRS file.
2. GvrsSummarize – Prints information from the Gvrs structure to standard output.
3. GvrsClose – closes a GVRS file and frees all memory associated with the "Gvrs" structure.

In the GVRS API, print statements are ***never*** included in the code, except in functions that include the string "summarize" in their name. By writing an "open" and a "summarize" function, we could confirm that we were accessing the header elements correctly.

Finally, we implemented logic to access the content of a GVRS file. This effort involved a number of key components. The C programming language does not provide a way of specifying scope for functions and elements. Even so, some software components are treated as "public" and some are treated as "internal".

- Gvrs – the main structure used to hold resources such as file pointers, allocated memory, and data elements needed for file access. Essentially, this structure is public in scope.
- GvrsElement – public structures and functions that allow application code to access data elements from a GVRS raster (integer or floating point values at a particular grid cell location).
- Tile-Directory – internal structures and functions that let the API find the location of a particular tile (subsection of the overall grid) within the body of the data file.

- Tile-Cache – internal structures and functions that maintain tiles in memory for rapid access by application code.
- GvrsMetadata – public structures and elements that allow application code to access descriptive information from a GVRS file. The ability to access metadata was not essential to reading the main data from a GVRS file, so we deferred working on metadata until late in the implementation process.
- Metadata-Directory – internal structures that let the API find the location of a particular metadata element in the file.  The ability to access metadata was not essential to reading the main data from a GVRS file, so we deferred working on metadata until late in the implementation process.

## Phase 1 implement basic read operations

The following table lists the tasks that were performed when porting the Java API to C. The hour column gives estimates of the time a single developer spent writing code.

| Task | Hours | Description |
|---|---|---|
| Read Data Primitives | 16 | Functions (.c files) and header (.h file) to read GVRS data primitives from the SampleDataPrimitives.dat file. Implement stand-alone test program to verify. A reference C implementation of the test program is provided by Test000_ReadPrimitives.c |
| Read Variable-Length Records | 8 | Stand-alone program to traverse a GVRS file, read the record headers to determine record type and length, compute the position of the next record, and advance through file. Verified ability to read data primitives and navigate file successfully in random-access sequence. |
| Read Header (preliminary) | 16 | Define structure named "Gvrs" containing the most important elements of header. Implement test program to print content of header (involved GvrsOpen, GvrsSummarize, GvrsClose functions). |
| Extend header-reading to read tile directory | 16 | Define structure for tile-directory. Read file directory from file and include a reference to it in the "Gvrs" structure. Also include logic to clean up resources in GvrsClose. |
| Read content of grid (preliminary) | 24 | Define structure for GvrsElement, implement functions to read integer-type data from one of the sample files. Because no tile cache is implemented at this stage, access is very slow. The important thing is to confirm correct data access. |
| Implement tile-cache for reading | 24 | Implement a tile cache. As an enhancement, included a hash table for mapping tile index to cache entries Ito expedite cache searches). GVRS C implements its own hash table. Other environments might include suitable hash implementations as part of their standard libraries (a C++ implement could have used the standard template library, etc.). |
| Performance and refinement | 24 | Exercise code, tabulate initial performance statistics, refine computations, code, etc. Verify resource disposal and memory "free" operations. |
| Implement checksums (optional) | 8 | Each record in a GVRS file carries a checksum that allows the detection of file errors. Since transmission errors are rare in modern hardware, we treated this step as optional. A lot of the code was cut-and-pasted from the original Java source and then modified for the C implementation. |

### Phase 1a implement metadata

Code for access metadata was a secondary concern for the C port, so it was deferred until later in the process. Also, performance was a much lower concern for metadata, so this implementation generally used simple data structures, sequential search logic, and “brute force” coding.

| Task | Hours | Description |
|---|---|---|
| Read metadata directory | 4 | Read metadata directory as part of GvrsOpen |
| Read specific metadata records | 4 | Check metadata directory to see if there is a match for a specific metadata ID (name and integer number), find file position and read content. |
| API to provide metadata content to calling modules | 16 | Provide metadata to calling modules. Since metadata has several data types (short, integer, float, double, string), this step involved a lot of extra functions. |

## Phase 2 read compressed data

Samples of the ETOPO1 global-scale elevation data file are attached to the current release of the Java API at https://github.com/gwlucastrig/gridfour/releases. These files provide good test subjects for developing data compression

| Task | Hours | Description |
|---|---|---|
| Bit-access functions | 12 | Implement bit-access functions for reading bits and bytes from a compressed data stream |
| M-32 codes | 8 | Implement functions to read GVRS M-32 codes from byte streams. |
| Integer predictors | 4-to-8 | Implement predictors to be used in decoding integer data types from source data (differencing predictor, linear predictor, and triangle predictor). To a large degree, this was a code cut-and-paste operation. |
| Basic access for compressed data | 8 | Implement logic to detect when data is compressed, invoke the appropriate decompressor, and extract data. |
| Huffman coding for integer data | 20 | Implement logic to decompress Huffman-coded data streams. This task required extra time due to performance issues. Ideally, future developers will be able to leverage the lessons-learned from existing code to expedite their efforts. |
| Deflate for integer data | 4-to-24 | Integrate the 3rd party Deflate software library into the GVRS API for decompressing data stored in Deflate format. The level of effort depends on the availability of Deflate on the local software build environment. If Deflate is available, this should be straightforward. |
| Floating-point compression | 16 | Implement ability to decompress floating-point data. The floating-point compression logic leverages from the work implemented for integer compression, so this tasks requires the integer implementation. |

## Phase 1 and 2 completion: documentation, user support

At the end of phase 1 and continuing into phase 2, we undertook an effort to extend the documentation (using doxygen) and provide user support via the Gridfour wiki pages and development notes cited above.  We also undertook additional testing and example code implementations. This is an on-going process and we do not have level-of-effort estimates available. Developers who wish to port the Gridfour API should estimate time to write documentation based on their own standards and requirements.

## Phase 3 implement basic write operations

One of the more challenging parts of the GVRS API is the file-space management logic required to allow GVRS to allocate, use, and re-use segments of the data file as the size of different data objects change. Most of this functionality could be stubbed out or partially implemented in the early phases. A full implementation was not required until data compression and metadata writing were introduced.

| Task | Hours | Description |
|---|---|---|
| Write data primitives | 8 | Functions (.c files) and header (.h file) to write GVRS data primitives. Implement stand-alone test program to verify. |
| Start Gvrs Builder | 4 | Create initial structure and functions for creating new Gvrs files. Implement shell functions for GvrsBuilderCreate and GvrsBuilderFree. Define grid and tile sizes. Populate opened-for-writing element in output GVRS file. |
| Write initial GVRS file Header | 4 | GvrsBuilder to write a GVRS file header and return a Gvrs structure. Extend GvrsClose to stub final write operations when closing a file, zero-out opened-for-writing element. |
| Element specifications | 16 | Add functions for GvrsBuilder to add element specifications when creating a new GVRS file. |
| Write data values to raster | 24 | Using row and column for grid cell, write data to appropriate tile. Extend tile cache logic to initialize new tiles or fetch existing tiles. Extend |
| Implement checksums (optional) | 8 | Add code to GvrsClose to process records and compute checksums. |

## Phase 4 write compressed data

| Task | Hours | Description |
|---|---|---|
| Bit-access functions | 8 | Extend the bit-access functions to store bit and byte sequences |
| M-32 codes | 4 | Extend the GVRS M-32 functions to store codes to byte streams. |
| Integer predictors | 4-to-8 | Implement predictors to be used in encoding integer data types from source data (differencing predictor, linear predictor, and triangle predictor). To a large degree, this was a code cut-and-paste operation. |
| Basic access for compressed data | 8 | Modify tile-writing operations to compress data before storage |
| File-space management code | 16-32 | Implement file-space management logic. The C logic was sufficiently different from the original Java that this was essentially new code. However, new porting efforts should be able to leverage this work to reduce development time. |
| Huffman coding for integer data | 12 | Implement logic to compress data using Huffman-coding data streams. Much of the work should be a straight-forward port of the C code. |
| Deflate for integer data | 4 | Integrate the 3rd party Deflate software library into the GVRS API for compressing data stored in Deflate format. |

| | | |
|---|---|---|
| Floating-point compression | 8 | Implement ability to compress floating-point data. The floating-point compression logic leverages from the work implemented for integer compression, so this tasks requires the integer implementation. |