# VIRGA

## Virtual Data-file Access

V3.Ø.Ø3

Author/Programmer: Drew Wade

Writer/Editor: Michael Morgan

Design-File Document No. 2.1Ø Rev 3

(VIRGA.TXT, VIRGA.OUT)

TABLE OF CONTENTS

# LIST OF FIGURES

# 1 INTRODUCTION

VIRGA constitutes a general purpose data manager, somewhere in between a full-fledged database manager and a virtual memory manager. It was originally written to facilitate GATEMASTER gate array storage and access, and it is still used for that purpose. For instance, when the GATEMASTER Layout Editor user edits a .LED file, which holds the gate array information, that file is a VIRGA file. The meaning of the name VIRGA (Virtual Gate Array Access) reflects the system's origin.

VIRGA gives "users" (programmers) the advantage of high speed (read and write) data access, efficient use of central memory, and efficient use of the hard disk. For instance, the LED program uses VIRGA routines to access gate array data, and to control memory buffers. The Polygon Editor uses VIRGA routines very similarly.

NOTE: In this manual, "user" indicates a user of the database, i.e., a programmer. Customers use LED, but it is programmers who use VIRGA routines in their programs.

VIRGA routines are used both in interactive editor programs and in batch programs. For instance, in the GATEMASTER system, not only LED uses VIRGA. The output program, MAKE, also uses VIRGA routines to access the database.

Information about VIRGA is confidential.

## 1.1 A General Access Tool

The source code for the VIRGA routines is insensitive to the detailed structure of a VIRGA database file. VIRGA routines simply search through the data frames (or use an index for quicker access) and find the information requested. Despite this generality of use, VIRGA can be used to create a database file which is efficiently structured and quick to access.

VIRGA allows access to disk and memory data via a buffering scheme (described in a section below). The buffers are handled automatically, but the user has the capability to control that management when necessary. The user can think of the buffer control procedures as disk read/write procedures or as memory management procedures; both aspects are relevant. For example, the user can open a temporary file, and access data frames within that file as if it were a virtual heap. In fact, several such files can be used. On the other hand, the user can use VIRGA simply to read some data from a VIRGA file (such as a .LED file) on disk, and write back new data.

The detailed contents of particular VIRGA database files are described in documents other than this one. For instance, the schema for the .LED file is described in the GATEMASTER database document, GMDB.TXT. That document includes data frame and field definitions, with annotations included.

## 1.2 Features

The main goals for the VIRGA design are:

Functionality
Ease of Use
Performance
Robustness

Functionality means that the user can read and write the data. Ease of use means that the user doesn't have to write inordinate amounts of code to read and write the data, or spend inordinate amounts of time understanding it. Performance means speed in execution. Robustness means that, ideally, nothing could ever go wrong.

The VIRGA design assumes that the user is cooperative and capable, and therefore may prefer to trade off robustness in favor of performance or ease of use.

The VIRGA routines are not intended to constitute a full database management system. Many of the features familiar in such a system are not provided here; e.g., transaction logging, backup and recovery, searching for data according to complex context-sensitive relations, etc.

## 1.3 Restrictions

VIRGA does not support concurrent access to a file by different processes. If, in the future, concurrency is necessary, the procedures for data base access could be converted to run as a separate task, with the current procedure interface unchanged. However, this re-design would require substantial modification of the code.

If a program that calls VIRGA is divided into code overlays, the programmer must insure that the code for particular database manipulations is available for any overlays that require them. The simplest approach would be to include the routines in the permanently-memory-resident part of the code.

## 1.4 GATEMASTER Use of VIRGA

In the GATEMASTER system, some VIRGA routines are called indirectly through PDI (Procedural Database Interface). PDI routines, such as PDI_READ_MACRO_CELL_LIST, are high level procedures which call several VIRGA routines in order to access a variety of data structures.

GATEMASTER database files, i.e., .BASE (base array), .MLIB (macros), .LED (full gate array), and .BMP (bit map), are all files in VIRGA format.

## 1.5 PDQ use of VIRGA

Another way of accessing VIRGA files, besides calling VIRGA routines in a program, is to use the PDQ program. PDQ is invoked by a user at the command level simply by typing "PDQ". It is a general user interface to any VIRGA file. In fact, most PDQ commands are simply VIRGA procedures that can be invoked individually at the user level.

In PDQ, a user can use simple commands to directly access the data frames in a VIRGA file. The user can view the effects on the database of a program with VIRGA routines or directly edit the contents of the database.

## 1.6 Critical Routines

The most important 2Ø routines are the first fifteen (the file and frame access routines), and the first 5 of the 7 index routines. Some readers may want to skip all of the buffer control routines and utility routines since none of those routines are required for VIRGA usage. (Please read the table of contents, if you haven't already, in order to see the document organization.)

## 2 STRUCTURE: MODULES & FRAMES

A VIRGA file is structured as shown below. It begins with a root, which is followed by one or more modules. The root contains file control information used by VIRGA:

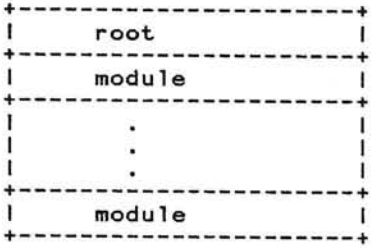



Figure 2 - 1 A VIRGA File

Within a module are frames:

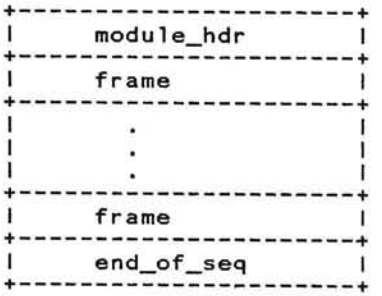



Figure 2 - 2 A VIRGA Module

Note that a VIRGA data module is a dynamic database structure that is controlled by VIRGA routines. The meaning of the word "module" here is entirely different from its use in the phrase, "PLM source code module".

Within the frames are data, arranged according to the user's schema design:

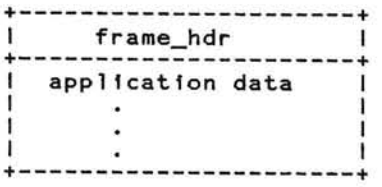



Figure 2 - 3 A VIRGA data frame

Every disk access reads or writes a whole module. Frames are the logical data containers that are used to hold the desired data. A VIRGA search routine, when called, may be able to find the desired data in a module that is presently in a memory buffer, or it may have to copy a module from the hard disk into a buffer.

Both frames and modules are typed. That is, the user can find a separate logical container of data for each frame type, with each container containing zero or more frames. Each frame type belongs to a single module type: each data frame of a given type always belongs to the same module type. A module type may contain one or more frame types.

This typing allows some "sorting" of the data in the file. There is no guarantee that any module (of any type) has any particular location in the file; however, it is guaranteed that a given type of frame is found in only one module type.

When a file is created, the user specifies all the frame types and the module types to which they belong. This gives the user

the flexibility to impose the ordering desired. This decision is a trade-off between, on the one hand, space optimization and, on the other hand, ease and speed of access. [Space optimization is a concern both in the amount of core memory consumed and in the amount of hard disk required for the permanent file.] If every frame were assigned to only one (the same) module type, then a minimum of space would be wasted, but then it is not possible to read only one frame type without getting other frame types interspersed. Alternatively, if each frame were assigned to a separate module, then it would be simple to read only frames of one type, but there will be a tendency for modules to be only partly full. For example, if module type 7 contains only one frame type, 12, and module type 9 contains only one frame type, 6, and if there is only one frame of type 12 and one of type 6, then there will be two modules, one of type 9 and one of type 7, each of which contains only one single frame.

It is expected, but not required, that each frame type contains data with a certain format. For example, a net_frame might contain a net, which is described by a certain structure literal.

Generally, there is no restriction on the number of frames of a given type, nor on the number of modules of a given type, nor on the total number of modules or frames. The one restriction is that all such types are specified by one-byte integers, and hence are limited to 255 in number. However, the user is required to estimate some of these sizes to help VIRGA [see Buffer Management and the CREATE procedure].

The first two module types and first four frame types are reserved by VIRGA for its own use. Module type Ø and frame type Ø signify empty modules/frames. Module type 1 contains control information used by VIRGA. This information consists of three types of frames: the module inventory, the frame type inventory, and the frame inventories. [See next section.] Note, also, that VIRGA uses the root for some control information, and that, in order to avoid crossing CAF boundaries, it will force the first module (a control module) to have a size equal to the module_size minus the root_size. The control frames are reserved for internal VIRGA use, but the user must reference them in accessing portions of a frame or module header. The user references the control modules as literals:

```
GDB_FIRST_MODULE    LITERALLY   '1',
GDB_FIRST_FRAME     LITERALLY   '3',
```

The user should, for example, use GDB_FIRST_FRAME+n and GDB_FIRST_MODULE+n, where n > Ø.

As an example, we present the early design of the frame and

module types for the .LED file:

| Module types: | Frame Types: |
|---|---|
| GDBM_EMPTY | GDBF_EMPTY |
| GDBM_CONTROL | GDBF_MOD_INV<br>GDBF_FRAME_TYPE_INV<br>GDBF_FRAME_INV |
| GDBM_NET | GDBF_NET |
| GDBM_COMPONENT | GDBF_COMPONENT |
| GDBM_MACRO | GDBF_MACRO |
| GDBM_ROUGH_DATA | GDBF_ROUGH_BITMAP<br>GDBF_ROUGH_GRID<br>GDBF_ROUGH_OTHER |
| GDBM_STATIC | GDBF_DED_PAGE_LIST<br>GDBF_CELL_ARRAY<br>GDBF_LAYER_INFO<br>GDBF_BITMAP_INFO<br>GDBF_OBSTRUCTION<br>GDBF_UNDERPASS<br>GDBF_FIXED_VIA |

Figure 2 - 4 LED Modules and Frames

The module and frame numbers are simply one-byte positive integers, starting at Ø and advancing up to, but not exceeding, 254. Hence, the above identifiers represent literals (symbolic constants).

Modules must begin with the module_header that follows:

```
GDB_MODULE_HDR LITERALLY
    'OPCODE         BYTE,   /* skip */
    HDR_SIZE        WORD,
    ID              WORD,   /* module ID */
    SIZE            WORD,   /* size of module */
    TYPE            BYTE,   /* module type */
    MARK            WORD,   /* offset to frame last deleted or added */
    FREE_BYTES      WORD,   /* total bytes available (VIRGA internal only)
    BUSY_FRAMES     WORD,   /* number of frames used (VIRGA internal only)
    DUMMY(2)        BYTE',  /* reserved for future use (must be zero)
```

Figure 2 - 5 Module Header Structure

Frames must begin with the frame_header that follows:

```
GDB_FRAME_HEADER LITERALLY
    'OPCODE         BYTE,   /* usually skip or similar */
    HDR_SIZE        WORD,
    SIZE            WORD,       /* frame size */
    ID              WORD,       /* frame ID */
    TYPE            BYTE,       /* frame type */
    GDB_RECTANGLE',

GDB_RECTANGLE LITERALLY     /* recognition area */
    '(X1,Y1)        WORD,
     (X2,Y2)        WORD',
```

Figure 2 - 6 Frame Header Structure

Every VIRGA frame has a data structure for a geometrical recognition area, whether needed or not. The recognition area does not have to have good data if spatial (geometrical) data is not needed to help define the object that the frame represents. However, every frame must begin with the exact header shown, since VIRGA will look for user-designed application data just below the fields reserved for the recognition rectangle coordinates.

The frame header must always be created, whether by VIRGA, or

by the user. The user must reference data fields by their offset beyond the header size.

The header may be extended if necessary. However, if the frame_header contains added fields, the header_size must reflect that, so that application data will written after the end of the header.

## 2.1 Sizes of the File, Modules, and Frames

Each file is created with a default module_size, which size is used for the file's CAF (contiguous allocation factor -- see the DAISY file system). Normally, modules will be created with that size. However, it is possible to create a larger module implicitly (by creating a frame larger than the module_size), and it is possible to explicitly create a module of any size (though the size will be rounded up to a multiple of the file system's block size, which is 512 bytes). Note, however, that performance is enhanced by using a module size that is an integer multiple of the CAF. When creating larger modules implicitly, VIRGA always uses a multiple of the default module size; e.g., if the default is 8K, then the module might have size 16K or 24K or a larger multiple.

Frames can be any size.

By increasing module_size, the number of disk accesses can normally be minimized, and performance enhanced. However, when the size is so large that there is insufficient memory to hold

the modules needed at one time, then they will be swapped in and out, slowing down execution. Also, memory fragmentation can cause a module allocation to be unsuccessful, especially when the module size approaches the size of the memory pool.

Some limited measurements and experience indicate that a reasonable module size is 8K or 16K bytes. We expect sizes of that order. Frame sizes should be chosen at least an order of magnitude smaller to accomodate memory management within a module. For example, LED schema's largest frame is the net frame, which is expected to be on the order of 512 bytes.

## 2.2 Module Buffering

VIRGA accesses a file by reading or writing a module at a time. A module is always pulled into memory for access, and there it resides in a buffer.

All data read and write requests operate on a module buffer. Data is read from a buffer and written to a buffer. Data is not written to disk until necessary or until specifically requested.

When data is requested and the module holding that data is not in memory, VIRGA gets a buffer from the system, and reads the module from disk into that buffer. There is a maximum number of buffers (a configuration parameter), and when that maximum is reached, no more buffers will be allocated. Also, there is limited memory available from the system. When the maximum

number of buffers is reached or insufficient memory exists for a new module, then one of the previously read modules (the least accessed one) will be removed from memory, its buffer will be de-allocated (returned to the system) and the buffer for the new module will be allocated. If necessary, multiple old buffers will be removed.

When a buffer is removed, its dirty flag is checked, and if that flag is set (true) then the module is first written back to disk, then removed from core memory. The dirty flag is automatically set by the procedures that write data [see sections below] or can be explicitly set by the user. [if dirty, then write before de-allocate]

NOTE: buffers for all files open in VIRGA are accounted for in a global buffer inventory. This means that a read request for a module in one file might cause a module of a different file to be swapped out, since the LRU aging algorithm operates globally over all buffers (see below).

A module buffer may be locked in memory by an explicit user request. A locked buffer will never be removed from memory except upon a close file or terminate request.

Modules may be removed from memory automatically or by explicit user request. They will be automatically removed when there is a request for a new buffer and insufficient resources exist (e.g., too many buffers or insufficient memory), or when the user closes the file or terminates VIRGA.

When it becomes necessary to remove an old buffer from memory, VIRGA chooses which buffer to remove according to an LRU (least-recently-used) algorithm. The first buffer removed will be the one with the longest "age".

In the aging algorithm, whenever any buffer is accessed via VIRGA, that buffer's age (LRU counter) is reset to Ø (youngest) while all other buffers age one unit (counter is incremented). Thus, the buffer chosen for removal is the oldest one.

[Actually, a slightly different counting mechanism is used. One global counter is incremented for each access. Each buffer maintains a local counter. When a buffer is accessed, its counter is set to the global counter's value. Then, the effective age of a buffer is given by the global counter less the buffer's counter. The "oldest" buffer is chosen as that one which has the smallest local counter, which is equivalent to the above algorithm, but is faster. Note that the same effect could be achieved by maintaining a (doubly) linked list for the buffers. The trade-off is that the counter approach is faster for accesses which do not involve a disk read or write, while the linked list approach is faster for the disk access case. VIRGA uses the counter mechanism.]

FIND_ routines in VIRGA return pointers to the data that is in the module buffer. The data is not copied out of the buffer. The user must consider this pointer temporary, since the

buffer may be swapped by any later VIRGA request. Also, the user should consider this data as read-only. Specific write requests guarantee that the data will be written, eventually, to the correct place on disk. However, if the calling program uses the pointer and modifies the data in the buffer, the data might not be written to disk.

The user has the capability to directly modify that data in the buffer, but one should do this ONLY by first locking the module (or frame) in memory, and then setting the dirty flag. The locking will assure that the data in the buffer will actually remain in the buffer (until the user unlocks it). Setting the dirty flag insures that the data will eventually be written back to disk.

Thus, the user may employ VIRGA in various modes as they suit the application. To illustrate, we present two approaches.

(1) It is possible to use VIRGA as a simple file access method. The user simply follows each find request with a copy request, resulting in a copy of the desired data from disk. Similarly, the write requests can be viewed as writing to disk.

(2) The user can view VIRGA as a virtual memory manager which can supply buckets of memory (modules). If the buckets overflow physical memory, then they will be retained on disk until needed again. In this approach, the user can either explicitly choose a particular file for the virtual memory and

save it between invocations, or allow VIRGA to choose a unique filename and can purge it when no longer needed. In this mode of use, the user must lock modules that the user wants to keep in memory.

Buffers are managed globally to the calling task; i.e., all open files share the same buffer pool. A request to read a module from one file may cause a module from another file to be swapped out. This improves the usage of available system memory.

# 3 INTERFACE GUIDE

## 3.1 File Manipulation

In this section we present the procedures which manipulate a file or the global VIRGA data structures.

All VIRGA procedures return an INTEGER status, which is negative if an error occurred. Each procedure, when actually invoked in a program, would have to be preceded by the integer STATUS variable, as in 'STATUS = gdb_routine(parameters)'. However, in this document, the "STATUS" return has been dropped for simplicity of reading.

The data structures referred to in the Interface Guide are defined as literals in the file GDFIL.ELT.

The GDB_INITIATE and GDB_TERMINATE procedures must be used by any and all users of VIRGA. The calls need be made only once regardless of the number of files accessed.

### 3.1.1 GDB_GET_CONFIG

The procedure GDB_GET_CONFIG first sets default values for the configuration table, then checks the disk to see if a table with good values is already there. If a configuration table exists on disk, it uses those values. Configuration fields include MAX_BUFS (maximum number of buffers), and MAX_FILES (maximum number of files).

The user may use TEC to reset the values of an existing configuration disk file, if desired. The configuration table is defined in the source module, GDB_INT.

This routine must be called before GDB_INITIATE.

Invocation:

```
GDB_GET_CONFIG (PARAM_BLOCK_PTR, @CONFIG_PTR);
```

Parameters:

PARAM_BLOCK_PTR is returned by SYS_GET_PARAMETER_OBJECT_PTR with selection = SYS_JOB_PARAMETER_OBJECT.

CONFIG_PTR points to the structure, GDB_CONFIG_DEF.

### 3.1.2 GDB_INITIATE

Allocate buffer control structures and initialize global data tables.

This routine must be called following the GET_CONFIG call.

Invocation:

```
GDB_INITIATE (@GDB_GLOBAL_PTR, GDB_CONFIG_PTR);
```

Parameters:

GDB_GLOBAL_PTR must be retained by the caller, never modified, and passed, eventually, to the termination procedure.

GDB_CONFIG_PTR must point to the structure, GDB_CONFIG_DEF, which has been initalized with all the configuration fields required by VIRGA.

### 3.1.3 GDB_TERMINATE

De-allocate all global structures.

This routine must be called before the program which initiated VIRGA terminates.

Invocation:

```
GDB_TERMINATE (GDB_GLOBAL_PTR);
```

Parameter:

GDB_GLOBAL_PTR was returned by GDB_INITIATE, and must be

supplied to this routine.

### 3.1.4 GDB_CREATE

Creates a VIRGA file with user-specified module & frame types. If a file already exists by that name, then no action will be taken and an error will be returned.

Invocation:

```
GDB_CREATE (@GDB_ROOT_PTR, GDB_GLOBAL_PTR, PATH_NAME_PTR,
                GDB_FILE_DESC_PTR);
```

Parameters:

GDB_ROOT_PTR must be maintained by the calling program, never modified, and passed to each and every procedure that accesses the file. GDB_GLOBAL_PTR must have been returned from a previous initiate request. PATHNAME_PTR may either point to an existing path name or be nil (Ø). If nil, then VIRGA will create a unique name in the temporary directory /TEMP. Otherwise, VIRGA will use the specified pathname for the file (pathname includes filename).

GDB_FILE_DESC_PTR points to the structure, GDB_FILE_DESC_DEF, which supplies a description of the specific module and frame structure desired for the new VIRGA file:

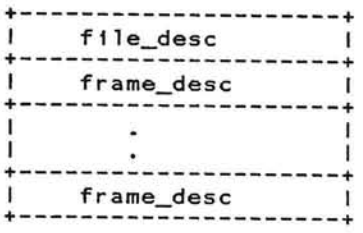



Figure 3 - 1 File Descriptor Table

All fields in this structure must be filled in by the caller. The structure has the following format:

```
GDB_FILE_DESC_DEF   LITERALLY
  'MODULE_SIZE        WORD,     /* in bytes */
  MOD_INV_SIZE        WORD,     /* initial # modules allowed */
  MOD_INV_EXT         WORD,     /* extension size */
  NUM_FRAME_TYPES     WORD',    /* no. of frame_desc's to follow */

GDB_FRAME_DESC_DEF  LITERALLY
  'MOD_TYPE           BYTE,     /* which module owns this one */
  INV_SIZE            WORD,     /* initial # frames allowed */
  INV_EXT_SIZE        WORD',    /* extension size */
```

Figure 3 - 2 File and Frame Descriptor Structures

MODULE_SIZE will be used as both the CAF and the default module size for the file; however, if the configuration table (specified in the GDB_INITIATE call) has the field CAF <> Ø, then that CAF will be used instead. MOD_INV_SIZE will be used for the size of the initial module inventory. MOD_INV_EXT is the amount by which the module_inventory will increase when it overflows. Since all inventories are dynamic, these numbers need not be exact, but are merely helpful hints to get VIRGA started off on the right foot.

For MOD_INV_SIZE, make a rough guess of the expected number of modules. For MOD_INV_EXT, use a small number if you want to conserve table space and expect that modules will rarely be added, or a larger number if you expect modules to be added often and want to minimize the overhead of dynamically expanding the module inventory.

NUM_FRAME_TYPES should specify the number of types that you want to define via GDB_FRAME_DESC_DEF blocks. Remember that

the first four frame types are reserved for VIRGA itself, so the first frame-descriptor block you supply will specify frame_type number 4. If, for example, you need 5 frame types, then NUM_FRAME_TYPES should be 5, and they will describe numbers 4 through 8.

The literal,

```
GDB_FIRST_FRAME_TYPE LITERALLY '3',
```

can be used when defining your frame_type literals;
e.g., you may use:

```
GDBF_FOO_FRAME LITERALLY 'GDB_FIRST_FRAME+1',
```

Each GDB_FRAME_DESC_DEF block specifies which module type the frame_type should be placed into. The frame INV_SIZE and INV_EXT_SIZE are similar to those for the module inventory described above except that here the numbers apply to the expected number of frames of this type. For example, the schema for LED describes a DED_PAGE_LIST frame type. If you expect that there will be relatively few frames of that type, you might specify an inv_size of, say, 10, and an ext_size of, say, 2. For nets, however, on a 10-equivalent gate chip you might specify 2500 for the inv_size, and 100 for the ext_size.

Note: Never specify inventory size < 4 (although 2 should work, 4 is safer). Similarly, for module_inventory size, specify at least 4.

### 3.1.5 GDB_OPEN

Makes a pre-existing VIRGA file available for access. If the file does not exist, no action will be taken, and an error will be returned.

Invocation:

```
GDB_OPEN (@GDB_ROOT_PTR, GDB_GLOBAL_PTR, PATH_NAME_PTR,
          MODE);
```

Parameters:

The root_ptr is created by VIRGA as part of VIRGA's internal management system. It must be saved, never modified, and passed to VIRGA for each and every request concerning this file. The global_ptr must have been returned by a previous initiate request. The path_name_ptr must point to the name of an existing file. The MODE parameter should be fsys_read_mode, fsys_write_mode, or other modes defined by the Daisy file system.

### 3.1.6 GDB_CLOSE

Make a specified file unavailable for access via VIRGA until the user again calls GDB_OPEN. VIRGA files MUST be closed via this call.

If any modules (or frames) are locked, the routine will still execute successfully. The status return will be positive but will indicate that locks were encountered.

Invocation:

```
GDB_CLOSE (GDB_ROOT_PTR, PURGE_FLAG);
```

Parameters:

GDB_ROOT_PTR must have been returned from a previous open or create request. PURGE_FLAG, if TRUE, will cause the file to be purged (removed, erased, deleted, whatever your favorite word is). (Purge_flag is BOOLEAN data type; see SYSCOM.ELT.)

## 3.2 Data Access

There are three ways to access data: through direct access, sequential access, or module access. The first two ways are frame-specific access methods.

In this section we describe the routines that provide data access.

FOR ALL ROUTINES:

As always, GDB_ROOT_PTR is the pointer returned from the open (or create) request, and specifies the file to be accessed. Module_ptr and frame_ptr point to the module (in which the frame resides) and to the frame. Frame_type and ID are self-explanatory.

All routines respect a frame-type's name index, if it exists. (Name indexes are described in a section below). This simply means that if a frame is modified, the name-index which indexes that frame-type (if it exists) will be updated in whatever manner is appropriate. For instance, if a frame is added, the new frame's name and ID are added to the name index for that frame-type.

A. Direct Access

### 3.2.1 GDB_FIND_FRAME

Find the desired frame, given a frame-type and frame ID. If necessary, the module containing the frame will be read from disk.

Find_frame uses an inventory to achieve direct access to the frame's module (maximum one disk access). Then, it performs a linear search within that module to find the specified frame.

Find_frame doesn't know or care about name-indexes.

Invocation:

```
GDB_FIND_FRAME (GDB_ROOT_PTR, @MODULE_PTR, @FRAME_PTR,
                FRAME_TYPE, FRAME_ID);
```

Parameters:

The routine ignores the input value of module_ptr and frame_ptr. From the frame_type and frame ID it will find the specified frame. Upon return, module_ptr and frame_ptr will point to the module and frame which were found. If the frame could not be found, the pointers will be set to nil and a negative status will be returned.

WARNING: Note that find (and similar procedures) return a pointer directly into the module buffer. This should be considered a TEMPORARY pointer only. Any later request could invalidate that pointer, either because the module was swapped

out or because the frame was moved. Also, that pointer must not be used to modify the contents of the frame directly (unless the user sets the lock and dirty flags). [See Buffer Control, below, for lock and dirty flag requests.]

### 3.2.2 GDB_ADD_FRAME

The ADD routine adds a new frame (with new data) to the file.

Although an empty frame may be available, if necessary, the routine will allocate a new frame, since it must copy from the given frame into a frame. If the frame can not be allocated (e.g., invalid frame type), the pointers will be set to nil and a negative status will be returned.

If the database FRAME_COPY.ID field is GDB_NIL_WORD (ØFFFFH), then VIRGA will allocate a unique ID for the frame; otherwise, the routine try to use the specified ID. If that ID is already in use, the call will fail with negative status. IDs which are automatically allocated by VIRGA never decrease; e.g., if you add a frame and VIRGA gives it ID 93, add another (94), delete frame 94, and then add another, that last frame will be given ID 95.

The routine always sets the dirty flag on the module containing the frame, so that the modified module will eventually be written to disk.

The routine will add the frame name to an existing index; if

the name is a duplicate, it will be added anyway, but a negative status will be returned.

Invocation:

```
GDB_ADD_FRAME (GDB_ROOT_PTR, @MODULE_PTR, @FRAME_PTR);
```

Parameters:

MODULE_PTR and FRAME_PTR must specify the location of the frame to be added. From that frame, the procedure will extract the frame_type, find a suitable module, allocate space for the frame, and copy the frame into its new location.

Upon return, MODULE_PTR will point to the module in which the frame was placed. FRAME_PTR will point to the new frame in its module The original copy of the frame (to which the input frame pointer pointed) is completely unchanged, but FRAME_PTR no longer points to it.

### 3.2.3 GDB_UPDATE_FRAME

Overwrites an existing frame in the file. The routine will allocate a new frame if necessary.

The routine always copies from one frame into another frame. It always sets the dirty flag on the module containing the frame, so that the modified module will eventually be written to disk.

It will check an existing index to make sure the new name is identical to the old name. If the index name is not identical, the routine will NOT update the frame, and will return negative status.

Invocation:

```
GDB_UPDATE_FRAME (GDB_ROOT_PTR, @MODULE_PTR, @FRAME_PTR,
                  NEW_FRAME_PTR);
```

Parameters:

The routine assumes that MODULE_PTR and FRAME_PTR point to the existing module and frame. NEW_FRAME_PTR must point to a copy of the new frame.

Upon return, MODULE_PTR and FRAME_PTR point to the newly updated module and frame.

### 3.2.4 GDB_GROW_FRAME

Increase (or decrease) the size of a frame. If size is decreased, data is lost from the area that was decreased. If size is increased, the new data area is not initialized.

The frame, after being modified, may end up in a different module. When using this routine, the user must assume that the frame is always moved, whereas in GDB_UPDATE_FRAME, the routine may leave it in the same place.

The routine will always set the dirty flag on the module containing the frame, so that the modified module will eventually be written to disk.

NOTE: a user cannot call GDB_GROW_FRAME on a locked frame, since the growing process requires moving the frame.

Invocation:

```
GDB_GROW_FRAME (GDB_ROOT_PTR, @MOD_PTR, @FRAME_PTR,
                NEW_SIZE);
```

Parameters:

MODULE_PTR and FRAME_PTR must point to the existing module and frame. NEW_SIZE is a WORD-valued field that specifies the byte-count of the new frame. The new frame's FRAME_HEADER.SIZE field will be updated to the new size.

Upon return, MODULE_PTR and FRAME_PTR point to the expanded or shrunk frame in its module.

### 3.2.5 GDB_DELETE_FRAME

The procedure removes a frame from the file.

It also removes a name the frame-type's name index, if it exists. If the name is not in the index, the frame is still deleted, but a negative status is returned.

The routine will always set the dirty flag on the module containing the frame, so that the modified module will eventually be written to disk.

Invocation:

```
GDB_DELETE_FRAME (GDB_ROOT_PTR, MODULE_PTR, FRAME_PTR);
```

Parameters:

The routine assumes that MODULE_PTR and FRAME_PTR point to an existing module and frame. The frame will be deleted. The user must assume that other data frames may be moved by this routine.

### 3.2.6 GDB_DELETE_BY_TYPE

Delete all frames of the specified type.

The routine will set the dirty flag on the modules containing the frames, so that the modified modules will eventually be written to disk.

Invocation:

```
GDB_DELETE_BY_TYPE (GDB_ROOT_PTR, FRAME_TYPE);
```

Parameters:

GDB_ROOT_PTR must be the same one as always. FRAME_TYPE must be the number of the frame_type that you want deleted.

B. Sequential Access

Sequential-access routines, described below, are primarily used in batch-type programs such as MAKE and GET.

The routines allow the user to step through all the frames of a given type. FIND_FIRST locates the "first" frame of the specified type; FIND_NEXT locates the "next" one, if any. The ordering found is simply the order in which the frames exist in the file, which is appropriate for speed, but does not have any particular meaning. Note that some modules might contain frames of various TYPE in no particular order.

Note also that other calls (e.g., update_frame) might change this order, so the user must not intersperse sequential calls with any other calls that might change the frame ordering. If it is in fact necessary to modify frames in between the sequential requests, the user must find the next frame, remember its ID, then modify the previous one, then make a (direct) find_frame request for the ID of the "next frame", etc. Also, be aware that any interspersed calls (even for other module types or other files) could cause the module of interest to be swapped out, so the pointers would no longer be valid.

### 3.2.7 GDB_FIND_FIRST

Given a frame-type, return pointers to the first frame found.

If the desired frame cannot be found then the returned pointers will be set to nil and a negative status will be returned.

Invocation:

```
GDB_FIND_FIRST (GDB_ROOT_PTR, @MODULE_PTR, @FRAME_PTR,
                FRAME_TYPE);
```

Parameters:

As always, GDB_ROOT_PTR is the returned pointer from open or create.

In this routine, the input MODULE_PTR and FRAME_PTR are ignored. The frame_type is used to locate the first frame of that type.

Upon return, MODULE_PTR and FRAME_PTR point to the desired module and frame.

### 3.2.8 GDB_FIND_NEXT

Given pointers to a frame, find the next frame of the same type.

If the desired frame cannot be found then the return pointers will be set to nil and a negative status will be returned.

Invocation:

```
GDB_FIND_NEXT (GDB_ROOT_PTR, @MODULE_PTR, @FRAME_PTR);
```

As always, GDB_ROOT_PTR is the returned pointer from open or create.

The input MODULE_PTR and FRAME_PTR must point to a valid frame and its module. Upon return they will point to the next frame (and its module) of the same type.

C. Module Access

### 3.2.9 GDB_GET_MODULES_BY_TYPE

This call gets all modules of a given type into memory and locks them there. This insures that later requests for frames within those modules will not have to go to disk (unless the module has subsequently been swapped out). [See Buffer Control, below, for lock requests.]

If there is insufficient memory to read in all modules, VIRGA will get as many as possible and then return a negative status.

Invocation:

```
GDB_GET_MODULES_BY_TYPE (GDB_ROOT_PTR, MODULE_TYPE);
```

Parameters:
As always, GDB_ROOT_PTR is the returned pointer from an open or create request.

MODULE_TYPE specifies the type of modules to be read. If MODULE_TYPE = gdb_nil_byte (= ØFFH), then all modules of all types will be read and locked (error returned, if not enough space).

## 3.3 Buffer Control

In the following section we discuss procedures that allow the user to control VIRGA's buffer handling. However, it is quite possible to use VIRGA without ever using any of the following calls. Straightforward use of find_frame, add_frame, update_frame, and delete_frame will take care of everything for you.

Yet, the user may desire to directly control memory buffering in order to improve performance or to simplify memory-dependent tasks. If so, please read on.

A. Flush, Allocate, or Dispose of Modules

### 3.3.1 GDB_FLUSH

This routine applies to the file that GDB_ROOT_PTR points to. It will write any dirty modules to disk and clear their dirty flags. The root will also be flushed. All modules that were in memory will remain there.

The updating effect is exactly as if the file had been closed and then re-opened, but with less overhead.

Invocation:

```
GDB_FLUSH (GDB_ROOT_PTR);
```

Discussion:

Under perfect conditions this call would never be needed, since VIRGA makes sure that dirty modules are eventually written to disk. However, a hardware or software failure might result in loss of data and might even leave the file in an inconsistent state.

To help avoid file corruption problems, the user should flush the file buffers occasionally. In particular, call GDB_FLUSH after major data updates have been made.

The next two routines are really memory-management (rather than buffer control) procedures. They allow the user to control the memory (and disk-space) allocation of VIRGA.

### 3.3.2 GDB_ALLOCATE_MODULE

Create a new module of the specified type and size.

If an error occurs (e.g., invalid module type) the returned pointer will be nil and a negative status will be returned.

Invocation:

```
GDB_ALLOCATE_MODULE (GDB_ROOT_PTR, @MODULE_PTR,
                     MODULE_TYPE, SIZE);
```

Parameters:

GDB_ROOT_PTR must be the same as always.

The input value of MODULE_PTR is ignored.

MODULE_TYPE and SIZE specify the desired module type and size.

Upon return, MODULE_PTR will point to the module (in memory).

### 3.3.3 GDB_ALLOCATE_FRAME

Create a new frame of the specified type and size and within the specified module.

Note that the GDB_ADD_FRAME routine will also create a frame of a specified type and size, but it will put it wherever VIRGA thinks best (and will copy the specified data into it). This procedure puts the frame in the module specified by the user.

If an error occurs (e.g., insufficient space within the module) then the returned pointer will be nil and a negative status will be returned.

Invocation:

```
GDB_ALLOCATE_FRAME (GDB_ROOT_PTR, MODULE_PTR, @FRAME_PTR,
                    FRAME_TYPE, SIZE);
```

Parameters:

GDB_ROOT_PTR is as always the returned pointer from an open or create.

MODULE_PTR points to the module (in memory) in which the frame will be allocated.

The input value of FRAME_PTR is ignored.

FRAME_TYPE and SIZE specify the type and size for the new frame.

Upon return, FRAME_PTR will point to the new frame.

### 3.3.4 GDB_DISPOSE_MODULE

This procedures de-allocates buffer space. Before the buffer is de-allocated it will be written to disk (if dirty). Then, the buffer is removed from memory (and the memory is returned to the system). This may be used to free up memory for another purpose. Note that a locked module will NOT be disposed.

Note that modules are not removed from the disk file, only from memory.

Invocation:

```
GDB_DISPOSE_MODULE (GDB_ROOT_PTR, MODULE_PTR);
```

Parameters:

As always, GDB_ROOT_PTR is the returned pointer from open or create.

MODULE_PTR must point to an in-memory module.

### 3.3.5 GDB_DISPOSE_MODULES_BY_TYPE

This procedures de-allocates buffer space. Before the buffers are de-allocated they will be written to disk (if dirty). Then, the buffers are removed from memory (and the memory is returned to the system). This may be used to free up memory for another purpose. Note that locked modules will NOT be disposed.

Note that modules are not removed from the disk file, only from memory.

Invocation:

```
GDB_DISPOSE_MODULES_BY_TYPE (GDB_ROOT_PTR, MODULE_TYPE);
```

Parameters:

As always, GDB_ROOT_PTR is the returned pointer from open or create.

MODULE_TYPE specifies the desired type. No error occurs if no such buffers exist.

If module_type = gdb_nil_byte (ØFFH), then all types will be disposed, except for type gdb_control (which holds VIRGA's inventories). Note that locked modules will not be disposed.

B. Locking and Unlocking of frames and modules

Locking allows the user to force some data to remain in memory. This can simplify user programming (since the user need not make multiple find requests to be sure that the desired data is in memory and can use direct memory access techniques within the locked memory) and can improve performance (since users know best what their code does, a user can best determine what should remain in memory and what shouldn't).

Locking exists on two levels of granularity: modules and frames. Since users generally manipulate frames (the structures containing the data) rather than modules, the frame-level locks may be more appropriate. However, module-level locks are also available and may be appropriate for users controlling their own memory allocation at the per-module level.

Since the module is the unit that is actually swapped to and from disk, it is the only unit that is really locked. To simulate frame locks, the following approach is used. Each time a frame is locked, the corresponding module's lock counter is incremented [see buffer control inventory in the Buffer Management section, above]. Also, each frame (in the frame inventory) has a lock flag which is set. Any module with a non-zero lock counter is considered locked. An unlock request on a frame will decrement the module counter and clear the frame's lock flag; however, if the frame was not locked,

the lock counter will not be decremented (and an error will be returned). One additional flag is maintained to indicate a module lock.

The effect is that the user can lock frames and be assured that they will stay in memory, and similarly for modules. However, it is then the user's responsibility to unlock all frames and modules that were locked.

Structure your code to avoid the following incorrect use of VIRGA. Suppose you have two blocks of code, each of which finds a frame, locks it, uses it, then unlocks it. Suppose that one of those code blocks invokes the other, and it happens that both code blocks are using the same frame. Then, the second code block would unlock the frame which the first code block considers still locked. The frame could subsequently be swapped out, resulting in a crash. To resolve this, you could either keep a resource record which contains all locked frames (and allocated system memory, etc.) and share that record among all code blocks. Or, when you lock a frame, you can check to see if it's already locked (in which case VIRGA will return a non_negative, non_success status). Then, if it had already been locked, don't unlock it.

ALL routines: GDB_ROOT_PTR is as always the returned pointer from open or create.

### 3.3.6 GDB_LOCK_FRAME

Lock a particular frame that is in memory.

An error is returned if the frame is already locked or not in memory.

Invocation:

```
GDB_LOCK_FRAME (GDB_ROOT_PTR, MODULE_PTR, FRAME_PTR);
```

Parameters:

MODULE_PTR and FRAME_PTR specify the desired frame and module.

### 3.3.7 GDB_UNLOCK_FRAME

Unlock a frame that is in memory.

Returns an error if the frame is already unlocked.

Invocation:

```
GDB_UNLOCK_FRAME (GDB_ROOT_PTR, MODULE_PTR, FRAME_PTR);
```

Parameters:

MODULE_PTR and FRAME_PTR specify the desired frame and module.

### 3.3.8 GDB_LOCK_MODULE

Lock a particular module that is already in memory.

Returns an error is the module is already locked or not in memory.

Invocation:

```
GDB_LOCK_MODULE (GDB_ROOT_PTR, MODULE_PTR);
```

Parameters:

MODULE_PTR and FRAME_PTR specify the desired frame and module.

### 3.3.9 GDB_UNLOCK_MODULE

Unlock a module that is in memory.

An error is returned if the module is not already locked.

Invocation:

```
GDB_UNLOCK_MODULE (GDB_ROOT_PTR, MODULE_PTR);
```

Parameters:

MODULE_PTR and FRAME_PTR specify the desired frame and module.

### 3.3.1Ø GDB_LOCK_MODULES_BY_TYPE

Lock all modules of the specified type that are already in memory. This forces that type of module to remain in memory until unlocked.

Invocation:

```
GDB_LOCK_MODULES_BY_TYPE (GDB_ROOT_PTR, MODULE_TYPE);
```

Parameters:

MODULE_TYPE is the desired type. If MODULE_TYPE = gdb_nil_byte (ØFFH), this means ALL types.

### 3.3.11 GDB_UNLOCK_MODULES_BY_TYPE

Unlock all modules of the specified type. This allows that type of module to then be flushed to disk.

Invocation:

```
GDB_UNLOCK_MODULES_BY_TYPE (GDB_ROOT_PTR, MODULE_TYPE);
```

Parameters:

MODULE_TYPE is the desired type. If MODULE_TYPE = gdb_nil_byte (ØFFH), this means ALL types, except that type = nil will not unlock VIRGA's internal inventories (module_type = gdbm_control).

### 3.3.12 GDB_SET_DIRTY_MODULE

Set the dirty flag on the specified module so that it will (eventually) be written back to disk. This procedure is provided to give the user direct control of the write-back of a module.

Invocation:

```
GDB_SET_DIRTY_MODULE (GDB_ROOT_PTR, MODULE_PTR);
```

Parameters:

MODULE_PTR = the module to be written to disk (eventually).

### 3.3.13 GDB_CLEAR_DIRTY_MODULE

Clear the dirty flag on a module. WARNING: this is a very dangerous thing to do, so other updates to that module may be lost.

Invocation:

```
GDB_CLEAR_DIRTY_MODULE (GDB_ROOT_PTR, MODULE_PTR);
```

Parameters:

MODULE_PTR = the module whose dirty flag is to be cleared.

## 3.4 Higher Level

Here we describe the available procedures which are a little smarter than the basic data access procedures described above.

Currently, there is but one. This call is provided for moving to the detailed view, for panning, and for a pick (select).

### 3.4.1 GDB_FIND_FRAMES_IN_WINDOW

This routine is not generally used, but it is available.

Invocation:

```
GDB_FIND_FRAMES_IN_WINDOW (GDB_ROOT_PTR, RECTANGLE_PTR,
                           FRAME_TYPE, RESULT_SIZE, @RESULT);
```

The procedure will locate all frames of a specified type whose "recognition area" falls within a rectangular grid area. It accepts a pointer to a rectangle (x1,y1,x2,y2) and the type of frame desired and proceeds as follows: all frames of the specified type are checked to see if the frame's recognition-area intersects the specified rectangle (where boundaries are considered to be included both in the recognition area and in the rectangle). If there is indeed an intersection (any overlap at all between recognition area and the defined window), then the frame's ID will be saved.

The RESULT is structured as follows (all WORD fields):

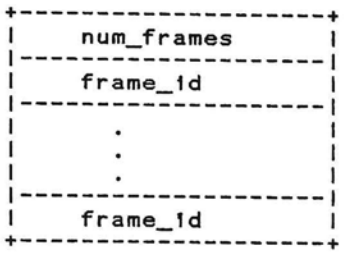



Figure 3 - 3 Frames-In-Window Returned Structure

If the size specified in the result_size parameter is exceeded, the call will terminate unsuccessfully.

## 3.5 Utility Routines

The following utility routines are provided. As with all of the above procedures, these return integer status respecting Daisy standards.

### 3.5.1 GDB_GET_PATHNAME

Invocation:

```
GDB_GET_PATHNAME (ROOT_PTR, @PATHNAME_PTR);
```

Given a file (specified by the root_ptr), returns pointer to the pathname for that file, where pathname means a structure of type cmd_path_name_def. Note that the pointer is to the pathname right in the root of the file. That pathname is READ_ONLY! Do NOT modify it! Rather, if you need to modify it, copy it to your own buffer, and then modify your own copy.

[FIND_FIRST and FIND_NEXT are usually easier to use than these next two.]

### 3.5.2 GDB_MAX_ID

Invocation:

```
GDB_MAX_ID (ROOT_PTR, FRAME_TYPE, @MAX_ID);
```

Given a file (specified by root_ptr), and a frame type, returns the (word value) maximum ID that has ever been allocated for that frame type. For example, if add_frame is called with FRAME_ID = GDB_NIL_WORD (so that VIRGA will allocate the ID automatically), and frames 7, 8 and 9 are thus allocated, and then frames 8 and 9 are deleted, the maximum ID returned will be 9. If no ID has been allocated, MAX_ID = Ø.

### 3.5.3 GDB_NUM_FRAMES

Invocation:

```
GDB_NUM_FRAMES (ROOT_PTR, FRAME_TYPE, @NUM);
```

Given a file (specified by root_ptr) and a frame type, returns the (word value) number of IDs currently in use (the number of frame-id's of that type). If no frames have been deleted, and if no frames have been allocated with IDs out of sequence (by specifying your own IDs), then this number will equal MAX_ID+1, where MAX_ID is the value returned by GDB_MAX_ID (unless num_frames = Ø, in which case MAX_ID = Ø, too).

### 3.5.4 GDB_GET_TIMESTAMP

Invocation:

```
GDB_GET_TIMESTAMP (ROOT_PTR, @TIMESTAMP);
```

Given a file (specified by root_ptr), returns the timestamp indicating when file was last flushed. Timestamp is in Daisy standard format (see UTILP3); it requires 6 bytes, which the caller must allocate. See UTILP3 for routines to translate timestamp to ascii string, etc.

### 3.5.5 GDB_GET_AFT

Given a file (specified by root_ptr), the routine returns the AFT number of the file and the EOF_MARKER. Here, AFT stands for the magic (32-bit) number which the Daisy file system allocates to each open file. That number is usually declared as a pointer, even though it is not a pointer.

Invocation:

```
GDB_GET_AFT (ROOT_PTR, @AFT, @WORK_AREA_PTR,
                        @MOD_SIZE, @EOF_MARKER);
```

The work_area_ptr is used for file system access; see file system documentation. Roughly, the work_area_ptr and the AFT together allow you to access the file directly via the file system. However, if you do that, VIRGA will probably screw up.

The mod_size is the WORD-valued default module size.

The eof_marker is the (dword value) byte offset of the end_of_file, which is equivalent to the size of the file in bytes. Note that the caller must allocate space for both the AFT and the eof_marker, and that both require 4 bytes.

## 3.6 Name Indexes

### 3.6.1 Index Structures

VIRGA allows an optional name-index for each frame_type. If you wish to use an index, you must create the index via GDB_CREATE_INDEX. Later, you may delete the index via GDB_DELETE_INDEX. These routines may be invoked at any time; e.g., you might create_index before adding any frames; or you might add frames to a database without an index, invoke CREATE_INDEX, and then add more frames. After an index exists, VIRGA will update it automatically.

Name indexes are critical to any normal use of VIRGA. For the purpose of testing new programs which call VIRGA routines, you will want to get name indexes which correspond to your test database files, or create them through PDQ. If the name-indexes didn't come with your GDB database, you can also create them with an appropriate SUBMIT .CSD file.

All the VIRGA requests other than index routines operate successfully whether there is an index or not. If there is a name-index, then add_frame, update_frame, and delete_frame respect it in the following sense: Add_frame will add the new frame's name to the index; if it is a duplicate name, it will be added anyway, but negative status will be returned. Delete_frame will remove the frame's name from the index; if it is not in the index, then the frame will still be deleted, but negative status will be returned. Update_frame first

checks if the name and key2 (see below) are unchanged; if either has changed, it will refuse the update request.

The name index itself allows the VIRGA user a second method of direct access to frames. The first method is, of course, to call GDB_FIND_FRAME, which uses the specified frame ID and FRAME_TYPE to find the frame. The specified frame, if it exists, is always found. A maximum of one disk access may be required to get the module with the frame.

The name index looks up a frame by its byte-string NAME, and its WORD-valued KEY2, a variable. Both of these are data within the index frame. When the name_index is created, the user specifies where these data (NAME, NAME_LEN, and KEY2) are located within the frame.

The name_index occupies one or more frames, each of which is a normal frame in all senses. Those index frames, like all frames, have a frame_type, which we call the index_frame_type. The frames whose names are being indexed also have a frame_type, which we call the frame_type. Except when creating an index, the user always specifies the frame_type of the frame whose name is being indexed, rather than the type of the index frames themselves.

Each name_index always has a frame with ID Ø (zero); it is the base index frame and contains slightly more information than the other index frames. It begins with this header (see GDNAM.ELT):

```
    /* index frame Ø continues with this structure */
GDB_INDEX_BASE_HDR  LITERALLY
    'GDB_INDEX_HDR,
    GDB_INDEX_DESC_DEF,
    FRAME_TYPE          BYTE,
    INDEX_FRAME_TYPE    BYTE,
    DUMMYØ(14)          BYTE',
```

Figure 3 - 4 Index Base Structure

where

```
/* each index frame starts with following, after frame hdr */
GDB_INDEX_HDR_1     LITERALLY
    'INDEX_OPCODE  BYTE,
    INDEX_HDR_SIZE WORD,
    SYM_STRUC_DEF,
    OFFSET_SYMTAB WORD,
    MAX_AVAIL     WORD',

GDB_INDEX_HDR_2     LITERALLY
    'DELTA_ENTRIES WORD,
    BASE_FLAGS    WORD,
    KEY2          WORD,
    OFFSET_BUF    WORD,
    LINK_FORWARD  WORD,
    LINK_BACK     WORD,
    DUMMY(17)     BYTE',

GDB_INDEX_HDR       LITERALLY
    'GDB_INDEX_HDR_1,
    GDB_INDEX_HDR_2',
```

Figure 3 - 5 Index Header Structures

and

```
/* application uses this to define index params in create_index */
GDB_INDEX_DESC_DEF      LITERALLY
    'NAME_LEN        WORD;
    NAME_OFFSET      WORD,
    NAME_LEN_OFFSET WORD,
    KEY2_OFFSET      WORD,
    EXTRA_BYTES      WORD,
    FLAGS            WORD,
    INIT_ENTRIES     WORD,
    INCR_ENTRIES     WORD',
```

Figure 3 - 6 Index Descriptor Structure

The GDB_CREATE_INDEX routine is invoked with a pointer to the index_desc[riptor] block, which is copied into the base index

frame. All other index frames begin with the index_hdr, which contains enough information about the index structure that the base index is not usually needed. The descriptor defines where in the frame the name, name_len, and key2 are to be found, some flags (of which only the upper_case flag is implemented), the initial size for the index (in number of entries), and the incremental size (number of entries for which space is reserved whenever the index overflows and needs to be enlarged).

If no KEY2 is desired, the descriptor will contain KEY2_OFFSET = GDB_NIL_WORD, in which case the entire index is contained in one frame. Otherwise, there will be one frame for each value of KEY2. Whenever a frame is added, its key2 value is checked and its name is added to the index_frame with ID equal to that KEY2; if that index frame doesn't yet exist, it is created.

NOTE: an existing NAME or KEY2 must have an offset which clears the header section and puts data clearly into the data area just beyond the header. That is, both NAME_OFFSET and KEY2_OFFSET must be greater than 16.

The general approach to find-by-name, then, is to find the index_frame type (from the frame_types_inventory), find the index_frame for desired key2, and then search that index. This involves a maximum of one disk access, and results in the ID of the desired frame. That frame may then be obtained via find_frame, which could involve a maximum of one more disk access.

Each index frame contains an index in the format of a symtab (see standard Daisy documentation). The symtab consists of two parts: the sym_structure, which is in the frame's index_hdr and contains the size of each entry, number of entries, etc.; and the symtab itself, which is an array of structures, each of which contains a name followed by other information, the index_entry_def.

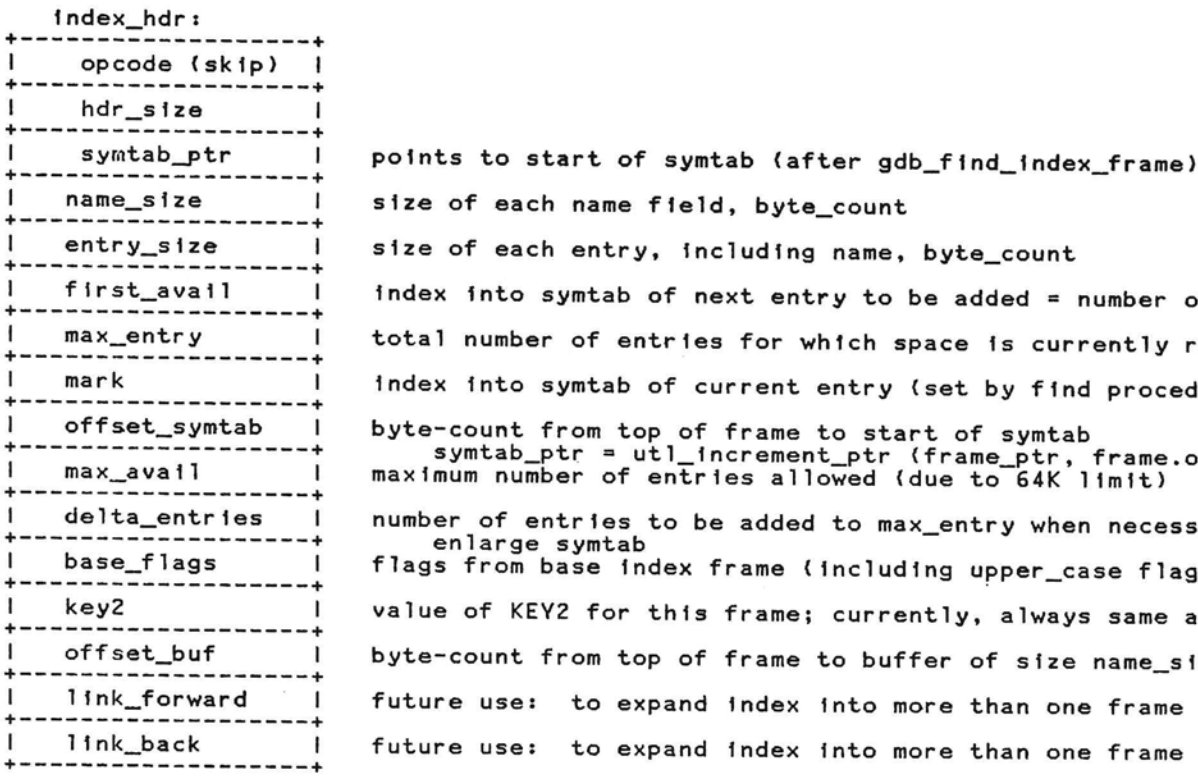



Figure 3 - 7 Index Header Diagram

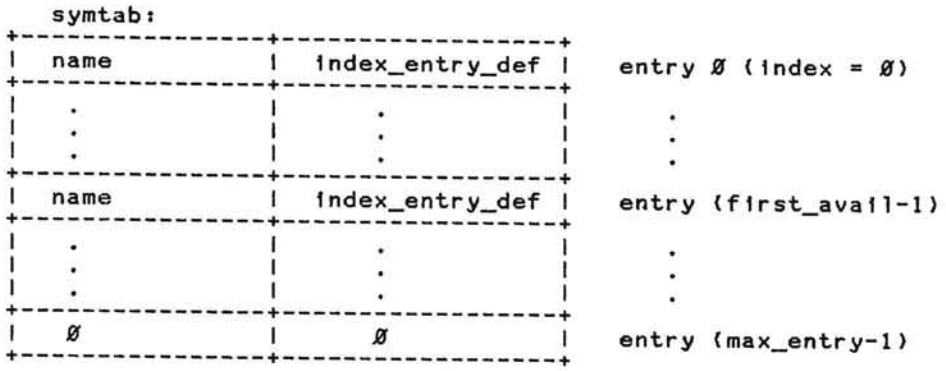
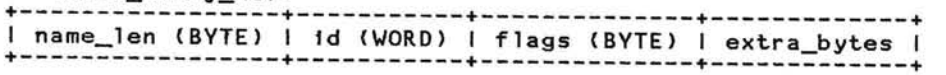



Figure 3 - 8 Symtab Diagram




where the extra_bytes field is N bytes, N specified in descriptor when index is created, for use by application.

Figure 3 - 9 Index Entries Diagram

Currently, the index for a given KEY2 is limited to one frame, which means it is limited to somewhat less than 64K (64K less sizes of module_hdr, frame_hdr, index_hdr, module's end_of_sequence byte, and perhaps minimum frame_size). If there are no extra_bytes in the index_entry_def and the name_size is 16, then this leaves room for about 3000 entries in the index. See the Theory of Operation section for more discussion on sizes.

Almost all VIRGA procedures act on one VIRGA file, which is specified by the file's root_ptr. However, the SEARCH procedures, below, act on a list of files. This list is specified by gdb_files_def (see GDFIL.ELT):

```
GDB_FILES_DEF   LITERALLY
     'NUM_FILES     WORD,
     MARK           WORD,
     FILES_PTR      POINTER',   /* to array of root ptrs */
```

Figure 3 - 1Ø Files Definition Structure

The first field specifies the total number of files, while the last is a pointer to an array of root_ptrs, one for each file. The mark field is an index into this array which can be set by the user or by the SEARCH routines to indicate which file to use or which file contained the found name. It is similar to the MARK field in the sym_structure, but differs in that it is initialized to zero (rather than sym_initial_mark = FFFFH). Also, to allow gdb_search_next_name to continue searching within the same file in which the previous name was found, searches begin with the file to which mark points, rather than the file after that one. That is, the sym_struc.mark indicates one BEFORE the next entry to scan, while the files_def.mark indicates EXACTLY the next file to be searched.

### 3.6.2 Creating and Deleting Name Index

The following procedures allow the user to create and delete a name_index.

### 3.6.3 GDB_DELETE_INDEX

Invocation:

```
GDB_DELETE_INDEX (ROOT_PTR, FRAME_TYPE);
```

This procedure deletes a name_index. The frame_type is the type of the frame whose name is being indexed, not the type of the index_frame itself. If there is no index for specified frame_type, then a non-negative status (gdb_stat_no_index) is returned. Else, all index frames are deleted and the index_type byte in the frames inventory for the specified frame_type is reset to zero, indicating no name_index for that frame_type. The frames of that frame_type are left untouched.

### 3.6.4 GDB_CREATE_INDEX

Invocation:

```
GDB_CREATE_INDEX (ROOT_PTR, FRAME_TYPE, INDEX_FRAME_TYPE,
                  DESC_PTR);
```

This request creates a name_index. If an index already exists, it first deletes it, then recreates it anew. The names of all frames of the specified FRAME_TYPE are added to the index, so that the index is up to date when this call returns.

When names are added to the index, they are mapped to upper-case if GDB_UPPER_CASE_BIT is set in the flags word of the descriptor.

Parameters:

ROOT_PTR specifies the VIRGA file; returned from open or create request.

FRAME_TYPE specifies the type of frame whose name is to be indexed.

INDEX_FRAME_TYPE specifies the type for the index frames. This is the only time the user needs to know this type; after this call, VIRGA maps the frame_types automatically. This frame_type must be included in those specified as legal when the file was created.

DESC_PTR points to a structure of type GDB_INDEX_DESC_DEF (see above and file GDNAM.ELT). Its fields must be initialized by the user before invoking create_index. They have the following meaning, where "offset" is always a byte-count from the top of

the frame (so offset = Ø is the first byte of the frame_hdr, etc.):

```
'NAME_LEN         WORD,   length reserved in symtab for each name entry
NAME_OFFSET       WORD,   offset into frame of first byte of name
NAME_LEN_OFFSET   WORD,   offset into frame of name_len byte
KEY2_OFFSET       WORD,   offset into frame of key2 word's first byte; or
EXTRA_BYTES       WORD,   number of extra bytes to reserved in each symta
FLAGS             WORD,   option bits:  upper_case_bit
INIT_ENTRIES      WORD,   number of entries for which space should be res
INCR_ENTRIES      WORD',  number of entries for which additional space sh
                              be reserved whenever it is necessary to enl
```

Figure 3 - 11 Index Descriptor explained

If KEY2_OFFSET is set to GDB_NIL_WORD, then VIRGA assumes there is no key2, and the name itself constitutes the full index key. In that case, the entire name index is contained in the base index frame (ID = Ø).

Note that this block tells the index routines how to handle the symtab in searching for a frame name, and where to look for the frame's KEY2. .cm

### 3.6.5 Finding Entries in Name Index

The following procedures enable user to locate frames by name. They search the appropriate file(s)' index(ices) to find the entry for the specified name, and return a pointer to that entry. That entry, then, contains the full, exact name, NAME_LEN, frame ID, and (optionally) extra_bytes of application-defined information.

When searching for a name, the user may specify a name with wild cards. The wild cards available are those defined by the Daisy symtab facility. Currently, these are:

| WILD_CARD | Meaning |
|---|---|
| * | match any string |
| ? | match any single character |

The symtab name_match procedure is invoked directly so the results from these VIRGA procedures will be as you expect from symtab. It would be very simple to expand the code to use full regular expressions (as used in TEC). That has not been done partly because there is no current need for that, and partly because it would degrade performance.

### 3.6.6 GDB_FIND_INDEX_FRAME

Invocation:

```
GDB_FIND_INDEX_FRAME (ROOT_PTR, @MOD_PTR, @FRAME_PTR,
                      FRAME_TYPE, KEY2);
```

This procedure is not really necessary. The user may directly invoke first_name, next_name, etc. However, the user may also invoke this procedure to obtain directly the symtab of names for a given frame_type and key2. The procedure maps frame_type to index_frame_type, locates the frame with ID = key2 (or = Ø if key2 = gdb_nil_word), invokes gdb_find_frame to find the frame, and sets the frame's index_hdr.symtab_ptr appropriately. The user may, if desired, lock the frame into memory via the standard gdb_lock_frame request.

To control searching, the user can lock the index into memory (prevents the index from being swapped out of memory), perform an explicit search of the symtab, or directly set the INDEX_HEADER.MARK field, using the FIRST_NAME/NEXT_NAME routines.

Parameters:

ROOT_PTR specifies a VIRGA file; returned from OPEN or CREATE.

MOD_PTR and FRAME_PTR return. They point to the found frame.

FRAME_TYPE is the type of the frame whose name is indexed.

KEY2 is the value of the key2 for which the index is desired; gdb_nil_word if no key2 for that name index.

### 3.6.7 GDB_FIRST_NAME

Invocation:

```
GDB_FIRST_NAME (ROOT_PTR, FRAME_TYPE, NAME_LEN, NAME_PTR,
                KEY2, @INDEX_NAME_PTR, @INDEX_INFO_PTR);
```

This procedure finds the first name in the name index which matches the specified name (name_len, name_ptr) and has the specified key2 (gdb_nil_word if none). It returns two pointers: index_name_ptr points to the name in the index frame so that the caller can see the full, exact name if the input name contained wild_cards; and index_info_ptr points to the gdb_index_entry_def part of the entry in the symtab, which contains the length of the name and the frame ID.

Here, "first" means that it starts at the beginning of the name index (for specified key2). The name_index is not ordered in any meaningful way. VIRGA will reset the mark field in the symtab to sym_initial_mark.

This procedure sets the index_frame_hdr.mark field to point to the found entry.

Parameters:

ROOT_PTR specifies the VIRGA file; returned from gdb_open or gdb_create.

FRAME_TYPE specifies the type of frame whose name is indexed.

NAME_LEN, NAME_PTR specify the name to find. It must be set by the caller. It may contain wild_cards (see above) as supported by Daisy symtabs.

KEY2 specifies the value of the KEY2 for the desired frame. It may be set to gdb_nil_word if there is no key2 for that name index.

INDEX_NAME_PTR is set by VIRGA to point to the found name in the index.

INDEX_INFO_PTR is set by VIRGA to point to the gdb_index_entry_def structure in the found index entry. It contains the length of the name and the frame's ID.

### 3.6.8 GDB_NEXT_NAME

Invocation:

```
GDB_NEXT_NAME(ROOT_PTR, FRAME_TYPE, NAME_LEN, NAME_PTR,
              KEY2, @INDEX_NAME_PTR, @INDEX_INFO_PTR);
```

Find the next name in the index which matches the specified input name (name_len, name_ptr) and has the specified value of key2. Performs the same function as first_name, but searches the name index starting at the entry whose index is (MARK+1). (to see the MARK field, see the Symtab description several pages above). If desired, the caller may directly set the mark field, and thereby determine the starting point of the search.

A first_name followed by successive next_name calls will find all names which match the specified input name. When no more names are found, VIRGA returns gdb_no_such_frame.

Note that this first/next sequence has the same limitation that gdb_find_first and gdb_find_next has: namely, any intervening VIRGA calls could swap out the index frame and invalidate the next_name function. If this is not desired, the user may either: (1) lock the index frame (after finding it with gdb_find_index_frame); or (2) write the application code so that no VIRGA calls intervene between first_name and next_name calls; or (3) remember the MARK field after each first/next call and reset it before succeeding next_name call.

Parameters: -- just like first_name --

### 3.6.9 GDB_SEARCH_FIRST_NAME

Invocation:

```
GDB_SEARCH_FIRST_NAME (FILES_PTR, FRAME_TYPE, NAME_LEN,
        NAME_PTR, KEY2, @INDEX_NAME_PTR, @INDEX_INFO_PTR);
```

This procedure performs the same function as first_name, but it searches a list of files rather than just one file. The list is specified by the gdb_files_def structure, which the caller must initialize. After the call, the files_def.mark will be set to the index of the file in which the name was found.

Parameters:

FILES_PTR points to structure GDB_FILES_DEF, which the caller initializes. At the start of the call, VIRGA will reset the files_def.mark to Ø (first file); at the end of the call, VIRGA will set files_def.mark to the index of the file in which the name was found. If any of the root_ptrs is zero, VIRGA will simply ignore that file and continue on to next file.

-- remainder of parameters just like gdb_first_name --

### 3.6.1Ø GDB_SEARCH_NEXT_NAME

Invocation:

```
GDB_SEARCH_NEXT_NAME(FILES_PTR, FRAME_TYPE, NAME_LEN,
        NAME_PTR, KEY2, @INDEX_NAME_PTR, @INDEX_INFO_PTR);
```

This procedure operates just like search_first_name, except that it continues where the last search_first_name or search_next_name left off. The effect of search_first_name followed by successive search_next_name requests will be to find all matching names in all files.

In more detail: search_first_name will reset the files_def.mark to Ø (the first file in the list), but search_next_name will instead start its search in whatever file files_def.mark currently indicates. Similarly, search_first_name will re-start at the beginning of the index for each file as it steps down the file list, while search_next name will continue searching in the name_index of the current file wherever previous search left off (at the entry whose index is MARK+1). Then, if search_next fails to find the name in that file, it will continue to the next file, starting at the beginning of the index for that file.

Another point of view is that search_next_name is just like gdb_next_name, except that it searches a list of files.

Parameters:

-- just like search_first_name --

# 4 THEORY OF OPERATION

## 4.1 Caveats

In this section we discuss caveats (warnings and exceptions) and general considerations of interest to the user. Where appropriate, references to other sections are in [square brackets].

The following caveats are important!!

1. Find/Next Utilities return a TEMPORARY pointer

[Data Access, Buffer Management, and Buffer Control sections]

The result of all the data find procedures is a pointer to the frame, relative to its module. This pointer should be considered temporary for two reasons:

a. The module resides in a buffer. Any later access request could cause that buffer to be re-used for another module. Therefore, the pointer may no longer be valid.

b. If any later request updates or deletes the frame to which the pointer points, then the frame may be moved, perhaps even to a different module. A pointer saved from an earlier find request may now point to another frame, to the middle of a frame, or to the middle of empty space in the module.

The rule is: Consider the pointer temporary. If you need to retain the data, copy it.

Lock requests can be used to assure that particular frames or modules remain in memory, but then the user must be sure to unlock the specified module or frame. Therefore, the best policy is to:

A. Lock
B. work with the frame
C. Unlock

2. "Next" is read-only.

[Data Access section]
This "next" facility is provided for read-only use; e.g., for scanning through all the nets. It will NOT work if the next requests are interleaved with other requests. The frame_ptr returned points directly to a frame within the module and should be considered temporary.

Given that all FIND_ routines search through the data frames linearly, the definition of "next" is the next frame found that is of the desired type.

An example of the temporary pointer is if you plan to read a net, modify it, then continue with the "next" net: the next facility will not work out. The process of modifying the previous net will invalidate the old pointer to it. The update request will return the new pointer, but if you ask for the "next" net following that one, you may have skipped many nets from the old position of the net (or you may re-examine previous nets), since the update routine may have moved the working frame to a different position.

NOTE: the next_frame request assumes that the module_ptr points to an in-memory module. If any access requests intervene between two next_frame requests, then the module may be swapped out.

There are various ways around the temporary pointer problem. For example, you could make the next request immediately, make a copy of the frame ID, and then work on the current frame. After updating the current frame, make a direct (find_frame) request for the next and loop. Or, you could lock frames or modules.

3. Disk Update

[Buffer Management, Buffer Control sections]
With the pointer returned from find, next, etc., the user has the capability to modify the data to which it points. However, this must be done ONLY with caution. First, since that pointer is temporary (see caveat above), it is the user's responsibility to insure that in fact the pointer is still valid (perhaps by locking the frame or module). Second, it is the user's responsibility to set the dirty_module flag.

4. What if the system crashes?

During the course of LED's execution (or the execution of whatever program uses VIRGA) various parts of the data base are being modified. Some of those changes will be updated on disk; others won't be. If the system crashes, the file will be left partially updated, resulting in a possibly invalid data base. This is very undesirable.

One alternative would be to write to disk every little change as it occurs, but that would entail too much overhead. Another possibility (the one used in DED) would be to write nothing to the disk until the main program terminates. That would insure that the disk file is always consistent, though any work from the current editing session would be lost. Unfortunately, that is not possible, since there is insufficient memory to hold the entire data base.

At present, LED will flush in-memory structures whenever appropriate. Although this flushing-to-disk will automatically take care of the essential cases, some applications may want to specifically flush at important points. For example, the PLACER might periodically flush macro information to disk during the placement process.

Some of the information in the file (and all of it in other files; e.g., the bitmaps) is redundant and could be recovered. Therefore, it is possible to use a recovery program (DOCTOR) that can help resurrect an LED data base after a crash, though

some cases will probably be unrecoverable. At present, DOCTOR is not completed, and patches must be done through PDQ, in hex.

5. Locking is binary

Note that the process of locking and unlocking is binary; that is, a lock bit is either set or cleared. For example, there is no difference between locking a frame once and locking it twice (except that the second lock request will return with a non_negative, non_success status which indicates that the frame was already locked). For example, if you lock a frame twice and unlock it once, the frame is completely unlocked (and might get swapped out by VIRGA).

6. Toggling of Search Direction

This is a performance consideration. When searching sequentially through a set of frames, it is desirable to toggle the direction of search before starting on the next frame-type; e.g., if the last time you searched you searched starting with net A and ending with net B, then next time start with net B and end with net A. This consideration only applies to searches for data frames, which of course are in modules in main memory. This does not apply to searches through a name index.

Alternating the direction of search will improve the interaction of sequential searches with the LRU algorithm used for buffer replacement. If each frame-type search is in the same direction, the search will reach the end of the buffers, and have to read again from disk. Then, each new module to be read will cause a fault and a disk read; with this toggling, fewer disk reads will be necessary in the case of a multiple frame-type search.

## 4.2 Internal File Structure

This section is a description of VIRGA's internal tables. VIRGA readers should not really read any of this, since it is recorded only for the purpose of maintaining VIRGA's internal code. Certainly, no users should write any code depending on these structures since a main advantage of VIRGA is its application-independence.

On a per-file basis, four inventories are maintained to speed access to individual modules and frames. First, each file has an inventory of all its modules. Second, each frame-type has an inventory of its kind of frames and what modules they are in. Third, a table exists (the Frame-Types Inventory) which has one entry for each frame-type, and has pointers to each frame-type's frame-inventory. Finally, there is a global buffer inventory which contains information on all memory buffers used for all files. The first three are file-resident and permanent. The last one is created when VIRGA is initiated, and discarded when VIRGA is terminated.

### 4.2.1 The module inventory

Every VIRGA module has an ID in its module header which is a direct index into the module inventory array. That is, module Ø has an ID which points into the array entry for module Ø, the next entry is for module 1, etc.

The module inventory is used by the buffer manager (GDBUFR, gdb_read_module, gdb_write_module) to read/write a module from/to disk. In particular, the inventory has a disk marker which specifies where each module is on disk. The module inventory is also used by the GDB_ALLOCATE_FRAME routine in deciding if a module has sufficient space for a new frame without having to read from disk.

The module inventory (per file) contains an entry for (at least) every module in the file, and has the following structure:

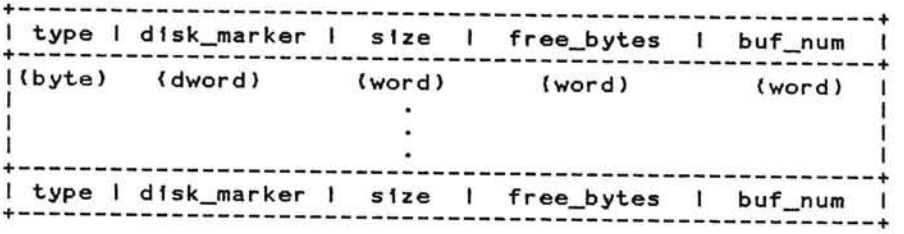



Figure 4 - 1 The Module Inventory for a File

The type field is a byte which specifies the module number (ØFFH [GDB_NIL_BYTE] if module doesn't exist)

The disk_marker field is DWORD, and specifies the start of the module on disk as a byte offset from the start of the file.

The remaining fields are of type WORD and specify:

the module size,

the number of free_bytes remaining in the module, and

the number of the buffer in memory in which the module

currently resides (ØFFFFH [GDB_NIL_WORD] if module is not in memory).

### 4.2.2 The Frame Inventory

Each frame-type has an inventory, called the Frame Inventory, which contains at least one entry for every frame of that type. Every frame-type has this inventory.

The format is as follows:

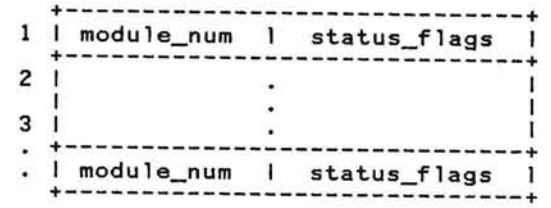



Figure 4 - 2 The Frame Inventory

The module_num (WORD) specifies which module contains this frame (ØFFFFH [GDB_NIL_WORD] if none). The status flags specify various status conditions (e.g., frame is locked).

### 4.2.3 The Frame Types Inventory

For each file there is one table besides the Module Inventory, with an entry for each frame type. This is the Frame-Types Inventory.

Its format follows:

An Inventory of Frame Inventories:

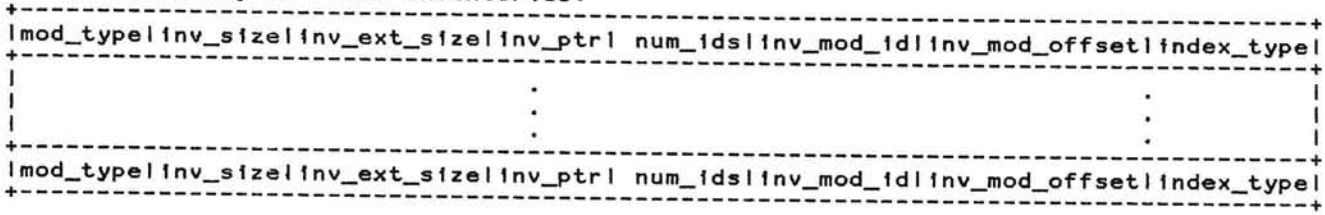



Figure 4 - 3 The Frame-Types Inventory

The Frame-Types Inventory is used to find, for a given frame type, the associated frame inventory. The inv_ptr is an in-memory pointer which is initialized when the file is opened.

The first three fields specify the module type, the current size of the inventory, and the size by which it will be expanded when it overflows.

The num_ids field specifies the total number of frame IDs (of that frame_type) allocated so far.

The inv_mod_id and inv_mod_offset specify the location of the frame containing this frame_type's frame-inventory.

The index_type field contains the frame-type of the index, if an index exists, or a zero if no index exists. The index type, if there is no name index, is set to zero. If there is a name index, this field holds the type of frame (frame_type) that holds the index.

### 4.2.4 The Buffer Inventory

Finally, there is an inventory of memory buffers. It has an entry for each buffer. It is a single global inventory for all files. The inventory is created only during run-time when GDB_INITIATE is invoked, and is discarded when GDB_TERMINATE is called.

The inventory has the following form:

A (global) Buffer Inventory:

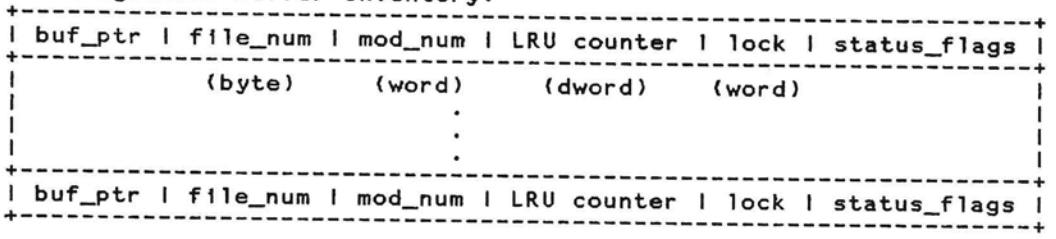



Figure 4 - 4 The Buffer Inventory

Here, the buf_ptr points to the in-memory buffer holding the module (Ø if none).

The file_num (BYTE) specifies which file and the mod_num (WORD) which module of that file occupies the buffer.

The LRU counter (DWORD) is used (as described in the section, Module Buffering) for the buffer-replacement algorithm. Note that all files contribute to the buffer inventory, and the LRU counter operates over all buffers. This means that a read request for a module in one file might cause a module of a different file to be swapped out.

The lock (WORD) counts how many locks have been nested on this module [see Locking/Unlocking, in the Buffer Control section].

The status_flags indicate various status conditions; e.g., whether the module is dirty, locked, module-locked (as opposed to frame-locked), etc.

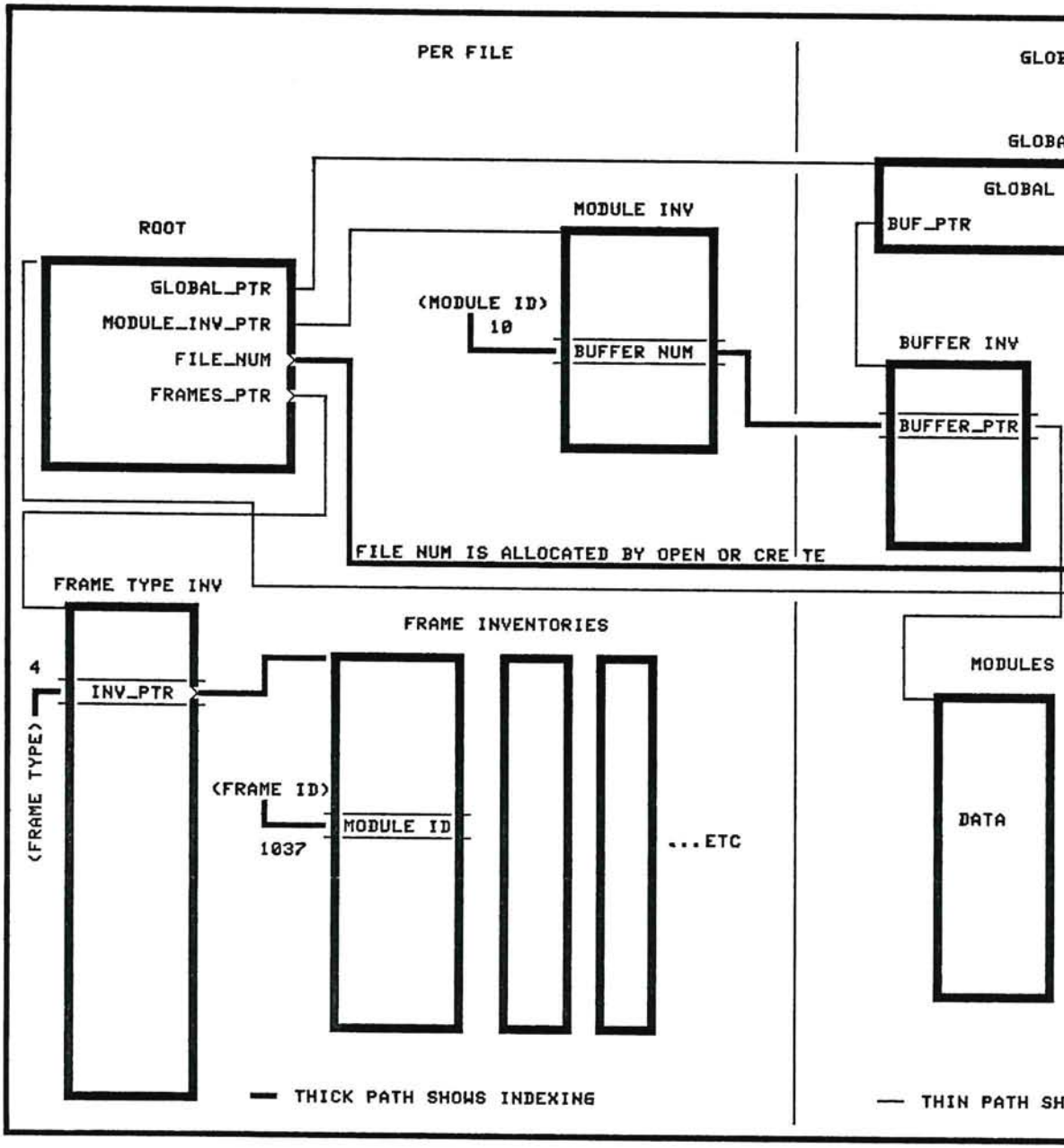

PER FILE
GLOB
GLOBA
GLOBAL
BUF_PTR
MODULE INV
ROOT
GLOBAL_PTR
MODULE_INV_PTR
FILE_NUM
FRAMES_PTR
(MODULE ID)
10
BUFFER NUM
BUFFER INV
BUFFER_PTR
FILE NUM IS ALLOCATED BY OPEN OR CRE TE
FRAME TYPE INV
FRAME INVENTORIES
MODULES
4
INV_PTR
(FRAME TYPE)
(FRAME ID)
1037
MODULE ID
...ETC
DATA
THICK PATH SHOWS INDEXING
THIN PATH SH

Figure 4 - 5 VIRGAs Internal Tables

### 4.2.5 The find-frame-algorithm

The Frame Inventory is used by GDB_FIND_FRAME as follows: the routine starts at the root module, by using the root pointer. From the root, it gets the frames_ptr to find the Frame-Types Inventory. It uses the user-specified frame-type to index into it, which yields a pointer to the correct Frame Inventory.

It then uses the user-specified frame-ID to index into the frame inventory and find the module ID. Using the root.module_inv_ptr, it finds the module inventory and, using the module ID, indexes into it to find the buffer number.

Using the already known global_ptr, it finds the buffer-inventory and uses the buf_num to look up the buffer_ptr, which points to the necessary module in memory. (Note: if buf_num = GDB_NIL_WORD, the module is not in memory so it reads from disk.) Finally, within the module in memory, it does a linear search to find the desired frame.

## 4.3 Source Code Modules

GDNTFC - An interface (.A86) that allows VIRGA routines to call LED routines

GDINIT - Initiate/Terminate VIRGA operation

GDSCFG - Set Configuration Table, assuming that the Symtab for it already exists.

GDCNFG - Create and Initialize Configuration Table

GDHILR - ('HIgher Level Routines') At present, this is just one routine: GDB_FIND_FRAMES_IN_WINDOW.

GDNIDX - Name Index manipulation routines

GDNAME - Find and match names in name index

GDNCRT - Create/Delete a Name Index

GDFRAM - Find/Add/Update/Delete frames

GDLOCK - Lock/Unlock frames and modules

GDDRTY - Set/Clear the dirty flag on a frame or module

GDNEXT - Find first or next frame

GDUTL - A tollection of utility routines

GDFIL - Create/Open/Close VIRGA files

GDEXP - Expand inventory frames and modules

GDALOC - Allocate/De-allocate frames and modules

GDMOD - Intra-module memory manager

GDBUFR - Buffer Manager: Read/Write/Swap/Flush/Get modules.

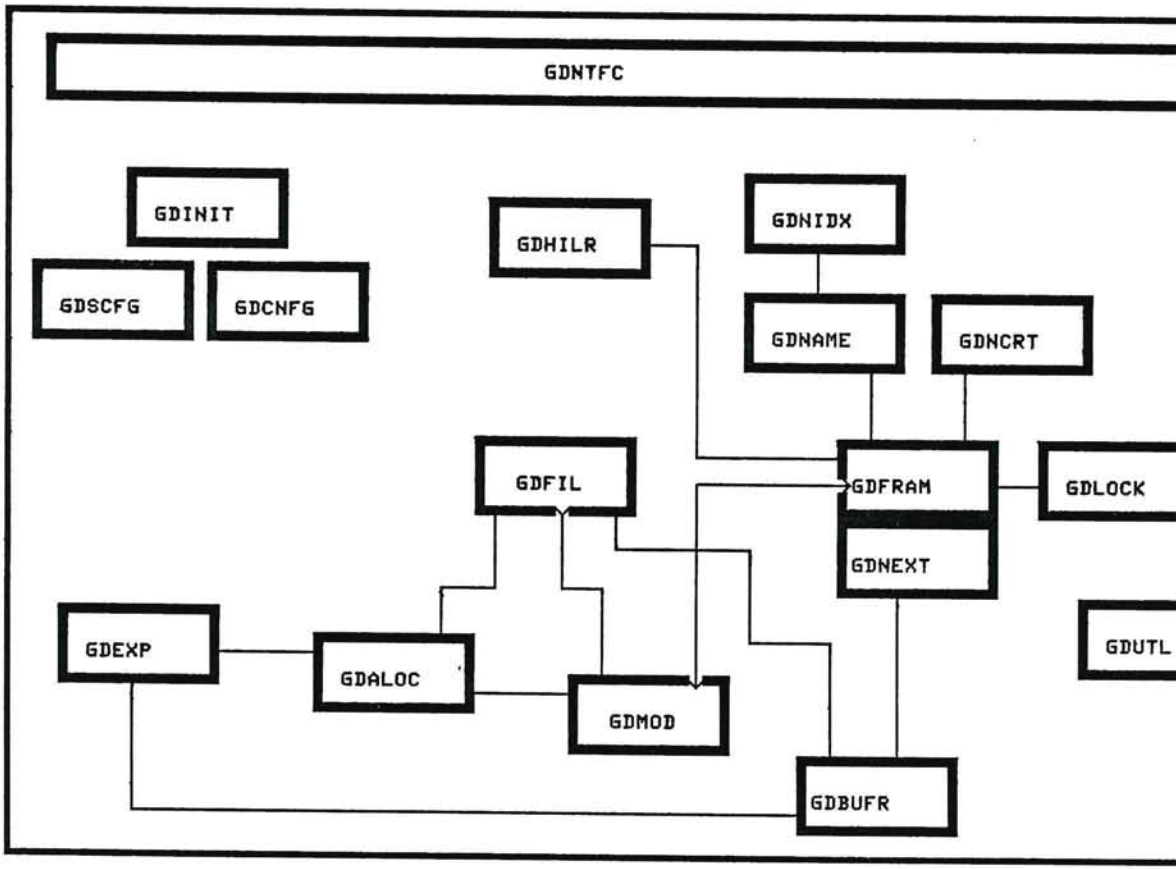



Figure 4 - 6 The Source Modules

## 4.4 Enhancement Considerations

### 1. Window Sequential Search

Note that the window call is a sequential search through all frames (of specified type) on disk. This search could become a performance bottleneck.

If a bottleneck occurs, two enhancements can be undertaken. First, double buffering could be used in a new call that would display the nets as it reads them. Users of the window request should structure their code so that an upgrade to this double buffered method would be simple.

Second, a structure could be maintained which would describe EXACTLY which nets cross each area of the chip, where "area" is a suitably defined subset of the chip, and the set of all areas covers the chip. [Anything less than exact (i.e., any approximate descriptions or "hints") would still require a sequential search to make sure we didn't miss anything.] This would involve adding a data structure to hold this information, and the add_frame call would have to be upgraded to update that data structure. The areas could be chosen as a grid of rectangles (similar to checks, but a derived data structure); they could be chosen to coincide with the rough router's divisions of the chip, with the organization of the static data, with the cached page buffers used to speed panning, or with some other structure.

2. Net Locality on Disk

[net locality is presently being used in PE-1Ø, but in that application, it is called, "geometric optimization"]

Net locality on disk would enhance VIRGA performance, since multiple get/put requests could be performed on the nets without actually going to disk. At the outset no extra code will be written to achieve net locality. The natural tendency to use a group of nets which are localized on the chip will, we expect, tend to localize them on disk (since the put requests will tend to write them into the same buffers). Before optimizing everywhere, we will first wait to see how much locality is achieved by this natural use of nets. Should this prove to be insufficient, then we can implement a forced-locality scheme, such as the one we now describe. Note, however, that (at least some) nets are NOT local; i.e., they wander over the whole chip. And other nets, even though local, may not be local to the chosen areas; e.g., the area boundary might cut across a channel and therefore divide some nets. Because of this and because of the disk management and desire to minimize disk space, we see no way to insure absolute locality. Thus, it should never be assumed that some set of nets are in fact located in the same or neighboring modules.

To achieve locality we could divide the chip into areas (probably the same areas chosen for the window optimization, described above). Each such area will be assigned a set of modules. [If the expected number of modules per area is less

than one, then the set could be a single module and two or more areas could be assigned to the same module.] Then, add_frame is modified so that, when it assigns a net to a module, it first applies a rule to determine which area "owns" the net and then tries to put the net in that area's module(s). If the module(s) is(are) full, then a new (overflow) module could be allocated. The overflow modules could be maintained intelligently (e.g., linked to the corresponding area), or could just be used randomly. The rule that assigns nets to areas could be dumb (e.g., pick any point on the net and choose the area in which that point lies) or progressively smarter (e.g., choose a point on the net which is closest to the centroid of the net). These rules, even if rather fancy, need not be table look-ups, but could be simple arithmetic formulae, and therefore new redundant data structures need not be created. For example, if the net lies in area A, I is the first net module, M is the module size, S is the expected size for nets in an area (possibly padded if the variation is large), then the net should be put into module number:

[for M > S] (A / (M/S) ) + I

[for M < S] (A * (S/M) ) + I + j

[where Ø <= j <= (S/M - 1)]

3. Expanded name index

For the expected application, the present name index size is sufficient; e.g., in GM-1Ø, this will limit the total number of macros to 3ØØØ, and perhaps the number of components or nets per DED page to 3ØØØ. Since the original design limit of GM-1Ø was 2ØØØ nets altogether, 3ØØØ per page doesn't sound like a severe limit. However, if necessary, the code can be enhanced to create a linked list of index frames. In that case, the link_forward (and perhaps link_backward) fields can be used to store the frame ID of the next (previous) frame in the list, while the key2 field specifies to which key2 this frame belongs. Still, probably, the frame with ID = key2 would be the head of that linked list. When the first index frame for a given key2 is created, it may be necessary to re-label (change the ID of) a pre-existing index frame which is using that ID but belongs to another key2. Again, all this is a description of hooks for future expansion; today, each index for a given key2 is wholly contained in one frame, and the code assumes that fact.

# 5 APPENDICES

## 5.1 Summary of Procedures

All procedures return integer status.

Section III:

```
GDB_GET_CONFIG (PARAM_BLOCK_PTR, @CONFIG_PTR)
GDB_INITIATE (@GDB_GLOBAL_PTR, GDB_CONFIG_PTR);
GDB_TERMINATE (GDB_GLOBAL_PTR);

GDB_CREATE (@GDB_ROOT_PTR, GDB_GLOBAL_PTR, PATH_NAME_PTR,
                    GDB_FILE_DESC_PTR);
GDB_OPEN (@GDB_ROOT_PTR, GDB_GLOBAL_PTR, PATH_NAME_PTR, MODE);
GDB_CLOSE (GDB_ROOT_PTR, PURGE_FLAG);
```

Section IV:

```
GDB_FIND_FRAME (GDB_ROOT_PTR, @MODULE_PTR, @FRAME_PTR,
                    FRAME_TYPE, FRAME_ID);
GDB_ADD_FRAME (GDB_ROOT_PTR, @MODULE_PTR, @FRAME_PTR);
GDB_UPDATE_FRAME (GDB_ROOT_PTR, @MODULE_PTR, @FRAME_PTR,
                    NEW_FRAME_PTR);
GDB_GROW_FRAME (GDB_ROOT_PTR, @MOD_PTR, @FRAME_PTR, NEW_SIZE);

GDB_DELETE_FRAME (GDB_ROOT_PTR, MODULE_PTR, FRAME_PTR);
GDB_DELETE_BY_TYPE (GDB_ROOT_PTR, FRAME_TYPE);

GDB_FIND_FIRST (GDB_ROOT_PTR, @MODULE_PTR, @FRAME_PTR, FRAME_TYPE);
GDB_FIND_NEXT (GDB_ROOT_PTR, @MODULE_PTR, @FRAME_PTR);
GDB_GET_MODULES_BY_TYPE (GDB_ROOT_PTR, MODULE_TYPE);
```

Section V:

```
GDB_FLUSH (GDB_ROOT_PTR);

GDB_ALLOCATE_MODULE (GDB_ROOT_PTR, @MODULE_PTR,
                        MODULE_TYPE, SIZE);
GDB_ALLOCATE_FRAME (GDB_ROOT_PTR, MODULE_PTR, @FRAME_PTR,
                        FRAME_TYPE, SIZE);

GDB_DISPOSE_MODULE (GDB_ROOT_PTR, MODULE_PTR);
GDB_DISPOSE_MODULES_BY_TYPE (GDB_ROOT_PTR, MODULE_TYPE);

GDB_LOCK_FRAME (GDB_ROOT_PTR, MODULE_PTR, FRAME_PTR);
GDB_UNLOCK_FRAME (GDB_ROOT_PTR, MODULE_PTR, FRAME_PTR);
GDB_LOCK_MODULE (GDB_ROOT_PTR, MODULE_PTR);
GDB_UNLOCK_MODULE (GDB_ROOT_PTR, MODULE_PTR);
GDB_LOCK_MODULES_BY_TYPE (GDB_ROOT_PTR, MODULE_TYPE);
GDB_UNLOCK_MODULES_BY_TYPE (GDB_ROOT_PTR, MODULE_TYPE);
```

```
GDB_SET_DIRTY_MODULE (GDB_ROOT_PTR, MODULE_PTR);
GDB_CLEAR_DIRTY_MODULE (GDB_ROOT_PTR, MODULE_PTR);
```

Section VI:

```
GDB_FIND_FRAMES_IN_WINDOW (GDB_ROOT_PTR, RECTANGLE_PTR,
                           FRAME_TYPE, RESULT_SIZE, @RESULT);
```

Section VII:

```
GDB_GET_PATHNAME (ROOT_PTR, @PATHNAME_PTR);

GDB_MAX_ID (ROOT_PTR, FRAME_TYPE, MAX_ID_PTR);
GDB_NUM_FRAMES (ROOT_PTR, FRAME_TYPE, @NUM);

GDB_GET_TIMESTAMP (ROOT_PTR, @TIMESTAMP);

GDB_GET_AFT (ROOT_PTR, @AFT_NUM, @WORK_AREA_PTR, @MOD_SIZE,
              @EOF_MARKER);
```

Section VIII:

```
GDB_DELETE_INDEX(ROOT_PTR, FRAME_TYPE);
GDB_CREATE_INDEX(ROOT_PTR, FRAME_TYPE, INDEX_FRAME_TYPE,
                           DESC_PTR);

GDB_FIND_INDEX_FRAME(ROOT_PTR, @MOD_PTR, @FRAME_PTR,
                     FRAME_TYPE, KEY2);

GDB_FIRST_NAME(ROOT_PTR, FRAME_TYPE, NAME_LEN, NAME_PTR, KEY2,
        @INDEX_NAME_PTR, @INDEX_INFO_PTR);
GDB_NEXT_NAME(ROOT_PTR, FRAME_TYPE, NAME_LEN, NAME_PTR, KEY2,
        @INDEX_NAME_PTR, @INDEX_INFO_PTR);

GDB_SEARCH_FIRST_NAME(FILES_PTR, FRAME_TYPE, NAME_LEN, NAME_PTR,
                      KEY2, @INDEX_NAME_PTR, @INDEX_INFO_PTR);
GDB_SEARCH_NEXT_NAME(FILES_PTR, FRAME_TYPE, NAME_LEN, NAME_PTR,
                     KEY2, @INDEX_NAME_PTR, @INDEX_INFO_PTR);
```

## 5.2 Summary of Public Modules

The relevant ELT and EXT files are listed below:

```
ELTS:
   GDINT.ELT  --  internal to VIRGA; do NOT use this.
   GDFIL.ELT  --  all constants and structures except name-index
   GDNAM.ELT  --  name-index-related constants and structures
   GDERR.ELT  --  error (status) literals

EXTS:
   GDFIL.EXT  --  file-access (create, open, close, etc.)
   GDDAT.EXT  --  data-access (find_frame, add_frame, etc.)
   GDBUF.EXT  --  buffer-control (lock_frame, dispose, etc.)
   GDHIL.EXT  --  higher-level (window)
   GDUTL.EXT  --  utilities (max_id, get_pathname, etc.)
   GDIDX.EXT  --  name-index create, delete, etc.
   GDNAM.EXT  --  name-index (find_index, first_name, etc.)
```

## 5.3 Version Changes

Version changes are listed here, going backwards, chronologically.

** Changes V3.Ø.Ø1 to V3.Ø.Ø3 **

1. Made changes necessary for O.S. 4.Ø

2. Updated the INCLUDE of GDINT.ELT to reflect /fØ/gm1Ø.

3. Made several routines re-entrant: e.g., GET_MODULE, FIND_FRAME.

4. Added .index_type to the frames_inventory.

5. Added GDB_ADD_FRAME_TYPE.

** Changes V1.Ø.5 to V3.Ø.Ø1 **

1. LINIT was replaced by LINPUT, which was replaced by SLIDE. SLIDE also incorporates the functionality of LUPDATE; i.e., it performs engineering update. Also, FNL was incorporated into SING, so the .FNL file is really the .SING file.

2. Get_modules_by_type not only gets the modules but also locks them.

3. Name-indices were added, along with the corresponding calls to create/delete name index, find entries in the name index, etc.

4. Added gdb_grow_frame, gdb_delete_by_type.

## ** Changes V1.Ø.Ø to V1.Ø.5 **

This document (Rev 1) describes VIRGA V1.Ø.5, released on 9/14/82. In this section we list the changes that have been made since the initial release (V1.Ø.Ø, 2/18/82).

1. ID and SIZE fields inverted (V1.Ø.2)

In the gdb_frame_hdr, the order of the ID and SIZE fields were inverted. Old code need only be recompiled. Old databases must be converted or discarded. Note that VIRGA will recognize data bases as incompatible in this sense, and will refuse to open them.

2. Open accepts MODE (V1.Ø.2)

The gdb_open procedure accepts an extra parameter, the MODE. This mode is exactly the same as the Daisy file system mode; i.e., it should be fsys_read_mode, fsys_write_mode, or fsys_nqa_mode.

3. By_type procedures accept type = nil for ALL types (V1.Ø.5)

The various _by_type procedures now accept type = gdb_nil_byte (=ØFFH) to mean ALL types. These procedures are:

```
 GDB_GET_MODULES_BY_TYPE
*GDB_DISPOSE_MODULES_BY_TYPE
 GDB_LOCK_MODULES_BY_TYPE
*GDB_UNLOCK_MODULES_BY_TYPE
```

Those preceeded by an asterisk (*) interpret nil type to mean all types except gdb_control, which type contains VIRGA's internal inventories. Thus, you may safely dispose or unlock

all types, and VIRGA's inventories will still remain in memory (as they must for VIRGA to execute).

4. Add_frame uses caller_specified ID (V1.Ø.5)

Add_frame now uses the ID specified by the caller (in the new_frame, the copy of the new frame), unless the caller specifies ID = gdb_nil_word, in which case VIRGA will automatically allocate a unique ID. If the specified ID is already in use, then VIRGA will refuse the add_frame request and return a negative status.

*** INDEX ***