## Contents

1. Preamble 3
2. The i/o abstraction layer 5
3. Our audience 7
4. Definitions 9
5. GMT resources 11
   5.1 Data tables 11
   5.2 GMT grids 12
   5.3 GMT images 13
   5.4 Color palette tables (CPT) 13
   5.5 PostScript document 14
   5.6 Text tables 14
   5.7 User data matrices 15
   5.8 User data columns 15
6. Overview of the GMT C Application Program Interface 17
   6.1 Example code 18
   6.2 Compilation 18
   6.3 Plugins 19
   6.4 List of API functions 19
7. The GMT C Application Program Interface 21
   7.1 Initialize a new GMT session 21
   7.2 Register input or output resources 22
   7.3 Create empty resources 25
   7.4 Duplicate resources 27
   7.5 Convert between resource types 28
   7.6 Import Data Sets 28
   7.7 Record-by-record input 31
   7.8 Manipulate data 34
   7.9 Message and Verbose Reporting 36
   7.10 Special GMT modules 37
   7.11 Adjusting headers and comments 37
   7.12 Export Data Sets 38
The Generic Mapping Tools
C/C++ Application Programming Interface
Pål (Paul) Wessel
SOEST, University of Hawai’i at Manoa
Walter H. F. Smith
Laboratory for Satellite Altimetry, NOAA/NESDIS/STAR
Remko Scharroo
EUMETSAT, Darmstadt, Germany
Joaquim F. Luis
Universidade do Algarve, Faro, Portugal
Florian Wobbe
Alfred Wegener Institute, Germany
Prior to version 5, the bulk of GMT functionality was coded directly in the standard GMT C program modules (e.g., surface.c, psxy.c, grdimage.c, etc.). The GMT library only offered access to low-level functions from which those high-level GMT programs were built. The standard GMT programs have been very successful, with tens of thousands of users worldwide. However, the design of the main programs prevented developers from leveraging GMT functionality from within other programming environments since access to GMT tools could only be achieved via system calls\(^1\). Consequently, all data i/o had to be done via temporary files. The design also prevented the GMT developers themselves from taking advantage of these modules directly. For instance, the tool pslegend needed to make extensive use of system calls to psxy and pstext in order to plot the lines, symbols, and text that make up a map legend, making it a very awkward program to maintain.

Starting with GMT version 5, all standard GMT programs have been rewritten into separate function

---

\(^1\) or via a very confusing and ever-changing myriad of low-level library functions for bold programmers.
“modules” invoked by a single driver program called gmt.c. The gmt executable simply calls the corresponding GMT modules; it is these modules that do all the work. These new functions have been placed in a new GMT high-level API library and can be called from a variety of environments (C/C++, Fortran, Julia, Python, MATLAB, Visual Basic, R, etc.).\(^2\) For example, the main program blockmean.c has been reconfigured as a high-level function GMT_blockmean(), which does the actual spatial averaging and can pass the result back to the calling program (or write it to file). The previous behavior of blockmean.c is achieved by calling gmt blockmean, i.e., the module is now just the first argument to the gmt executable. For backwards compatibility with older GMT (4) scripts we optionally install numerous symbolic links to the gmt executable with names such as blockmean, psxy, surface, etc. The gmt executable is smart enough to understand when it is being invoked via one of these links and then knows which module to call upon. Consequently, blockmean.c and other files do in fact no longer exist.

\(^2\) Currently, C/C++, FORTRAN, MATLAB and Julia are being tested.
In order for the API to be as flexible as possible we have generalized the notions of input and output. Data that already reside in an application’s memory may serve as input to a GMT module and we refer to such data as “Virtual Files”. Other sources of input may be file pointers and file descriptors (as well as the standard mechanism for passing file names). For standard data table i/o, the GMT API takes care of the task of assembling any combination of files, pointers, and memory locations into a single virtual data set from which the GMT module may read (a) all records at once into memory, or (b) read one record at a time. Likewise, GMT functions may write their output to a virtual destination, which might be a memory location in the user’s application (another Virtual File), a file pointer or descriptor, or an output file. The GMT modules are unaware of these details and simply read from a “source” and write to a “destination”. Thus, the standard concept of file-based input/output so familiar to any GMT user carries over to the API, except for the generalization that files can be virtual files already in memory. Because of this design we will see that we need to associate these virtual files with special filenames that we may pass to modules, and the modules will faithfully treat these as real files. However, under the hood the API layer will take care of the differences between real and virtual files.

Users who wish to maintain their own data types and memory management can also use the GMT modules, but some limitations and requirements do apply: The user’s data can either be provided as (1) a 2-D matrix (of any data type, e.g., float, integer, etc.) and in any memory layout configuration (e.g., row-major or column-major layout) or as (2) a set of column vectors that each may be of any type. These custom arrays will need to be hooked onto the GMT containers \texttt{GMT\_MATRIX} and \texttt{GMT\_VECTOR}, respectively. Such objects can then be treated as virtual files for either input of output.
Here, we document the new functions in the GMT API library for application developers who wish to call these functions from their own custom programs. At this point, only the new high-level GMT API is fully documented and intended for public use. The structure and documentation of the under-lying lower-level GMT library is not finalized. Developers using these functions may risk disruption to their programs due to changes we may make in the library in support of the GMT API. However, developers who wish to make supplemental packages to be distributed as part of GMT will (other than talk to us) probably want to access the entire low-level GMT library as well. It is unlikely that the low-level library will ever be fully documented.

There are two classes of development that users can pursue:

1. Building stand-alone custom executables that link with the shared GMT API. Our examples in this documentation are of this kind. There programs are likely to address a user’s special data formats or processing needs by leveraging high-level GMT modules to do some of the heavy lifting.

2. Building shared library plugins to extend the breadth of GMT. Users who wish to build one or more new modules and distributed then via a plugin that is dynamically loaded at run-time can now do so. At the present, all the modules in the official GMT supplement are compiled into a single plugin that can be accessed at run-time. Similarly, developers may add additional plugin libraries with any number of GMT-like modules and these will then be available from the gmt command (as well as from derived interfaces such as the GMT/MATLAB toolbox and the Python module). An example of plugin development is given by the GSFML extension to GMT.
CHAPTER 4

Definitions

For the purpose of this documentation a few definitions are needed:

1. “Standard GMT program” refers to one of the traditional stand-alone command-line executables known to all GMT users, e.g., blockmean, psxy, grdimage, etc. Prior to version 5, these were the only GMT executables available. In GMT 5, these are accessed via the gmt executable.

2. “GMT module” refers to the function in the GMT API library that is responsible for all the action taken by the corresponding standard GMT program. All such modules are given the same names as the corresponding programs e.g., “blockmean”, but are invoked via the GMT_Call_Module function.

3. “GMT application” refers to a new application written by any developer. It uses the API, perhaps for custom i/o, and may call one or more GMT functions to create a new GMT-compatible executable.

4. “GMT plugin library” refers to a collection of one or more new custom GMT-like modules that are presented as a plugin library. It such libraries are placed in the official GMT plugin directory or their path is added to the GMT defaults parameter GMT_CUSTOM_LIBS then the gmt executable can find them.

5. “Family” refers to one of the many high-level GMT data types (e.g., grids, CPTs) and is typically a required argument to some API functions.

6. “Method” refers to one of several ways in which data can be read or written in the API, including from existing memory variables.

7. “Direction” is typically either GMT_IN (for reading) or GMT_OUT (for writing).

8. In the API description that follows we will use the type int to mean a 4-byte integer. All integers used in the API are 4-byte integers with the exception of one function where an 8-byte integer is used. Since different operating systems have their own way of defining 8-byte integers we use C99’s int64_t for this purpose; it is guaranteed to yield the correct type that the GMT function expects.

In version 5, the standard GMT programs are themselves simple invocations of the gmt application with the function name as argument. However, some of these modules, such as pslegend, gmtconvert,
grdblend, grdfilter and others may call several additional modules.
GMT resources

The GMT API knows how to create, duplicate, read and write six types of data objects common to GMT operations: Pure data tables (ASCII or binary), grids, images, color palette tables (also known as CPT), PostScript documents, and text tables (ASCII, usually a mix of data and free-form text). In addition, we provide two data objects to facilitate the passing of simple user arrays (one or more equal-length data columns of any data type, e.g., double, char) and 2-D or 3-D user matrices (of any data type and column/row organization). We refer to these data types as GMT resources. There are many attributes for each of these resources and therefore we use a top-level structure for each object to keep them all within one container. These containers are given or returned by GMT API functions using opaque pointers (void *). Below we provide a brief overview of these containers, listing only the most critical members. For complete details, see Appendix A. We will later present how they are used when importing or exporting them to or from files, memory locations, or streams. The first six are the standard GMT objects, while the latter two are special data containers to facilitate the passing of user data in and out of GMT modules. These resources are defined in the include file gmt_resources.h; please consult this file to ensure correctness in case the documentation is not up-to-date. Note than in all instances the fundamental data variable is called “data”.

5.1 Data tables

Much data processed in GMT come in the form of ASCII, netCDF, or native binary data tables. These may have any number of header records (ASCII files only) and perhaps segment headers that separate groups of points or lines and polygons. GMT programs will read one or more such tables when importing data. However, to avoid memory duplication or data limitations some programs may prefer to read such records one at the time. The GMT API has functions that let you read your data record-by-record by presenting a virtual data set that combines all the data tables specified as input. This simplifies record processing considerably. Programs reading an entire data set will encounter several structures: A data set (struct GMT_DATASET) may contain any number of tables (struct GMT_DATATABLE), each with any number of segments (struct GMT_DATASEGMENT), each segment with any number of records, and each record with any number of (fixed) columns. Thus, the arguments to GMT API

---

3 At the moment, GMT does not have native support for 3-D grids.
functions that handle such data sets expect a struct `GMT_DATASET`. All segments are expected to have the same number of columns.

```c
struct GMT_DATASET {
    /* Single container for an array of GMT tables (files) */
    uint64_t n_tables; /* The total number of tables contained */
    uint64_t n_columns; /* The number of data columns */
    uint64_t n_segments; /* The total number of segments across all tables */
    uint64_t n_records; /* The total number of data records across all tables */
    double *min; /* Minimum coordinate for each column */
    double *max; /* Maximum coordinate for each column */
    struct GMT_DATATABLE **table; /* Pointer to array of tables */
};
```

The top-level dataset structure for pure data tables contains the table structure, as defined below:

```c
struct GMT_DATATABLE {
    /* Single container for an array of data segments */
    unsigned int n_headers; /* Number of table header records (0 if no header) */
    uint64_t n_columns; /* Number of columns (fields) in each record */
    uint64_t n_segments; /* Number of segments in the array */
    uint64_t n_records; /* Total number of data records across all segments */
    double *min; /* Minimum coordinate for each column */
    double *max; /* Maximum coordinate for each column */
    char **header; /* Array with all table header records, if any */
    struct GMT_DATASEGMENT **segment; /* Pointer to array of segments */
};
```

Finally, the table structure depends on a structure for individual data segments:

```c
struct GMT_DATASEGMENT {
    /* For holding segment lines in memory */
    uint64_t n_rows; /* Number of points in this segment */
    uint64_t n_columns; /* Number of fields in each record (>= 2) */
    double *min; /* Minimum coordinate for each column */
    double *max; /* Maximum coordinate for each column */
    double **data; /* Data x,y, and possibly other columns */
    char *label; /* Label string (if applicable) */
    char *header; /* Segment header (if applicable) */
};
```

Data sets may have different geometries, such as representing a set of points, one or more lines, or closed polygons.

### 5.2 GMT grids

GMT grids are used to represent equidistant and organized 2-D surfaces. These can be processed or plotted as contour maps, color images, or perspective surfaces. Because the native GMT grid is simply a 1-D float array with metadata kept in a separate `struct GMT_GRID_HEADER` header, we pass this information via a `struct GMT_GRID`, which is a container that holds both items. Thus, the arguments to GMT API functions that handle GMT grids expect this type of variable.

```c
struct GMT_GRID {
    /* A GMT float grid and header in one container */
    struct GMT_GRID_HEADER *header; /* The full GMT header for the grid */
    float *data; /* Pointer to the float grid array */
};
```

The top-level grid structure, holding both header and data array, depends on the grid header header structure:

```c
struct GMT_GRID_HEADER {
    uint32_t n_columns; /* Number of columns */
    uint32_t n_rows; /* Number of rows */
    uint32_t registration; /* GMT_GRID_NODE_REG (0) for node grids,
                             GMT_GRID_PIXEL_REG (1) for pixel grids */
    double w[4]; /* Min/max x and y coordinates */
};
```

(continues on next page)
5.3 GMT images

 GMT images are used to represent bit-mapped images typically obtained via the GDAL bridge. These can be reprojected internally, such as when used in grdimage. Since images and grids share the concept of a header, we use the same header structure for grids as for images; however, some additional metadata attributes are also needed. Finally, the image itself may be of any data type and have more than one band (channel). Both image and header information are passed via a struct GMT_IMAGE, which is a container that holds both items. Thus, the arguments to GMT API functions that handle GMT images expect this type of variable. Unlike the other objects, writing images has only partial support via grdimage. For the full definition, see GMT_IMAGE.

```
struct GMT_IMAGE {
    enum GMT_enum_type type; /* Data type, e.g. GMT_FLOAT */
    int *colormap; /* Array with color lookup values */
    int n_indexed_colors; /* Number of colors in a color-mapped image */
    struct GMT_GRID_HEADER *header; /* Pointer to full GMT header for the image */
    unsigned char *data; /* Pointer to actual image */
};
```

5.4 Color palette tables (CPT)

The color palette table files, or just CPTs, contain colors and patterns used for plotting data such as surfaces (i.e., GMT grids) or symbols, lines and polygons (i.e., GMT tables). GMT programs will generally read in a color palette table, make it the current palette, do the plotting, and destroy the table when done. The information is accessed via a pointer to struct GMT_PALETTE. Thus, the arguments to GMT API functions that handle palettes expect this type of variable. It is not expected that users will wish to manipulate the CPT directly, but rather use this mechanism to hold them in memory and pass as arguments to GMT modules. Developers are unlikely to actually manipulate the contents of CPT structures but if needed then the full definition can be found in GMT_PALETTE.

```
struct GMT_PALETTE {
    /* Holds color-related parameters for look-up */
    unsigned int n_headers; /* Number of CPT header records (0 if no header) */
    unsigned int n_colors; /* Number of colors in the data array */
    unsigned int mode; /* Flags controlling use of BFN colors */
    struct GMT_LUT *data; /* CPT lookup data with color information */
    struct GMT_BFN bfn[3]; /* Structures with back/fore/nan fills */
    char **header; /* Array with all CPT header records, if any */
};
```

---

4. This may change in later releases.
## 5.5 PostScript document

Normally, GMT modules producing PostScript will write to standard output or a designated file. Alternatively, you can tell the API to write to a memory buffer instead and then receive a structure with the final plot (or partial plot) represented as a long text string. The full structure definition can be found in `GMT_POSTSCRIPT`.

```c
struct GMT_POSTSCRIPT {    /* Single container for a chunk of PostScript text */
    unsigned int n_headers;    /* Number of PostScript header records (0 if no header) */
    size_t n_alloc;            /* Size of array allocated so far */
    size_t n_bytes;           /* Length of data array so far */
    unsigned int mode;        /* Bit-flag for header (1) and trailer (2) */
    char *data;               /* Pointer to actual PostScript text */
    char **header;            /* Array with all PostScript header records, if any */
};
```

## 5.6 Text tables

Some data needed by GMT are simply free-form ASCII text tables. In many respects these are handled similarly to data tables. E.g., they may have any number of header records and even segment headers, and GMT programs can read one or more tables or get text records one at a time. A `struct GMT_TEXTSET` may contain any number of `struct GMT_TEXTTABLE`, each with any number of `struct GMT_TEXTSEGMENT`, and each segment with any number of text records. Thus, the arguments to GMT API functions that handle such data sets expect this type of variable. The user’s program may then parse and process such text records as required. This resource is particularly useful when your data consist of a mix of data coordinates and ordinary text since regular data tables will be parsed for floating-point columns only. For the full definition, see `GMT_TEXTSET`.

```c
struct GMT_TEXTSET {    /* Single container for an array of GMT text tables (files) */
    uint64_t n_tables;    /* The total number of tables in the set */
    uint64_t n_segments;  /* The total number of segments in the set */
    uint64_t n_records;   /* The total number of data records in the set */
    struct GMT_TEXTTABLE **table;    /* Pointer to array of tables */
};
```

The top-level structure for mixed data/text data sets depends on the text table structure:

```c
struct GMT_TEXTTABLE {    /* Single container for an array of text segments */
    unsigned int n_headers;  /* Number of table header records (0 if no header) */
    uint64_t n_segments;     /* Number of segments in the table */
    uint64_t n_records;      /* Total number of data records in the table */
    char **header;           /* Array with all table header records, if any */
    struct GMT_TEXTSEGMENT **segment;    /* Pointer to array of segments */
};
```

Finally, the text table structure depends on the structure for text segments:

```c
struct GMT_TEXTSEGMENT {    /* For holding segment text records in memory */
    uint64_t n_rows;        /* Number of rows in this segment */
    char **data;            /* Array of text records */
    char *label;            /* Label string (if applicable) */
    char *header;           /* Segment header (if applicable) */
};
```
5.7 User data matrices

Users may write programs that need to call GMT modules but may keep their data in separate 2-D arrays that the allocate and maintain independent of GMT. For instance, a program may have built an integer 2-D matrix in memory and wish to use that as the input grid to the grdfilter module, which normally expects a struct GMT_GRID with floating point data via an actual or virtual file. To handle this case we create a struct GMT_MATRIX container (see Create empty resources), assign the appropriate union pointer to your data matrix and provide information on dimensions and data type. We then open this container as a virtual file and pass its filename to any module. The full structure definition can be found in GMT_MATRIX.

```c
struct GMT_MATRIX { /* Single container to hold a user matrix */
    uint64_t n_rows;    /* Number of rows in the matrix */
    uint64_t n_columns; /* Number of columns in the matrix */
    enum GMT_enum_fmt shape; /* 0 = C (rows) and 1 = Fortran (cols) */
    enum GMT_enum_reg registration; /* 0 for gridline and 1 for pixel registration */
    size_t dim;             /* Allocated length of longest C or Fortran dim */
    size_t size;            /* Byte length of data */
    enum GMT_enum_type* type; /* Array with data type for each vector */
    union GMT_UNIVECTOR* data; /* Array with unions for each column */
    double range[6];       /* Contains xmin/xmax/ymin/ymax[/zmin/zmax] */
};
```

The enum types referenced in GMT_VECTOR and Table GMT_MATRIX and summarized in Table types.

5.8 User data columns

Likewise, programs may instead be manipulating a set of custom column vectors. For instance, the user’s program may have allocated and populated three column arrays of type float and wishes to use these as the input source to the surface module, which normally expects double precision triplets via a struct GMT_DATASET read from an actual or virtual file Simply create a new GMT_VECTOR container (see section Create empty resources) and assign the union array pointers (see univector) to your data columns and provide the required information on length, data types, and optionally range. Again, once we open this data as a virtual file we can pass its filename to any module expecting such data. The full structure definition can be found in GMT_VECTOR.

```c
struct GMT_VECTOR { /* Single container to hold user vectors */
    uint64_t n_columns;  /* Number of vectors */
    enum GMT_enum_reg registration; /* 0 for gridline and 1 for pixel registration */
    enum GMT_enum_type* type; /* Array with data type for each vector */
    union GMT_UNIVECTOR* data; /* Array with unions for each column */
    double range[2]; /* The min and max limits on t-range (or 0,0) */
};
```
Overview of the GMT C Application Program Interface

Users who wish to create their own GMT application based on the API must make sure their program goes through the steps below. The details for each step will be revealed in the following chapter. We have kept the API simple: In addition to the GMT modules, there are only 57 public functions to become familiar with, but most applications will only use a very small subset of this selection. Functions either return an integer error code (when things go wrong; otherwise it is set to \texttt{GMT\_NOERROR (0)}) , or they return a void pointer to a GMT resource (or NULL if things go wrong). In either case, the API will report what the error is. The layout here assumes you wish to use virtual files as input sources (i.e., data you already have in memory); if the data must be read from actual data files then things simplify considerably.

To keep things as simple as possible we will assume you are writing an application that will read in table data, call a module using the data in memory as input, and then save the output from the module back into another memory location. No actual processing of the data or further calculation will be done here (so a bit of a boring program, but the point is to develop something short we can test). Also, to keep the code short we completely ignore the return codes of the modules for now. We will call our program \texttt{example1.c}. Here are the steps:

1. Initialize a new GMT session with \texttt{GMT\_Create\_Session}, which allocates a hidden GMT API control structure and returns an opaque pointer to it. This pointer is a required argument to all subsequent GMT API function calls within the session.

2. Read a data set (or grid, etc.) into memory with \texttt{GMT\_Read\_Data}, which, depending on data type, returns one of the data structures discussed earlier.

3. Associate your data with a virtual file using \texttt{GMT\_Open\_VirtualFile}. This steps returns a special filename that you can use to tell a module where to read its input. No actual file is created.

4. Open a new virtual file to hold the output using \texttt{GMT\_Open\_VirtualFile}. This step also returns a special filename for the module to send its output.

5. Prepare required arguments (including the two virtual file names) and call the GMT module you wish to use via \texttt{GMT\_Call\_Module}.

6. Obtain the desired output object via \texttt{GMT\_Read\_VirtualFile}, which returns a data structure of requested type.
7. Close the virtual files you have been using with `GMT_Close_VirtualFile`.

8. We terminate the GMT session by calling `GMT_Destroy_Session`.

### 6.1 Example code

For the example code to run, make sure the data table “table_5.11” from gallery example 14 is placed in the current directory, then compile and run this program:

```c
#include "gmt.h"
int main (int argc, char *argv[]) {
    void *API;   /* The API control structure */
    struct GMT_DATASET *D = NULL;   /* Structure to hold input dataset */
    struct GMT_GRID *G = NULL;      /* Structure to hold output grid */
    char input[GMT_STR16] = "";      /* String to hold virtual input filename */
    char output[GMT_STR16] = "";     /* String to hold virtual output filename */
    char args[128] = "";             /* String to hold module command arguments */

    /* Initialize the GMT session */
    API = GMT_Create_Session (*"test", 2U, 0, NULL);
    /* Read in our data table to memory */
    D = GMT_Read_Data (API, GMT_IS_DATASET, GMT_IS_FILE, GMT_IS_PLP, GMT_READ_NORMAL, NULL, "table_5.11", NULL);
    /* Associate our data table with a virtual file */
    GMT_Open_VirtualFile (API, GMT_IS_DATASET, GMT_IS_PLP, GMT_IN, D, input);
    /* Create a virtual file to hold the resulting grid */
    GMT_Open_VirtualFile (API, GMT_IS_GRID, GMT_IS_SURFACE, GMT_OUT, NULL, output);
    /* Prepare the module arguments */
    sprintf (args, "-R0/7/0/7 -I0.2 -D1 -St0.3 %s -G%s", input, output);
    /* Call the greenspline module */
    GMT_Call_Module (API, "greenspline", GMT_MODULE_CMD, args);
    /* Obtain the grid from the virtual file */
    G = GMT_Read_VirtualFile (API, output);
    /* Close the virtual files */
    GMT_Close_VirtualFile (API, 0, input);
    GMT_Close_VirtualFile (API, 0, output);
    /* Write the grid to file */
    GMT_Write_Data (API, GMT_IS_GRID, GMT_IS_FILE, GMT_IS_SURFACE, GMT_READ_NORMAL, NULL, "junk.nc", G);
    /* Destroy the GMT session */
    GMT_Destroy_Session (API);
}
```

### 6.2 Compilation

To compile this program (we assume it is called example1.c), we use the gmt-config script to determine the correct compile and link flags and then run gcc:

```bash
inc=`gmt-config --cflags`
lib=`gmt-config --libs`
gcc example1.c $inc $lib -o example1
./example1
```

This obviously assumes you have already installed GMT and that it is in your path. If you run example1 it will take a moment (this is mostly due to the gridding performed by greenspline) and then it stops. You should find the resulting grid junk.nc in the current directory. Plot it to see if it makes sense, e.g.
If you intend to write applications that take any number of data files via the command line then there will be more book-keeping to deal with, and we will discuss those steps later. Likewise, if you need to process a file record-by-record then more lines of code will be required.

### 6.3 Plugins

Developers who wish to make custom plugin libraries that are compatible with GMT should examine the fully functioning examples of more involved code, available from the repository gmt-custom, obtainable via

```
svn checkout svn://gmtserver.soest.hawaii.edu/gmt-custom/trunk gmt-custom
```

### 6.4 List of API functions

The following is an alphabetical listing of all the public API functions in GMT. Click on any of them to see the full syntax of each function.

The C/C++ API is deliberately kept small to make it easy to use.

<table>
<thead>
<tr>
<th>constant</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GMT_Alloc_Segment</td>
<td>Allocate data and text segments</td>
</tr>
<tr>
<td>GMT_Append_Option</td>
<td>Append new option structure to linked list</td>
</tr>
<tr>
<td>GMT_Begin_IO</td>
<td>Enable record-by-record i/o</td>
</tr>
<tr>
<td>GMT_Call_Module</td>
<td>Call any of the GMT modules</td>
</tr>
<tr>
<td>GMT_Convert_Data</td>
<td>Convert between compatible data types</td>
</tr>
<tr>
<td>GMT_Close_VirtualFile</td>
<td>Close a virtual file</td>
</tr>
<tr>
<td>GMT_Create_Args</td>
<td>Convert linked list of options to text array</td>
</tr>
<tr>
<td>GMT_Create_Cmd</td>
<td>Convert linked list of options to command line</td>
</tr>
<tr>
<td>GMT_Create_Data</td>
<td>Create an empty data resource</td>
</tr>
<tr>
<td>GMT_Create_Options</td>
<td>Convert command line options to linked list</td>
</tr>
<tr>
<td>GMT_Create_Session</td>
<td>Initialize a new GMT session</td>
</tr>
<tr>
<td>GMT_Delete_Option</td>
<td>Delete an option structure from the linked list</td>
</tr>
<tr>
<td>GMT_Destroy_Args</td>
<td>Delete text array of arguments</td>
</tr>
<tr>
<td>GMT_Destroy_Cmd</td>
<td>Delete text command of arguments</td>
</tr>
<tr>
<td>GMT_Destroy_Data</td>
<td>Delete a data resource</td>
</tr>
<tr>
<td>GMT_Destroy_Group</td>
<td>Delete a group of data resources</td>
</tr>
<tr>
<td>GMT_Destroy_Options</td>
<td>Delete the linked list of option structures</td>
</tr>
<tr>
<td>GMT_Destroy_Session</td>
<td>Terminate a GMT session</td>
</tr>
<tr>
<td>GMT_Duplicate_Data</td>
<td>Make an identical copy of a data resources</td>
</tr>
<tr>
<td>GMT_Encode_Options</td>
<td>Encode option arguments for external interfaces</td>
</tr>
<tr>
<td>GMT_Expand_Option</td>
<td>Expand option with explicit memory references</td>
</tr>
<tr>
<td>GMT_End_IO</td>
<td>Disable further record-by-record i/o</td>
</tr>
<tr>
<td>GMT_FFT</td>
<td>Take the Fast Fourier Transform of data object</td>
</tr>
<tr>
<td>GMT_FFT_1D</td>
<td>Take the Fast Fourier Transform of 1-D float data</td>
</tr>
<tr>
<td>GMT_FFT_2D</td>
<td>Take the Fast Fourier Transform of 2-D float data</td>
</tr>
</tbody>
</table>

Continued on next page
### Table 1 – continued from previous page

<table>
<thead>
<tr>
<th>constant</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GMT_FFT_Create</td>
<td>Initialize the FFT machinery</td>
</tr>
<tr>
<td>GMT_FFT_Destroy</td>
<td>Terminate the FFT machinery</td>
</tr>
<tr>
<td>GMT_FFT_Option</td>
<td>Explain the FFT options and modifiers</td>
</tr>
<tr>
<td>GMT_FFT_Parse</td>
<td>Parse argument with FFT options and modifiers</td>
</tr>
<tr>
<td>GMT_FFT_Wavenumber</td>
<td>Return wavenumber given data index</td>
</tr>
<tr>
<td>GMT_Find_Option</td>
<td>Find an option in the linked list</td>
</tr>
<tr>
<td>GMT_Get_Common</td>
<td>Determine if a GMT common option was set</td>
</tr>
<tr>
<td>GMT_Get_Coord</td>
<td>Create a coordinate array</td>
</tr>
<tr>
<td>GMT_Get_Default</td>
<td>Obtain one of the API or GMT default settings</td>
</tr>
<tr>
<td>GMT_Get_Index</td>
<td>Convert row, col into a grid or image index</td>
</tr>
<tr>
<td>GMT_Get_Matrix</td>
<td>Obtain pointer to user matrix from container</td>
</tr>
<tr>
<td>GMT_Get_Pixel</td>
<td>Get grid or image node</td>
</tr>
<tr>
<td>GMT_Get_Record</td>
<td>Import a single data record</td>
</tr>
<tr>
<td>GMT_Get_Row</td>
<td>Import a single grid row</td>
</tr>
<tr>
<td>GMT_Get_Values</td>
<td>Convert string into coordinates or dimensions</td>
</tr>
<tr>
<td>GMT_Get_Vector</td>
<td>Obtain pointer to user vector from container</td>
</tr>
<tr>
<td>GMT_Init_IO</td>
<td>Initialize i/o given registered resources</td>
</tr>
<tr>
<td>GMT_Init_VirtualFile</td>
<td>Reset a virtual file for reuse</td>
</tr>
<tr>
<td>GMT_Make_Option</td>
<td>Create an option structure</td>
</tr>
<tr>
<td>GMT_Message</td>
<td>Issue a message, optionally with time stamp</td>
</tr>
<tr>
<td>GMT_Open_VirtualFile</td>
<td>Select memory location as input or output for module</td>
</tr>
<tr>
<td>GMT_Option</td>
<td>Explain one or more GMT common options</td>
</tr>
<tr>
<td>GMT_Parse_Common</td>
<td>Parse the GMT common options</td>
</tr>
<tr>
<td>GMT_Put_Matrix</td>
<td>Put user matrix into container</td>
</tr>
<tr>
<td>GMT_Put_Record</td>
<td>Export a data record</td>
</tr>
<tr>
<td>GMT_Put_Row</td>
<td>Export a grid row</td>
</tr>
<tr>
<td>GMT_Put_Vector</td>
<td>Put user vector into container</td>
</tr>
<tr>
<td>GMT_Read_Data</td>
<td>Import a data resource or file</td>
</tr>
<tr>
<td>GMT_Read_Group</td>
<td>Import a group of data resources or files</td>
</tr>
<tr>
<td>GMT_Read_VirtualFile</td>
<td>Access the output from a module via memory</td>
</tr>
<tr>
<td>GMT_Register_IO</td>
<td>Register a resources for i/o</td>
</tr>
<tr>
<td>GMT_Report</td>
<td>Issue a message contingent upon verbosity level</td>
</tr>
<tr>
<td>GMT_Set_Default</td>
<td>Set one of the API or GMT default settings</td>
</tr>
<tr>
<td>GMT_Set_Comment</td>
<td>Assign a comment to a data resource</td>
</tr>
<tr>
<td>GMT_Set_Columns</td>
<td>Specify how many output columns to use for rec-by-rec</td>
</tr>
<tr>
<td>GMT_Set_Geometry</td>
<td>Specify data geometry for rec-by-rec i/o</td>
</tr>
<tr>
<td>GMT_Set_Index</td>
<td>Convert row, col into a grid or image index</td>
</tr>
<tr>
<td>GMT_Status_IO</td>
<td>Check status of record-by-record i/o</td>
</tr>
<tr>
<td>GMT_Update_Option</td>
<td>Modify an option structure</td>
</tr>
<tr>
<td>GMT_Write_Data</td>
<td>Export a data resource</td>
</tr>
</tbody>
</table>

Summary of all the API functions and their purpose.
CHAPTER 7

The GMT C Application Program Interface

7.1 Initialize a new GMT session

Advanced programs may be calling more than one GMT session and thus run several sessions, perhaps concurrently as different threads on multi-core machines. We will now discuss these steps in more detail. Throughout, we will introduce upper-case GMT C enum constants *in lieu* of simple integer constants. These are considered part of the API and are available for developers via the gmt_resources.h include file.

Most applications will need to initialize only a single GMT session. This is true of all the standard GMT programs since they only call one GMT module and then exit. Most user-developed GMT applications are likely to only initialize one session even though they may call many GMT modules. However, the GMT API supports any number of simultaneous sessions should the programmer wish to take advantage of it. This might be useful when you have access to several CPUs and want to spread the computing load. In the following discussion we will simplify our treatment to the use of a single session only.

To initiate the new GMT session we use

```c
void *GMT_Create_Session (const char *tag, unsigned int pad, unsigned int mode, int (*print_func) (FILE *, const char *));
```

and you will typically call it like this:

```c
void *API = NULL;    /* Opaque pointer to GMT controls */
API = GMT_Create_Session ("Session name", 2, 0, NULL);
```

where API is an opaque pointer to the hidden GMT API control structure. You will need to pass this pointer to *all* subsequent GMT API functions; this is how essential internal information is passed around. The key task of this initialization is to set up the GMT machinery and internal variables used for map projections, plotting, i/o, etc. The initialization also allocates space for internal structures used to keep track of data. The pad argument specifies how many rows and columns should be used as padding for grids and images so that boundary conditions can be applied. GMT uses 2 and we strongly recommend that you use that value. In particular, if you choose 0 or 1 there may be certain GMT modules that will

---

5 However, there is no thread-support yet, so you will need to manage your own threads.
be unable to do their work properly as they count on those boundary rows and columns in the grids. Note that this setting has no effect on what is written to a grid file; the padding is an internal feature. The `mode` argument is only used for external APIs that need to communicate their special needs during the session creation. This integer argument is a sum of bit flags and the various bits control the following settings:

1. Bit 1 (1): If set, then GMT will not call the system exit function when a serious problem has been detected but instead will simply return control to the calling environment. For instance, this is required by the GMT/MATLAB toolbox since calling exit would also exit MATLAB itself. Unless your environment has this feature you should leave this bit alone.

2. Bit 2 (2): If set, then it means we are calling the GMT API from an external API, such as MATLAB, Octave, or Python. Normal C/C++ programs should leave this bit alone. Its effect is to enable two additional modules for reading and writing GMT resources from these environments (those modules would not make any sense in a Unix command-line environment).

3. Bit 3 (4): If set, then it means the external API uses a column-major format for matrices (e.g., MATLAB, Fortran). If not set we default to row-major format (C/C++, Python, etc.).

4. Bit 4 (8): If set, then we enable GMT's modern run-mode (where `-O` `-K` are not allowed and PostScript is written to hidden temp file). Default is the GMT classic run-mode.

The `print_func` argument is a pointer to a function that is used to print messages from GMT via `GMT_Message` or `GMT_Report` from external environments that cannot use the standard printf function (this is the case for the GMT/MATLAB toolbox, for instance). For all other uses you should simply pass NULL for this argument. Should something go wrong during the API initialization then API will be returned as NULL. Finally, `GMT_Create_Session` will examine the environmental parameter TMPDIR (TEMP on Windows) to set the GMT temporary directory [/tmp on Unix, current directory on Windows].

Below is a bare-bones minimalistic GMT program `hello.c` that initializes and destroys a GMT session:

```c
#include "gmt.h"
int main (int argc, char *argv[]) {
    void *API; /* The API control structure */
    /* Initialize the GMT session */
    API = GMT_Create_Session ("test", 2U, 0, NULL);
    /* And now for something original: */
    GMT_Message (API, GMT_TIME_NONE, "hello, world\n");
    /* Destroy the GMT session */
    GMT_Destroy_Session (API);
    return 0;
}
```

While not super-exiting, this code demonstrates the two essential API calls required to initiate and later terminate a GMT session. In between we do what all basic programs are supposed to do: print “Hello, world”. The user is of course allowed to do whatever custom processing before the GMT session is created and can do all sorts of stuff after the GMT session is destroyed, as long as no GMT functions or resources are accessed. It may be convenient to isolate the GMT-specific processing from the custom part of the program and only maintain an active GMT session when needed.

### 7.2 Register input or output resources

When using the standard GMT programs, it is common to specify input files on the command line or via special program options (e.g., `-Intensity.nc`). The outputs of the programs are either written to standard output (which you may redirect to files or pipes into other programs) or to files specified by specific program options (e.g., `-Goutput.nc`). Alternatively, the GMT API allows you to specify input (and output) to be associated with open file handles or virtual files. We will examine this more closely.
below. Registering a resource is a required step before attempting to import or export data that do not come from files or standard input/output.

### 7.2.1 Resource initialization

All GMT programs dealing with input or output files given on the command line, and perhaps defaulting to the standard input or output streams if no files are given, must call the i/o initializer function GMT_Init_IO once for each direction required (i.e., input and output separately). For input it determines how many input sources have already been registered. If none has been registered then it scans the program arguments for any filenames given on the command line and register these input resources. Finally, if we still have found no input sources we assign the standard input stream as the single input source. For output it is similar: If no single destination has been registered we specify the standard output stream as the output destination. Only one main output destination is allowed to be active when a module writes data (some modules also write additional output via program-specific options). The prototype for this function is

```c
int GMT_Init_IO (void *API, unsigned int family, unsigned int geometry, unsigned int direction, unsigned int mode, unsigned int n_args, void *args);
```

where `family` specifies what kind of resource is to be registered, `geometry` specifies the geometry of the data, `direction` is either GMT_IN or GMT_OUT, and `mode` is a bit flag that determines what we do if no resources have been registered. The choices are

- **GMT_ADD_FILES_IF_NONE** (1) means “add command line (option) files if none have been registered already”.
- **GMT_ADD_FILES_ALWAYS** (2) means “always add any command line files”.
- **GMT_ADD_STDIO_IF_NONE** (4) means “add std* if no other input/output have been specified”.
- **GMT_ADD_DEFAULT** (6) means “always add any command line files first, and then add std* if no other input/output were specified”.
- **GMT_ADD_STDIO_ALWAYS** (8) means “always add std* even if resources have been registered”.
- **GMT_ADD_EXISTING** (16) means “only use already registered resources”.

The standard behavior is GMT_ADD_DEFAULT (6). Next, `n_args` is 0 if `args` is the head of a linked list of options (further discussed in Prepare modules opts); otherwise `args` is an array of `n_args` strings (i.e., the int argc, char *argv[] model).

Many programs will register an export location where results of a GMT function (say, a filtered grid) should be returned, but may then wish to use that variable as an input resource in a subsequent module call. This is accomplished by re-registering the resource as an input source, thereby changing the direction of the data set. The function returns 1 if there is an error; otherwise it returns 0.

### 7.2.2 Resource registration

Should your program need to add additional sources (or a destination) to the list of items to be considered you will need to register them manually. This is considered a low-level activity and most applications are unlikely to have to resort to this step. We document it here in case your situation calls for such action. Registration involves a direct or indirect call to
where \( \text{family} \) specifies what kind of resource is to be registered, \( \text{method} \) specifies how we to access this resource (see Table \text{methods} for recognized methods), \( \text{geometry} \) specifies the geometry of the data, \( \text{ptr} \) is the address of the pointer to the named resource. If \( \text{direction} \) is \( \text{GMT\_OUT} \) and the \( \text{method} \) is not related to a file (filename, stream, or handle), then \( \text{ptr} \) must be NULL. Note there are some limitations on when you may pass a file pointer as the method. Many grid file formats cannot be read via a stream (e.g., netCDF files) so in those situations you cannot pass a file pointer [and GMT_Register_IO would have no way of knowing this]. For grid (and image) resources you may request to obtain a subset via the \( \text{wesn} \) array; otherwise, pass NULL (or an array with at least 4 items all set to 0) to obtain the entire grid (or image). The \( \text{direction} \) indicates input or output and is either \( \text{GMT\_IN} \) or \( \text{GMT\_OUT} \). Finally, the function returns a unique resource ID, or \( \text{GMT\_NOTSET} \) if there was an error.

<table>
<thead>
<tr>
<th>( \text{family} )</th>
<th>( \text{source points to} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{GMT_IS_DATASET}</td>
<td>A [multi-segment] data file</td>
</tr>
<tr>
<td>\text{GMT_IS_GRID}</td>
<td>A grid file</td>
</tr>
<tr>
<td>\text{GMT_IS_IMAGE}</td>
<td>An image</td>
</tr>
<tr>
<td>\text{GMT_IS_PALETTE}</td>
<td>A color palette table [CPT]</td>
</tr>
<tr>
<td>\text{GMT_IS_POSTSCRIPT}</td>
<td>A GMT PostScript object</td>
</tr>
<tr>
<td>\text{GMT_IS_TEXTSET}</td>
<td>A [multi-segment] text file</td>
</tr>
<tr>
<td>\text{GMT_IS_MATRIX}</td>
<td>A custom user data matrix</td>
</tr>
<tr>
<td>\text{GMT_IS_VECTOR}</td>
<td>A custom user data vector</td>
</tr>
</tbody>
</table>

GMT constants used to specify a data family.

<table>
<thead>
<tr>
<th>( \text{method} )</th>
<th>( \text{how to read/write data} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{GMT_IS_FILE}</td>
<td>Pointer to name of a file</td>
</tr>
<tr>
<td>\text{GMT_IS_STREAM}</td>
<td>Pointer to open stream (or process)</td>
</tr>
<tr>
<td>\text{GMT_IS_FDESC}</td>
<td>Pointer to integer file descriptor</td>
</tr>
<tr>
<td>\text{GMT_IS_DUPLICATE}</td>
<td>Pointer to memory we may \textit{duplicate} data from</td>
</tr>
<tr>
<td>\text{GMT_IS_REFERENCE}</td>
<td>Pointer to memory we may \textit{reference} data from</td>
</tr>
</tbody>
</table>

GMT constants used to specify how data will be read or written.

<table>
<thead>
<tr>
<th>( \text{geometry} )</th>
<th>( \text{description} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{GMT_IS_NONE}</td>
<td>Not a geographic feature</td>
</tr>
<tr>
<td>\text{GMT_IS_POINT}</td>
<td>Multi-dimensional point data</td>
</tr>
<tr>
<td>\text{GMT_IS_LINE}</td>
<td>Geographic or Cartesian line segments</td>
</tr>
<tr>
<td>\text{GMT_IS_POLYGON}</td>
<td>Geographic or Cartesian closed polygons</td>
</tr>
<tr>
<td>\text{GMT_IS_PLP}</td>
<td>Either points, lines, or polygons</td>
</tr>
<tr>
<td>\text{GMT_IS_SURFACE}</td>
<td>2-D gridded surface</td>
</tr>
</tbody>
</table>

GMT constants used to specify the geometry of the data object.
### GMT constants used for domain array indexing.

<table>
<thead>
<tr>
<th>index</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GMT_XLO</td>
<td>x_min (west) boundary of grid subset</td>
</tr>
<tr>
<td>GMT_XHI</td>
<td>x_max (east) boundary of grid subset</td>
</tr>
<tr>
<td>GMT_YLO</td>
<td>y_min (south) boundary of grid subset</td>
</tr>
<tr>
<td>GMT_YHI</td>
<td>y_max (north) boundary of grid subset</td>
</tr>
<tr>
<td>GMT_ZLO</td>
<td>z_min (bottom) boundary of 3-D matrix subset</td>
</tr>
<tr>
<td>GMT_ZHI</td>
<td>z_max (top) boundary of 3-D matrix subset</td>
</tr>
</tbody>
</table>

7.3 Create empty resources

If your application needs to build and populate GMT resources in ways that do not depend on external resources (files, memory locations, etc.), or you have data read in separately and you wish to build a GMT resource from scratch, then you can obtain an empty object by calling

```c
void *GMT_Create_Data (void *API, unsigned int family, unsigned int geometry,
                       unsigned int mode, uint64_t par[], double *wesn, double *inc,
                       unsigned int registration, int pad, void *data)
```

which returns a pointer to the allocated resource. Pass a valid `family` selection. Also pass a compatible `geometry`. Depending on the family and your particular way of representing dimensions you may pass the additional parameters in one of two ways:

1. Actual integer dimensions of items needed (which depends on the `family`).
2. Physical distances and increments of each dimension.

For the first case you should pass both `wesn` and `inc` as NULL (or as arrays with elements all set to 0), and pass the `par` array with contents as indicated below:

**GMT_IS_GRID.** An empty `GMT_GRID` structure with a header is allocated; the data array is NULL. Use `registration` to choose either gridline (`GMT_GRID_PIXEL_REG`) or pixel (`GMT_GRID_NODE_REG`) registration. The domain can be prescribed on one of two ways: (1) The `par` argument is NULL. Then `wesn` and `inc` can also be NULL but only if the common GMT options `-R` and `-I` have been set because they are required to get the necessary info. If they were not set, then `wesn` and `inc` must in fact be transmitted. If `wesn` and `inc` are set (directly or indirectly) then `par` is ignored, even if not NULL. (2) The `par` argument is not NULL but both `wesn` and `inc` are NULL. Now, `par[0]` must have the number of columns and `par[1]` must have the number of rows in the grid. Internally, `inc` will be set to 1/1 and `wesn` will be set to 0/n_columns/0/n_rows. As an option, add `GMT_GRID_XY` to `mode` and we also allocate the grid's x and y coordinate vectors.

**GMT_IS_IMAGE.** Same procedure as for `GMT_IS_GRID` but we return an empty `GMT_IMAGE` object. In either way of specification you may use `par[2]` to pass the number of image bands [1].

**GMT_IS_DATASET.** We allocate an empty `GMT_DATASET` structure consisting of `par[0]` tables, each with `par[1]` segments, each with `par[2]` rows, all with `par[3]` columns. The `wesn`, `inc`, and `registration` argument are ignored. The `data` argument should be NULL.
GMT_API Documentation, Release 5.4.5

GMT_IS_TEXTSET. We allocate an empty GMT_TEXTSET structure consisting of par[0] tables, each with par[1] segments, all with par[2] text records (rows). The wesn, inc, and registration arguments are ignored and should be NULL/0. The data argument should be NULL.

GMT_IS_PALETTE. We allocate an empty GMT_PALETTE structure with par[0] palette entries. The wesn, inc, and registration arguments are ignored and should be NULL/0. The data argument should be NULL.

GMT_IS_POSTSCRIPT. We allocate an empty GMT_POSTSCRIPT structure with a text buffer of length par[0]. Give par[0] = 0 if the PostScript string is allocated or obtained by other means. The wesn, inc, and registration arguments are ignored and should be NULL/0. The data argument should be NULL.

GMT_IS_VECTOR. We allocate an empty GMT_VECTOR structure with par[0] column entries. The number of rows can be specified in one of two ways: (1) Set the number of rows via par[1]. Then, wesn, inc, and registration arguments are ignored. (2) Specify wesn, inc, and registration and the number of rows will be computed from these parameters instead. The data argument should be NULL. If you have custom vectors you wish to use then pass par[1] = 0 to avoid any allocation and use GMT_Put_Vector to hook up your vectors.

GMT_IS_MATRIX. We allocate an empty GMT_MATRIX structure. The domain can be prescribed on one of two ways: (1) Here, par[0] is the number of columns while par[1] has the number of rows. Also, par[2] indicates the number of layers for a 3-D matrix, or pass 0, 1, or NULL for a 2-D matrix. (2) Pass wesn, inc, registration and we compute the dimensions of the matrix. The data argument should be NULL. As for vectors, give dimensions as 0 and hook your custom matrix in via a call to GMT_Put_Matrix.

For grids and images you may pass pad to set the padding, or -1 to accept the prevailing GMT default. The mode determines what is actually allocated when you have chosen grids or images. As for GMT_Read_Data you can pass GMT_CONTAINER_AND_DATA to initialize the header and allocate space for the array; here data must be NULL. Alternatively, you can pass GMT_CONTAINER_ONLY to just initialize the grid or image header, and later call GMT_Create_Data a second time, now passing GMT_DATA_ONLY, to allocate space for the array. In that second call you pass the pointer returned by the first call as data and specify the family; all other arguments should be NULL or 0. Normally, resources created by this function are considered to be input (i.e., have a direction that is GMT_IN). The exception to this is for containers to hold results from GMT which need have a direction set to GMT_OUT. Such empty containers are requested by passing mode = GMT_IS_OUTPUT and setting all dimension arguments to 0 or NULL. The function returns a pointer to the data container. In case of an error we return a NULL pointer and pass an error code via API->error.

7.3.1 Hooking user arrays to objects

If you have custom column vector or matrices and you want them to be used as input to GMT modules, you will need to create a GMT_VECTOR or GMT_MATRIX container and hook your items to them. Likewise, if you want to receive the output of GMT modules into user arrays or matrices then you will need to access those data. The following utility functions are used for these tasks:

```c
int GMT_Put_Matrix (void *API, struct GMT_MATRIX *M, unsigned int type, void _
˓→*matrix);
```
where \( M \) is a \texttt{GMT\_MATR\_x} created by \texttt{GMT\_Create\_Data}, the \texttt{type} is one of the recognized data \texttt{types}, and \texttt{matrix} is your custom matrix. To extract a custom matrix from an output \texttt{GMT\_MATR\_x} you can use

```c
void *GMT_Get_Matr\_x (void *API, struct GMT\_MATR\_x *M);
```

which simply returns a pointer to the right union pointer. For vectors the same principles apply:

```c
int GMT_Put_Vect\_x (void *API, struct GMT\_VECTR\_x *V, unsigned int col, unsigned int type, void *vector);
```

where \( V \) is the \texttt{GMT\_VECTR\_x} created by \texttt{GMT\_Create\_Data}, \texttt{col} is the vector column in question, \texttt{type} is one of the recognized data \texttt{types} used for this vector, and \texttt{vector} is a pointer to this custom vector. To extract a custom vector from an output \texttt{GMT\_VECTR\_x} you can use

```c
void *GMT_Get_Vect\_x (void *API, struct GMT\_VECTR\_x *V, unsigned int col);
```

where \texttt{col} is the vector number you wish to obtain a pointer to.

### 7.3.2 Manually add segments

If you do not know the number of rows in the segments or you expect different segments to have different lengths then you should set the row dimension to zero in \texttt{GMT\_Create\_Data} and add the segments manually with \texttt{GMT\_Alloc\_Segment}, which allocates a new \texttt{GMT\_DATASET} or \texttt{GMT\_TEXTSET} segment for such multi-segment tables.

```c
void *GMT Alloc\_Segment (void *API, unsigned int family, uint64_t n_rows, uint64_t n_columns, char *header, void *S);
```

where \texttt{header} is the segment’s desired header (or NULL) and \texttt{family} selects which kind of resource is desired, which in this case should either be \texttt{GMT_IS\_DATASET} or \texttt{GMT_IS\_TEXTSET}. If \( S \) is not NULL then we simply reallocate the lengths of the segment; otherwise a new segment is first allocated.

For a \texttt{GMT\_DATASET} there is also the option of controlling the allocation of the segment array by setting \texttt{n\_rows = 0}. This would allow external arrays (double-precision only) to connect to the \( S->\text{data}[\text{col}] \) arrays and not be freed by GMT’s garbage collector.

### 7.4 Duplicate resources

Often you have read or created a data resource and then need an identical copy, presumably to make modifications to. Or, you want a copy with the same dimensions and allocated memory, except data values should not be duplicated. Alternatively, perhaps you just want to duplicate the header and skip the allocation and duplication of the data entirely. These tasks are addressed by

```c
void *GMT Dupl\_icate\_Data (void *API, unsigned int family, unsigned int mode, void *data);
```

which returns a pointer to the allocated resource. Specify which \texttt{family} and select \texttt{mode} from \texttt{GMT\_DUPLICATE\_DATA}, \texttt{GMT\_DUPLICATE\_ALLOC}, and \texttt{GMT\_DUPLICATE\_NONE}, as discussed above (also see \texttt{mode} discussion above). For \texttt{GMT\_GRID} you may add \texttt{GMT\_DUPLICATE\_RESET} which will ensure the duplicate grid will have normal padding (useful when the original has non-standard padding). For \texttt{GMT\_DATASET} and \texttt{GMT\_TEXTSET} you can add modifiers \texttt{GMT\_ALLOC\_VERTICAL} or \texttt{GMT\_ALLOC\_HORIZONTAL} to the \texttt{mode} if you wish to put all the data into a single long table or
to paste all tables side-by-side, respectively (thus getting one wide table instead). Additional note for

**GMT_DATASET**: Normally we allocate the output given the corresponding input dimensions. You can override these by specifying your alternative dimensions in the input dataset’s variable `dim[]`. The `data` is a pointer to the resource you wish to duplicate. In case of an error we return a NULL pointer and pass an error code via `API->error`.

### 7.5 Convert between resource types

Having a resource in memory you may want to convert it to an alternative representation. For instance, you may have a `GMT_DATASET` in memory but for an application you need the equivalent information in `GMT_TEXTSET` format. Or, you have read a `GMT_DATASET` but need to strip the information from the data into a `VECTOR` format, dropping all the segment header information, so that your custom algorithm or other non-GMT functions can be used on the data. In this case you will use

```c
void *GMT_Convert_Data (void *API, void *In, unsigned int family_in, 
                        void *Out, unsigned int family_out, unsigned int flag[]);
```

which returns a pointer to the converted resource. Specify the needed `family` for both the input and output resources and set the (up to) three flags passed via the `flag` array. The first `flag[0]` determines how table headers and segment headers should be handled. By default (`flag[0] = 0`) they are preserved (to the extent possible). E.g., converting a `GMT_DATASET` to `MATRIX` always means table headers are skipped whereas segment headers are converted to NaN-records. Other values for this flag is 1 (Table headers are not copied, segment headers are preserved), 2 (Headers are preserved, segment headers are reset to blank), or 3 (All headers headers are eliminated). Note that this flag only affects duplication of headers. If the new object is written to file at a later stage then it is up to the GMT default setting if headers are written to file or not. The second `flag[1]` controls how many columns to expect when converting `GMT_TEXTSET` only. If 0 then we try to determine the number of columns from the first text record. If `family_in` is not `GMT_IS_TEXTSET` then `flag[1]` is ignored. The third `flag[2]` controls restructuring of tables and segments within a set. For `flag[2] = 0` we retain the original layout. Other selections are `GMT_WRITE_TABLE SEGMENT` (combine all segments into a single segment in a single table), `GMT_WRITE_TABLE` (collect all segments into a single table), and `GMT_WRITE_SEGMENT` (combine segments into one segment per table). Many family combinations are simply not allowed, such as grid to color palette, dataset to image, etc.

### 7.6 Import Data Sets

If your program needs to import any of the six recognized data types (data table, grid, image, CPT, PostScript, or text table) you will use the `GMT_Read_Data` or `GMT_Read_VirtualFile` functions. The former is typically used when reading from files, streams (e.g., `stdin`), or an open file handle, while the latter is only used to read from memory. Because of the similarities of these six import functions we use an generic form that covers all of them.

All input functions takes a parameter called `mode`. The `mode` parameter generally has different meanings for the different data types and will be discussed below. However, one bit setting is common to all types: By default, you are only allowed to read a data source once; the source is then flagged as having been read and subsequent attempts to read from the same source will result in a warning and no reading takes place. In the unlikely event you need to re-read a source you can override this default behavior by adding `GMT_IO_RESET` to your `mode` parameter. Note that this override does not apply to sources that are streams or file handles, as it may not be possible to re-read their contents.
7.6.1 Import from a file, stream, or handle

To read an entire resource from a file, stream, or file handle, use

```c
void *GMT_Read_Data (void *API, unsigned int family, unsigned int method,
    unsigned int geometry, unsigned int mode, double wesn[], const char *input,
    void **ptr);
```

- **API**
- **family**
- **method**
- **geometry**
- **mode** – see below
- **wesn**
- **input** – a pointer to char holding the file name to read, or NULL if stdin
- **ptr** – NULL or the pointer returned by this function after a first call (when reading grids in two steps)

**Return:** Pointer to data container, or NULL if there were errors (passed back via API->error)

where `ptr` is NULL except when reading grids in two steps (i.e., first get a grid structure with a header, then read the data). Most of these arguments have been discussed earlier. This function can be called in three different situations:

1. If you have a single source (filename, stream pointer, etc.) you can call `GMT_Read_Data` directly; there is no need to first register the source with `GMT_Register_IO` or gather the sources with `GMT_Init_IO`. Furthermore, for `GMT_DATASET` and `GMT_TEXTSET` you can also specify a filename that contains UNIX wildcards (e.g., “all_*_[ab]?.txt”) and these will all be read to produce a single multi-table `GMT_DATASET` or `GMT_TEXTSET` (for other datatypes, see `GMT_Read_Group` instead).

2. If you want to specify stdin as source then pass input as NULL.

3. If you already registered all desired sources with `GMT_Init_IO` then you indicate this choice by passing the invalid `geometry` = 0.

Space will be allocated to hold the results, as needed, and a pointer to the object is returned. If there are errors we simply return NULL and report the error. The `mode` parameter has different meanings for different data types.

**Color palette table.** `mode` contains bit-flags that control how the CPT’s back-, fore-, and NaN-colors should be initialized. Select 0 to use the CPT resource’s back-, fore-, and NaN-colors, 2 to replace these with the current GMT default values, or 4 to replace them with the color table’s entries for highest and lowest value.

**Data table.** `mode` is currently not used.

**Text table.** `mode` is currently not used.

**GMT grid or image.** Here, `mode` determines how we read the grid: To read the entire grid and its header, pass `GMT_CONTAINER_AND_DATA`. However, if you may need to extract a sub-region you must first read the header by passing `GMT_CONTAINER_ONLY`, then examine the header structure range attributes, specify a subset via the array `wesn`, and finally call `GMT_Read_Data` a second time, now with `mode = GMT_DATA_ONLY`, passing your `wesn`
array and the grid structure returned from the first call as `ptr`. In the event your data array should be allocated to hold both the real and imaginary parts of a complex data set you must add either `GMT_GRID_IS_COMPLEX_REAL` or `GMT_GRID_IS_COMPLEX_IMAG` to `mode` so as to allow for the extra memory needed and to stride the complex value-pairs correctly. If your grid is huge and you must read it row-by-row, set `mode` to `GMT_CONTAINER_ONLY` | `GMT_GRID_ROW_BY_ROW`. You can then access the grid row-by-row using `GMT_Get_Row`. By default, the rows will be automatically processed in sequential order. To completely specify which row to be read, pass `GMT_GRID_ROW_BY_ROW_MANUAL` instead. Finally, as an option you may add `GMT_GRID_XY` to the `mode` and we also allocate the `x` and `y` coordinate vectors for the grid or image.

PostScript. `mode` is currently not used.

If you need to read the same resource more than once you should add the bit flag `GMT_IO_RESET` to the given `mode`.

### 7.6.2 Import a group of data sets

To read a group of resources, you may instead use

```c
void *GMT_Read_Group (void *API, unsigned int family, unsigned int method, unsigned int geometry, unsigned int mode, double wesn[], void *input, unsigned int *n_items, void *ptr);
```

- `API`  
- `family`  
- `method`  
- `geometry`  
- `mode` – see below  
- `wesn`  
- `input` – Contents depends on the value of `n_items`. If it is zero then we expect a pointer to char holding UNIX wildcard file name(s) to read, otherwise we expect a pointer to an array of character strings (`n_items` in total) with names of all the files to read. If `n_items` is NULL then we assume 0 but cannot return the number found.  
- `ptr` – NULL or the pointer returned by this function after a first call (applies when reading grids or images in two steps)  
- Return: Pointer to array of data container, or NULL if there were errors (passed back via API-error)

where `ptr` is NULL except when reading grids in two steps (i.e., first get a grid structures with a header, then read the data arrays). Most of these arguments have been discussed earlier. It is useful when you need to read a series of files (e.g., from a list with filenames) or want to specify the items to read using a UNIX wildcard specification. Note: If used with `GMT_DATASET` or `GMT_TEXTSET` then you will receive an array of structures as well. Typically, many data/text files are read into separate tables that all form part of a single SET (this is what `GMT_Read_Data` does), but if `GMT_Read_Group` is used on the same arguments then an array of one-table sets will be returned instead. The purpose of your application will dictate which form is more convenient.
7.6.3 Open a virtual file (memory location)

If you have read in or otherwise obtained a data object in memory and you now wish for it to serve as input to a GMT module, you will have to associate that object with a “Virtual File”. This step assigns a special filename to the memory location and you can then pass this filename to any module that needs to read that data. It is similar for writing, and you can pass NULL as the object to have GMT automatically allocate the output resource. The full syntax is

```c
void GMT_Open_VirtualFile (void *API, unsigned int family, unsigned int geometry, unsigned int direction, void *data, char *filename);
```

Here, `data` is the pointer to your memory object. The function returns the desired filename via `filename`. This string must be at least GMT_STR16 bytes (16). The other arguments have been discussed earlier. Simply pass this filename in the calling sequence to the module you want to use to indicate which file should be used for reading or writing.

7.6.4 Import from a virtual file

Once the module completes it will have written its output to the virtual file you initialized with `GMT_Open_VirtualFile`. To use the actual data you will need to “read” it into your program. Of course, the data are already in memory but to access it you need to use `GMT_Read_VirtualFile`, which expects the output filename you obtained from `GMT_Open_VirtualFile`. The syntax is

```c
void GMT_Read_VirtualFile (void *API, char *filename);
```

The function requires the output filename via `filename` and then returns the data object, similar to what `GMT_Read_Data` does.

7.6.5 Reset a virtual file for reuse

Should you need to read a virtual file again then you must first reset it to its original state with `GMT_Init_VirtualFile`. The syntax is

```c
int GMT_Init_VirtualFile (void *API, unsigned int mode, const char *filename);
```

The function requires the virtual file’s `filename` and then resets the internal counters (e.g., record numbers and other book-keeping parameters). The `mode` is presently not used.

7.6.6 Close a virtual file

Once you have finished using a virtual file you need to close it. This will reset its internal settings back to what it was before you used it as a virtual file. The syntax is

```c
int GMT_Close_VirtualFile (void *API, char *filename);
```

where `filename` is the name of the virtual file.

7.7 Record-by-record input

In the case of data and text tables you have the option of selecting record-by-record reading or writing. As a general rule, your program development simplifies if you can read entire resources into mem-
ory with \texttt{GMT\_Read\_Data} or \texttt{GMT\_Read\_VirtualFile}. However, if this leads to unacceptable memory usage or if the program logic is particularly simple, you may obtain one data record at the time via \texttt{GMT\_Get\_Record} and write one at the time with \texttt{GMT\_Put\_Record}. For row-by-row i/o for grids there is the corresponding function \texttt{GMT\_Get\_Row}. There are additional overhead involved in setting up record-by-record processing, which is the topic of this section.

### 7.7.1 Enable Data Import

Once all input resources have been registered, we signal the API that we are done with the registration phase and are ready to start the actual data import. This step is only required when reading one record at the time. We initialize record-by-record reading by calling \texttt{GMT\_Begin\_IO}. This function enables data and text record-by-record reading and prepares the registered sources for the upcoming import. The prototype is

\begin{verbatim}
int GMT_Begin_IO (void *API, unsigned int family, unsigned int direction,
   unsigned int header);
\end{verbatim}

where \texttt{family} specifies the resource type to be read or written (only \texttt{GMT\_IS\_DATASET} and \texttt{GMT\_IS\_TEXTSET} are available for record-by-record handling). The \texttt{direction} is either \texttt{GMT\_IN} or \texttt{GMT\_OUT}, so for import we obviously use \texttt{GMT\_IN}. The function determines the first input source and sets up procedures for skipping to the next input source in a virtual data set. The \texttt{GMT\_Get\_Record} function will not be able to read any data before \texttt{GMT\_Begin\_IO} has been called. As you might guess, there is a companion \texttt{GMT\_End\_IO} function that completes, then disables record-by-record data access. You can use these several times to switch modes between registering data resources, doing the importing/exporting, and disabling further data access, perhaps to do more registration. We will discuss \texttt{GMT\_End\_IO} once we are done with the data import. The final \texttt{header} argument determines if the common header-block should be written during initialization; choose between \texttt{GMT\_HEADER\_ON} and \texttt{GMT\_HEADER\_OFF}. The function returns 1 if there is an error; otherwise it returns 0.

### 7.7.2 Set data geometry

Typically only done for output data written record by record, we designate the data set’s geometry by calling

\begin{verbatim}
int _GMT_Set_Geometry (void *API, unsigned int direction, unsigned int geometry);
\end{verbatim}

where \texttt{direction} is either \texttt{GMT\_IN} or \texttt{GMT\_OUT} and \texttt{geometry} sets the geometry that will be produced (or read).

### 7.7.3 Importing a data record

If your program will read data table records one-by-one you must first enable this input mechanism with \texttt{GMT\_Begin\_IO} and then read the records within a loop, repeatedly using

\begin{verbatim}
void *GMT_Get_Record (void *API, unsigned int mode, int *nfields);
\end{verbatim}

where the returned value is either a pointer to a double array with the current row values or to a character string with the current row, depending on \texttt{mode}. In either case these pointers point to ephemeral memory internal to GMT and should be considered read-only. When we reach end-of-file, encounter conversion problems, read header comments, or identify segment headers we instead return a NULL pointer. The
nfields integer pointer will return the number of fields returned; pass NULL if your program should ignore this information.

Normally (i.e., mode = GMT_READ_DATA), we return a pointer to a double array. To read text records, supply instead mode = GMT_READ_TEXT and we will return a pointer to the text record. However, if you have input records that mixes organized floating-point columns with text items you could pass mode = GMT_READ_MIXED. Then, GMT will attempt to extract the floating-point values from as many columns as needed; you can still access the original record string, as discussed below. Finally, if your application needs to be notified when GMT closes one file and opens the next, add GMT_FILE_BREAK to mode and check for the status code GMT_IO_NEXT_FILE (by default, we treat the concatenation of many input files as a single virtual file). Using GMT_Get_Record requires you to first initialize the source(s) with GMT_Init_IO. For certain records, GMT_Get_Record will return NULL and sets status codes that your program will need to examine to take appropriate response. Table IO-status lists the various status codes you can check for, using the GMT_Status_IO function (see next section).

### 7.7.4 Examining record status

Programs that read record-by-record must be aware of what the current record represents. Given the presence of headers, data gaps, NaN-record, etc., the developer may want to check the status after reading the current record. The internal i/o status mode can be interrogated with the function

```c
int GMT_Status_IO (void *API, unsigned int mode);
```

which returns 0 (false) or 1 (true) if the current status is reflected by the specified mode. There are 11 different modes available to programmers; for a list see Table IO-status. For an example of how these may be used, see the test program testgmtio.c. Developers who plan to import data on a record-by-record basis may also consult the source code of, say, blockmean or pstext, to see examples of working code.

<table>
<thead>
<tr>
<th>mode</th>
<th>description and return value</th>
</tr>
</thead>
<tbody>
<tr>
<td>GMT_IO_DATA_RECORD</td>
<td>1 if we read a data record</td>
</tr>
<tr>
<td>GMT_IO_TABLE_HEADER</td>
<td>1 if we read a table header</td>
</tr>
<tr>
<td>GMT_IO_SEGMENT_HEADER</td>
<td>1 if we read a segment header</td>
</tr>
<tr>
<td>GMT_IO_ANY_HEADER</td>
<td>1 if we read either header record</td>
</tr>
<tr>
<td>GMT_IO_MISMATCH</td>
<td>1 if we read incorrect number of columns</td>
</tr>
<tr>
<td>GMT_IO_EOF</td>
<td>1 if we reached the end of the file (EOF)</td>
</tr>
<tr>
<td>GMT_IO_NAN</td>
<td>1 if we only read NaNs</td>
</tr>
<tr>
<td>GMT_IO_GAP</td>
<td>1 if this record implies a data gap</td>
</tr>
<tr>
<td>GMT_IO_NEW_SEGMENT</td>
<td>1 if we enter a new segment</td>
</tr>
<tr>
<td>GMT_IO_LINE_BREAK</td>
<td>1 if we encountered a segment header, EOF, NaNs or gap</td>
</tr>
<tr>
<td>GMT_IO_NEXT_FILE</td>
<td>1 if we finished one file but not the last</td>
</tr>
</tbody>
</table>

The various modes used to test the status of the record-by-record machinery.

### 7.7.5 Importing a grid row

If your program must read a grid file row-by-row you must first enable row-by-row reading with GMT_Read_Data and then use the GMT_Get_Row function in a loop; the prototype is
int GMT_Get_Row (void *API, int row_no, struct GMT GRID *G, float *row);

where row is a pointer to a pre-allocated single-precision array to receive the current row, G is the grid in question, and row_no is the number of the current row to be read. Note this value is only considered if the row-by-row mode was initialized with GMT GRID ROW BY ROW MANUAL. The user must allocate enough space to hold the entire row in memory.

### 7.7.6 Disable Data Import

Once the record-by-record input processing has completed we disable further input to prevent accidental reading from occurring (due to poor program structure, bugs, etc.). We do so by calling GMT End IO. This function disables further record-by-record data import; its prototype is

```c
int GMT_End_IO (void *API, unsigned int direction, unsigned int mode);
```

and we specify direction = GMT IN. At the moment, mode is not used. This call will also reallocate any arrays obtained into their proper lengths. The function returns 1 if there is an error (whose code is passed back with API->error), otherwise it returns 0 (GMT NOERROR).

### 7.8 Manipulate data

Once you have created and allocated empty resources, or read in resources from the outside, you may wish to manipulate their contents. This section discusses how to set up loops and access the important variables for each of the supported families. For grids and images it may in addition be required to determine what the coordinates are at each node point. This information can be obtained via arrays of coordinates for each dimension, obtained by

```c
double *GMT_Get_Coord (void *API, unsigned int family, unsigned int dim, void *data);
```

where family must be GMT IS GRID or GMT IS DATASET, dim is either GMT IS X or GMT IS Y, and data is the grid or image pointer. This function will be used below in our example on grid manipulation.

Another aspect of dealing with grids and images is to convert a row and column 2-D reference to our 1-D array index. Because of grid and image boundary padding the relationship is not straightforward, hence we supply

```c
int64_t GMT_Get_Index (struct GMT GRID_HEADER *header, int row, int col);
```

where the header is the header of either a grid or image, and row and col is the 2-D position in the grid or image. We return the 1-D array position; again this function is used below in our example. Likewise, for images with many layers we also define

```c
int64_t GMT_Get_Pixel (struct GMT GRID_HEADER *header, int row, int col, int layer);
```

where the header is the header of an image, and row, col and layer (= 1 for grids) is the position in the grid or image.
7.8.1 Manipulate grids

Most applications wishing to manipulate grids will want to loop over all the nodes, typically in a manner organized by rows and columns. In doing so, the coordinates at each node may also be required for a calculation. Below is a snippet of code that shows how to do visit all nodes in a grid and assign each node the product $x \times y$:

```c
int row, col, node;
double *x_coord = NULL, *y_coord = NULL;
/*... create a grid G or read one ... */
x_coord = GMT_Get_Coord (API, GMT_IS_GRID, GMT_X, G);
y_coord = GMT_Get_Coord (API, GMT_IS_GRID, GMT_Y, G);
for (row = 0; row < G->header->n_rows) {
    for (col = 0; col < G->header->n_columns; col++) {
        node = GMT_Get_Index (G->header, row, col);
        G->data[node] = x_coord[col] * y_coord[row];
    }
}
```

Note the use of `GMT_Get_Index` to get the grid node number associated with the `row` and `col` we are visiting. Because GMT grids have padding (for boundary conditions) the relationship between rows, columns, and node indices is more complicated and hence we hide that complexity in `GMT_Get_Index`. Note that for trivial procedures such setting all grid nodes to a constant (e.g., -9999.0) where the row and column does not enter you can instead do a single loop:

```c
int node;
/*... create a grid G or read one ... */
for (node = 0; node < G->header->size) G->data[node] = -9999.0;
```

Note we must use `G->header->size` (size of allocated array) and not `G->header->nm` (number of nodes in grid) since the latter is smaller due to the padding and a single loop like the above treats the pad as part of the “inside” grid. Replacing `size` by `nm` would be a bug.

7.8.2 Manipulate data tables

Another common application is to process the records in a data table. Because GMT considers the GMT_DATASET resources to contain one or more tables, each of which may contain one or more segments, all of which may contain one or more columns, you will need to have multiple nested loops to visit all entries. The following code snippet will visit all data records and add 1 to all columns beyond the first two (x and y):

```c
uint64_t tbl, seg, row, col;
struct GMT_DATATABLE *T = NULL;
struct GMT_DATASEGMENT *S = NULL;
/*... create a dataset D or read one ... */
for (tbl = 0; tbl < D->n_tables; tbl++) {
    /* Convenient shorthand for current table */
    T = D->table[tbl];
    for (seg = 0; seg < T->n_segments; seg++) {
        /* Convenient shorthand for current segment */
        S = T->segment[seg];
        for (row = 0; row < S->n_rows; row++) {
            /* For all rows in segment */
            for (col = 2; col < T->n_columns; col++) {
                /* For all cols > 1 */
                S->data[col][row] += 1.0; /* Just add one */
            }
        }
    }
}
```
7.8.3 Manipulate text tables

When the data files contain text mixed in with numbers you must open the file as a GMT_TEXTSET and do your own parsing of the data records. The following code snippet will visit all text records and print them out with some counters:

```c
uint64_t tbl, seg, row, col;
struct GMT_TEXTTABLE *T = NULL;
struct GMT_TEXTSEGMENT *S = NULL;

/* ... create a textset D or read one ... */
for (tbl = 0; tbl < D->n_tables; tbl++) {
    /* For each table */
    T = D->table[tbl]; /* Convenient shorthand for current table */
    for (seg = 0; seg < T->n_segments; seg++) { /* For all segments */
        S = T->segment[seg]; /* Convenient shorthand for current segment */
        for (row = 0; row < S->n_rows; row++) { /* For each text record */
            printf ("T=%d S=%d R=%d : %s \n", tbl, seg, row, S->data[row]);
        }
    }
}
```

7.9 Message and Verbose Reporting

The API provides two functions for your program to present information to the user during the run of the program. One is used for messages that are always written (optionally with a time stamp) while the other is used for reports whose verbosity level must exceed the verbosity settings specified via -V.

7.9.1 Verbose reporting

```c
int GMT_Report (void *API, unsigned int level, const char *message, ...);
```

This function takes a verbosity level and a multi-part message (e.g., a format statement and zero or more variables as required by the format string). The verbosity level is an integer in the 0–5 range; these levels are listed in Table timemodes. You assign an appropriate verbosity level to your message, and depending on the chosen run-time verbosity level set via -V your message may or may not be reported. Only messages whose stated verbosity level is lower or equal to the -V level will be printed. These messages are typically progress reports, etc., and are sent to standard error.

<table>
<thead>
<tr>
<th>constant</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GMT_MSG_QUIET</td>
<td>No messages whatsoever</td>
</tr>
<tr>
<td>GMT_MSG_NORMAL</td>
<td>Default output, e.g., warnings and errors only</td>
</tr>
<tr>
<td>GMT_MSG_COMPAT</td>
<td>Compatibility warnings</td>
</tr>
<tr>
<td>GMT_MSG_VERBOSE</td>
<td>Verbose level</td>
</tr>
<tr>
<td>GMT_MSG_LONG_VERBOSE</td>
<td>Longer verbose</td>
</tr>
<tr>
<td>GMT_MSG_DEBUG</td>
<td>Debug messages for developers mostly</td>
</tr>
</tbody>
</table>

The different levels of verbosity that can be selected.

7.9.2 User messages

For custom messages to the user that should always be printed, we use
This function always prints its message to the standard output. Use the `mode` value to control if a time stamp should preface the message, and if selected how the time information should be formatted. See Table `timemodes` for the various modes.

<table>
<thead>
<tr>
<th>constant</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GMT_TIME_NONE</td>
<td>Display no time information</td>
</tr>
<tr>
<td>GMT_TIME_CLOCK</td>
<td>Display current local time</td>
</tr>
<tr>
<td>GMT_TIME_ELAPSED</td>
<td>Display elapsed time since last reset</td>
</tr>
<tr>
<td>GMT_TIME_RESET</td>
<td>Reset the elapsed time to 0</td>
</tr>
</tbody>
</table>

The different types of message modes.

### 7.10 Special GMT modules

There are some differences between calling modules on the command line and using them via the API. These are discussed here.

#### 7.10.1 API-only modules

There are two general-purpose modules that are not part of the command-line version of GMT. These are the read and write modules. Both take an option to specify what GMT resource is being read or written: `-Tc|d|g|i|p|t`, which selects CPT, dataset, grid, image, PostScript, or textset, respectively. In addition both modules accept the `infile` and `outfile` argument for source and destination. These may be actual files of memory locations, of course.

#### 7.10.2 PostScript Access

The GMT module `psconvert` is normally given one or more PostScript files that may be converted to other formats. When accessed by the API it may also be given the special file name `="`, which means we are to use the internal PostScript string produced by the latest GMT plotting instead of any actual file name. The module can access this string which must be a complete plot (i.e., it must have header, middle, and trailer and thus be a valid PostScript file). This allows the API to convert plots to a suitable image format without any duplication and manipulation of the PostScript itself.

### 7.11 Adjusting headers and comments

All header records in incoming datasets are stored in memory. You may wish to replace these records with new information, or append new information to the existing headers. This is achieved with

```c
int GMT_Set_Comment (void *API, unsigned int family, unsigned int mode, void *arg, void *data);
```

Again, `family` selects which kind of resource is passed via `data`. The `mode` determines what kind of comment is being considered, how it should be included, and in what form the comment passed via `arg` is provided. Table `comments` lists the available options, which may be combined by adding (bitwise...
“or”). The `GMT_Set_Comment` function does not actually output anything but sets the relevant comment and header records in the relevant structure. When a file is written out the information will be output as well (Note: Users can always decide if they wish to turn header output on or off via the common GMT option `-h`). For record-by-record writing you must enable the header block output when you call `GMT_Begin_IO`.

<table>
<thead>
<tr>
<th>constant</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GMT_COMMENT_IS_TEXT</td>
<td>Comment is a text string</td>
</tr>
<tr>
<td>GMT_COMMENT_IS_OPTION</td>
<td>Comment is a linked list of GMT_OPTION structures</td>
</tr>
<tr>
<td>GMT_COMMENT_IS_COMMAND</td>
<td>Comment is the command</td>
</tr>
<tr>
<td>GMT_COMMENT_IS_REMARK</td>
<td>Comment is the remark</td>
</tr>
<tr>
<td>GMT_COMMENT_IS_TITLE</td>
<td>Comment is the title</td>
</tr>
<tr>
<td>GMT_COMMENT_IS_NAME_X</td>
<td>Comment is the x variable name (grids only)</td>
</tr>
<tr>
<td>GMT_COMMENT_IS_NAME_Y</td>
<td>Comment is the y variable name (grids only)</td>
</tr>
<tr>
<td>GMT_COMMENT_IS_NAME_Z</td>
<td>Comment is the z variable name (grids only)</td>
</tr>
<tr>
<td>GMT_COMMENT_IS_COLNAMES</td>
<td>Comment is the column names header</td>
</tr>
<tr>
<td>GMT_COMMENT_IS_RESET</td>
<td>Comment replaces existing information</td>
</tr>
</tbody>
</table>

The modes for setting various comment types.

The named modes (`command`, `remark`, `title`, `name_x,y,z` and `colnames`) are used to distinguish regular text comments from specific fields in the header structures of the data resources, such as `GMT_GRID`. For the various table resources (e.g., `GMT_DATASET`) these modifiers result in a specially formatted comments beginning with “Command: ” or “Remark: “, reflecting how this type of information is encoded in the headers.

### 7.12 Export Data Sets

If your program needs to write any of the six recognized data types (CPTs, data tables, text tables, grids, images, or PostScript) you can use the `GMT_Write_Data` function.

Both of these output functions takes a parameter called `mode`. The `mode` parameter generally takes on different meanings for the different data types and will be discussed below. However, one bit setting is common to all types: By default, you are only allowed to write a data resource once; the resource is then flagged to have been written and subsequent attempts to write to the same resource will quietly be ignored. In the unlikely event you need to re-write a resource you can override this default behavior by adding `GMT_IO_RESET` to your `mode` parameter.

#### 7.12.1 Exporting a data set

To have your program accept results from GMT modules and write them separately requires you to use the `GMT_Write_Data` function. It is very similar to the `GMT_Read_Data` function encountered earlier.

**Exporting a data set to a file, stream, or handle**

The prototype for writing to a file (via name, stream, or file handle) is
int GMT_Write_Data (void *API, unsigned int family, unsigned int method, unsigned int geometry, unsigned int mode, double wesn[], void *output, void *data);

- **API**
- **family**
- **method**
- **geometry**
- **mode** – specific to each data type *(see below)*
- **wesn**
- **output** –
- **data** – A pointer to any of the six families.

Return: 0 on success, otherwise return -1 and set API->error to reflect to cause.

where data is a pointer to any of the four structures discussed previously.

**Color palette table** mode controls if the CPT’s back-, fore-, and NaN-colors should be written (1) or not (0).

**Data table** If method is GMT_IS_FILE, then the value of mode affects how the data set is written:

- GMT_WRITE_SET The entire data set will be written to the single file [0].
- GMT_WRITE_TABLE Each table in the data set is written to individual files [1]. You can either specify an output file name that must contain one C-style format specifier for a int variable (e.g., “New_Table_%06d.txt”), which will be replaced with the table number (a running number from 0) or you must assign to each table i a unique output file name via the D->table[i]->file[GMT_OUT] variables prior to calling the function.
- GMT_WRITE_SEGMENT Each segment in the data set is written to an individual file [2]. Same setup as for GMT_WRITE_TABLE except we use sequential segment numbers to build the file names.
- GMT_WRITE_TABLE_SEGMENT Each segment in the data set is written to an individual file [3]. You can either specify an output file name that must contain two C-style format specifiers for two int variables (e.g., “New_Table_%06d_Segment_%03d.txt”), which will be replaced with the table and segment numbers, or you must assign to each segment j in each table i a unique output file name via the D->table[i]->segment[j]->file[GMT_OUT] variables prior to calling the function.
- GMT_WRITE_OGR Writes the dataset in OGR/GMT format in conjunction with the -a setting [4].

**Text table** The mode is used the same way as for data tables.

**GMT grid** Here, mode may be GMT_CONTAINER_ONLY to only update a file’s header structure, but normally it is simply GMT_CONTAINER_AND_DATA so the entire grid and its header will be exported (a subset is not allowed during export). However, in the event your data array holds both the real and imaginary parts of a complex data set you must add either GMT_GRID_IS_COMPLEX_REAL or GMT_GRID_IS_COMPLEX_IMAG to mode so as to export the corresponding grid values correctly. Finally, for native binary grids you may skip writing the grid header by adding GMT_GRID_NO_HEADER; this setting is ignored for all other grid formats. If your output grid is huge and you are building it row-by-row, set mode to
You can then write the grid row-by-row using `GMT_Put_Row`. By default the rows will be automatically processed in order. To completely specify which row to be written, use `GMT_GRID_ROW_BY_ROW_MANUAL` instead; this requires a file format that supports direct writes, such as netCDF. Finally, if you are preparing a geographic grid outside of GMT you need to add the mode `GMT_GRID_IS_GEO` to ensure that the proper metadata will be written to the netCDF header, thus letting the grid be recognized as such.

Note: If `method` is `GMT_IS_FILE`, `family` is `GMT_IS_GRID`, and the filename implies a change from NaN to another value then the grid is modified accordingly. If you continue to use that grid after writing please be aware that the changes you specified were applied to the grid.

### 7.13 Record-by-record output

In the case of data and text tables, you may also consider the `GMT_Put_Record` function for record-by-record writing. As a general rule, your program organization may simplify if you can write the entire resource with `GMT_Write_Data`. However, if the program logic is simple or already involves using `GMT_Get_Record`, it may be better to export one data record at the time via `GMT_Put_Record`. For grids there is the corresponding `GMT_Put_Row` function.

#### 7.13.1 Enable Data Export

Similar to the data import procedures, once all output destinations have been registered, we signal the API that we are done with the registration phase and are ready to start the actual data export. As for input, this step is only needed when dealing with record-by-record writing. Again, we enable record-by-record writing by calling `GMT_Begin_IO`, this time with `direction = GMT_OUT`. This function enables data export and prepares the registered destinations for the upcoming writing.

**Specifying the number of output columns**

For record-based input/output you will need to specify the number of output columns, unless it equals the number of input columns. This is done with the `GMT_Set_Columns` function:

```c
void *GMT_Set_Columns (void *API, unsigned int n_columns, unsigned int mode);
```

The `n_columns` is a number related to the number of output columns you plan to write, while `mode` controls what that number means. Here, `mode = GMT_COL_FIX` means it is the actual number of output columns, `mode = GMT_COL_ADD` means it should be added to the known number of input columns to arrive at the number of final output columns, while `mode = GMT_COL_SUB` means this value should be subtracted from the number of input columns to find the number of output columns.

#### 7.13.2 Exporting a data record

If your program must write data table records one-by-one you must first enable record-by-record writing with `GMT_Begin_IO` and then use the `GMT_Put_Record` function in a loop; the prototype is

```c
int GMT_Put_Record (void *API, unsigned int mode, void *rec);
```

where `rec` is a pointer to either (a) a double-precision array with the current row. Then, `rec` is expected to hold at least as many items as the current output column setting, which represents the number of columns in the output destination. Alternatively (b), `rec` points to a text string. The `mode` parameter
must be set to reflect what is passed. Using \texttt{GMT\_Put\_Record} requires you to first initialize the destination with \texttt{GMT\_Init\_IO}. Note that for families \texttt{GMT\_IS\_DATASET} and \texttt{GMT\_IS\_TEXTSET} the methods \texttt{GMT\_IS\_DUPLICATE} and \texttt{GMT\_IS\_REFERENCE} are not supported since you can simply populate the \texttt{GMT\_DATASET} structure directly. As mentioned, \texttt{mode} affects what is actually written:

\textbf{GMT\_WRITE\_DATA.} Normal operation that builds the current output record from the numerical values in \texttt{rec}.

\textbf{GMT\_WRITE\_TEXT.} For ASCII output mode we write the text string \texttt{rec}. If \texttt{rec} is \texttt{NULL} then we use the current (last imported) text record. If binary output mode we quietly skip writing this record.

\textbf{GMT\_WRITE\_TABLE\_HEADER.} For ASCII output mode we write the text string \texttt{rec}. If \texttt{rec} is \texttt{NULL} then we write the last read header record (and ensures it starts with \#). If binary output mode we quietly skip writing this record.

\textbf{GMT\_WRITE\_SEGMENT\_HEADER.} For ASCII output mode we use the text string \texttt{rec} as the segment header. If \texttt{rec} is \texttt{NULL} then we use the current (last read) segment header record. If binary output mode instead we write a record composed of NaNs.

The function returns \texttt{1} if there was an error associated with the writing (which is passed back with \texttt{API->error}), otherwise it returns \texttt{0} (\texttt{GMT\_NOERROR}).

\subsection{Exporting a grid row}

If your program must write a grid file row-by-row you must first enable row-by-row writing with \texttt{GMT\_Read\_Data} and then use the \texttt{GMT\_Put\_Row} function in a loop; the prototype is

\begin{verbatim}
int GMT_Put_Row (void *API, int row_no, struct GMT_GRID *G, float *row);
\end{verbatim}

where \texttt{row} is a pointer to a single-precision array with the current row, \texttt{G} is the grid in question, and \texttt{row_no} is the number of the current row to be written. Note this value is only considered if the row-by-row mode was initialized with \texttt{GMT\_GRID\_ROW\_BY\_ROW\_MANUAL}.

\subsection{Disable Data Export}

Once the record-by-record output has completed we disable further output to prevent accidental writing from occurring (due to poor program structure, bugs, etc.). We do so by calling \texttt{GMT\_End\_IO}. This function disables further record-by-record data export; here, we obviously pass \texttt{direction} as \texttt{GMT\_OUT}.

\subsection{Destroy allocated resources}

If your session imported any data sets into memory then you may explicitly free this memory once it is no longer needed and before terminating the session. This is done with the \texttt{GMT\_Destroy\_Data} function, whose prototype is

\begin{verbatim}
int GMT_Destroy_Data (void *API, void *data);
\end{verbatim}

where \texttt{data} is the address of the pointer to a data container, i.e., not the pointer to the container but the \texttt{address} of that pointer (e.g. \&pointer). Note that when each module completes it will automatically free
memory created by the API; similarly, when the session is destroyed we also automatically free up memory. Thus, `GMT_Destroy_Data` is therefore generally only needed when you wish to directly free up memory to avoid running out of it. The function returns 1 if there is an error when trying to free the memory (the error code is passed back with `API->error`), otherwise it returns 0 (`GMT_NOERROR`).

### 7.15 Destroy groups of allocated resources

If you obtained an array of resources via `GMT_Read_Group` then you will need to destroy these resources with `GMT_Destroy_Group` instead, whose prototype is:

```c
int GMT_Destroy_Group (void *API, void *data, unsigned int n);
```

where `data` is the address of the array with data containers, i.e., not the array to the containers but the address of that array (e.g., `&array`), and `n` is the number of containers.

### 7.16 Terminate a GMT session

Before your program exits it should properly terminate the GMT session, which involves a call to:

```c
int GMT_Destroy_Session (void *API);
```

which simply takes the pointer to the GMT API control structure as its only arguments. It terminates the GMT machinery and deallocates all memory used by the GMT API book-keeping. It also unregisters any remaining resources previously registered with the session. The GMT API will only close files that it was responsible for opening in the first place. Finally, the API structure itself is freed so your main program does not need to do so. The function returns 1 if there is an error when trying to free the memory (the error code is passed back with `API->error`), otherwise it returns 0 (`GMT_NOERROR`).

### 7.17 Presenting and accessing GMT options

As you develop a program you may wish to rely on some of the GMT common options. For instance, you may wish to have your program present the `-R` option to the user, let GMT handle the parsing, and examine the values. You may also wish to encode your own custom options that may require you to parse user text into the corresponding floating point dimensions, constants, coordinates, absolute time, etc. The API provides several functions to simplify these tedious parsing tasks. This section is intended to show how the programmer will obtain information from the user that is necessary to do the task at hand (e.g., special options to provide values and settings for the program). In the following section we will concern ourselves with preparing arguments for calling any of the GMT modules.

#### 7.17.1 Display usage syntax for GMT common options

You can have your program menu display the standard usage message for a GMT common option by calling the function:

```c
int GMT_Option (void *API, const char *options);
```

where `options` is a comma-separated list of GMT common options (e.g., “R,J,O,X”). You can repeat this function with different sets of options in order to intersperse your own custom options within an overall alphabetical order; see any GMT module for examples of typical layouts.
7.17.2 Parsing the GMT common options

The parsing of all GMT common option is done by on call to

```c
int GMT_Parse_Common (void *API, const char *args, struct GMT_OPTION *list);
```

where `args` is a string of the common GMT options your program is allowed to use. An error will be reported if any of the common GMT options fail to parse, and if so we return 1; if no errors we return 0. All other options, including file names, will be silently ignored. The parsing will update the internal GMT information structure that affects module operations.

7.17.3 Inquiring about the GMT common options

The API provide only a limited window into the full GMT machinery accessible to the modules. You can determine if a particular common option has been parsed and in some cases determine the values that were set with

```c
int GMT_Get_Common (void *API, unsigned int option, double *par);
```

where `option` is a single option character (e.g., ‘R’) and `par` is a double array with at least a length of 6. If the particular option has been parsed then the function returns the number of parameters passed back via `par`; otherwise we return -1. For instance, to determine if the -R was set and to obtain the specified region you may call

```c
if (GMT_Get_Common (API, 'R', wesn)) != -1) {
    /* wesn now contains the boundary information */
}
```

The `wesn` array could now be passed to the various read and create functions for GMT resources.

7.17.4 Parsing text values

Your program may need to request values from the user, such as distances, plot dimensions, coordinates, date/time strings and other data. The conversion from such text to actual distances, taking units into account, is tedious to program. You can simplify this by using

```c
int GMT_Get_Values (void *API, const char *arg, double par[], int maxpar);
```

where `arg` is the text item with one or more values that are separated by commas, spaces, tabs, semicolons, or slashes, and `par` is an array of length `maxpar` long enough to hold all the items you are parsing. The function returns the number of items parsed with a maximum of `maxpar`, or -1 if there is an error. For instance, assume the character string `origin` was given by the user as two geographic coordinates separated by a slash (e.g., "35:45W/19:30:55.3S"). We obtain the two coordinates in decimal degrees by calling

```c
n = GMT_Get_Values (API, origin, pair, 2);
```

Your program can now check that `n` equals 2 and then use the values in `pair` separately. Note: Dimensions given with units of inches, cm, or points are converted to the current default unit set via `PROJ_LENGTH_UNIT`, while distances given in km, nautical miles, miles, feet, or survey feet are returned in meters. Arc lengths in minutes and seconds are returned in decimal degrees, and date/time values are returned in seconds since the current epoch [1970].
7.17.5 Get or set an API or GMT default parameter

If your program needs to determine one or more of the current API or GMT default settings you can do so via

```c
int GMT_Get_Default (void *API, const char *keyword, char *value);
```

where `keyword` is one such keyword (e.g., PROJ_LENGTH_UNIT) and `value` must be a character string long enough to hold the answer. In addition to the long list of GMT defaults you can also inquire about the API parameters `API_PAD` (the current pad setting), `API_IMAGE_LAYOUT` (the order and structure of image memory storage), `API_GRID_LAYOUT` (order of grid memory storage), `API_VERSION` (the API version string), `API_CORES` (the number of cores seen by the API), `API_BINDIR` (the API (GMT) executable path), `API_SHAREDIR` (the API (GMT) shared directory path), `API_DATADIR` (the API (GMT) data directory path), and `API_PLUGINDIR` (the API (GMT) plugin path). Depending on what parameter you selected you could further convert it to a numerical value with `GMT_Get_Values` or just use it in a text comparison.

To change any of the API or GMT default settings programmatically you would use

```c
int GMT_Set_Default (void *API, const char *keyword, const char *value);
```

where as before `keyword` is one such keyword (e.g., PROJ_LENGTH_UNIT) and `value` must be a character string with the new setting. Note that all settings must be passed as text strings even if many are inherently integers or floats.

For indexed access to custom grids and images we may need to know the internal matrix layout. You can change this information via

```c
int64_t GMT_Set_Index (struct GMT_GRID_HEADER *header, char *code);
```

where the `header` is the header of either a grid or image, and `code` is a three-character code indication ...

7.18 Call a module

One of the advantages of programming with the API is that you have access to the high-level GMT modules. For example, if your program must compute the distance from a node to all other nodes in the grid then you can simply set up options and call `grdmath` to do it for you and accept the result back as an input grid. All the module interfaces are identical and are called via

```c
int GMT_Call_Module (void *API, const char *module, int mode, void *args);
```

Here, `module` is the name of any of the GMT modules, such as `psxy` or `grdvolume`. All GMT modules may be called with one of three sets of `args` depending on `mode`. The three modes differ in how the options are passed to the module:

- `mode = GMT_MODULE_EXIST`. Return GMT_NOERROR (0) if the module exists, nonzero otherwise.
- `mode = GMT_MODULE_PURPOSE`. Just print the one-line purpose of the module; `args` must be NULL.
- `mode = GMT_MODULE_LIST`. Just prints a list of all modules (including those given as plugins); `args` must be NULL.
mode = GMT_MODULE_OPT. Expects args to be a pointer to a doubly-linked list of objects with individual options for the current program. We will see how API functions can help prepare and maintain such lists.

mode = GMT_MODULE_CMD. Expects args to be a single text string with all needed options.

mode > 0. Expects args to be an array of text strings and mode to be a count of how many options are passed (i.e., the argc, argv[] model used by the GMT programs themselves).

7.18.1 Set program options via text array arguments

When mode > 0 we expect an array args of character strings that each holds a single command line option (e.g., “-R120:30/134:45/8S/3N”) and interpret mode to be the count of how many options are passed. This, of course, is almost exactly how the stand-alone GMT programs are called (and reflects how they themselves are activated internally). We call this the “argc-argv” mode. Depending on how your program obtains the necessary options you may find that this interface offers all you need.

7.18.2 Set program options via text command

If mode = 0 then args will be examined to see if it contains several options within a single command string. If so we will break these into separate options. This is useful if you wish to pass a single string such as “-R120:30/134:45/8S/3N -JM6i mydata.txt -Sc0.2c”. We call this the “command” mode and it is extensively used by the modules themselves.

7.18.3 Set program options via linked structures

The third, linked-list interface allows developers using higher-level programming languages to pass all command options via a pointer to a NULL-terminated, doubly-linked list of option structures, each containing information about a single option. Here, instead of text arguments we pass the pointer to the linked list of options mentioned above, and mode must be passed as GMT_MODULE_OPT. Using this interface can be more involved since you need to generate the linked list of program options; however, utility functions exist to simplify its use. This interface is intended for programs whose internal workings are better suited to generate such arguments – we call this the “options” mode. The order in the list is not important as GMT will sort it internally according to need. The option structure is defined below.

```c
struct GMT_OPTION {
    char option; /* Single option character (e.g., 'G' for -G) */
    char *arg; /* String with arguments (NULL if not used) */
    struct GMT_OPTION *next; /* Next option pointer (NULL for last option) */
    struct GMT_OPTION *prev; /* Previous option (NULL for first option) */
};
```

7.18.4 Convert between text and linked structures

To assist programmers there are also two convenience functions that allow you to convert between the two argument formats. They are

```c
struct GMT_OPTION *GMT_Create_Options (void *API, int argc, void *args);
```

7.18. Call a module
This function accepts your array of text arguments (cast via a void pointer), allocates the necessary space, performs the conversion, and returns a pointer to the head of the linked list of program options. However, in case of an error we return a NULL pointer and set API->error to indicate the nature of the problem. Otherwise, the pointer may now be passed to the relevant GMT module. Note that if your list of text arguments were obtained from a C main() function then argv[0] will contain the name of the calling program. To avoid passing this as a bad file name option, call GMT_Create_Options with argc-1 and argv+1 instead. If you wish to pass a single text string with multiple options (in lieu of an array of text strings), then pass argc = 0. When no longer needed you can remove the entire list by calling

```c
int GMT_Destroy_Options (void *API, struct GMT_OPTION **list);
```

The function returns 1 if there is an error (which is passed back with API->error), otherwise it returns 0 (GMT_NOERROR).

The inverse function prototype is

```c
char **GMT_Create_Args (void *API, int *argc, struct GMT_OPTION *list);
```

which allocates space for the text strings and performs the conversion; it passes back the count of the arguments via argc and returns a pointer to the text array. In the case of an error we return a NULL pointer and set API->error to reflect the error type. Note that argv[0] will not contain the name of the program as is the case the arguments presented by a C main() function. When you no longer have any use for the text array, call

```c
int GMT_Destroy_Args (void *API, int argc, char **argv);
```

to deallocate the space used. This function returns 1 if there is an error (which is passed back with API->error), otherwise it returns 0 (GMT_NOERROR).

Finally, to convert the linked list of option structures to a single text string command, use

```c
char *GMT_Create_Cmd (void *API, struct GMT_OPTION *list);
```

Developers who plan to import and export GMT shell scripts might find it convenient to use these functions. In case of an error we return a NULL pointer and set API->error, otherwise a pointer to an allocated string is returned. When you no longer have any use for the text string, call

```c
int GMT_Destroy_Cmd (void *API, char **string);
```

to deallocate the space used. This function returns 1 if there is an error (which is passed back with API->error), otherwise it returns 0 (GMT_NOERROR).

### 7.18.5 Manage the linked list of options

Several additional utility functions are available for programmers who wish to manipulate program option structures within their own programs. These allow you to create new option structures, append them to the linked list, replace existing options with new values, find a particular option, and remove options from the list. Note: The order in which the options appear in the linked list is of no consequence to GMT. Internally, GMT will sort and process the options in the manner required. Externally, you are free to maintain your own order.
Make a new option structure

`GMT_Make_Option` will allocate a new option structure, assign values given the option and arg parameters (pass NULL if there is no argument for this option), and return a pointer to the allocated structure. The prototype is

```
struct GMT_OPTION *GMT_Make_Option (void *API, char option, const char *arg);
```

Should memory allocation fail the function will print an error message pass an error code via API->error, and return NULL.

Append an option to the linked list

`GMT_Append_Option` will append the specified option to the end of the doubly-linked list. The prototype is

```
struct GMT_OPTION *GMT_Append_Option (void *API, struct GMT_OPTION *option, struct GMT_OPTION *list);
```

We return the list back, and if list is given as NULL we return option as the start of the new list. Any errors result in a NULL pointer with API->error holding the error type.

Find an option in the linked list

`GMT_Find_Option` will return a pointer ptr to the first option in the linked list starting at list whose option character equals option. If not found we return NULL. While this is not necessarily an error we still set API->error accordingly. The prototype is

```
struct GMT_OPTION *GMT_Find_Option (void *API, char option, struct GMT_OPTION *list);
```

If you need to look for multiple occurrences of a certain option you will need to call `GMT_Find_Option` again, passing the option following the previously found option as the list entry, i.e.,

```
list = *ptr->next;
```

Update an existing option in the list

`GMT_Update_Option` will replace the argument of current with the new argument arg and otherwise leave the option at its place in the list. The prototype is

```
int GMT_Update_Option (void *API, struct GMT_OPTION *current, const char *arg);
```

An error will be reported if (a) current is NULL or (b) arg is NULL. The function returns 1 if there is an error, otherwise it returns 0 (GMT_NOERROR).

Delete an existing option in the linked list

You may use `GMT_Delete_Option` to remove the current option from the linked list. The prototype is

```
void GMT_Delete_Option (void *API, struct GMT_OPTION *current);
``"
int GMT_Delete_Option (void *API, struct GMT_OPTION *current, struct GMT_OPTION **head);

We return 1 if the option is not found in the list and set API->error accordingly. Note: Only the first occurrence of the specified option will be deleted. If you need to delete all such options you will need to call this function in a loop until it returns a non-zero status.

Specify a file via a linked option

To specify an input file name via an option, simply use < as the option (this is what GMT_Create_Options does when it finds filenames on the command line). Likewise, > can be used to explicitly indicate an output file. In order to append to an existing file, use ). For example the following command would read from file.A and append to file.B:

```
```

These options also work on the command line but usually one would have to escape the special characters <, > and ) as they are normally used for file redirection.

Encode option arguments for external interfaces

Developers writing interfaces between GMT and external platforms such as other languages (Python, Java, Julia, etc.) or tools (MATLAB, Octave, etc.) need to manipulate linked options in a special way. For instance, a GMT call in the MATLAB or Octave application might look like

```
table = gmt('blockmean -R30W/30E/10S/10N -I2m', [x y z]);
grid = gmt('surface -R -I2m -Lu', table, high_limit_grid);
grid2 = gmt('grdmath ? LOG10 ? MUL', grid, grid);
```

Most of the time our implicit rules will take care of the ordering. The rule says that all required input data items must be listed before any secondary input data items, and all primary output items must be listed on the left hand side before any secondary output items. There are three situations where the parsing will need further help; (1) Specifying the positions of memory arguments given to gmtmath, (2) specifying the positions of memory arguments given to grdmath, and (3) using -R? when passing a memory grid to the -R option (since just -R means use the previous region in the command history). Thus, in the gmtmath call we we needed to specify where the specific arguments should be placed among the operators. API developers will rely on GMT_Open_VirtualFile to convert the above syntax to correct options for GMT_Call_Module. The prototype is

```
struct GMTRESOURCE *GMT_Encode_Options (void *API, const char *module, int n_in, struct GMT_OPTION **head, int *n_items);
```

where module is the name of the module whose linked options are pointed to by *head, n_in contains the number of input objects we have to connect (or -1 if not known) and we return an array that contains specific information for those options that (after processing) contain explicit memory references. The number of items in the array is returned via the n_items variable. The function returns NULL if there are errors and sets API->error to the corresponding error number. The GMT_RESOURCE structure is defined below:

```
struct GMTRESOURCE {
    /* Information for passing external resources */
    enum GMT_enum_family family; /* GMT data family */
    enum GMT_enum_geometry geometry; /* One of the recognized GMT geometries */
}
```

(continues on next page)
API developers will need to provide specific code to handle the registration of native structures in their language or application and to translate between the GMT resources and the corresponding native items. Developers should look at an existing and working interface such as the GMT/MATLAB toolbox to see the required steps.

**Expand an option with explicit memory references**

When the external tool or application knows the name of the special file names used for memory references the developer should replace the place-holder ? character in any option string with the actual reference name. This is accomplished by calling `GMT_Expand_Option`, with prototype

```c
int GMT_Expand_Option (void *API, struct GMT_OPTION *option, const char *name);
```

where `option` is the current option and `name` is the special file name for the memory reference.
The GMT FFT Interface

While the i/o options presented so far lets you easily read in a data table or grid and manipulate them, if you need to do the manipulation in the wavenumber domain then this chapter is for you. Here, we outline how to take the Fourier transform of such data, perform calculations in the wavenumber domain, and take the inverse transform before writing the results. To assist programmers we also distribute fully functioning demonstration programs that takes you through the steps we are about to discuss; these demo programs may be used as your starting point for further development and can be found in the gmt-custom repository.

8.1 Presenting and parsing the FFT options

Several GMT programs that use the FFTs present the same unified option and modifier sets to the user. The API makes these available as well. If your program needs to present the FFT option usage you can call

```c
unsigned int GMT_FFT_Option (void *API, char option, unsigned int dim,
                            const char *string);
```

Here, `option` is the unique character used for this particular program option (most GMT programs have standardized on using ‘N’ but you are free to choose whatever letter you want except existing GMT common options). The `dim` sets the dimension of the transform; currently you must choose 1 or 2, while `string` is a one-line message that states what the option does; you should tailor this to your program. If NULL then a generic message is placed instead.

To parse the user’s selection you call

```c
void *GMT_FFT_Parse (void *API, char option, unsigned int dim, const char *arg);
```

which accepts the user’s string option via `arg`; the other arguments are the same as those above. The function returns an opaque pointer to a structure with the chosen parameters.
8.2 Initializing the FFT machinery

Before you can take any transforms you must initialize the FFT machinery. This process involves a series of preparatory steps that are conveniently performed for you by

```c
void *GMT_FFT_Create (void *API, void *X, unsigned int dim,
                      unsigned int mode, void *F);
```

Here, X is either your dataset or grid pointer, dim is the dimension of the transform (1 or 2 only), mode passes various flags to the setup, such as whether the data is real, imaginary, or complex, and F is the opaque pointer previously returned by `GMT_FFT_Parse`. Depending on the option string you passed to `GMT_FFT_Parse`, the data may have a constant level or a trend removed, mirror reflected and extended by various symmetries, padded and tapered to desired transform dimensions, and possibly temporary files are written out before the transform takes place. See the grdfft man page for a full explanation of the options presented by `GMT_FFT_Option`.

8.3 Taking the FFT

Now that everything has been set up you can perform the transform with

```c
void *GMT_FFT (void *API, void *X, int direction, unsigned int mode, void *K);
```

which takes as direction either `GMT_FFT_FWD` or `GMT_FFT_INV`. The mode is used to specify if we pass a real (`GMT_FFT_REAL`) or complex (`GMT_FFT_COMPLEX`) data set, and K is the opaque pointer returned by `GMT_FFT_Create`. The transform is performed in place and returned via X. When done with your manipulations (below) you can call it again with the inverse direction to recover the corresponding space-domain version of your data. The FFT is fully normalized so that calling forward followed by inverse yields the original data set. The information passed via K determines if a 1-D or 2-D transform takes place; the key work is done via `GMT_FFT_1D` or `GMT_FFT_2D`, as explained below.

8.4 Taking the 1-D FFT

A lower-level 1-D FFT is also available via the API, i.e.,

```c
int GMT_FFT_1D (void *API, float *data, uint64_t n, int direction,
                 unsigned int mode);
```

which takes as direction either `GMT_FFT_FWD` or `GMT_FFT_INV`. The mode is used to specify if we pass a real (`GMT_FFT_REAL`) or complex (`GMT_FFT_COMPLEX`) data set, and data is the 1-D data array of length n that we wish to transform. The transform is performed in place and returned via data. When done with your manipulations (below) you can call it again with the inverse direction to recover the corresponding space-domain version of your data. The 1-D FFT is fully normalized so that calling forward followed by inverse yields the original data set.

8.5 Taking the 2-D FFT

A lower-level 2-D FFT is also available via
which takes as direction either GMT_FFT_FWD or GMT_FFT_INV. The mode is used to specify if we pass a real (GMT_FFT_REAL) or complex (GMT_FFT_COMPLEX) data set, and data is the 2-D data array in row-major format, with row length n_columns and column length n_rows. The transform is performed in place and returned via data. When done with your manipulations (below) you can call it again with the inverse direction to recover the corresponding space-domain version of your data. The 2-D FFT is fully normalized so that calling forward followed by inverse yields the original data set.

8.6 Wavenumber calculations

As your data have been transformed to the wavenumber domain you may wish to operate on the various values as a function of wavenumber. We will show how this is done for datasets and grids separately. First, we present the function that returns an individual wavenumber:

$$\text{double GMT_FFT_Wavenumber (void *API, uint64_t k, unsigned int mode, void *K);}$$

where k is the index into the array or grid, mode specifies which wavenumber we want (it is not used for 1-D transform but for the 2-D transform we can select either the x-wavenumber (0), the y-wavenumber (1), or the radial wavenumber (2)), and finally the opaque vector created by GMT_FFT_Create.

8.6.1 1-D FFT manipulation

[To be added after gmtfft has been added as new module, probably in 5.4.]

8.6.2 2-D FFT manipulation

The number of complex pairs in the grid is given by the header’s nm variable, while size will be twice that value as it holds the number of components. To visit all the complex values and obtain the corresponding wavenumber we simply need to loop over size and call GMT_FFT_Wavenumber. This code snippet multiples the complex grid by the radial wavenumber:

```c
uint64_t k;
for (k = 0; k < Grid->header->size; k++) {
    wave = GMT_FFT_Wavenumber (API, k, 2, K);
    Grid->data[k] *= wave;
}
```

Alternatively, you may choose to be more specific about which components are real and imaginary (especially if they are to be treated differently), and set up the loop this way:

```c
uint64_t re, im;
for (re = 0, im = 1; re < Grid->header->size; re += 2, im += 2) {
    wave = GMT_FFT_Wavenumber (API, re, 2, K);
    Grid->data[re] *= wave;
    Grid->data[im] *= 2.0 * wave;
}
```
8.7 Destroying the FFT machinery

When done you terminate the FFT machinery with

```c
double GMT_FFT_Destroy (void *API, void *K);
```

which simply frees up the memory allocated by the FFT machinery with `GMT_FFT_Create`. 
FORTRAN Support

FORTRAN 90 developers who wish to use the GMT API may use the same API functions as discussed in Chapter 2. As we do not have much (i.e., any) experience with modern Fortran we are not sure to what extent you are able to access the members of the various structures, such as the GMT_GRID structure. Thus, this part will depend on feedback and for the time being is to be considered preliminary and subject to change. We encourage you to take contact should you wish to use the API with your Fortran 90 programs.

9.1 FORTRAN 77 Grid i/o

Because of a lack of structure pointers we can only provide a low level of support for Fortran 77. This API is limited to help you inquire, read and write GMT grids directly from Fortran 77. To inquire about the range of information in a grid, use

```c
int gmt_f77_readgrdinfo (unsigned int dim[], double limits[], double inc[],
                        char *title, char *remark, const char *file)
```

where `dim` returns the grid width, height, and registration, `limits` returns the min and max values for x, y, and z as three consecutive pairs, `inc` returns the x and y increments, while the `title` and `remark` return the values of these strings. The `file` argument is the name of the file we wish to inquire about. The function returns 0 unless there is an error. Note that you must declare your variables so that `limits` has at least 6 elements, `inc` has at least 2, and `dim` has at least 4.

To actually read the grid, we use

```c
int gmt_f77_readgrd (float *array, unsigned int dim[], double wesn[],
                    double inc[], char *title, char *remark, const char *file)
```

where `array` is the 1-D grid data array, `dim` returns the grid width, height, and registration, `limits` returns the min and max values for x, y, and z, `inc` returns the x and y increments, and the `title` and `remark` return the values of the corresponding strings. The `file` argument is the name of the file we wish to read from. The function returns 0 unless there is an error. Note on input, `dim[2]` can be set to 1, which means we will allocate the array for you; otherwise we assume space has already been secured.
Also, if $\text{dim}[3]$ is set to 1 we will in-place transpose the array from C-style row-major array order to Fortran column-major array order.

Finally, to write a grid to file you can use

```
int gmt_f77_writegrd(float *array, unsigned int dim[], double wens[], double inc[],
                       const char *title, const char *remark, const char *file)
```

where `array` is the 1-D grid data array, `dim` specifies the grid width, height, and registration, `limits` may be used to specify a subset (normally, just pass zeros), `inc` specifies the x and y increments, while the `title` and `remark` supply the values of these strings. The `file` argument is the name of the file we wish to write to. The function returns 0 unless there is an error. If $\text{dim}[3]$ is set to 1 we will in-place transpose the array from Fortran column-major array order to C-style row-major array order before writing. Note this means `array` will have been transposed when the function returns.
Developers may want to access GMT modules from external programming environments, such as MATLAB, Octave, Julia, Python, R, IDL, etc., etc. These all face similar challenges and hence this section will speak in somewhat abstract terms. Specific language addressing the challenges for some of the above-mentioned environments will follow below.

The C/C++ API for GMT makes it possible to call any of the ~100 core modules, the 40 or so supplemental modules, and any number of custom modules provided via shared libraries (e.g., the gsfml modules). Many of the external interfaces come equipped with methods to call C functions directly. The key challenges pertain to specifying the input to use in the module and to receive what is produced by the module. As we know from GMT command line usage, all GMT modules expect input to be given via input files (or stdin, except for sources like grids and images). Similarly, output will be written to a specified output file (or stdout if the data type supports it). Clearly, external interfaces could do the same thing. The problem is that most of the time we already will have the input data in memory and would prefer the output to be returned back to memory, thus avoiding using temporary files. Here, we will outline the general approach for using the GMT API. We will describe a relatively low-level approach to calling GMT modules. Once such an interface exists it is simpler to build a more flexible and user-friendly layer on top that can handle argument parsing in a form that makes the interface seem more of a natural extension of your external environment than a forced fit to GMT’s command-line heritage. Before we describe the interface it is important to understand that the GMT modules, since the beginning or time, have done the i/o inside the modules. While these steps are helped by i/o library functions, the i/o activities all take place inside the modules. This means that external environments in which the desired input data already reside in memory and the desired results should be returned back to memory pose a trickier challenge. We will see the solution to this involves the concept of virtual files.

### 10.1 Plain interface

While the syntax of your external environment’s language will dictate the details of the implementation, we will in general need to build a function (or class, or method) that allows you to issue a call like this:

\[
\text{[results]} = \text{gmt} (\text{module, options, inputs})
\]
Fig. 1: GMT Modules can read and write information in may ways. The GMT command line modules can only access the methods in white, while all methods are available via the C API. External interfaces will preferentially want the methods in orange.

where results (i.e., objects returned back to memory) is optional and may be one or more items grouped together, depending on language syntax. If no output is required then no left-hand side assignment will be present. Likewise, inputs is optional and may be one or more comma-separated objects present in memory. In most cases, options will be required and this is a string with options very similar to the arguments given on the GMT command line. Finally, module is required since you must specify which one you want to call. The coding of the gmt method, class, or function above may be written entirely in C, partly in C and the external scripting language, or entirely in the scripting language, depending on restrictions on what needs to be done and where this is most easily accomplished. How this is accomplished may vary from environment to environment.

Fig. 2: Data pass in and out of the gmt interface which may be written in the scripting language used by the external interface. The native data will need to be encapsulated by GMT containers and this step may be done by a C parser but could also be done by the gmt interface directly. Either of these communicate directly with the C functions in the GMT API.

10.2 Data containers

The external interface developer will need to create native data classes or structures that are capable of containing the information associated with the 6 GMT objects: data tables, grids, images, color palette tables, PostScript documents, and text tables. In other words, how your external environment will represent these data in memory. Some of these “containers” may already exist, while others may need to be designed. Most likely, you will end up with a set of six containers that can hold the various GMT data objects and related metadata. In addition, it may be convenient to also consider the two GMT helper objects MATRIX and VECTOR, which may be closer to the native representation of your data than, for instance, the native GMT_DATASET.

10.3 Input from memory

Whether input comes from memory or from external files, the call to a GMT module is the same: we have to specify filenames to provide the input data. Thus, the game is to provide virtual file names that represent our in-memory data. The process is relatively simple and may need to be done in a snippet of C code that can be called by a function written in your environments scripting language. The steps go like this:
1. Create a GMT C container marked for input and copy or reference your data provided by your external environment into this container.

2. Open a virtual file using this container to represent the input source.

3. Insert this virtual file name in the appropriate location in the GMT option string. If the module imports data from `stdin` then we can use the hidden option `-<filename`.

When the GMT module is run it will know how to make the connections between the virtual file names and the actual data via information stored inside the C API. When the module completes you should close any open virtual files that were used by the module.

### 10.4 Output to memory

As the case for selecting input, GMT modules only know about writing results to a file (or stdout). Hence, we must follow the same paradigm as we did for input and identify virtual files to represent the output destinations. The steps are:

1. Create an empty GMT C container of the right type marked for output.

2. Create a virtual file name to represent this output destination.

3. Place this file name in the appropriate location in the GMT option string. If the module exports data to `stdout` then we can use the hidden option `->filename`.

When the GMT module is run it will know how to make the connections between the memory allocated by the module and the virtual file names stored inside the C API. Once the module call has completed you can access the results in the external environment by using `GMT_Read_VirtualFile` with the virtual filename you created earlier. This will return a GMT C container with the results, and you can now populate you external data containers with data produced by the GMT module.

### 10.5 The magic of knowing

External developers have access to the two extra API functions `GMT_Encode_Options` and `GMT_Expand_Option`. Your `gmt` will need to call `GMT_Encode_Options` to obtain information about what the selected module expects, what its options are, which were selected, and what data types are expected. It may possibly modify the options, such as adding the `filename` “?” to options that set `required` input and output files and returns an array of structures with specific information about all inputs and outputs. If sources and destinations were missing from your `options` string it is taken to mean that you want to associate these sources and destinations with memory locations rather than actual files. The second function `GMT_Expand_Option` can then then used to replace these place-holder names with the virtual filenames you created earlier.

#### 10.5.1 The MATLAB interface

We have built a MATLAB/Octave interface to GMT called the toolbox. It was our first attempt to use the C API from an external environment and its development influenced how we designed the final GMT C API. MATLAB represents most data as matrices but there are also structures that can hold many different items, including several matrices and text strings. Thus, we designed several native mex structures that represent the six GMT objects. The main `gmt` function available in MATLAB derives from a small MATLAB script (gmt.m) which handles basic argument testing and then passes the arguments to our
C function gmtmex.c. Most of the high-level parsing of options and arguments is done in this function, but we also rely on a C library (gmtmex_parser.c) that hides the details of the implementation. It is this library that does most of the work in translating between the GMT and MATLAB object layouts. Knowing what types are represented by the different sources and destinations is provided by the array of structures returned by \texttt{GMT Encode Options}.

\subsection*{10.5.2 The Julia interface}

Unlike the MATLAB interface, the Julia interface is written entirely in the Julia language.

\subsection*{10.5.3 The Python interface}

To be defined shortly.

\section*{10.6 Appendix A: GMT resources}

We earlier introduced the six standard GMT resources (dataset, grid, image, color palette table, PostScript, textset) as well as the user vector and matrix. Here are the complete definitions of these structures, including all variables accessible via the structures.

\subsection*{10.6.1 Data set}

Each data set is represented by a \texttt{GMT\_DATASET} that consists of one or more data tables represented by a \texttt{GMT\_DATATABLE}, and each table consists of one or more segments represented by a \texttt{GMT\_DATASEGMENT}, and each segment contains one or more rows of a fixed number of columns. If the dataset originated from an OGR/GMT file then there will be items such as an \texttt{GMT\_OGR} structure per table and a \texttt{GMT\_OGR\_SEG} structure per segment.

```c
struct GMT_DATASET {
    /* Single container for an array of GMT tables (files) */
    /* Variables we document for the API: */
    uint64_t n_tables; /* Total number of tables (files) contained */
    uint64_t n_columns; /* Number of data columns */
    uint64_t n_segments; /* Total number of segments across all tables */
    uint64_t n_records; /* Total number of data records across all tables */
    double *min; /* Minimum coordinate for each column */
    double *max; /* Maximum coordinate for each column */
    struct GMT_DATATABLE **table; /* Pointer to array of tables */
    /* ---- Variables “hidden” from the API ---- */
    uint64_t id; /* The internal number of the data set */
    size_t n_alloc; /* The current allocation length of tables */
    uint64_t dim[4]; /* Used by GMT\_Duplicate\_Data to override dimensions */
    unsigned int geometry; /* The geometry of this dataset */
    unsigned int alloc_level; /* The level it was allocated at */
    enum GMT_enum_dest io_mode; /* -1: write OGR format (requires proper -a), 0: write everything to one destination [Default], 1: use table->file[GMT\_OUT] to write separate */
    unsigned int alloc_mode; /* Allocation mode [GMT\_ALLOCATED\_BY\_GMT] */
    char *file[2]; /* Name of file or source [0 = in, 1 = out] */
};
```
Here is the full definition of the GMT_DATATABLE structure:

```c
struct GMT_DATATABLE {
    /* To hold an array of line segment structures and header information */
    struct GMT_DATASEGMENT **segment; /* Pointer to array of segments */

    unsigned int n_headers; /* Number of file header records (0 if no header) */
    uint64_t n_columns; /* Number of columns (fields) in each record */
    uint64_t n_segments; /* Number of segments in the array */
    double *min; /* Minimum coordinate for each column */
    double *max; /* Maximum coordinate for each column */
    char *header; /* Array with all file header records, if any */
    struct GMT_DATASEGMENT **segment; /* Pointer to array of segments */

    unsigned int n_records; /* Total number of data records across all segments */
    double *min; /* Minimum coordinate for each column */
    double *max; /* Maximum coordinate for each column */
    char *header; /* Array with all file header records, if any */
    struct GMT_DATASEGMENT **segment; /* Pointer to array of segments */

    struct GMT_OGR *ogr; /* Pointer to struct with all things GMT/OGR 
                            (if MULTI-geometry and not MULTIPOINT) */

    char *file[2]; /* Name of file or source [0 = in, 1 = out] */
};
```

Here is the full definition of the GMT_DATASEGMENT structure:

```c
struct GMT_DATASEGMENT {
    /* For holding segment lines in memory */
    /* Variables we document for the API: */

    uint64_t n_rows; /* Number of points in this segment */
    uint64_t n_columns; /* Number of fields in each record (>= 2) */
    double *min; /* Minimum coordinate for each column */
    double *max; /* Maximum coordinate for each column */
    double **data; /* Data x,y, and possibly other columns */
    char *label; /* Label string (if applicable) */
    char *header; /* Segment header (if applicable) */

    /* ---- Variables "hidden" from the API ---- */

    enum GMT_enum_write mode; /* 0 = output segment, 1 = output header only, 2 = skip segment */
    enum GMT_enum_pol pol_mode; /* GMT_IS_PERIMETER [-Pp] or GMT_IS_HOLE [-Ph] (for polygons only) */
    uint64_t id; /* The internal number of the segment */
    size_t n_alloc; /* The current allocation length of each data column */
    double dist; /* Distance from a point to this feature */
    struct GMT_OGR_SEG *ogr; /* NULL unless OGR/GMT metadata exist for this segment */

    struct GMT_DATASEGMENT *next; /* NULL unless polygon has holes and pointing to next hole */

    char *file[2]; /* Name of file or source [0 = in, 1 = out] */
};
```

The definition of the GMT_OGR structure (not part of the API):

```c
struct GMT_OGR {
    /* Struct with all things GMT/OGR for a table */
    /* The first parameters are usually set once per data set and do not change */
    unsigned int geometry; /* @G: Geometry of this data set, if known [0 otherwise] */
    unsigned int n_aspatial; /* @T: Number of aspatial fields */
    char *region; /* @R: Region textstring [NULL if not set] */
    char *proj[4]; /* @J: The 1-4 projection strings [NULL if not set] */
    unsigned int *type; /* @T: Data types of the aspatial fields [NULL if not set] */
    char **name; /* @N: Names of the aspatial fields [NULL if not set] */

    /* The following are for OGR data only. It is filled during parsing (current segment) 
     * but is then copied to the segment header so it can be accessed later */

    enum GMT_enum_pol pol_mode; /* @P: GMT_IS_PERIMETER or GMT_IS_HOLE (for polygons only) */
    char *tvalue; /* @D: The text values of the current aspatial fields */
    double *dvalue; /* @D: Same but converted to double (assumed possible) */

    char *file[2]; /* Name of file or source [0 = in, 1 = out] */
};
```
The definition of the GMT_OGR_SEG structure (not part of the API):

```c
struct GMT_OGR_SEG {
    /* Struct with GMT/OGR aspatial data for a segment */
    enum GMT_enum_pol pol_mode; /* @P: GMT_IS_PERIMETER or GMT_IS_HOLE (for polygons only) */
    unsigned int n_aspatial; /* @T: Number of aspatial fields */
    char **tvalue; /* @D: Values of current aspatial fields */
    double *dvalue; /* @D: Same but converted to double (assumed possible) */
};
```

10.6.2 GMT grid

A grid is represented by a `GMT_GRID` that consists of a header structure represented by a `GMT_GRID_HEADER` and an float array `data` that contains the grid values.

```c
struct GMT_GRID {
    /* To hold a GMT float grid and its header in one container */
    struct GMT_GRID_HEADER *header; /* Pointer to full GMT header for the grid */
    float *data; /* Pointer to the float grid */
    /* ---- Variables "hidden" from the API ---- */
    unsigned int id; /* Internal number of the grid */
    unsigned int alloc_level; /* Level it was allocated at */
    enum GMT_enum_alloc alloc_mode; /* Allocation mode [GMT_ALLOCATED_BY_GMT] */
    double *x, *y; /* Vector of coordinates */
    void *extra; /* Row-by-row machinery information [NULL] */
};
```

The full definition of the GMT_GRID_HEADER structure. Most of these members are only used internally:

```c
struct GMT_GRID_HEADER {
    /* Variables we document for the API: */
    /* They are copied verbatim to the native grid header and must be 4-byte unsigned ints. */
    uint32_t n_columns; /* Number of columns */
    uint32_t n_rows; /* Number of rows */
    uint32_t registration; /* GMT_GRID_NODE_REG (0) or GMT_GRID_PIXEL_REG (1) */

    /* == The types of the following 12 elements must not be changed. */
    /* They are also copied verbatim to the native grid header. */
    double wesn[4]; /* Min/max x and y coordinates */
    double z_min; /* Minimum z value */
    double z_max; /* Maximum z value */
    double inc[2]; /* x and y increment */
    double z_scale_factor; /* grd values must be multiplied by this */
    double z_add_offset; /* After scaling, add this */
    char x_units[GMT_GRID_UNIT_LEN80]; /* units in x-direction */
    char y_units[GMT_GRID_UNIT_LEN80]; /* units in y-direction */
    char z_units[GMT_GRID_UNIT_LEN80]; /* grid value units */
    char title[GMT_GRID_TITLE_LEN80]; /* name of data set */
    char command[GMT_GRID_COMMAND_LEN320]; /* name of generating command */
    char remark[GMT_GRID_REMARK_LEN160]; /* comments re this data set */
    /* == End of "untouchable" header. */

    /* == Variables "hidden" from the API: */
    /* This section is flexible. It is not copied to any grid header */
    /* or stored in any file. It is considered private */
    unsigned int type; /* Grid format */
    unsigned int bits; /* Bits per value (e.g., 32 for ints/floats; 8 for bytes) */
    unsigned int complex_mode; /* Grid format */
    size_t nm; /* Number of data items in this grid (n_columns * n_rows) [padding is excluded] */
    ...}
```

(continues on next page)
10.6.3 GMT image

An image is similar to a grid except it may have more than one layer (i.e., band). It is represented by a GMT_IMAGE structure that consists of the GMT_GRID_HEADER structure and an char array data that contains the image values. The type of the array is determined by the value of type.
10.6.4 CPT palette table

A CPT is represented by a `GMT_PALETTE` structure that contains several items, such as a `GMT_LUT` structure that contains the color information per interval. The background, foreground, and NaN-color values have colors specified by the `GMT_BFN` array structure `bfn`. As each actual color may be specified in different ways, including as an image, each color slice is represented by the `GMT_FILL` structure.

```c
struct GMT_LUT {  
  double z_low, z_high, i_dz;  
  double rgb_low[4], rgb_high[4], rgb_diff[4];  
  double hsv_low[4], hsv_high[4], hsv_diff[4];  
  unsigned int annot;  
  /* true if CPT applies to categorical data */  
  unsigned int z_adjust[2];  
  /* 1 if +u<unit> was parsed and scale set, 3 if z has been adjusted, 0 otherwise */  
  unsigned int z_mode[2];  
  /* 1 if +U<unit> was parsed, 0 otherwise */  
  unsigned int z_unit[2];  
  /* Unit enum specified via +u<unit> */  
  double z_unit_to_meter[2];  
  /* Scale, given z_unit, to convert z from <unit> to meters */  
};
```

The full definition of the `GMT_LUT` structure.

```c
struct GMT_FILL {  
  char *label;  
  /* For patterns instead of color */  
  struct GMT_FILL *fill;  
  /* For patterns instead of color */  
  char skip;  
  /* true means skip this slice */  
};
```

The full definition of the `GMT_BFN` structure.
The full definition of the GMT_FILL structure. Note: Not part of the GMT API:

```
struct GMT_FILL { /* Holds fill attributes */
  double rgb[4]; /* Chosen color if no pattern + Transparency 0-→ 1 [0 = opaque] */
  double f_rgb[4], b_rgb[4]; /* Colors applied to unset and set bits in 1→bit image */
  bool use_pattern; /* true if pattern rather than rgb is set */
  int pattern_no; /* Number of a predefined pattern, or -1 if not */
  unsigned int skip; /* true means skip this slice */
  unsigned int dpi; /* Desired dpi of image building-block if use_pattern */
  char* pattern [GMT_BUFSIZ]; /* Full filename of user-defined raster pattern */
};
```

### 10.6.5 PostScript text

Bulk PostScript is represented by a GMT_POSTSCRIPT structure that contains data that points to the text array containing n_bytes characters of raw PostScript code. The mode parameter reflects the status of the PostScript document.

```
struct GMT_POSTSCRIPT { /* Single container for a chunk of PostScript code */
  unsigned int n_headers; /* Number of PostScript header records (0 if no header) */
  size_t n_alloc; /* Length of array allocated so far */
  size_t n_bytes; /* Length of data array so far */
  unsigned int mode; /* Bit-flag for header (1) and trailer (2) */
  char* data; /* Pointer to PostScript code */
  char** header; /* Array with all PostScript header records, if any */
  /* ---- Variables "hidden" from the API ---- */
  uint64_t id; /* The internal number of the data set */
  unsigned int alloc_level; /* The level it was allocated at */
  enum GMT_enum_alloc alloc_mode; /* Allocation mode [GMT_ALLOC_INTERNALLY] */
};
```

### 10.6.6 Text set

Each text set is represented by a GMT_TEXTSET that consists of one or more text tables represented by a GMT_TEXTTABLE, and each table consists of one or more segments represented by a GMT_TEXTSEGMENT, and each segment contains one or more rows of strings.

```
struct GMT_TEXTSET { /* Single container for an array of GMT text tables (files) */
  unsigned int n_tables; /* The total number of tables (files) contained */
  unsigned int n_segments; /* The total number of segments across all tables */
  unsigned int n_records; /* The total number of data records across all tables */
  unsigned int n_headers; /* Pointer to array of tables */
  struct GMT_TEXTTABLE **table; /* Pointer to array of tables */
  /* ---- Variables "hidden" from the API ---- */
  unsigned int alloc_level; /* The current allocation length of tables */
  unsigned int alloc_level; /* The geometry of this dataset */
};
```

(continues on next page)
The full definition of the GMT_TEXTTABLE structure:

```c
struct GMT_TEXTTABLE { /* To hold an array of text segment structures and header information */
    /* Variables we document for the API: */
    unsigned int n_headers; /* Number of file header records (0 if no header) */
    uint64_t n_segments; /* Number of segments in the array */
    uint64_t n_records; /* Total number of data records across all segments */
    char **header; /* Array with all file header records, if any */
    struct GMT_TEXTSEGMENT **segment; /* Pointer to array of segments */
    /* ---- Variables "hidden" from the API ---- */
    uint64_t id; /* The internal number of the table */
    size_t n_alloc; /* The current allocation length of segments */
    enum GMT_enum_write mode; /* 0 = output table, 1 = output header only, 2 = skip table */
    char *file[2]; /* Name of file or source [0 = in, 1 = out] */
};
```

The full definition of the GMT_TEXTSEGMENT structure:

```c
struct GMT_TEXTSEGMENT { /* For holding segment text records in memory */
    /* Variables we document for the API: */
    uint64_t n_rows; /* Number of rows in this segment */
    char **data; /* Array of text records */
    char *label; /* Label string (if applicable) */
    char *header; /* Segment header (if applicable) */
    /* ---- Variables "hidden" from the API ---- */
    uint64_t id; /* The internal number of the segment */
    enum GMT_enum_write mode; /* 0 = output segment, 1 = output header only, 2 = skip segment */
    size_t n_alloc; /* Number of rows allocated for this segment */
    char *file[2]; /* Name of file or source [0 = in, 1 = out] */
    char **tvalue; /* The values of the OGR/GMT aspatial fields */
};
```

### 10.6.7 Matrix

User matrices are represented by a *GMT_MATRIX* structure that contains data that points to an array of size *n_columns* by *n_rows*. The type indicates the memory type of the matrix, which is represented by the *GMT_UNIVECTOR* union.

```c
struct GMT_MATRIX {
    uint64_t n_rows; /* Number of rows in the matrix */
    uint64_t n_columns; /* Number of columns in the matrix */
    uint64_t n_layers; /* Number of layers in a 3-D matrix */
    enum GMT_enum_fmt shape; /* /\ 0 = C (rows) and 1 = Fortran (cols) */
    enum GMT_enum_reg registration; /* /\ 0 for gridline and 1 for pixel registration */
    size_t dim; /* Allocated length of longest C or Fortran dim */
    size_t size; /* Byte length of data */
    enum GMT_enum_type type; /* Data type, e.g. GMT_FLOAT */
    double range[6]; /* Contains xmin/xmax/ymin/ymax/[zmin/zmax] */
};
```
union GMT_UNIVECTOR data; /* Union with pointer to actual matrix of the chosen type */
char command[GMT_GRID_COMMAND_LEN320]; /* name of generating command */
char remark[GMT_GRID_REMARK_LEN160]; /* comments re this data set */
/* ---- Variables "hidden" from the API ---- */
unsigned_t id; /* The internal number of the data set */
unsigned int alloc_level; /* The level it was allocated at */
enum GMT_enum_alloc alloc_mode; /* Allocation mode [GMT_ALLOCATED_BY_GMT] */
};

10.6.8 Vectors

User vectors are represented by a GMT_VECTOR structure that contains data that points to an array of n_columns individual vectors. The type array indicates the memory type of each vector. Each vector is represented by the GMT_UNIVECTOR union which can accommodate any data type.

The full definition of the GMT_UNIVECTOR union that holds a pointer to any array or matrix type:

union GMT_UNIVECTOR {
    uint8_t *uc1; /* Pointer for unsigned 1-byte array */
    int8_t *sc1; /* Pointer for signed 1-byte array */
    uint16_t *ui2; /* Pointer for unsigned 2-byte array */
    int16_t *si2; /* Pointer for signed 2-byte array */
    uint32_t *ui4; /* Pointer for unsigned 4-byte array */
    int32_t *si4; /* Pointer for signed 4-byte array */
    uint64_t *ui8; /* Pointer for unsigned 8-byte array */
    int64_t *si8; /* Pointer for signed 8-byte array */
    float *f4; /* Pointer for float array */
    double *f8; /* Pointer for double array */
};

10.7 Appendix B: GMT constants

To increase readability we have encoded many simple integer constants as named enum. These are listed in the tables below and used as flags to various API functions.
<table>
<thead>
<tr>
<th>constant</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GMT_CHAR</td>
<td>int8_t, 1-byte signed integer type</td>
</tr>
<tr>
<td>GMT_UCHAR</td>
<td>int8_t, 1-byte unsigned integer type</td>
</tr>
<tr>
<td>GMT_SHORT</td>
<td>int16_t, 2-byte signed integer type</td>
</tr>
<tr>
<td>GMT_USHORT</td>
<td>uint16_t, 2-byte unsigned integer type</td>
</tr>
<tr>
<td>GMT_INT</td>
<td>int32_t, 4-byte signed integer type</td>
</tr>
<tr>
<td>GMT_UINT</td>
<td>uint32_t, 4-byte unsigned integer type</td>
</tr>
<tr>
<td>GMT_LONG</td>
<td>int64_t, 8-byte signed integer type</td>
</tr>
<tr>
<td>GMT_ULONG</td>
<td>uint64_t, 8-byte unsigned integer type</td>
</tr>
<tr>
<td>GMT_FLOAT</td>
<td>4-byte data float type</td>
</tr>
<tr>
<td>GMT_DOUBLE</td>
<td>8-byte data float type</td>
</tr>
</tbody>
</table>

The known data types in the GMT API.
Index

A
API, 1