TABLE OF CONTENTS

LIST OF FIGURES

Each frame-oriented chapter has a 'map' of the data structures that can be found in the frame.

1 INTRODUCTION

This document is almost entirely devoted to a description of the contents of the .LED file. Since the .LED file completely describes the current state of a user's gate array, it is the central file in the GATEMASTER cluster of files, and it is a critical focus for the actions of most GATEMASTER programs.

Chapters 1 - 3 of this document provide an overview of the GATEMASTER database. Chapter 2 contains a chart of the various database files, a figure depicting the frame types in the .LED file in their general relationship to other GATEMASTER data, and critical details of the database structure. Chapter 3 describes some basic data structures and constants that are used over and over again in the database.

After those preliminaries, this document proceeds to describe the 25 types of frames which may be found in the .LED file. Each frame 'type' is the model for every instance of that frame type in the file. For example, the chapter on 'The NET Frame' is a discussion of the data organization which is common to every NET frame.

Each type of frame has its own chapter in this manual. Each chapter describes every data field, the processes which set and access the field, and the meaning and value of the data in the field.

Since all the frame-oriented chapters were written by adding annotations to a copy of the file which literally declares the database schema, the major part of this document (consisting of chapters 4 to 30) is an exploded, annotated version of the actual database schema.

Since the .LED file is so important, it is simply referred to as 'the database'.

This document is intended to be good both for self-instruction and for quick reference. Since the database's many frame types are described in an order that makes sense for the self-instruction purpose, the reader who wants to find a particular frame type should look in the index at the document's end.

To save on space and avoid extensive redundancy, this document often refers to other GATEMASTER documents. The beginning GATEMASTER programmer must read these manuals in an order which helps comprehension. First, experiment with LED (the interactive editor) with the GATEMASTER Reference Manual in hand. Second, try using BASE (the input compiler) and see the GAIL manual for hints on putting data into the GATEMASTER system. Third, read the VIRGA manual in detail to learn the memory/database manager. Almost all the database files are in

VIRGA format. Fourth, skim the PDQ manual to see the possibilities for interactive query of a database file. After you have seen the GATEMASTER functionality and tools, the purpose and shape of the database in this manual will make sense.

The GATEMASTER database exists in the Daisy computer as a cluster of VIRGA-type files, and there are several ways to access its data. First, one may use PDI (high-level) or VIRGA (lower-level) to access the database by way of procedural routines in application programs. PDI documentation exists in the GATEMASTER group. The module-and-frame structure of VIRGA is documented in the VIRGA Technical Manual, Design File Doc. 4.21. Second, one may access the database through the PDQ query program at the command level. For that purpose, see the PDQ Technical Manual, Design File Doc. 4.22.

Basically, the .LED file contains the main data for GATEMASTER. This manual describes the schema (the logical organization) of the .LED file from the schema point of view (oriented toward data structures). The schema is a file which contains many PLM declarations for data fields; as a whole data structure, it is an exact model of the database.

One other point of view (besides the schema) is to look at the data from the perspective of programs and subsystems: to see how the different programs deal with all the fields. However, this kind of information is less stable than the schema, and more specialized. Thus, a programmer must go back to comments in the code or read a design spec for that point of view.

2 DATABASE OVERVIEW

2.1 The Database Files

Below is a chart of all the GATEMASTER database files.

TYPES OF FILES IN THE GATEMASTER SYSTEM

Which Type of File	Encoding & Format	Contents (What Data)	The Purpose of the File
.SING	Text. Blocks.	Circuit logic (path_id's, netlist, text, parameters).	Brings logical data and accompanying text into the GATEMASTER. Serves as one input to SLIDE.
.GAIL	Text. Blocks.	Either a base array or a macro library.	Allows gate array vendors to describe their base arrays and macros for input to the GATEMASTER.
.BASE	Binary. VIRGA modules, frames.	A compiled base array before its merger with data from the .SING file.	Holds a compiled base array before it is merged with the netlist data. Serves as one input to SLIDE.
.MLIB	Binary. VIRGA modules, frames.	A compiled library of macros.	Holds all data for macros that the LED user may want to use for placement. LED copies a macro into the .LED file when it is needed.
.LED	Binary. VIRGA modules, frames.	The SLIDE-integrated combination of the .SING file and the .BASE file.	Holds circuit logic, base array, macros, placement, & routing for fast access in during layout operations.
.BAK	Same as .LED	Same data as in the .LED file.	The .BAK file backs up the user's .LED file in case of accident.

TYPES OF FILES IN THE GATEMASTER SYSTEM (continued)

Which Type of File	Encoding & Format	Contents (What Data)	The Purpose(s) of the File
.BMP	Binary. VIRGA modules, frames.	Detail Bitmap. Has all the obstructions to vias, pins, and traces.	Allows the Detail Router to perform design-rule checks and detail routing quickly.
.RAIN	Binary. VIRGA modules, frames.	Rough Bitmap. Has all the rough routing capacities & utilizations.	Allows the Rough Router to perform its routing quickly.
.SCRN	Binary. Not a VIRGA file. Page Buffer Image.	Graphics for the entire gate array in all 3 views.	Permits fast access to the 3 graphic displays: Overview, Rough View, and Detail View.
.MAKE	Text. Blocks.	Logic, text, path_id's, parameters, placement, and routing from a .LED file.	1. Send a completed gate array layout to a manufacturer. 2. Send a partially completed gate array layout to a non-Daisy computer for placement and/or routing.
.GET	Text. Blocks.	A gate array layout with some placement or routing already performed.	Brings a gate array which has been partly placed or routed on another system into the GATEMASTER.

The database for the GATEMASTER system consists primarily of the .LED file. The .LED file contains a complete record of the current state of a gate array layout. In addition, temporary files are used: the .BMP file for the run-time Detail Bitmap, .RAIN for the Rough Bitmap, and the .SCRN file for the page buffer image.

VIRGA will support the .LED file, the .BMP (bitmap) file, and any other temporary files the user may wish to use with any content and structure (within the VIRGA framework).

Another VIRGA file is the .BASE (base array) file. The base array shows the gate array before SLIDE has merged the netlist with a gate array file. A typical GATEMASTER system is oriented toward a customer's choice of a particular gate array vendor. Furthermore, the system will have multiple .BASE and .MLIB files in order to handle the vendor's various gate array families. The base array formerly had the name, "virgin array". The reader might find this name in old documentation.

For each base array, a set of corresponding .MLIB files contain all the pre-defined macros that can be used.

Another relevant file is the SLIDE.SING file, which constitutes the input to SLIDE when it creates the .LED file or performs an engineering update. See the GATEMASTER Reference Manual for a description of SLIDE'S operation.

The .MAKE file and the .GET file are files in GAIL format. [The .GET file is not yet implemented. The .GAIL file will take placement and routing from a non-Daisy computer and put that placement and routing into the .LED file.]

When the user invokes the LED program, LED opens the specified .LED file, opens various .MLIB files, and generates the secondary files if they do not already exist.

2.2 The .LED file

The next page has a picture of the .LED file in its general relationship to other GATEMASTER files involved in the user's data flow. The NOTE frame is enclosed in parentheses to indicate that it is never found in the .LED disk file. It is a run-time data structure only.

Where does the data in the database come from? BASE and SLIDE fill in most of the data fields. Other information comes from the actions of the LED user.

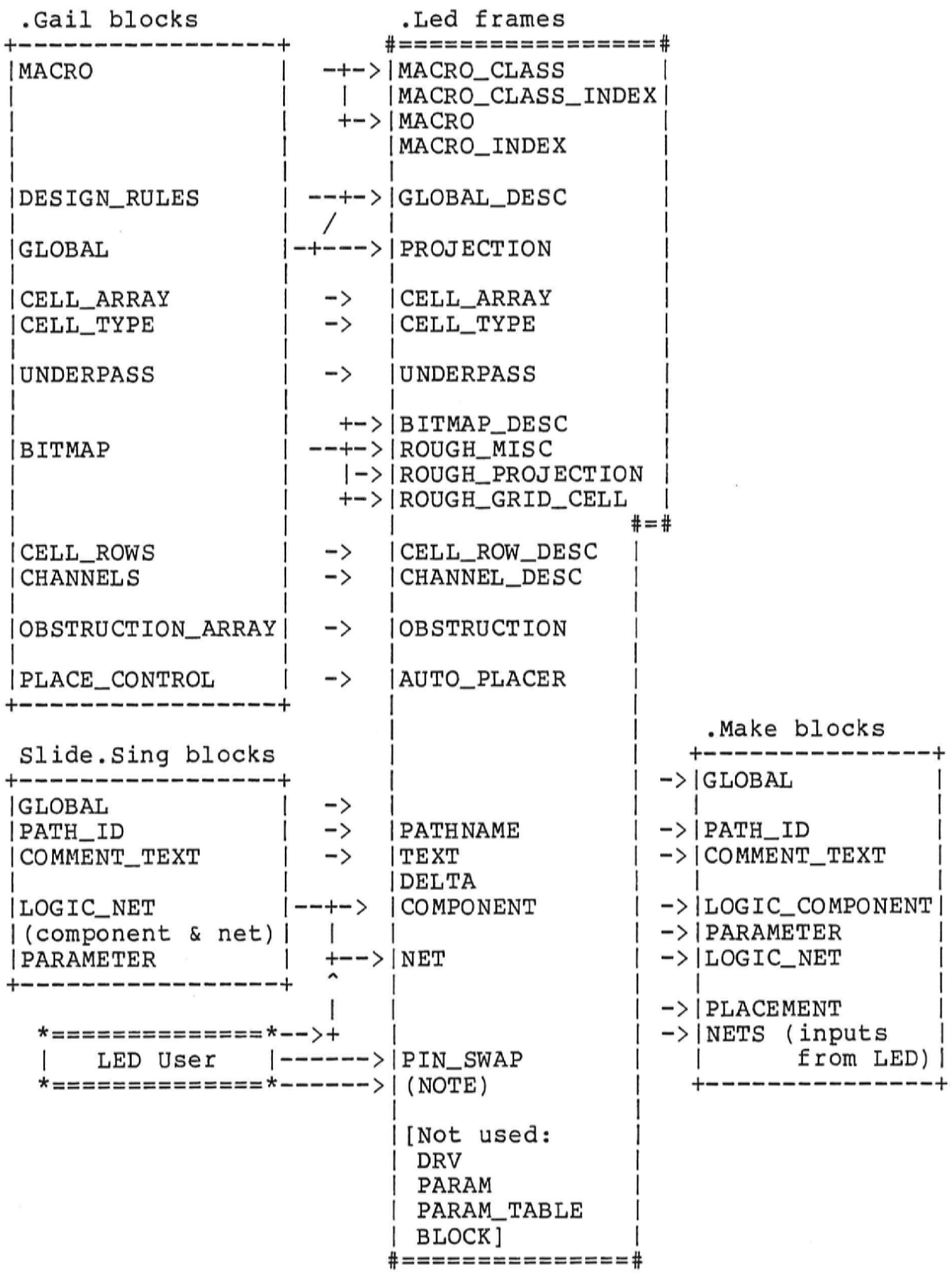


Figure 2 - 1 Overview of .LED File Data Input/Output

2.3 Accessing the Database

Almost all the programs in the GATEMASTER system use VIRGA routines to access the database. Although PDI (Procedural Database Interface) is a higher-level way of accessing the database, it is based on VIRGA routines, too. For this reason, it is essential that GATEMASTER programmers read the VIRGA manual thoroughly and experiment with PDQ in order to see the VIRGA database structure.

2.3.1 VIRGA

VIRGA is a set of file-access routines which can quickly retrieve or handle virtualized memory segments for a specially constructed file (a file in VIRGA format). VIRGA's functions are: efficiently put data into central memory; offer selective locking on all frame types; maintain name indexes and common pointers; manage a large data file on disk (tree-structured); provide variable length records, but with random access; provide fast search routines and database utilities. For further information, please see the VIRGA manual.

2.3.2 PDQ

PDQ is the only way of directly reading the database and writing to it from the user level. It is an R&D tool for database query; it is not available to Daisy customers. To learn how to open, read, and modify a database through PDQ, please see the PDQ manual.

2.3.3 PDI

PDI stands for Procedural Database Interface. It is a set of routines designed to access the GATEMASTER database. The routines are written at a relatively high level (higher than VIRGA's emphasis on frame-level access).

[PDI is only partly implemented. The first stage is broad READ functionality; the second stage is WRITE functionality (e.g., PLACE_COMPONENT, UNPLACE_COMPONENT).]

PDI is strongly oriented toward the actual kinds of data in the GATEMASTER system, whereas VIRGA is a general database tool. It is an indirect way of using VIRGA. By contrast with VIRGA, which returns pointers to database frames, PDI returns pointers to particular gate array objects.

Experienced VIRGA users can open PDI in VIRGA-access mode and use VIRGA routines to get frame-level information. However, new programmers will probably just open PDI with a call to

PDI_INITIATE and use only PDI routines thereafter.

The purposes of PDI:

1) Using PDI routines, programmers can write GATEMASTER code more quickly than if VIRGA routines were used directly.

2) PDI routines will serve as the foundation for parts of the newer GATEMASTER programs: PLUG, GET, the batch auto-placer, and the batch auto-router. PDI routines provide a basis for efficient batch access to the database.

3) PDI supplies an extra level of protection for the database: it protects the data from inappropriate use of VIRGA routines.

4) Since the PDI layer increases the separation between gate array application programs and the actual database, changes to the schema will be easier than previously. In the future, GATEMASTER application code will be relatively unaffected by schema revisions.

5) The PDI layer hides the Daisy-confidential structure of both VIRGA and the database from customers (customers may eventually write routines to read the database). PDI can be the non-confidential interface to the GATEMASTER database.

PDI capabilities are:

Process start-up (PDI_INITIATE in various modes)
File operations: Open, Close, and Save
Find operations: FIND_FIRST and FIND_NEXT on frames.
Read operations: read component data (such as pins, parameters, names, and cells), read net data, or read macro data.

Write operations: modify component placements, swap pins, or write a net.

Mixed-Mode calls: send files & pointers between VIRGA and PDI.
Coordinate Conversion: Daisy units to manufacturer's units and vice versa.

Detailed documentation on PDI is available in design specifications in the GATEMASTER group.

3 DATABASE CONVENTIONS

The database design depends on certain conventions. Conventions include matters of programming practice, constant data values, and standard literals for commonly used data objects. This chapter describes specific data structures and the generic objects they represent.

Topics will be:

- Spare Fields
- Schema Parameters
- Common Data Structures:
  - headers
  - structure definitions
  - rectangles, outlines, text, and coordinates
- Schema Data Constants

3.1 Spare Fields

In most of the database frames, a few data items are named SPARE or TYPE ("spare bits"). These fields reserve some space in the database. Because some space is reserved, the process of adding a data item to the database is easier than if the whole schema had to change in order to make room. Also, since the GATEMASTER group can agree to interpret the field specifically, one can avoid having to write more code for LVR (LED VERSION update command) and LVR has fewer tasks to perform during its update run.

Use of a spare field must be approved in a design review meeting. When new software is released, all the software must know about the new fields. Also, all code must at least be re-compiled with the new schema. The new LVR command must specifically create the new fields for old databases, and put data into the fields.

3.2 Schema Parameters

Although the following fields are constant code structures, they are not considered to be "data". Because they are code structures, they are declared in the GDSCH.ELT file along with all the database frames. "GDSCH" means Gatemaster Database Schema.

Data constants such as the layer constants are declared in the file, GDSCC. "GDSCC" means Gatemaster Database Schema Constants. Those data constants are documented in the next main chapter.

The three schema parameters below do not describe any part of a gate array chip. They strictly define properties of the database. Specifically, all three are size limitations on data.

GDB_NAME_LENGTH LITERALLY '016',
The vast majority of all name fields in the schema are 16 bytes long. However, rather than declare "16" as the length in all the byte arrays for the names, we use gdb_name_length. It represents 16, the maximum length of any name entry.

GDB_SHORT_NAME_LENGTH LITERALLY '4',
In three places in the database, a name length of 16 is much too big. A shorter name length (4 bytes) is the limit for the following fields: the TEXT field in the macro_label_def, the LABL field in the cell array's gdb_label_def, and net NAME in the gdb_pwrgnd_name_def. In the first two cases, GATEMASTER saves a lot of space by requiring that the names be short. In the case of power/ground nets, the rationale is simply that the names must be made shorter than 16 and this convention for short names has already been established.

GDB_MAX_STEP LITERALLY '4';
Used in the cell_array and underpass frames. Tells the maximum number of regular steps which can be specified in each direction. This parameter only applies to a STEP lattice's use of the GDB_ARRAY_DEF structure. (GDB_ARRAY_DEF is described later in this chapter.) If a lattice is totally even in one direction, its description only requires one step. If the alternate steps are different, then two steps are required. Steps three and four are not currently supported by GATEMASTER.

The principal function of these parameters is convenience; however, they also serve as a global change mechanism: when one literal declaration is changed, the value is changed for

all references to it.

3.3 Special constants and structures

1. Nil = ØFFFFH

PLM provides 32-bit pointers with only 20-bits of information. Often, instead of those pointers, we use 16-bit indices or offsets in their place, indicating relative memory location. A nil pointer is Ø (all bits = Ø). A nil index is ØFFFFH, to distinguish it from the quite valid index = Ø. Similarly, if a word value is uninitialized, it should be set to ØFFFFH. For example, if a component is un-placed, its macro_id should be ØFFFFH.

```
GDB_NIL              LITERALLY 'Ø',
  (To initialize a pointer on a 286 system, use
   "NIL", which is built into the PL/M-286 language.)
GDB_NIL_BYTE         LITERALLY 'ØFFH',
GDB_NIL_WORD         LITERALLY 'ØFFFFH',
GDB_NO_INFO          LITERALLY 'ØFFFFH',
```

2. Layer_mask

A one-word (16-bit) layer_mask is interpreted as follows: The LSB (right-most) corresponds to the first routing layer, next least significant bit to the second routing layer, etc. (In discussion, GATEMASTER staff usually refers to "first layer" and "second layer". As GATEMASTER code literals, the layers are named "GDBC_LAYER_Ø" and "GDBC_LAYER_1" (plus other combinations). In a GAIL text, the input data for the layers refers to "LAYER_1" and "LAYER_2".

A layer_mask indicate a set of layers; for example, it can indicate what layers a via connects. The most significant (left-most) bits indicate phantom or pseudo-layers. For example, if it is desired to indicate an internal connection between pins in a macro, one can set the phantom bit on a trace's layer-mask. No other bits should be set: the trace should exist on the phantom layer only.

Currently, only one phantom layer exists. Others could be used in the future.

3. DEF, HDR

PLM has typed variables and it has structures, but it doesn't conveniently provide typed structures. Instead, we agree to use text macros (defined via the LITERALLY keyword) to define structure types. In order to distinguish these from variables, constants, etc., we agree that such type declarations will end with _DEF or _HDR. HDR should indicate a structure that begins a larger compound_structure (e.g., net_hdr); DEF is a general structure (e.g., comp_pin_def). (This distinction has some

gray areas in which the user should use his/her own judgment.)

4. Units = Grid Co-ords (WORD)

Except as otherwise noted below, all length and location units are of type WORD and refer to the internally-defined grid See the global_desc and projection frames for information on input and output to the Daisy grid system.

Almost all Daisy grid coordinates are in the Cartesion upper-right coordinate quadrant and can therefore be represented by unsigned numbers. All Daisy coordinates are equivalent to detail-view grid points.

Exceptions follow (all are important!):

(a) Relative Coordinates. Some coordinates express relative coordinate offsets rather than absolute grid locations. Among the relative coordinates, some allow negative coordinates. Examples of relative coordinates are:

The cell_array_x and _y fields are in relative cell-array units.

The cell_type outline is gdb_relative_outline_def.

The underpass has relative traces and vias.

The obstruction frame has relative obstructions.

Coordinates within a macro definition are of type INTEGER and are relative to the origin of that macro. The origin corresponds to the origin of the cell in which the macro will be placed. Even the GDB_RECTANGLE in the frame header for the macro frame is interpreted as Integer. Since INTEGER type may include negative coordinates, macros can include routes, etc., that extend below or to the left of the macro origin.

(b) Specially signed coordinates. Besides all the INTEGER fields that hold the relative coordinates noted above, two places in the database use signed coordinates (using the twos-complement method):

Macro cell coordinates (of extension cells for macros) are signed BYTE fields, measured in CELL-UNITS with respect to the origin CELL.

Manufacturer's coordinates in the GDB_PROJECTION_DEF are signed DWORDS. Special care is needed in handling these numbers.

(c) Manufacturer's coordinates. There are two places where

manufacterer's coordinates are stored:

Projection coordinates in the GDB_PROJECTION_DEF are used for output conversion of the whole coordinate system. They are all in manufacturer's coordinates.

The 'LOCAL' field in the MACRO frame's pin definition, if present, is an offset from a Daisy grid point. The offset is measured in manufacturer's units.

5. Case-folding

Text strings may be entered in the data base in any case; i.e., upper or lower or mixed. Any program which accesses such text strings is responsible for doing its own case folding if and when it is necessary; the original in the data base should be left un-folded.

3.4 Common Data Structures

3.4.1 The Generic Frame Header

Every frame has the same frame header at the top. Its structure (its list of data fields) is constant, but its initialization and usage varies among frame types.

The frame header is the only part of the frame that VIRGA looks at. VIRGA doesn't know about the rest of the frame organization.

The literal, GDB_FRAME_HEADER, is declared in the file, GDFIL.ELT. The schema makes frequent references to that literal. Therefore, for the reader's convenience, it is listed in all of this manual's frame-oriented chapters.

The 3 most important items in the frame header are SIZE, ID, and TYPE.

```
GDB_FRAME_HDR       LITERALLY
    'OPCODE  BYTE,
    HDR_SIZE WORD,
    SIZE     WORD,
    ID       WORD,
    TYPE     BYTE,
    GDB_RECTANGLE';
```

OPCODE is the same for all frames. During graphic display, the LED Screen Handler (LSH) looks at this field to see if it says 'LSH_FRAME_OPCODE'. This confirms that the data is a VIRGA data frame.

The main purpose of LSH is to respond to Screen Handler requests. It writes into a buffer, and that buffer is passed rapidly to the page buffer. ALL GATEMASTER graphic operations go through LSH, since LSH is highly optimized for speed of graphic display. In particular, text format functions use LSH opcodes. The LSH text formats are listed in this document's last Appendix.

HDR_SIZE is used by LSH to jump to the 'item_OPCODE', the first field in the item header. This size (the size of the frame header) is constant. It is 16 bytes.

SIZE tells you how big the frame is in bytes. This information helps VIRGA bounce from frame to frame. It also tells VIRGA how much data you have (how much space you need) in the frame. When LSH receives a frame, it checks this SIZE against reasonable limits.

A program which needs to copy a frame must set SIZE to the

appropriate value. The new frame must be big enough to receive another frame's contents. The size of a particular frame is not deducible from the frame-type. For example, there are big NET frames and small NET frames.

Eventually, there is a limit on frame size. The default module size limit is 7 frames. Default frame-size is 8K bytes. The size of any frame is limited by the module limit. Therefore, the maximum (default) size of a frame is 56K bytes. Through special measures, the frame might be gotten up to the 64K limit on segments. See the VIRGA manual for details of setting frame and module sizes.

ID means frame id. This tells which particular frame it is within a frame type. For example, the database frame which holds nets has the frame type, GDBF_NET (means, roughly, "Gatemaster Data Base Frame -- Net"). ID only tells you which frame of a particular type; it makes no sense by itself. For example, if a net frame has an ID of Ø, that means that it probably was the first net frame entered into the database.

TYPE in each frame header tells VIRGA and LSH what kind of frame it is. When LSH responds to screen paint requests, it reads this type field, verifies that it matches the item_OPCODE which leads the item-header, and uses the TYPE to dispatch to a sub-task. LSH will modify the page buffer image in accordance with the needs of that frame-type. If the type is not a valid screen type, LSH will not display it.

LSH knows about twelve kinds of frames. For example, it knows about LSH_NET_OPCODE. It also knows a SKIP opcode. SKIP means that the frame has no graphics. The SKIP opcode indicates a non-graphic frame. The valid codes for LSH are listed in an Appendix to this document.

The many 'frames' described in this document are actually frame-types. They serve as models of what data organization a program can rely on finding in a frame. The number of instances of one frame-type (e.g., NET) in the database may be any number less than 16K.

3.4.2 The Item Header

The term "item header" simply means the specific header which always follows the frame header for a particular frame type. Most item headers have flags which govern the interpretation or access to the frame's data, and counts and offsets which are crucial to accessing repetitive records further down in the frame.

3.4.2.1 The item opcode

The first field in the item header MUST be an item opcode so that LSH can check it. For example, the NET_OPCODE field leads the Net Header. The item opcode is the same for all frames of one particular frame TYPE. It is redundant data. Nevertheless, LSH verifies it. The opcode must be greater than SCR_MAX_OPCODE and must match the TYPE field in the frame header above.

3.4.2.2 The item HDR_SIZE

The second item in a item header is always the frame's item_HDR_SIZE. Across all frames of the same type, it is constant. Nobody uses it. Since VIRGA does not read beyond the frame header, not even Virga looks at it.

The only reason to look at the item_HDR_SIZE field would be to know where data area starts. HOWEVER, you should instead use at the count/offset pairs in the header. R&D databases will have all these sizes set correctly by PDQ.

3.4.2.3 Item Header main part

The bulk of the item header must be used to store data which may be unique to this particular frame. The unique VIRGA frame ID within this frame type means that this frame is a unique entity. Some examples of critical data are name, name_length, location, layer, parameters, type bits, version, and time-stamp.

Little data items which come in clusters, such as net pins, need to be outside the item header. That way, the header size is constant and the lists or arrays of data elements, if empty, take up no space (they aren't there).

3.4.2.4 Counts, Offsets

An item header generally has count/offset pairs which point to repetitive data arrays at the bottom of the frame. The array records may be constant or variable in length. Nevertheless, the counts and offsets tell you how far to go and how many pieces of data to collect. For variable-length records, the counts/offset pair implicitly takes care of the problem of getting from one clump of data to another. Thus, the application program is spared the chore of calculating the size of each data record in order to skip over it.

The offset is ALWAYS FROM THE VERY TOP OF THE FRAME, including the generic frame header. Use the UTL_INCREMENT_PTR routine to add the offset to the VIRGA-known frame address (frame_ptr).

For example, take NET_HEADER.OFFSET_PIN (this is the byte offset) and increment the frame_ptr with it. PLM returns the sub-element when referenced as such. And then on the new pointer, you base the structure that you need to read the data.

3.4.3 Geometric Area Definitions

One very common structure is the geometric rectangle. It is required in every frame header.

3.4.3.1 The Absolute Rectangle

(GDB_RECTANGLE)

This structure is actually defined in GDFIL.ELT because VIRGA needs it as part of the generic frame header. However, it is presented here for convenience.

The absolute rectangle, shown below, is used in obstructions, in the bitmap frame, and in every instance of the generic frame header except the one for macros.

```
GDB_RECTANGLE     LITERALLY
    '(X1,Y1)  WORD,
     (X2,Y2)  WORD',
```

The four coordinates specify two points which are the lower-left and upper-right corners of a rectangle; x1 <= x2 and y1 <= y2 ALWAYS. Rectangle sides are parallel to x- and y-axes. The same holds true for line segments which have the same two-point, four-coordinate structure.

This rectangle defines the geometric area that a particular gate array object takes up on the chip. E.g., it may take up the chip area from (Ø,Ø) to (4,3). This will usually be the 'enclosing rectangle': the minimum rectangle which encloses the object. The rectangle is used for two purposes:

1. In Database Search, GDB_RECTANGLE is used as a recognition rectangle. The GDB_RECTANGLE is in the generic frame header for this purpose. The recognition rectangle allows a program to rapidly find which objects overlap a particular grid point. For example, when the LED user presses the SELECT or PLACE action key, LED code must look for objects which intersect the cursor position.

   Most programs which look at recognition rectangles use a search method which is optimized for a class of gate array objects (e.g., components). However, a program could call the VIRGA routine, GDB_FIND_FRAMES_IN_WINDOW, to look for intersections between a window and all objects.

   LSH does NOT read the GDB_RECTANGLE in the frame header.

The rectangle is not used for graphic display.

2. In Design Rule Checks, GDB_RECTANGLE represents an area taken up by an object. For example, when a macro is placed or unplaced, BUP code reads the macro rectangle, swells it by maximum pitch, and checks the spacing between between the rectangle and objects represented in the .BMP file. If the macro rectangle is 10 by 12 grids and max-pitch is 3, the BUP code will check for conflict between the swelled rectangle, 13 by 15, and any .BMP file obstructions. Please see the BUP code if more information on DRC algorithms is required.

3.4.3.2 The Relative Rectangle

The relative rectangle definition is the same as gdb_rectangle, but with respect to something. It has INTEGER variables in order to allow negative coordinates.

```
GDB_RELATIVE_RECTANGLE_DEF    LITERALLY
'(X1,Y1)        INTEGER,
 (X2,Y2)        INTEGER',
```

The above literal is used in the obstructions frame and in the macro frame for the macro outline.

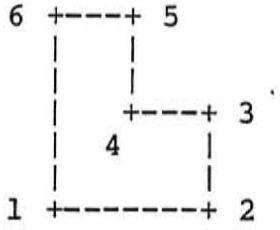
3.4.4 The Outline Definition

A graphical outline is used to define the boundaries of two kinds of gate array objects: macros and cell-types.

Polygon and/or rectangle outlines are represented as a count (and offset) and a sequence of (x,y) word or integer pairs. The outline can only have vertical and horizontal line segments; no angled lines are allowed.

If count=2, then it's a rectangle, and points are lower-left and upper-right corners of rectangle, in that order. Thus, the Outline Definition is a more general structure than the GDB_RECTANGLE since the outline can be a polygon or it can take on the rectangle form (two coordinates).

If count >=4, then it's a closed polygon with sides parallel to x- and y-axes. The points are the vertices. The polygon is drawn by making straight lines from each point to the next, and from the last point to the first (to close it). The first point is NOT repeated (the 'last' point is not counted). For example, the polygon below would require six coordinate pairs:



The count MUST be an even number.

There are two ways of declaring the outline. Each is used in some places in the database, depending on whether Integer or Word data type is more appropriate (integer type allows negative coordinates). The two different definitions are never used together.

3.4.4.1 Absolute Outline

```
GDB_OUTLINE_DEF  LITERALLY
'(X,Y)    WORD',   /* in chip grid units */
```

3.4.4.2 Relative Outline

```
GDB_RELATIVE_OUTLINE_DEF   LITERALLY
'(X,Y)    INTEGER',  /* with respect to something */
```

This is the same outline as in GDB_OUTLINE_DEF above, but defined as INTEGER-type data instead of WORD-type. ALL of the fields following which specify RELATIVE... are simply INTEGER versions of what began as WORD fields. Integer type is more convenient, but the word type declaration is kept in case some numbers must be very large.

The reason we need integers is because the relative coordinates for some objects need to be negative. A translation from relative (INTEGER) to absolute (WORD) is required at some point. For example, for macros the translation occurs when the macro is placed; all the resulting values must be >= zero. Some GAIL input values are relative (INTEGER), and other values are absolute (WORD).

3.4.5 Text Definition Structures

Text strings for graphics.

3.4.5.1 The Text Def.

(GDB_TEXT_DEF)

This text definition carries a little piece of text and tells LSH exactly how to handle it.

TEXT_X WORD,
the x coordinate of the text location point.

TEXT_Y WORD,
the y coordinate of the text location point

TEXT_FORMAT BYTE,
the "format" tells LSH how to orient the text, relative to the locating point. Text format options are explained in the GAIL manual, and the code literals to send to LSH are listed in an Appendix.

TEXT_LENGTH WORD,
the actual length of the string in the following field.

TEXT(GDB_NAME_LENGTH) BYTE,
this is the piece of text (usually a name). It can be up to 'GDB_NAME_LENGTH' bytes in length.

3.4.5.2 The Relative Text Def.

(GDB_RELATIVE_TEXT_DEF)

This structure is the same as the one above, but the coordinates are expressed relative to something, and the coordinates are INTEGER-type rather than WORD-type.

```
TEXT_X                  INTEGER,
TEXT_Y                  INTEGER,
TEXT_FORMAT             BYTE,
TEXT_LENGTH             WORD,
TEXT(GDB_NAME_LENGTH)   BYTE,
```

Codes for the text_format field are in LSHDEF.ELT (in the Appendix).

3.4.6 Coordinates and Their Structures

3.4.6.1 The Array Definition

(GDB_ARRAY_DEF)

The data in the array can be in 'STEP' form (represent a lattice of points in brief), or in 'PROJECTION' form (hold all the points in the fully expanded array). The X_TYPE and Y_TYPE fields below indicate which coordinate method representation is being used. Please see the GAIL Manual for a full explanation of the various coordinate methods.

TOTAL_COUNT WORD,
the total number of points in the lattice.

X_COUNT WORD,
number of columns of points in the lattice.

Y_COUNT WORD,
number of rows of points in the lattice.

X_ORIGIN WORD,
Y_ORIGIN WORD,
These origins specify the absolute, Daisy detail-view grid coordinates of the origin point from which the whole lattice proceeds (only relevant if type in that dimesion = step). This "first-point" location is critical to the linear transformation method. In either X or Y, if the coordinate method is LIST, the field is set to GDB_NIL_WORD.

X_LIST_COUNT WORD,
count of X coordinates in the X-list in the data area. If the array method in X is STEP, this is GDB_NIL_WORD.

OFFSET_X_LIST WORD,
offset to the X-list. If the array method in X is STEP, this is GDB_NIL_WORD.

Y_LIST_COUNT WORD,
count of the Y coordinates in the Y-list in the data area. If the array method in Y is STEP, this is GDB_NIL_WORD.

OFFSET_Y_LIST WORD,
offset to the Y-list. If the array method in Y is STEP, this is GDB_NIL_WORD.

X_TYPE BYTE,
Y_TYPE BYTE,
These two fields specify the method of lattice creation. They can contain literals defined in GDSCC.ELT (explained in the Schema Constants section. The valid array types are PROJECTION, RANDOM (never used), and STEP (actually, GDBC_ARRAY_STEP_ONE or GDBC_ARRAY_STEP_TWO). If the GAIL user specifies 'LIST', this field gets PROJECTION. If the GAIL user specifies 'ARRAY' and one step value, this field gets STEP1. If the GAIL user specifies 'ARRAY' and uses two step values, this field gets STEP2.

X_STEP(GDB_MAX_STEP) WORD,
Y_STEP(GDB_MAX_STEP) WORD,
These are arrays which specify the various deltas between successive points in a step-array type of lattice. Specifically, for each dimension, the field specifies: the first step, the second defined step, the third step, the fourth step. For example, a step could be "5". Currently, only the first two steps are ever used.

The Step method is really useful when you have a lattice of points. The GDB_ARRAY_DEF structure allows us to represent X*Y lattice points using only X+Y points in the lists.

If desired as part of a frame design, the structure must be a part of the item header. Also necessary are the lists which fill out the frame.

If the array type is STEP, the array is a lattice and it is fully described in the GDB_ARRAY_DEF. If the array type is PROJECTION, there are coordinate lists which appear further down in the frame. For PROJECTION type arrays, GDB_ARRAY_DEF specifies a generic pattern, but the coordinate lists describe where the pattern is put down. Please see the GAIL Manual for graphic examples of STEP and PROJECTION.

Lattices are used in the following frames: cell array, underpass, obstruction, bitmap. Thus, a frame of one of these types must have GDB_ARRAY_DEF declared in its header.

For the underpass and obstruction arrays, the generic pattern is replicated at all the lattice points. In the cell array, no pattern is replicated. Every lattice point has an associated unique object (a cell). Every cell can have individually assigned characteristics. For bitmaps, nothing is replicated at the lattice points. The lattice points determine vertical and horizontal grids which divide up the chip surface into bitmap rectangular areas.

Unfortunately, the database does not use a lattice to represent manufacturer's coordinates. In GAIL input to BASE, the user may use a lattice to describe the manufacturer's coordinates, but then those coordinates are stored in fully

expanded form using the GDB_PROJECTION_DEF structure. That structure is explained in the PROJECTION frame chapter.

3.4.6.2 The Coordinate Lists

(GDB_COORD_DEF)

The X and Y coordinate lists are arrays of the following:

```
COORD          WORD;
```

This structure specifies the WORD form of any X or Y coordinate.

It would be easier to represent the coordinate lists as arrays of data, rather than as arrays of structures. However, for database consistency's sake, the coordinates are represented in data structures which are of this type (just one word variable, COORD).

The same GDB_COORD_DEF structure is used for the X or Y coordinate of any lattice point. Whether the coordinate value is an X position or a Y position, its value sits in the same kind of field: COORD.

The GDB_COORD_DEF definition is declared just once, but the coordinate lists in many frames will use it. The important offset information for programmers is simply the count and offset figures for the X and Y coordinate lists. The offsets tell the programmer how far to go to get to the two clumps (arrays) of coordinate data.

3.5 Schema Data Constants

The fields in this chapter are based on GDSCC.ELT V1.2.5, which mates with GDSCH.ELT V5.0.02, 2/14/84. The file GDSCC.ELT holds declarations (PLM literals) for data constants which are often used in GATEMASTER software. This chapter is an annotated GDSCC.ELT.

The data constants are kept separate from both the program code and the database structures. They are data constants, and as such, they may change at any time. In particular, they may change on shorter notice than the code or the schema. Keeping them separate from the schema and the code helps keep them visible. Then we can see the interactions of various data constraints. Furthermore, all these constants are reserved. Changing them or declaring other data constants requires approval in a design review.

The declarations consist of declaring symbols in PLM which are equal to literal values. The same literal may be used to assign a constant to data types of various sizes. The compiler will fill out the bits as necessary. However, all the values declared are actual numerical constants. They are not patterns which will repeat in order to fill a larger data type. For example, GDBC_VIA_OBST is equal to '0000$0001B' (a binary 1). If a program assigns this word-length constant to a dword-length variable, the compiler will set the field to '1' and set all the preceding bits to zero.

The following symbol conventions are essential:

1) A 'B' in a declaration allows you to use a Binary value as the constant to be declared.

2) A '$' (dollar sign) in a declaration is ignored by the PLM compiler. It only serves to help human beings read each nibble (4 bits) separately.

3) An H in a declaration means hexadecimal input. For example, an '8H' gets written into the fields as '1000' (binary). Thus, an '8H' sets the high bit. An F is all 1's.

The constants are described below in groups. They are organized partly by what data fields use them and partly by what kind of information they declare.

The process goes as follows: a 'PLM literal' is a piece of code which the file GDSCC.ELT has literalized to a constant numerical value. No 'literals' are found in the database; only real data is found in the database. The software uses literals

such as GDBC_VIA_OBST to put numerical values into database fields and to compare against the values in database fields.

3.5.1 Marking Bit-encoded fields as In-Use

The next two masks (in byte and word versions) are used during database set-up to initialize all bit-encoded TYPE fields. In particular, we want to set the high bit to indicate that the other bits are really information and not garbage. The bit indicates that the field was initialized with a value.

Any program which allocates space in the database should set all data fields to zero, in particular the high bit. Then, when a program has some data to write into a bit-encoded field, it must set the high bit to indicate the presence of valid data.

HOWEVER, do not set the 'valid-info' bits for a layer-mask; that destroys the validity of the previous data.

```
/* byte length */
GDBC_MASK_VALID_TYPE_INFO    LITERALLY   '80H',

/* word length */
GDBC_MASK_VALID_WORD_INFO    LITERALLY   '8000H';
```

Other fields (not bit-encoded) should be initialized with all FF's (hex), which is all 1's.

3.5.2 Layer Specification

BASE puts these values into many fields in the database to indicate what layer(s) a gate array element is on, and what layers an option or rule applies to. For example, a LAYER_MASK field carries one of these values.

The layers are declared as shown below:

```
/* Each layer gets its own bit. */
GDBC_LAYER_0           LITERALLY  '0000$0000$0000$0001B',
GDBC_LAYER_1           LITERALLY  '0000$0000$0000$0010B',
GDBC_LAYER_0_1         LITERALLY  '0000$0000$0000$0011B',
GDBC_LAYER_PHANTOM     LITERALLY  '1000$0000$0000$0000B',
GDBC_LAYER_PHANTOM_0   LITERALLY  '1000$0000$0000$0001B';
```

In almost all GATEMASTER discussion and documentation, GDBC_LAYER_0 is referred to as "layer one". Likewise, GDBC_LAYER_1 is referred to as "layer two".

Some layer combinations have explicit literals to represent

them. The literal value in that case is the 'OR' function of the two individual layers. An example is GDBC_LAYER_0_1, above.

3.5.3 Obstruction Types

These constants tell what kinds of objects are prohibited.

'Route' means trace. GDB_ROUTE_OBST means that the grid point(s) should be obstructed to traces.

When it comes to obstructins, pins are treated as pieces of trace. The GDB_ROUTE_OBST flag will obstruct pins (except for special GAIL rules for pins landing on things).

```
/* For obstruction.type field in the obstruction frame,
   macro_obstruction, etc. */
GDBC_VIA_OBST                LITERALLY    '0000$0001B',
GDBC_ROUTE_OBST              LITERALLY    '0000$0010B',
```

The combined mask should not be used. The programmer should use an OR function in a program to test for the combined condition, or test for the first OR the second one. If the first two constants ever change, testing for the third one will be a crazy test.

```
/* Don't use the following.
   Use (gdbc_via_obst OR gdbc_route_obst). */
GDBC_VIA_ROUTE_OBST          LITERALLY    '0000$0011B';
```

3.5.4 Macro and Underpass Items

Used in the TYPE fields in the NET frame for the following structures: pins, detail trace, rough trace, and vias. These constants are internal to the software. They are not found in GAIL.

Each of the constants serves two purposes:

1) to prevent the LED user from altering macro trace/via in a placed component, or altering an underpass.
2) for confirmation check, upon un-placing a component: the bits tell LED that it must remove the indicated net elements from a NET frame.

Macro_item is used in the net frame to mark a routing element that came from a macro. Pins, traces, and vias in the NET frame that came from a macro are marked as such. This constant is not used on objects in the Macro frame itself, since their source, the macro itself, is already obvious.

```
GDBC_MASK_MACRO_ITEM          LITERALLY    '0000$0001B',
```

Underpass_item is used in the net frame to mark an object that came from an underpass. For instance, a net trace that came from an underpass is marked as such.

```
GDBC_MASK_UNDERPASS_ITEM      LITERALLY    '0000$0010B',
```

Macro-item and Underpass-item will never be assigned to the same object.

Fixed_item is just the OR of macro_item and underpass_item above. It is only needed when the LED user is deleting trace or vias. It is a useful fiction: it says that either macro or underpass stuff is involved so don't mess with it!

```
GDBC_MASK_FIXED_ITEM          LITERALLY    '0000$0011B',
```

In LED, the actual test is: IF (item).type AND GDBC_MASK_FIXED_ITEM <> 0, THEN (it's a fixed item -- don't touch).

3.5.5 Touch Underpass

A flag is built into GAIL to indicate that a net must be completed in order to connect a macro to an underpass. The GM_TOUCH_UNDERPASS option in GAIL will cause BASE to set the following constant on a macro trace (in the MACRO frame):

```
GDBC_MASK_ADD_UNDERPASS_ITEM LITERALLY  '0100$0000B',
```

If LED finds the bit is set, LED will search for an underpass to touch.

If LED finds that the component placement has a macro trace that does touch an underpass, LED follows a particular process: (1) copy the Macro trace into the Net frame; (2) set the 'macro-item' bit, to show that the trace came from a macro (is a macro subnet); (3) check to see if the trace touches underpass; (4) if so, check for the ADD_UNDERPASS bit; (5) then deal with the underpass. First, add it to the net. (6) Then set the bit on the trace for 'underpass-item'.

3.5.6 Pad Item

Used in net.pin_type and net.via_type in the pad.

Pad-item is a non-publicized GAIL option which allows special treatment of pads, but is complicated to implement. It alters design rules in the case of vias landing on pads.

```
GDBC_MASK_PAD_ITEM              LITERALLY    '0000$0100B',
```

3.5.7 Power/Ground Net Number and Offset

A power or ground net is always big. Currently, we break them up into one small net per component. Since the nets must always be correct- by- construction, the break-up into thousands of small nets requires a missing-pin scheme which necessitates routing action by the user.

An alternative to the thousands of missing pins is a pseudo-net scheme. That way of representing power/ground nets will be used in the future. It is discussed only in the GLOBAL_DESC frame.

Generally, pins in the net frame imply the logic of what needs to be connected. However, current power and ground nets pose a special problem where, in effect, a pin is missing in the net, so a flag is needed to indicate that the net information is incomplete. The code sets the net_type field in the NET frame if the a net is a power/ground net. If it is one, special treatment will be required in LED.

The constant below is word-length; it is used to set the net.net_type field. The TYPE_BYTE constant no longer exists.

```
GDBC_MASK_PWRGND_TYPE_WORD LITERALLY '0000$0000$0000$0111B',
```

If any bits are set, the net is a power/ground net. The bits tell LED that during component placement, macro traces need to be copied into the net frame to complete the power-or-ground net.

The three bits in the constant indicate one number (value can be 1 to 7). The value indicates the number of the power or ground net. A value of 0 means that the net is a regular signal net.

In order to get the macro trace, we have to be able to get to the macro. One must add the number represented by the 3 bits above to a constant value in order to generate the needed macro ID. The constant 'offset' (added to the ID value) is shown below:

```
GDBC_PWRGND_NET_OFFSET     LITERALLY '65000',
```

For example, if the net-type equals 7, one adds this number to 65000. The sum of 65007 means that the macro with id 65007 has trace which represents power or ground.

Remember: values 1 to 7 are in the net_type; the big numbers are macro id's. The macro id's are kept high in this way in order to avoid all the id's for normal macros.

The BASE program sets up the correspondence between the bits in the net_type field and the macro id's. SLIDE creates the power/ground nets. LED code must add or subtract the offset as necessary to find the structures it desires. For instance, during component placement, LED must subtract the offset from the macro id in order to get to the appropriate power/ground net in the NET frame. It can then copy the macro net's bits into the power/ground net. To get to the macro that corresponds to a particular power/ground net, LED must add the offset. Finally, if the macro subnet is incomplete (if the power/ground net has gaps in it), auto-routing must be used to complete it.

3.5.8 Temporary trace-reverse

Below is a bit which LED sets in its own code while it is modifying traces. The bit is reserved for use by LED only; it should not be used in the database.

```
GDBC_MASK_TRACE_REVERSE    LITERALLY   '0000$1000B',
```

3.5.9 Component Status

In the component header in the component frame, there is a STATUS field. Currently, one bit is used in the status field which represents the component's status vis-a-vis the PLACE program (fixed or unfixed).

SLIDE sets the field to a default of zero (means 'unfixed' component). However, an LED user may set a component's status to 'fixed', which then prohibits the PLACE program from moving it. The mask for this state is:

```
GDBC_MASK_COMP_FIXED    LITERALLY  '0000$0000$0000$0001B',
```

3.5.10 Pin type

Input and Output pin types are declared below. Bidirectional is represented as the OR of those two.

These constants are not only used in the database and in the code. They are "built-in" to GAIL. The GAIL user can refer to them.

The type bits are not set on component pins, nor on original net pins. BASE sets the type on unique macro pins, based on GAIL input. LED copies macro pins into net_pin records without their (input/output) TYPE, but sets other bits in the TYPE field to indicate their source (e.g., 'macro-item').

The following two constants are used in component.comp_pin.type and macro.macro_pin.type:

```
GDBC_MASK_INPUT        LITERALLY   '0001$0000B',
GDBC_MASK_OUTPUT       LITERALLY   '0010$0000B',
```

3.5.11 Route_Necessary Flag

The following constant is not used. It will be used in GAIL in the future. It is intended for macro_route.flags. If this bit is not set, then route is necessary only if one of pins on route is used. See the GAIL manual for further details.

```
GDBC_MASK_NECESSARY   LITERALLY  '0000$0010B';
```

3.5.12 Flags for RAIN (Rough Router)

The six flags here are for RAIN. The first two are still used; the other four are obsolete and should be dropped in the next schema change.

The two constants below are only used in the field X_OR_Y in Gdb_rough_projection_hdr. They are needed to distinguish the two Rough_Projection frames which hold the X and the Y coordinate lists, when VIRGA was not able to do that (early in GATEMASTER history).

```
GDBC_ROUGH_PROJ_X       LITERALLY  '0',
GDBC_ROUGH_PROJ_Y       LITERALLY  '1',
```

Formerly, the Rough Router used the following bit in rough_misc.status; it was set when the rough bitmap was initialized.

```
GDBC_MASK_ROUGH_INIT   LITERALLY '1',
```

The next three constants are obsolete.

```
GDBC_HORIZONTAL         LITERALLY  '0',
GDBC_VERTICAL           LITERALLY  '1',
GDBC_DIAGONAL           LITERALLY  '2';
```

3.5.13 X and Y Projection Frames

The two constants below are used to tell which of two PROJECTION frames has the X coordinate list, and which one has the Y coordinate list. The PROJECTION frame is entirely independent of the ROUGH_PROJECTION frame.

The following constants are used in the PROJ_TYPE field in the PROJECTION frame:

```
GDBC_PROJ_X      LITERALLY  '0000$0001B',
GDBC_PROJ_Y      LITERALLY  '0000$0010B';
```

3.5.14 Lattice and Array flags

In GAIL text, you have to use LIST or ARRAY keywords to declare the method of coordinate input. If using ARRAY, you can declare one or two steps.

The constants below declare these coordinate methods (refer to GDB_ARRAY_DEF).

The first two constants are used in cell_array and underpass_array x_ and y_type fields. Random array is not implemented.

```
GDBC_ARRAY_RANDOM       LITERALLY '0FFH',
GDBC_ARRAY_PROJECT      LITERALLY '0',
```

Another two constants declare, literally, the number of steps used.

```
GDBC_ARRAY_STEP_ONE     LITERALLY '1',
GDBC_ARRAY_STEP_TWO     LITERALLY '2',
/* n > 0 means n steps; only two steps currently used */
```

The last constant is used in cell_array.flags; if set, x_labl comes first in the label concatenation (relevant to cell arrays only.)

```
GDBC_MASK_X_LABL_FIRST LITERALLY '1';
```

3.5.15 Per-Layer Constants

All the constants below come from GAIL options. They are primarily for layer_info.flags.

The constants declare conditions which hold for a whole routing layer.

```
GDBC_MASK_HORIZONTAL        LITERALLY '0000$0000$0000$0001B',
```
Normally, preferred routing direction is vertical. However, the literal above is used to declare horizontal as the preferred direction.

```
GDBC_MASK_UNDERPASS         LITERALLY '0000$0000$0000$0010B',
```
Same as GAIL GM_UNDERPASS. It declares an underpass layer. A layer with underpasses can also contain programmable vias.

```
GDBC_MASK_NO_DOGLEGS        LITERALLY '0000$0000$0000$0100B',
```
No doglegs allowed on the layer.

```
GDBC_MASK_CELLS_ARE_PROTECTED LITERALLY '0000$0000$0001$0000B',
```
Same as GM_PINS_ON_OBSTRUCTIONS (lets pins dig into obstr.).

```
GDBC_MASK_NO_TRACE_ROUTE_TO_PIN LITERALLY '0000$0000$0010$0000B',
```
No traces on pins. (GM_NO_TRACES_ON_PINS).

```
GDBC_MASK_NO_VIA_ROUTE_TO_PIN LITERALLY '0000$0000$0100$0000B',
```
No vias on pins. (GM_NO_VIAS_ON_PINS).

3.5.16 Global Design Rule

Used for global_desc.via_rules.

```
GDBC_MASK_NO_FLOATING_VIA LITERALLY '0000$0000$0001$0000B';
```
Says that no floating vias are allowed; only fixed, base array vias are allowed. Set by the GAIL option, NO_VIAS_ON_PINS. Interpreted as prohibiting floating vias on any and all layers.

3.5.17 Motorola flags

Very specific feature bits are used for Motorola cells and components. They are NOT built-in, public constants in the BASE compiler; they are GATEMASTER confidential.

```
/* for features word in components, cell arrays, macros */
GDBC_MASK_FEATURE_RESISTOR LITERALLY '0000$0000$0000$0001B',
GDBC_MASK_FEATURE_MED_PWR  LITERALLY '0000$0000$0000$0010B',
GDBC_MASK_FEATURE_HIGH_PWR LITERALLY '0000$0000$0000$0100B',


/* Motorola pad pins */
GDBC_MASK_PIN_N       LITERALLY '0000$0000$0000$0100B',
GDBC_MASK_PIN_RC      LITERALLY '0000$0000$0000$0010B',
GDBC_MASK_PIN_A       LITERALLY '0000$0000$0000$1000B',
GDBC_MASK_PIN_B       LITERALLY '0000$0000$0001$0000B',
GDBC_MASK_PIN_AB      LITERALLY '0000$0000$0010$0000B',
GDBC_MASK_TDIODE      LITERALLY '0000$0000$0100$0000B';
```

A feature put on a component means that the feature is required. A feature put on a cell in the cell array frame means that the feature is available.

The "RC" feature means "resistor-capacitor network". The "TDIODE" feature means "temperature-sensing diode". The TDIODE (if required) can land in one specific pad cell only.

DML code and MKFIX code contain these literals hard-coded, and they give special treatment to components with those features. By contrast, BASE, SLIDE, and MAKE do not have them and do not know them. So far as BASE is concerned, it is the job of the GAIL user to make sure that the DML feature bits on components match the GAIL feature bits on cell arrays.

3.5.18 Delta-List

These constants, in the delta frame, describe exactly which atomic difference was discovered by SLIDE. The user must always eliminate the difference in LED. None of this is from GAIL.

```
/* used in delta.delta_type */
GDBC_DELTA_DELETED_COMPONENT LITERALLY '1',
GDBC_DELTA_ADDED_NET_PIN     LITERALLY '2',
GDBC_DELTA_DELETED_NET_PIN   LITERALLY '3',
GDBC_DELTA_DELETED_NET       LITERALLY '4',
GDBC_DELTA_UNPLACE_COMP      LITERALLY '5',
```

The following two flags tell who noticed the particular discrepancies and logged them into the delta frame. Only the first constant is currently used; SLIDE puts it in.

```
/* used in delta.flags */
GDBC_DELTA_CREATED_BY_SLIDE  LITERALLY '0',
GDBC_DELTA_CREATED_BY_LED    LITERALLY '1'; /* not used */
```

3.5.19 Pin-Swap

These constants are very similar to those for the Delta frame. These tell what kind of pin change occurred. Only two kinds of changes/discrepancies may occur in pin swap.

The Load-Set code in LED sets and reads these flags. In the future, SLIDE will use them to compare against the entries in the Delta frame, to see if there is redundancy which can be cancelled out.

```
/* used in pin_swap.type  */
GDBC_SWAP_LED_DELETED_NET_PIN  LITERALLY '1',
GDBC_SWAP_LED_ADDED_NET_PIN    LITERALLY '2';
```

3.5.20 Page flag on pathname

Not used by SLIDE. Intended use (long ago): to flag a pathname which names a DED schematic page.

```
GDBC_MASK_PAGE          LITERALLY '0000$0000$0000$0001B';
```

3.5.21 Text flags

The following constants are specific to Motorola gate arrays. Currently, MAKE does not use them. It looks at the name in the text frame to classify the text contents. However, these constants are still used in MKFIX.

```
/* used for TEXT_TYPE */
GDBC_TEXT_FIX           LITERALLY '0000$0001B',
GDBC_TEXT_NET           LITERALLY '0000$0010B';
```

3.5.22 Soft/Hard Parameters

These constants would be used on pseudo-net parameters (not currently used).

```
/* for parameters */
GDBC_PARAM_PIN_POWER_TYPE LITERALLY '0000$0000$0000$0001B',
GDBC_HARD_PARAM_WORD_TYPE LITERALLY '0000$0000$0000$0001B';
```

3.5.23 Macro Name Length

The following name-length applies only to macros, not in general. Also, unlike most of the other constants here, this name-length is just a yardstick for comparison. It does not enter the database.

```
GDBC_MACRO_NAME_LENGTH   LITERALLY '15';
```

In the schema, macro names are gdb_name_length (which currently equals 16). However, we only allow macro names of up to 15 characters to preserve future compatibility with the Library Utilities (LBU). The above literal is used for the <=15 check in SLIDE.

3.5.24 Data-creation flag from SLIDE

The flags below are set by SLIDE just for future comparison, since a zero would falsely indicate non-initialization. Currently, they are not read by any other code.

```
GDBC_SLIDE_CREATED_PATHNAME  LITERALLY '0000$0000$0000$0001B',
GDBC_SLIDE_CREATED_PARAMETER LITERALLY '0000$0001B',
GDBC_SLIDE_CREATED_TEXT      LITERALLY '1';
```

3.5.25 Off_Grid Pins

Off-grid pin information comes from GAIL. The customer can choose to have off-grid pins, but GATEMASTER only allows off-grid pins in one dimension: either in X or in Y.

The following constants are used in the global_desc frame to tell whether off-grid pins are used, and whether in X or in Y.

```
GDBC_OFF_GRID_TYPE_NULL  LITERALLY '0',
/* if off_grid_pin is not in use */

GDBC_OFF_GRID_TYPE_X  LITERALLY '1',
/* if macro_pin.local is x */

GDBC_OFF_GRID_TYPE_Y  LITERALLY '2';
/* if macro_pin.local is y */
```

The macro_pin.local field holds the actual offset.

3.5.26 Orientation (flip/rotation) flags

[Not yet implemented. No flipping or rotation is performed by any code, and no orientation fields are checked.]

These constants are intended for use in all orientation fields in the macro and cell_array frames. There are eight possible orientations. Each one is a combination of flip/not-flip and a rotation in that order.

```
GDBC_NO_LEGAL_TRANSFORM     LITERALLY '0000$0000B',
GDBC_ROTATE_0               LITERALLY '0000$0001B',
GDBC_ROTATE_90              LITERALLY '0000$0010B',
GDBC_ROTATE_180             LITERALLY '0000$0100B',
GDBC_ROTATE_270             LITERALLY '0000$1000B',
GDBC_FLIP_ROTATE_0          LITERALLY '0001$0000B',
GDBC_FLIP_ROTATE_90         LITERALLY '0010$0000B',
GDBC_FLIP_ROTATE_180        LITERALLY '0100$0000B',
GDBC_FLIP_ROTATE_270        LITERALLY '1000$0000B';
```

The 'ROTATE' movement is counter-clockwise. The flip axis and other details may be found in the design specification (authors: Suzanne Jacobs, Yosi Kliger).

3.5.27 Version update history

The following constants are used by the LED version command (LVR) to keep a history of schema update. LVR code sets a bit for each conversion. This allows LVR to know just how old a database is. Old databases may have bad data (produced by buggy programs). Or, early, hand-calculated bitmaps may have introduced incorrect data. If the database is old, LVR may need to re-calculate and correct some of the data.

```
GDBC_CONVERT_304_TO_405     LITERALLY '0000$0001B',
GDBC_CONVERT_405_TO_500     LITERALLY '0000$0010B';
```

3.5.28 File-type

Used in global_desc_2nd_hdr.file_type to tell a program what sort of file it is looking at. All programs should issue a warning if they encounter a file that is not the right type.

```
GDBC_UNDEFINED_FILE         LITERALLY '1',
GDBC_LED_FILE               LITERALLY '2',
GDBC_BASE_FILE              LITERALLY '3',
GDBC_MLIB_FILE              LITERALLY '4';
```

4 The NET Frame

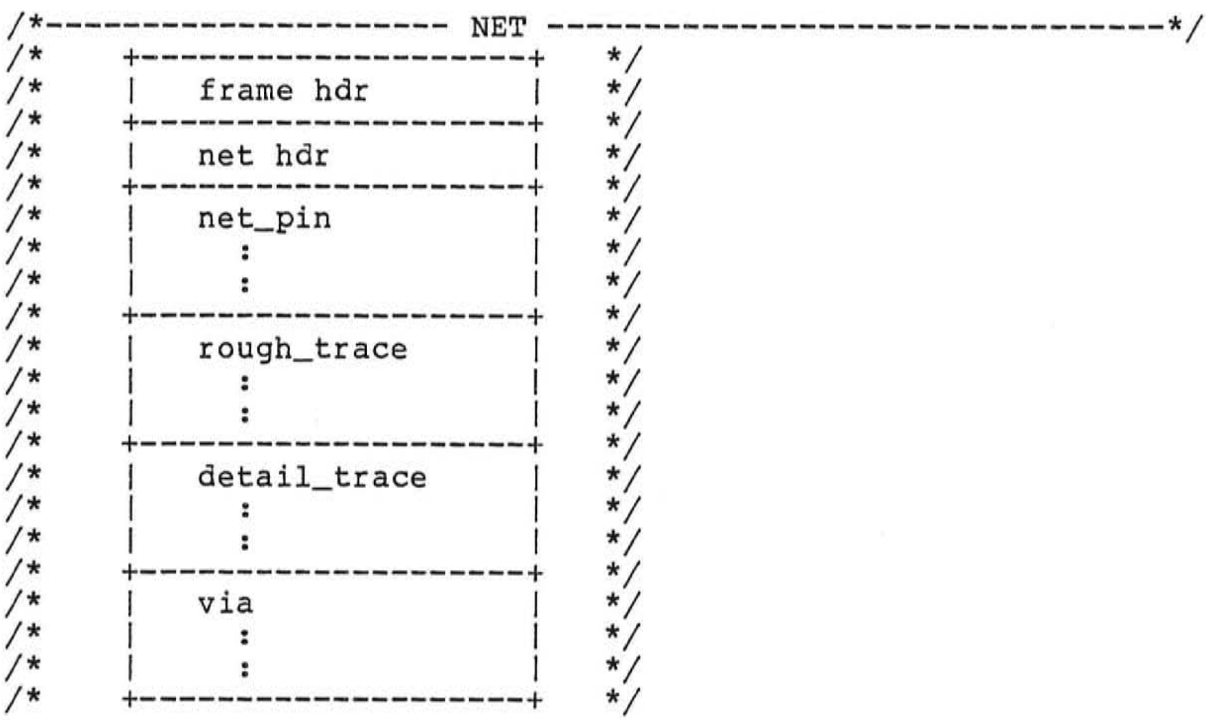
A net, fundamentally, is a set of pins, and a set of various kinds of interconnect. Interconnect is composed of rough traces, detail traces, and vias. All of these elements are associated with particular nets.

Specifically, the term "net" implies that the gate array elements which comprise all the data for a particular net must eventually be connected together. At any one time, none or some or all of those elements may have been placed or routed in the gate array layout, but eventually, they must all either touch or be made electrically common in some other fashion.

SLIDE creates the net frames when it carries the SLIDE.SING netlist into its creation of the .LED file. Each net frame represents a net (connectivity that needs to be implemented by the user in layout). Initially, the net frames contain just the logical connectivity (i.e., which pins or which components need to become electrically common). When the LED user places components and puts down traces, the LED program will modify the pins, traces, and vias in the relevant net frame to reflect the new state of the net.

Routing in a net can come from a source besides user layout: from component (macro) placement. In this case, LED copies the macro routing into a net frame. When a component is placed, macro information is copied into the net frame (e.g., multiple physical pins on the macro that correspond to a component's logical pin, macro traces and vias, any underpasses those pins on the macro macro, etc.). Later, when routing is done, manually, or automatically, additional traces and vias are added to the net frame.

4.1 Overview



4.2 The Generic Frame Header

Defined in gdfil.elt. Is here just for convenience.

```
GDB_FRAME_HDR       LITERALLY
    'OPCODE  BYTE,    set to LSH_FRAME_OPCODE.
    HDR_SIZE WORD,    always the same: 16 bytes.
    SIZE     WORD,    byte size of this particular frame.
    ID       WORD,    frame id within the frame-type.
    TYPE     BYTE,    frame-type is GDBF_NET.
    GDB_RECTANGLE';   enclosing rectangle for all net elements.
                      Anybody who changes the net must set this
                      using the routine, BNT_RECTANGLE.  Pass it
                      the frame; it will calculate the enclosing
                      rectangle and fill this structure.  For
                      example, the Rough Router, Detail Router,
                      the TRACE command in LED, and the batch
                      routers in BAR alter nets, and must call
                      the routine.
```

4.3 The Net Header

(GDB_NET_HDR)

NET_OPCODE BYTE,
Set to LSH_NET_OPCODE.

NET_HDR_SIZE WORD,
the size of this net header.

NET_TYPE WORD,
This field is bit-encoded. One bit is used to indicate a completed net; some others are used to indicate a power/ground net. In brief, there are three type constants which can be used here:

GDBC_MASK_NET_IS_COMPLETED.
GDBC_MASK_PWRGND_PRESET,
GDBC_MASK_PWRGND_TYPE_WORD,
(GDBC_PWRGND_NET_OFFSET is a power/ground constant; however, it is never found in the net_type field. It is subtracted from the macro id in order to get here.)

These literals are explained in the section called Schema Data Constants in the Database Conventions chapter.

Power/ground data: essentially, some pins need to be tied to power or to ground. However, the only way to indicate that the power/ground net needs to be completed is to specially mark the net elements. In this net_type field, one must set the bits that indicate that the connection to power or ground must eventually be accomplished. Please see the Schema Data Constants section for details.

CRITICALITY BYTE,
tells how important is it to route this net. If this net is critical, then you want to route it early, and thereby make sure that it gets routed, and make it as small as possible. The PLACE program uses this field. MAKE and MKFIX have the ability to print out the information.

Set by SLIDE, using the parameter called WEIGHT. The DED user enters the original parameter value on the schematic: high value indicates an important net. DML has been programmed to carry this parameter into the SLIDE.SING file. SLIDE will read that value and set it for the net for the rest of its life.

When SLIDE reads the SLIDE.SING file, it divides the DML value by 10 and puts it into this GM-10 field. That is, if SLIDE receives a 100 from DML, it puts a 10 in this field. DML checks that it supplies a value < 2550. SLIDE checks that the result of its divide-by-10 is less than 255. If SLIDE gets no value, it puts here a default value of 10. So, a 1 in this field is very low criticality; a 250 in

this field is very high criticality.

STATUS WORD,
bit-encoded. Not used. Could be used for completion status, but not currently needed.

DED_PAGE_ID WORD,
This is a number; it is the frame id for the pathname frame which contains the name of the DED page with this net. This frame id MIGHT be the same as the current DED id for that page. This frame ID helps you distinguish between nets, since net names by themselves are not unique (the same name can occur on various DED pages). However, on a particular page, the net name must be unique.

SLIDE assigns the frame id, starting with '1'. The page id which is indexed by this frame id will, at the start, be equal to this frame id, since SLIDE processes the nets and assigns these frame id's in page-id order. See the pathname frame for a full discussion of the complexities in assigning a page id.

NAME_SIZE WORD,
Holds the actual string length of the name below.

All 'name' fields in the database have actual character data. Usually, the data is not as long as the field, so we use another field to record the actual name length as well.

You might think that the SIZE field is unnecessary, since one could pad the NAME data with binary zeros (ASCII nulls) on the right, and scan it later, easily. However, GATEMASTER will soon allow any and all characters (except quote marks) to appear in the name. This includes ASCII nulls! The padding method would become a mess if you had to scan the name field for the end of the name.

NAME (GDB_NAME_LENGTH) BYTE,
This is the net (it is known by its name). Gdb_name_length is a constant, 16 bytes long. It's an array of 16 bytes.

ALIAS_LENGTH WORD,
This is the length of the following alias name.

ALIAS(GDB_NAME_LENGTH) BYTE,
This is a text string: another name for the net. Rationale: some systems have restrictions regarding name length, or legal characters. For instance, there are some special problems associated with preparing Motorola data for LOGCAP. The DML models have put out a unique alias name for each net, and the LOGCAP program will accept the alias names.

NET_PARAM WORD,

Waiting to be used. It might become necessary to associate another parameter with the net.

PARAM_ID WORD,
This is the ID for the parameter frame associated with this net. [However, that frame is not currently implemented.]

PIN_COUNT WORD,
number of pins. (Equivalent pins, too.)

OFFSET_PIN WORD,
offset to the pin data.

ROUGH_COUNT WORD,
number of rough traces.

OFFSET_ROUGH WORD,
offset to the rough trace data.

DETAIL_COUNT WORD,
number of detail traces.

OFFSET_DETAIL WORD,
offset to the detail trace data.

VIA_COUNT WORD,
number of vias.

OFFSET_VIA WORD,
offset to the via data.

All of the foregoing data fields are in the GDB_NET_HDR.

The following section contains all the net-specific data: pins, traces, and vias, but arranged in particular structures. Each kind of data is represented according to a particular structure. For example, a pin is represented according to the PLM structure, GDB_NET_PIN_DEF (below).

4.4 The Net Pin Definition

(GDB_NET_PIN_DEF)

TYPE BYTE,
This is the type of the net-pin: I, O, or BI (Input, Output, or Bidirectional). SLIDE always sets the bit for 'valid_info'. PLACE uses this field in creating its internal net-list. LED reads this field only as a consistency check during un-placement.

If this is an original net pin, the type is not set until the pin is placed. If the pin is later unplaced, its type is not erased.

Whether the net is user routing or macro-dependent routing, this net-pin type is copied in from the macro_pin type.

If the routing is from a macro subnet, this field also gets a high bit set which indicates that this is a macro item. If the component placement includes physically equivalent pins (creates multiple new records), their TYPE gets the 'macro-item' bit, too. Please refer to the TYPE field discussion at bottom of this NET chapter.

COMPONENT_ID WORD,
This is the frame id for the component frame that the pin is from.

PIN_ID WORD,
This is the actual pin id within the component frame (specified by the component_id). Important: this pin id is the same id as the (subnet) ID that is in the macro frame. Using this field and the field above, you can access, elsewhere in the database, the particular component pin which is relevant to this net.

LAYER_MASK WORD,
Used a lot! This is one word, bit_encoded. Tells you on which layers the pin exists. The layer_mask = Ø when not placed.

Layer two is "above" layer one, in the user's view of the layout.

The least significant bit in the layer_mask shows layer one, on or off. The next most significant bit shows layer two. The most significant bit indicates whether the phantom layer is on or off.

Picture:

(most-significant) <---> (least-significant)

Bit: 16 15 14 13 12 11 10 09 08 07 06 05 04 03 02 01

1 1 1
phantom layer two layer one
layer

None of the other bits should ever be set. They will be needed for additional real or phantom layers in future 3-layer chips.

All pins are always on the phantom layer. That bit is automatically set by BASE, and LED will copy the pin layer-mask from the macro pin. The phantom layer represents electrical connectedness between pins, among other things.

Fundamentally, the phantom layer makes it easier to indicate connections between net elements. Suppose you have two macro pins which need to be connected. Each pin also exists on the phantom layer automatically. A trace on the phantom layer (a 'phantom trace') may be used to connect the two pins.

Also, the fact that the pin id is the same as the macro pin id tells you to copy other macro pins/traces/vias with the same subnet-id into the net. This helps during component placement and unplacement.

(X,Y) WORD,
Shows the location of the pin! Equals GDB_NIL_WORD (FFFF) when the net pin has not been placed. (The layer mask tells you what layers the pin is on.)

4.5 Rough Traces in the Net

(GDB_ROUGH_TRACE_DEF)

Rough traces will NOT appear in the real gate array. They are merely a Daisy-internal way of representing areas in which the NET_ROUTE command is allowed to route.

Basically, all traces enter the net frame as a result of action in LED:

1) the TRACE command
2) the POINTS_ROUTE and NET_ROUTE commands.
   (The POINTS_ROUTE command has been turned off. For R&D purposes, it may be turned on by invoking LED with the DEBUG option.)
3) COMPONENT placement. Upon component placement, the macro detail traces are copied into two net definitions: the rough_trace_def, and the detail_trace_def.

When the user is in rough-view, automatic net routing (POINTS_ROUTE or NET_ROUTE) will always result in rough routing.

When the LED user is in the detail-view, automatic net routing will always result in detail routing. If, in detail-view, LED needs to call the Rough Router to move across bitmap boundaries before carrying out the routing with the Detail Router (called by POINT_ROUTE and NET_ROUTE commands), then rough routing will be entered here as well.

TYPE BYTE,
No bits are copied into here from the macro. If the net is from a macro subnet, this field gets the 'valid-item' bit (always set by LED) and the 'macro-item' bit. If the rough trace comes from user routing in LED, only the 'valid-item' bit is set.

(X1,Y1) WORD,
Location of one endpoint of the rough trace. In Daisy grids, absolute. These coordinates must be less than or equal to the respective coordinates for the other endpoint (below). [X1<=X2, Y1<=Y2].

(X2,Y2) WORD,
Location of the other endpoint of the rough trace. In Daisy grids, absolute.

4.6 Detail Traces in the Net

(GDB_DETAIL_TRACE_DEF)

This structure is just like the macro_trace, but with Word instead of Integer coordinate variables.

All detail traces are one grid-point in width. There is no width field which would alter this. Therefore, a trace takes up just one grid width all the way along its length (horizontal or vertical).

These fields show where metal ("trace") will be required in the physical gate array in order to implement electrical nets.

TYPE BYTE,
No bits are copied into here from the macro. If the net is from a macro subnet, this field gets the 'valid-item' bit (always set by LED) and the 'macro_item' bit. If the rough trace comes from user routing in LED, only the 'valid-item' bit is set.

LAYER_MASK WORD,
Shows the layers that this trace exists on. Only one layer should be set! (for each trace). This includes the phantom layer as one layer. The trace should be on layer one or on layer two or on the phantom layer. Traces must not be on more than one layer.

(X1,Y1) WORD,
Gives one endpoint of the detail trace. Daisy grids, absolute.

(X2,Y2) WORD,
Gives the second endpoint of the trace. Daisy grids, absolute.

4.7 Vias in the Net

(GDB_VIA_DEF)

This structure has a twin: GDB_FIXED_VIA.

The source of vias is either a macro (macro subnet) or user routing. The user routing may be manual (TRACE command) or automatic (DROUTE).

TYPE BYTE,
If the via is from a macro subnet, then this mask will get the macro_via.type, plus it will have the 'macro-item' bit (set by LED).

LAYER_MASK WORD,
Tells which layers are involved. The vast majority of vias go between layers one and two. Both the layer one and two bits would be turned on (set to 3) in that case. It is OK to have vias between the phantom layer and another layer, or between all three layers. The phantom bit should only be set in a via copied from a macro.

(X,Y) WORD;
Gives the location of the via in the Daisy grid, in absolute coordinates.

4.8 Discussion of the TYPE field

A bitmask is used to set the TYPE field in the following object definitions: pin, detail-trace, rough_trace, and via. The 'net_type' is an entirely different kind of field.

The TYPE field indicates whether the item is part of an underpass or a macro, whether it is a fixed item, and so on. LED will set the TYPE fields when it creates the net routing. E.g., -MACRO_ITEM, or -UNDERPASS_ITEM, or -PAD_ITEM (pad_item is used for Motorola only). PAD_ITEM should not be used for anything else.

The mask options available are defined in the section on Schema Data Constants in the Database Conventions chapter. Look there for the actual bitmasks.

TRACE_REVERSE is only temporarily set by LED during processing, and should never be found set in a static database.

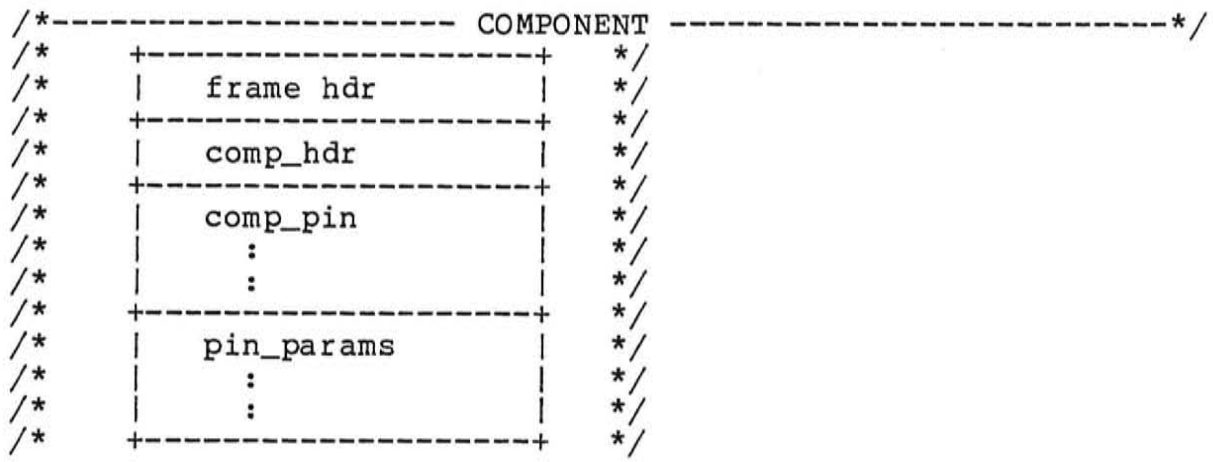
INPUT and OUTPUT bits are used to specify the pin type (this comes from the macro library). If both are set, this indicates bi-directional.

However, when a component is placed, a pin on the macro which instantiates the component can turn into a subnet. Thus, there may be multiple macro pins, with additional traces and vias. LED will copy the TYPE bits from the macro elements (except trace) into the net.

5 The COMPONENT Frame

The database will have one component frame for each component. SLIDE sets most of the component data from DML information.

5.1 Overview



5.2 The Generic Frame Header

```
GDB_FRAME_HDR       LITERALLY
    'OPCODE  BYTE,   set to LSH_FRAME_OPCODE.
    HDR_SIZE WORD,   always the same: 16 bytes.
    SIZE     WORD,   byte size of this particular frame.
    ID       WORD,   frame id within the frame-type.
                     The particular ID results from the
                     order of SLIDE processing and frame
                     creation.  The ID is critical in many
                     areas.  It tells which component!!
    TYPE     BYTE,   frame-type is GDBF_COMPONENT.
    GDB_RECTANGLE'; rectangle exists only when component
                     is placed! LED fills this upon placement.
                     Otherwise, set to FF's (GDB_NIL_WORD). If
                     the component is L-shaped, LED will
                     calculate the enclosing rectangle and put
                     it here. Tells where the component lies
                     in the gate array: it's a recognition
                     area for {SELECT} functions. When an LED
                     user selects a component in the
                     to-be-placed list, LED will find the
                     component by name. However, in a SELECT
                     operation, LED reads through component
                     rectangles like this one. In a PLACE
                     operation, LED reads cell array
                     rectangles until it finds the cell array.
```

5.3 The Component Header

(GDB_COMP_HDR)

The macro_id through cell_array_y fields are set only when the component is placed. When a component has not been placed or has been un-placed, their values should all be GDB_NIL_WORD. The component is placed if and only if the macro_id <> (does not equal) GDB_NIL_WORD.

COMP_OPCODE BYTE,
Set to LSH_COMPONENT_OPCODE.

COMP_HDR_SIZE WORD,
The size of this header.

MACRO_ID WORD,
This is the id of the macro that the component is using (when placed). If component is not placed, the field is GDB_NIL_WORD. Zero may be a legal ID. However, the ID can never be negative because of the WORD type. The macro ID points to a macro frame elsewhere in the .LED file. An ID here implies that the macro has been copied into the .LED file.

Macros reside not only in the .LED file, but also in the macro library files. The macros in the .LED file are copied in from the macro libraries. The macro is copied in when an LED user places a component, and the macro for that placement has not been used before, and thus is not already in the .LED file.

Deleting a macro: in the macro frame, a usage count is kept of how many times a macro has been used (how many components are using the macro). Sometimes this usage_count gets down to zero, but the macro is not deleted at that point. It is when the user runs the UPDATE command that macros with zero usage count are thrown out. UPDATE only deletes macros; it does not add new one.

(X,Y) WORD,
This is the cell-array locating point for the placed component (its location in the gate array), in Daisy grid coordinates. It is also the origin point (Ø,Ø) for the macro which instantiated this component.

Every macro has an origin point. It is always Ø,Ø, presently. The macro's origin point is put down on a cell array point. The (X,Y) field above tells the component where (on the grid) the cell array point is that the macro is attached to.

The next three fields are actually redundant data, since the cell array ID and cell-array- X and Y could be calculated from CELL_ARRAY frame information, but the information is repeated for you here, so that it is ready to access with no hassle.

CELL_ARRAY_ID WORD,
Tells which cell array has got this component placed on it. (Gives the cell_array frame ID.)

CELL_ARRAY_X WORD,
The X location of the relevant cell, in cell_coordinates. (In cell units, not grid units.) I.e., the cell in the cell array's lower left corner has the cell_array_x and y of zero. Its neighbor to the right has cell array coordinates, (1,Ø).

CELL_ARRAY_Y WORD,
The Y location of the cell in the array of cells.

STATUS WORD,
Flags on the component. Currently, only one flag exists:
GDBC_MASK_COMP_FIXED.
The SLIDE program sets status to zero, initially (default is 'unfixed'). If the LED user decides to fix a component (prohibit movement of this component by PLACE), then LED will set the mask. The PLACE and PIM programs will read this field and check it. Cell-rotation information is not carried here; it is carried by flags in the macro and cell_array frames.

ORIENTATION BYTE,
This field is not used. If the field were used, the flipping and rotation possibilities yield eight bit-encodable combinations. GM-1Ø currently fakes macro flips and rotation by: 1) putting more macros in the macro class. 2) checking cell-type id's.

Currently, no macros are explicitly flipped or rotated by GM-1Ø code. None of the code knows it when one macro is a flipped or rotated version of another.

The next five fields are just like the same five in the net frame.

DED_PAGE_ID WORD,
Tells the DED page from which the component came. This ID is somewhat easier to handle than the same field related to nets, since the component is always assigned to just one page, and does not span multiple pages.

NAME_LENGTH WORD,
The length of the component name (number of characters). E.g., "6".

NAME(GDB_NAME_LENGTH) BYTE,
Name of the component. E.G., "XCMP12". Comes from the schematic, via the SLIDE.SING file.

ALIAS_LENGTH WORD,
Length of the alias name.

ALIAS(GDB_NAME_LENGTH) BYTE,
The component's alias name.

MACRO_CLASS_NAME_SIZE WORD,
size of the following name.

MACRO_CLASS_NAME(GDB_NAME_LENGTH) BYTE,
the name of the macro-class that a component uses in its placement. E.g., "NAND2". This is essentially the function that was picked for the component on the DED page; the name comes from the schematic. The data in this field won't ever change.

Upon component SELECT, this macro-class name will come up in the menu with its list of macro names. However, upon component PLACE the macro-class name will not appear on the array. Only the component name and the macro name will appear on the array.

FEATURES WORD,
a bit-encoded word. The bit-mask indicates what features the component requires in order to be placed.

Featured problem: you might have two cells, and a hundred macros can go in them. And every macro except one can go into both cells, so you would really like to reduce the number of cell-types from two to one, and thereby cut the number of macros required in the library by half.

The use of feature bits allows you to individuate some macros and cell-types from others that would otherwise be identical. Feature bits are the something extra that saves on the number of macros that must be defined and kept in a library. Every macro can go into one cell-type only. GAIL contains the ability to code feature bits independently for every cell location.

Procedure for checking a placement, assuming that the component only uses one cell:

1) check that there is nothing else in that cell location. It has to be empty.
2) make sure that the macro is allowed to go into that cell-type.
3) make sure that the features match (see below).
4) perform design rule checks.

The features on the component must be a subset of the features in that cell location. All the feature bits which the component requires must be carried by the cell you are trying to put the macro in. The feature required may be high-power or resistor, for example. Caveat: a macro may need to encompass more than one cell location.

Therefore, two kinds of matching are needed:

1) The features in the origin cell must meet the requirements of the component.
2) The features in the extension cells that the macro occupies must meet the requirements of those cells in the macro cell-list.

Notice that for these multi-cell macros, features are required which are specified in two sources: component and macro.

Features in the component come from DML or from the schematic via the SLIDE.SING file. The features might represent, for example, user-assigned parameters or parameters of components in DED or CED. The component feature might be assigned explicitly by the user, or implicitly, as when DML adds features upon finding Motorola pad pins.

Features in the macro or in the cell array come from GAIL.

PARAM_COUNT WORD,
the count of the GDB_PIN_PARAM_DEF structures which exist following this section.

OFFSET_PARAM WORD,
the offset to the GDB_PIN_PARAM_DEF area.

PIN_COUNT WORD,
This is the count of GDB_COMP_PIN_DEF structures which follow this section.

OFFSET_PIN WORD,
the offset to the GDB_COMP_PIN_DEF area.

POWER WORD,
Not used. SLIDE sets it to zero, currently. In the future, this field might tell the actual voltage (power) on each component, since some manufacturers must perform power-balancing checks across the whole chip. Motorola personnel, when balancing power across the chip, read the macro name from MKFIX or MAKE output and use their own macro power information tables.

COMP_PARAM WORD,
Not used.

PARAM_ID WORD,
Currently, this field should always be set to GDB_NIL_WORD. It is not currently implemented. When implemented, the field will tell the ID of the parameter frame that holds soft parameters for this component.

PLACE_CLASS BYTE,
Used only by the constructive initial placement program, PLACE. SLIDE sets this field to GDB_NIL_BYTE. The PLACE program, when it sees NIL, will copy the place-class from the macro to here. At present this is redundant with that macro data, but in the future, DED/DML/SLIDE will be enhanced to allow custom place-class assignment, per component. In that case, the PLACE program will copy place-class from the macro when it finds NIL here, but if it finds a SLIDE-assigned place-class number here, it will not alter that number.

MACRO_NAME_SIZE WORD,
tells the size of the name following (if component is placed).

MACRO_NAME(GDB_NAME_LENGTH) BYTE,
Name is present only if the component is placed. This is the name of the macro which was used to instantiate the component (the same name as in the Macro frame).

SPARE (2) WORD,
spare space.

5.4 The Component Pin Def.

(GDB_COMP_PIN_DEF)

The component record will have one of these comp_pin structures for every logical pin (on this component).

SLIDE sets all component-pin data. No parts of the LED program except Pin-Swap code and Load-Set (Delta-menu modification) code will modify component-pin data.

ID WORD,
Matches the macro pin ID. The ID's can begin with any value, and need not be consecutive. All different macros in a macro class must be consistent in how the ID's are given to the pins. You want the same pin (e.g, ouputl), no matter which macro you are using.

TYPE BYTE,
A bit-encoded type field. Not used. Pin type is specified in the macro frame on the macro pins.

NET WORD;
Tells the Id of the net that this component pin is on. (ID allows quick access to the net frame).

5.4.1 The Pin Parameter Def.

(GDB_PIN_PARAM_DEF)

This is a separate structure for two reasons. One, every pin can have as many parameters as it needs. Second, since most pins do not have parameters, most pins will not need this accompanying structure -- saves space.

PIN_ID WORD,
This id must match pin ID given above.

PARAM_NUMBER BYTE,
Tells what parameter number it is. E.g., power. At present, only POWER is a parameter on component pins. (see the PIN_POWER_TYPE constant in the Data Constants chapter). Source: SLIDE.

TYPE BYTE,
Bit-encoded byte. At present, if set, tells 'valid-info' and 'SLIDE-created' (see Data Constants). Source: SLIDE.

VALUE DWORD;
The actual value of the parameter, as input by the LOGICIAN user. Source: SLIDE. At present, can only be a value for the amount of power on the pin (or, the pin_param_def structure will not exist). For example: 5, 50, or 200.

Motorola usage is power in % relative to nominal (100). A value of 100 is 100% of nominal, so no explicit entry is needed. By contrast, a value of 200 is 200%, so an explicit entry (200) is needed.

6 The GLOBAL_DESCRIPTOR FRAME

There is one global descriptor frame in each file. And there MUST be one!

When BASE is compiling GAIL data, and therefore is moving data from the GLOBAL block in GAIL into the GLOBAL_DESC frame in its target file, it only compares a few fields explicitly:

1) base chip name.
2) manufacturer name.
3) macro library name.

When BASE processes the user's GAIL text for MACRO frames, it will check to see if there are any differences between incoming macros and the ones which exist in the .BASE and .MLIB files. If BASE finds any discrepancy between the new data and existing data, it will perform two further checks to see whether the user knows that discrepancies are present. First, it will look at the VERSION field in the MACRO frame in the relevant old file and make sure that the incoming macro's individual "version" number is a higher one. Second, if we are checking a .MLIB file, it will look at the MLIB_REV_NUMBER field in the GLOBAL_DESC frame and compare against the REV_NUMBER set by the user in the new GAIL text. The user should increment these rev. numbers to explicitly indicate that macro(s) have changed. If the rev. numbers are not higher, BASE will generate an error message.

The .BASE and .LED files and all the .MLIB file must have a global descriptor frame. Some of the information in their GLOBAL_DESC frames will be compared. BASE will compare between the .MLIB file and the .BASE file.

The schema version in the MLIB file may differ from the BASE file (they use different literals). Since the GLOBAL_DESC frame in the .MLIB file is essentially never used (the only frame used in .MLIB is 'MACRO'), its general structure doesn't really matter. Except for the fields which are checked, BASE lets the user input a global_desc which differs from an existing one. However, user documentation and support will always tell the user to use exactly the same (GAIL text) GLOBAL block for both!

6.1 Overview

```
/*---------------------- GLOBAL_DESC ------------------------*/
/*     +--------------------+   */
/*     |   frame hdr        |   */
/*     +--------------------+   */
/*     |   global_desc_hdr  |   */
/*     +--------------------+   */
/*     |   global_desc_2nd_hdr  */  /* compiler limitation */
/*     +--------------------+   */
/*     |   power_info       |   */
/*     |      :             |   */
/*     |      :             |   */
/*     +--------------------+   */
/*     |   layer_info       |   */
/*     |      :             |   */
/*     |      :             |   */
/*     +--------------------+   */
/*     |   pwgn_name        |   */
/*     |      :             |   */
/*     |      :             |   */
/*     +--------------------+   */
```

6.2 The Generic Frame Header

```
GDB_FRAME_HDR        LITERALLY
    'OPCODE  BYTE,   set to LSH_FRAME_OPCODE.
    HDR_SIZE WORD,   always the same: 16 bytes.
    SIZE     WORD,   byte size of this particular frame.
    ID       WORD,   the ID ought to be zero, but it
                     shouldn't matter.  No code should be
                     reading it, since there is only one
                     frame to look at!
                     (Get the first frame of this type.)
    TYPE     BYTE,   Type is GDBF_GLOBAL_DESC.
    GDB_RECTANGLE'; not used -- no geometric info.
                     All 4 words = GDB_NIL_WORD.
```

6.3 The Global Descriptor Header

(GDB_GLOBAL_DESC_HDR)

GLOB_OPCODE BYTE,
Set to LSH_SKIP_OPCODE.

GLOB_HDR_SIZE WORD,
Tells the size of the whole global header: the total size of the first and second global headers together.

None of the following fields are specified by the GAIL user. The fields are controlled automatically by the BASE compiler.

SCHEMA_VERSION_SIZE WORD,
size of the version string given below.

SCHEMA_VERSION (GDB_NAME_LENGTH) BYTE,
this tells the version number of the schema that this file was compiled with. This is significant for .LED and .BASE and .MLIB files. LED, when it opens macro libraries, checks that the .MLIB file has the correct schema version.

All code opening a file must check this version number. All code except LED must immediately reject files with the wrong schema version in this field. All files to be used must have the latest version number.

Caveat: LED contains the code which must update the database, so it must take appropriate action: warn the user of wrong version and update the file (upgrade the database structure). LED will call a sub-task, LVR (LED VeRsion update), which will carry out the database upgrade.

The LVR sub-task called by the LED program will update both the .BASE and the .MLIB files. The stand-alone LVR program will update only the .MLIB files.

After updating the database structure and data, LVR updates the schema version and the size field above.

When upgrading a very old version, LVR operates iteratively. That is, it will update version 3.0.04 to version 4.0.05, and then it will update from that version to 5.0.02. The latest LVR does not handle anything older than 3.0.04.

REVISION WORD,
Tells the revision level of the .LED file. When the BASE compiler is run in update mode (incrementally update some macros in the .MLIB file), it will increment this number by one. Also, if LVR is called, it will increment this number by one. The UPDATE command in LED is only for macro update; it will not increment this number.

NUM_UPDATES WORD,
This number is incremented by Slide every time it updates the .LED file.

WRAP_FLAGS WORD,
Not used. The field was intended to carry options for a "WRAP" program (never written; replaced by MAKE).

MACRO_LIB_NAME_SIZE WORD,
Not used. If used, it would specify the size of the name

in the next field. Might be set by the BASE compiler.

MACRO_LIB_NAME(GDB_NAME_LENGTH) BYTE,
Not used, though it is set by BASE. The reason the field is not used is that it names only one macro library. The GATEMASTER already supports multiple libraries: a list of all macro libraries is given in the PROFILE file (under /PROJECT). The actual LED search process for macro data is described in the Macro chapter.

MACRO_LIB_PATH_ID WORD,
Not used. See above.

CHIP_NAME_SIZE WORD,
size of the chip name given below.

CHIP_NAME(GDB_NAME_LENGTH) BYTE,
a user-given chip name. E.g., "Ignition-Control". This name is just carried through to MAKE output.

BASE_CHIP_NAME_SIZE WORD,
size of the base-chip name given below.

BASE_CHIP_NAME(GDB_NAME_LENGTH) BYTE,
the name of the base chip being used for gate array layout. E.g., "MOTOROLA_600". Comes from GAIL text. The LED program, when opening macro libraries, checks this name against the same name in the .MLIB files being opened. Inappropriate libraries will be closed.

BASE_CHIP_PATH_ID WORD,
Not used.

MAX_X_GRID WORD,
Gives the extent of the gate array chip in the Daisy X dimension.

MAX_Y_GRID WORD,
Gives the extent of the gate array chip in the Daisy Y dimension.

The next four fields go together with two fields much farther down in this Header: COORD_TRANS_METHOD_X and _Y. Together, they specify the process of output conversion from Daisy coordinates to manufacturer's coordinates. All six fields come from the GLOBAL_DESCRIPTOR frame in the GAIL text.

The GAIL global descriptor frame also specifies information which ends up in the schema's PROJECTION frames (if the method is projection). If the method of conversion is linear, then the origins and factors shown below are used to calculate the linear transformation.

MFR_ORIGIN_X WORD,
In GM-1Ø coordinates, tells the location to us (Daisy) of the manufacturer's origin (of their coordinate system).

MFR_ORIGIN_Y WORD,
Y value, corresponding to the above.

GM_TO_MFR_X_FACTOR WORD,
this factor used only for linear conversion. It's a multiplier.

GM_TO_MFR_Y_FACTOR WORD,
this factor used only for linear conversion.

SPARE_Ø BYTE,
Spare space. (This used to be one particular field, but it has now been turned into a spare.)

OFF_GRID_PIN_TYPE BYTE,
comes straight from GAIL.

BITMAP_COUNT WORD,
from the BITMAP block in GAIL. It is calculated by multiplying the number of bitmap rectangles along the X dimension times the number along the Y dimension. The product of that multiplication is entered here.

The next three file stamps are set by programs other than BASE or SLIDE. They give information about whether a needed file exists, and whether it is up to date. If the file stamp is zero, then the relevant file needs to be created anew for use in the LED environment.

BITMAP_FILE_STAMP DWORD,
stamp on the .BMP file. Used by LED, the automatic routers, and the design rule checkers.

LUSH_FILE_STAMP DWORD,
stamp on the .SCRN file. Used by LSH to maintain currency of the screen display.

ROUGH_FILE_STAMP DWORD,
stamp on the .RAIN file. Used by the Rough Router.

BASE sets the next seven fields according to GAIL-user input in the GLOBAL_DESCRIPTOR frame. See the GAIL manual for more detailed discussion on all the following fields.

VIA_RULES WORD,
Indicates whether fixed or floating vias.

RESISTANCE_UNIT INTEGER,
a number which is an exponent on the resistance value. Represents log-10 ohms.

CAPACITANCE_UNIT INTEGER,
a number which is an exponent on the capacitance value. Represents log-10 farads.

DELAY_TIME_UNIT INTEGER,
a number which is an exponent on the delay-time value. Represents log-10 seconds.

LENGTH_UNIT WORD,
Not used. Was intended to represent grid units per micron. The current Linear-or-Projection functionality makes this unit obsolete. The unit could only be of value if an array is of stepl in BOTH X and Y, and if the step size is the same, and if an even number of Daisy grids fit into a physical micron. That means never.

ROUGH_X_DIVISOR BYTE,
A number which is used as Rough View divisor by LED.

ROUGH_Y_DIVISOR BYTE,
A number which is used as Rough View divisor by LED.

OVERVIEW_X_DIVISOR BYTE,
A number which is used as Overview divisor by LED.

OVERVIEW_Y_DIVISOR BYTE,
A number which is used as Overview divisor by LED.

POWER_COUNT WORD,
Not used. Will be the count of POWER_INFO_DEF records.

OFFSET_POWER_INFO WORD,
Not used. Will be the offset to the POWER_INFO_DEF data.

LAYER_COUNT WORD,
Count of LAYER_INFO_DEF records.

OFFSET_LAYER_INFO WORD,
Offset to the LAYER_INFO_DEF data.

The next five fields go together and specify when this file was created. The sequence of fields is identical to the order of the information that one gets by calling the system timestamp routine. For comparison, later, the timestamp must match Daisy standard. That is, year = 82, for example.

The BASE program writes to these fields when it creates .BASE file or updates that file's information. SLIDE writes to the creation fields when it first builds the .LED file. The LED program does not modify them.

```
CREATION_DAY        BYTE,
CREATION_MONTH      BYTE,
CREATION_YEAR       WORD,
   E.g., year = '82'.
CREATION_HOUR       BYTE,
CREATION_MINUTE     BYTE,
```

SING_TIMESTAMP DWORD,
This timestamp is not used at all. It was intended for receiving the date of the DML run, but that information is not obtainable by SLIDE.

PACKAGE WORD,
Used only by MKFIX. Held for output only. This is a mask which indicates the type of package. The type of package (ceramic/plastic, no. of pins, etc.) can make nasty differences in legal combinations of pad and pin, features, and id's. Please see the GAIL manual for further details. This field really needs to be a text string so that the manufacturer's designation can be input. E.g., package "6ØLAC".

MANUFACTURER_NAME_SIZE WORD,
Size of the name following.

MANUFACTURER_NAME (GDB_NAME_LENGTH) BYTE,
Tells the name of the company which will fabricate this gate array. E.g., "MOTOROLA". See GAIL.

COMPANY_NAME_SIZE WORD,
size of the name following.

COMPANY_NAME (GDB_NAME_LENGTH) BYTE,
Intended to hold the name of the company which is using LED to design this chip. (Not currently accessible by the user.)

PARAM_ID WORD,
Not used. In the future, when soft parameters are implemented, this particular ID will point to the parameter frame holding global parameters (parameters which apply to the whole chip).

COORD_TRANS_METHOD_X BYTE,
Tells whether linear or projection method should be used for output conversion, along this dimension of the coordinates.

If the field above specifies PROJECTION method, then the projection array values from the GAIL GLOBAL_DESCRIPTOR frame have been copied into the PROJECTION frame in this file, and are not held in this section.

The effect of the COORD_TRANS_METHOD_X field above is independent of the field below. Each dimension (X or Y) has its own method of transformation.

COORD_TRANS_METHOD_Y BYTE,
tells whether linear or projection method should be used for output conversion, along this dimension of the coordinates. If this item equals PROJECTION, then the effects are the same as noted above.

DATA_VERSION WORD,
Not used. The schema version number is used to indicate changes in the database instead. The data version number could be used to indicate that GM-1Ø code has been altered in such a way that some data will now have a different meaning, yet no structural database changes have occurred.

MLIB_REV_NUMBER WORD,
The user-defined revision number on the macro library. The BASE program reads the user's GAIL input for a .MLIB run, and sets this field accordingly. The BASE program will issue an error if it finds that any macro in an existing .BASE file does not agree with an incoming macro of the same name, and the user-defined Rev_Number (in GAIL) is not greater than the number found in this field.

LED_CORRUPT_FLAG BYTE,
Not currently used. In the future, PBM will set the flag in the .LED file when it tries to create .BMP and finds corruption (illegal routing). All code opening .LED should check this flag. If the flag is set, some data is dangerously wrong, and the process should issue a warning and possibly (depending on a configuration flag) decide not to run.

PWRGND_NAME_COUNT WORD,
count of PWRGND_NAME_DEF structures below. The count is either 'Ø' (zero) or '7'. To avoid monstrous amounts of processing, the user has been limited to 7 very large nets, whether power or ground.

OFFSET_PWRGND_NAME WORD,
offset to the PWRGND_NAME_DEF area below, which currently carries the power and ground names as they are known in

DED, prior to being broken into GATEMASTER nets.

LVR_CONVERT_FLAG BYTE,
A mask set by the LVR sub-task when it runs. The mask reveals the exact history of database upgrades, and indicates the age of the database. See the Database Constants chapter for the literals.

FILE_TYPE BYTE,
This field answers the question, "Is this file .MLIB, .BASE, or .LED?" The SLIDE program copies the .BASE file into a .LED file and changes the file-type. See the Database Constants chapter for the literals. This lets GM-1Ø programs know the file-type even if the user has screwed around with the filename extension. Currently, this field is only checked by SLIDE, which will refuse to read a .MLIB-type file.

SPARE BYTE,
a spare field.

SPARE1 (123) DWORD,
a bunch of spare space.

6.4 The Secondary Global Header

(GDB_GLOBAL_DESC_2ND_HDR)

```
LITERALLY
'GDB_GLOBAL_DESC_2ND_HDR_1,
 GDB_GLOBAL_DESC_2ND_HDR_2',
```

The fields contained in the '2ND_HDR' cannot be put into the main global header due to PLM limitations on the size of any one structure.

This "2ND HDR" comes immediately after the first GLOBAL_DESC_HDR in the database. Programs must create a pointer to the second header by using the PLM-intrinsic SIZE routine to get the size of the first header, and then incrementing the frame pointer by the frame header size and the 1st HDR size. The pointer arithmetic looks as follows: Second_hdr_ptr = utl_increment_ptr (global_desc_ptr, size(frame_hdr) + size(first_hdr)).

6.5 The Power Information Def.

(GDB_POWER_INFO_DEF)

This structure is not yet used. The GDB_PWRGND_NAME_DEF further down in the frame is used currently. It involves many small nets and a missing-pin connection method.

In the future, power/ground net information will be contained in an array of these records. The nets in this section may be called pseudo-nets, because they are handled differently from real nets. Pseudo-nets will allow power and ground nets to have connections without requiring much user routing. The structures for this pseudo-net method are currently implemented in the schema, but not implemented in the code.

Essentially, we would like to flag all the necessary power and ground connections, and carry this information through to the manufacturer, but avoid treating all the connections as one net.

Basically, there are two ways to handle power/ground nets:

1) connect to a pseudo-pin (carries a pseudo subnet), or
2) connect to a piece of rail (directly to a pseudo-net).

Alternative number 2 is the method which is currently used.

[None of the fields below are used yet.]

ID WORD,
This will be the id of the pseudo-net that can be found on the component pin. Start the ID's backwards so that any possible conflict with the regular nets is extremely unlikely. (ØFFFE, ØFFFD, ...) However, it's very safe: since a chip will only have 2 or 3 pseudo-nets, there can only be a few such ID's.

The code will have to read the subnet id from the component pin, and look in this field to see whether the net id is here or not. If it is not, then the net is a regular net. If the id is here, then the net is a pseudo-net (power or ground).

TYPE BYTE,
Not used. Just 8 bits for whatever may be needed.

POWER INTEGER,
This is the voltage in the pseudo-net. E.g., + or - 5 Volts. D.C. Ground nets might have zero voltage, or negative.

NAME_SIZE WORD,
the size of the name following.

NAME(GDB_NAME_LENGTH) BYTE,
the name of the net. E.g., "POWER_ONE".

6.6 Layer Information Def.

(GDB_LAYER_INFO_DEF)

An array of these records holds spacing rules for design rule checking. There is one set of data (one record) per layer.

All fields in this definition are set by BASE, carried through by SLIDE, and not touched by LED.

FLAGS WORD,
Holds only layer-options from GAIL input. It's a bit-encoded mask, equal to the inclusive-OR of all the layer options.

TYPE BYTE,
Not used. Waits for bit-encoding needs.

TRACE_WIDTH WORD,
Holds the trace-width from GAIL input. BASE insures that this is a positive integer. Please see the GAIL manual for further information.

All the following design rule distances are set by BASE according to the GAIL Design_Rules block. LED only reads them.

```
EAST_WEST_PIN_PIN        BYTE,
EAST_WEST_PIN_TRACE      BYTE,
EAST_WEST_PIN_VIA        BYTE,
EAST_WEST_TRACE_TRACE    BYTE,
EAST_WEST_TRACE_VIA      BYTE,
EAST_WEST_VIA_VIA        BYTE,

NORTH_SOUTH_PIN_PIN      BYTE,
NORTH_SOUTH_PIN_TRACE    BYTE,
NORTH_SOUTH_PIN_VIA      BYTE,
NORTH_SOUTH_TRACE_TRACE  BYTE,
NORTH_SOUTH_TRACE_VIA    BYTE,
NORTH_SOUTH_VIA_VIA      BYTE,
```

6.7 Power-Ground Name Def.

(GDB_PWRGND_NAME_DEF)

To really understand power and ground nets, and what data the user puts in to indicate them, please see the GAIL manual.

This is the structure which is currently used for power and ground name information. An array of these structures holds the names of active power and ground nets. A net is 'active' if it is in use (if something is connected to it). Each power or ground "net" may have many small NET frames associated with it. Currently, each component which makes a power/ground connection gets a corresponding NET frame to represent a missing pin.

Either there are none of these, or the full allotment of seven records exists. Each record holds the DED-schematic name of a power or ground net.

NAME_SIZE WORD,
size of the name following.

NAME (GDB_SHORT_NAME_LENGTH) BYTE,
This is a net name (a power or ground net only). For instance, a name might be "PWR1".

The name is kept short because the net names from the SLIDE.SING file have junk appended. Therefore, GATEMASTER must use a shorter length name, and by convention, short names in GATEMASTER are GDB_SHORT_NAME_LENGTH (currently = 4).

BASE will set up net data which is specified in GAIL. However, the power/ground net names which the user inputs in the GAIL text must match the DED net names which reach the SLIDE.SING file through DML.

The order of the net names is implicit. SLIDE receives net names and sets up corresponding power/ground net numbers in net.net_type. For example, if 'PWR1' is the first power/ground net name that SLIDE reads, the macro which has the big power/ground trace for 'PWR1' will have an id of 65001. For more explanation, see the Schema Constants chapter and the GAIL Manual.

Code considerations:

1) MAKE still needs the ability to find all these little pieces and output them as one huge net with one name.

2) Also, we don't want the menus to cough up thousands of tiny nets and make the user route them. So a little special handling is necessary in the "NETS" menu code. That is, the LOAD_SET command in LED will display power/ground nets in a separate menu.

3) How to tie pins to ground? Having a ground pin on each component is awkward, so all ground connections are made to a grounding trace that comes from the macro. However, the incoming netlist is supposed to have a complete list of subnets which need routing. In most cases, each pin is assumed to be a subnet in itself. The pin's coordinates are filled in when the component is placed.

All the pins on one component which belong to the same power/ground net taken together form one net. Usually, however, only one pin connects to power/ground.

Component pins therefore serve double duty: they are pins which come from the macro, and each becomes one of the subnets that the net is suppose to contain. This might have been split out into two fields, but that would require severe code changes.

The subnet in the macro does not have a pin on it. The missing pin is indicated by a bitmask in the NET_TYPE field. SLIDE sets 3 bits in that field to indicate that a net is a power net, and does NOT add the implicit pin.

This whole structure is needed because so many pins connect to power and ground that the huge size of the needed power and ground nets would be unmanageable. (A design might require that a pin on every component in the chip be tied to ground, and that all unused pins be tied LOW, to ground. That results in far too many pins in one net to be handled easily!) The structure attempts to save on the space needed to tie lots of similar info to many nets.

The existing net frame could have been used to represent these little nets, but that means that too many tiny net connections would come up in LED for the user to route. This would be very user-hostile. Therefore, power and ground nets had to be broken up into little pieces. Each component has its own little power and ground nets.

7 The MACRO FRAME

All data in a macro frame comes from GAIL text. The macro frame in the .MLIB file is read-only. LED will copy macros from the .MLIB file into the .LED file as needed, but LED will not change macro data. LED will change only the macro usage_count field.

A .MLIB file has very few of the frame types that a .LED file has. Specifically, a .MLIB file will have the following frames: one GLOBAL_DESC frame, many MACRO frames, a few MACRO_CLASS frames, one macro-name index frame, and one macro-class-name index frame. Every macro that needs to be in a macro library is initially represented only in that .MLIB file.

A .BASE file may also have some macros in it. If Empty and Permanent macros are needed as a part of the base chip, the user must enter them as a part of the GAIL input for the .BASE file. The BASE program reads that data and builds macro frames for those Empty and Permanent macros in the .BASE file. Then, SLIDE, when it builds the .LED file, will carry those macros from the .BASE file into the .LED file.

When the LED program is first invoked, it follows a certain search process for macro data. LED looks in the PROFILE file to get the list of relevant macro libraries (.MLIB files). It opens each .MLIB file and first, checks its schema version number against the .LED schema version. Second, it checks the base_chip_name against the .BASE filename. In brief, LED will leave the right libraries open, and close inappropriate ones.

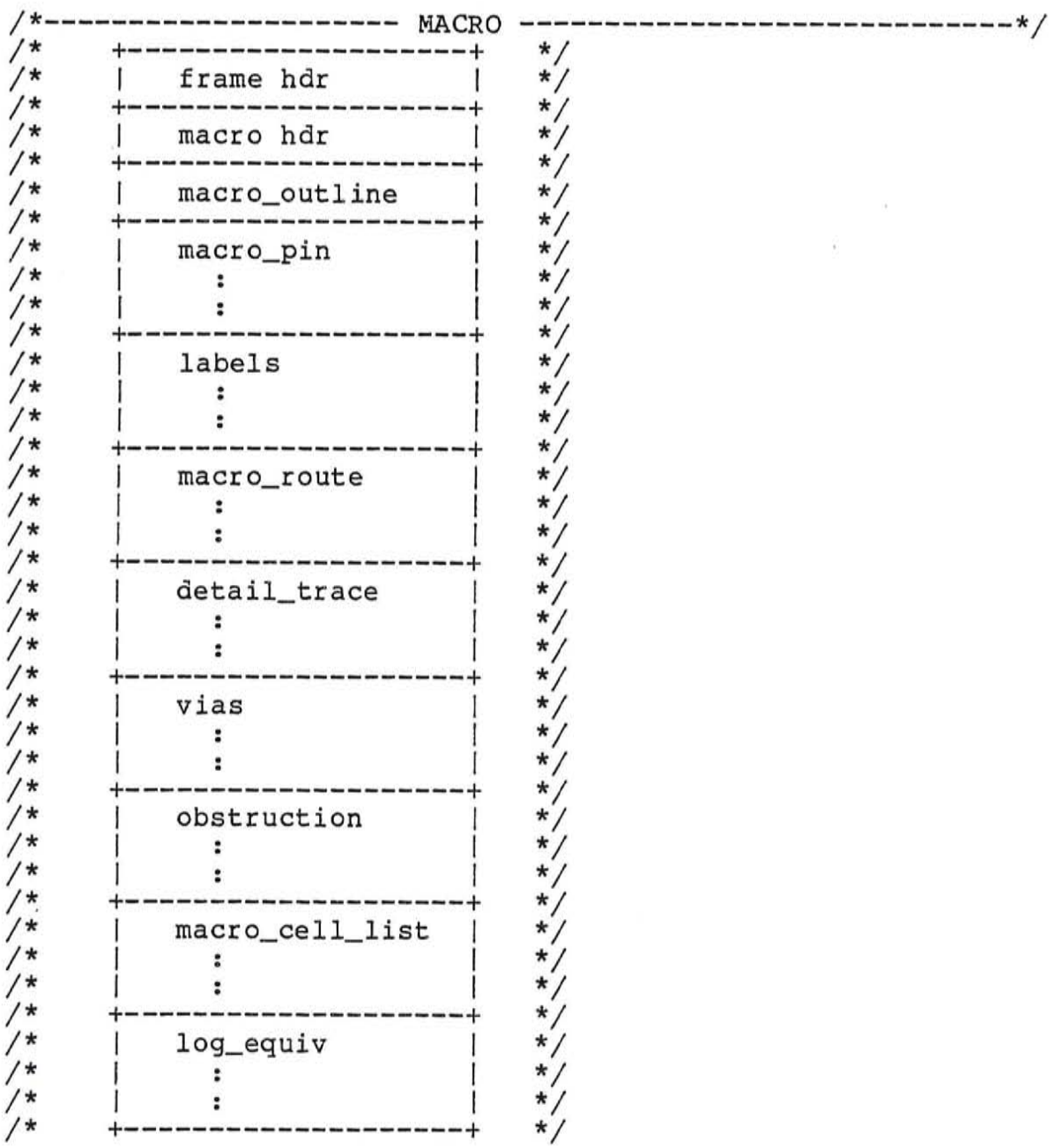
Upon the user's SELECT of a component, LED uses the macro-class index to find the macro-class for the component, and displays the macro-class name and its list of macros which can implement that component's function. Upon the user's SELECT of a particular macro, LED will blink the legal cell locations.

When the user attempts a component placement, LED carries out a search: see if there is a macro which can instantiate that component at that location. First, upon the user's SELECT of a component, LED finds the appropriate macro-class frame (using macro-class name-index), and displays the list of suitable macros. Second, upon the user's selection of a macro, LED finds the specific macro by name (using macro name-index). LED will search the .LED file first and then all the open .MBLIB files in the order they are listed in the PROFILE file.

When the user hits PLACE to place the macro, LED will copy the appropriate macro frame from the .MBLIB file into the .LED file. LED will also increment the macro's USAGE_COUNT field by one. USAGE_COUNT is the only macro field that LED writes into.

Every subsequent placement of a component which makes use of that macro will not result in a transfer process. When LED uses a macro that it already has in the .LED file, it merely increments the macro's USAGE_COUNT. When a component is unplaced and the macro is removed, the count is decremented but the macro frame is not removed from the .LED file.

7.1 Overview



7.2 The Frame Header

```
GDB_FRAME_HDR          LITERALLY
    'OPCODE   BYTE,    set to LSH_FRAME_OPCODE.
    HDR_SIZE  WORD,    always the same: 16 bytes.
    SIZE      WORD,    byte size of this particular frame.
    ID        WORD,    frame ID within the frame-type.
                       (uniquely identifies a macro).
    TYPE      BYTE,    type is GDBF_MACRO.
    GDB_RECTANGLE';    enclosing rectangle for physical
                       macro data. Declared using the Relative
                       Rectangle Definition (see Conventions).
                       Two kinds of macro data exist: physical
                       (where things are), and graphical (display
                       stuff). This rectangle encompasses all the
                       physical elements, and most of the
                       graphical display.
```

The GDB_RECTANGLE in the frame header is declared as WORD, but the data is actually encoded as integer since some macro rectangle coordinates may have to be negative. Therefore, the programmer must equivalence this WORD-valued GDB_RECTANGLE to INTEGER, and then read the correct integer coordinates. The bit-pattern is correct even when the coordinates are negative.

7.3 The Macro Header

(GDB_MACRO_HDR)

MAC_OPCODE BYTE,
Set to LSH_MACRO_OPCODE.

MAC_HDR_SIZE WORD,
size of the macro header.

VERSION WORD,
Tells the version number of this particular macro. Comes straight from GAIL. Please see the GAIL manual. This refers only to a revision level of the macro data, not to database structure. Not to be confused with schema version! This field is used only by the UPDATE command, when the user updates macros which have bad data. Programs which check this field need to shut down if the version level is wrong. The UPDATE command in LED compares this version number in the .LED file with the same macro field in the .MLIB file. If it finds a macro which has been used but which is now obsoleted by an incoming macro, it will generate a message in the log file that there is an "un-updated macro" in the file. It will not alter a macro which is being used in a placement. UPDATE will only alter the database if the macro which has been obsoleted is not being used. It will delete the macro's frame from the .LED file; it will NOT add the new macro frame.

GDB_RELATIVE_TEXT_DEF,
This definition holds the macro name, its location, format, and name-length; the structure is defined in the Database Conventions chapter. Many programs may look in here to verify the macro name. LSH will use the name and name-length in this structure for graphic display.

The type and features fields in the macro's cell-list entries will specify the macro's legal placement. The macro's name no longer specifies the macro's class or its cell-type for placement. The macro name is only a unique name. The information comes straight from GAIL input, and is set by BASE. Please see the GAIL manual for further information.

MACRO_TYPE WORD,
Not used. The TYPE field in the macro's GDB_CELL_LIST DEF records says which cell-type is legal for each cell.

LEGAL_ORIENTATIONS BYTE,
Not currently used. This field used to be 'legal_transforms'.

ACTUAL_ORIENTATION BYTE,
Not currently used. This byte will be set in the future, whenever the component is placed.

FEATURES WORD,
Not used. Features are specified in GDB_CELL_LIST_DEF. This field is to be deleted in the next schema change.

PARAM_ID WORD,
Not used.

USAGE_COUNT WORD,
Shows the number of times that this macro has been used to instantiate its associated components. When this macro is used to PLACE a component, LED looks for the macro frame in the .LED file, copies it in from a .MLIB file if necessary, and increments the usage count.

If the user performs a subsequent unplace, the usage count is decremented by one. The macro frame is not removed from the .LED file. Unused macros are eventually cleaned out of the .LED file by the UPDATE command.

PLACE_CLASS BYTE,
Set by BASE from GAIL input. Indicates the place-class for initial placement (PLACE program) only. For further discussion, please see the GATEMASTER Reference manual (chapter on Automatic Placers and Routers) and the GAIL manual.

SPARE BYTE,
a spare field.

OUTLINE_COUNT WORD,
number of points in the macro outline; if 2, the outline is a rectangle.

OFFSET_OUTLINE WORD,
offset to the outline data.

PIN_COUNT WORD,
count of macro pins.

OFFSET_PIN WORD,
offset to pin data.

ROUTE_COUNT WORD,
count of macro-route records. (Not yet used.)

OFFSET_ROUTE WORD,
offset to macro-route data. (Not yet used.)

TRACE_COUNT WORD,
count of macro detail traces.

OFFSET_TRACE WORD,
offset to detail trace data.

VIA_COUNT WORD,
count of macro vias.

OFFSET_VIA WORD,
offset to via data.

OBSTRUCTION_COUNT WORD,
count of obstructions.

OFFSET_OBSTRUCTION WORD,
offset to obstruction data.

CELL_LIST_COUNT WORD,
count of cell-list definitions.

OFFSET_CELL_LIST WORD,
offset to cell-list data.

LOG_EQUIV_COUNT WORD,
count of logical equivalences.

OFFSET_LOG_EQUIV WORD,
offset to logical equivalence data.

ROUGH_X_BLOCKAGE WORD,
number of rough routing tracks blocked by the macro in the X dimension. See GAIL. Only the rough router uses this field.

ROUGH_Y_BLOCKAGE WORD,
number of rough routing tracks blocked by the macro in the Y dimension. See GAIL. Only the rough router uses this field.

COMP_NAME_FORMAT BYTE,
from GAIL. The format, together with the (X,Y) position below, determines the position and orientation of the component name in the macro display. Used only by LSH.

COMP_NAME_X INTEGER,
Specifies the X in the name's (x,y) shift relative to macro's origin. Source: GAIL. Used only by LSH.

COMP_NAME_Y INTEGER,
Specifies the Y in the name's (x,y) shift relative to macro's origin. Source: GAIL. Used only by LSH.

LABELS_COUNT WORD,
count of labels.

OFFSET_LABELS WORD,
offset to label data.

Most of the structures below are looked at by many programs. BUP code (design rule checker) looks at the pins, traces, vias, and obstructions. It has to check all data that has a physical side. After LED has copied the macro into the .LED file, code from LED, BUP, and LSH will look at the detail (x,y) coordinates.

The FORMAT fields are only used by LSH. LOGICAL_EQUIVALENCE stuff is only read by the Pin Swap program. LED placement code is the only code that actually copies macro traces and vias into nets.

7.4 The Macro Outline

The macro outline follows the macro header (right here).

The macro outline is an array of (x,y) pairs (gdb_relative_outline_def records).

7.5 The Macro Pin Def.

(GDB_MACRO_PIN_DEF)

Macro pins are set up in GAIL text. That data is read-only. However, LED may add macro pins to NET frames when a component using this macro is placed. LED and BUP code will read the macro pin data.

There may be more than one pin with the same id. In that case all pins are on the same subnet (they must be routed together, if not already routed together with traces).

ID WORD,
Macro Pin id. Matches component_pin.id.

TYPE BYTE,
Tells the type of pin. See the GDSCC type constants.

LAYER_MASK WORD,
Tells what layers the pin is on.

(X,Y) INTEGER,
The location of the pin, relative to the macro origin.

LOCAL INTEGER,
This is offset to the off_grid location (in mfr's units, relative to macro origin). The offset is present only if offset pins are being used.

7.6 The Macro Label Def.

(GDB_MACRO_LABEL_DEF)

This structure holds the labels for the macro pins and correlates them to the pin id. Everywhere in the database, the ID is used. Only when dealing with external ASCII text (in SLIDE.SING, .MAKE, or in menu display) is the correlation needed.

ID WORD,
The macro pin id.

X INTEGER,
Y INTEGER,
The origin point for the label, in Daisy coordinates.

FORMAT BYTE,
Tells how to orient the label.

TEXT (GDB_SHORT_NAME_LENGTH) BYTE,
This is the pin's label (a text string). No length field is specified above. The name is padded with ASCII-nulls (binary Ø's).

SLIDE looks at the textual label from GAIL input and converts it into an ID. Thus, SLIDE makes the necessary correlation between the pin label and the subnet ID (using GGU routines). The MAKE program and the DELTA-LIST code need the correlation.

LSH looks at all except the ID. LSH doesn't need the correlation. It just puts the label on the screen and doesn't read the ID.

7.7 Macro Routing Definitions

BASE reads GAIL input and enters macro-routing records accordingly. Upon component placement, LED copies the macro detail traces into the appropriate net frames twice: both as detail traces and as rough traces.

(GDB_MACRO_ROUTE_DEF)

This structure is not yet used. It will tell the system about a macro subnet which must be routed even though it is incomplete in the macro's definition and does not belong to any net outside the macro (does not contain any pins on DED nets). Please see the GAIL explanation.

ID WORD,
This id matches pin id. This id is serving as subnet id.

FLAGS WORD,
One word for future bit-encoding.

(GDB_MACRO_TRACE_DEF)

This structure is the same as gdb_detail_trace_def except that word variables are integer instead. (plus there's an ID.)

ID WORD,
Subnet id.

TYPE BYTE,
This may have GDBC_MASK_UNDERPASS_ITEM.

LAYER_MASK WORD,
Which layer the trace is on.

(X1,Y1) INTEGER,
(X2,Y2) INTEGER,
The two endpoints of the detail trace.

(GDB_MACRO_VIA_DEF)

This structure is the same as GDB_VIA_DEF except that word variables are integer instead. (Plus there's an ID).

ID WORD,
The subnet id.

TYPE BYTE,
This field may have GDBC_MASK_UNDERPASS_ITEM (will be implemented soon).

LAYER_MASK WORD,
What layers the via is on.

(X,Y) INTEGER,
The location of the via in the Daisy grid.

7.8 Macro Obstruction Def.

(GDB_MACRO_OBSTRUCTION_DEF)

This structure is the same as the Relative Obstruction Def. in the Obstruction frame except that word variables are integer.

Obstructions are only read by the Bitmap design rule checker. The Rough Router does not read this macro obstruction data. It rads the rough_x_blockage and rough_y_blockage in the macro header above.

GDB_RELATIVE_OBSTR_DEF,
This structure is defined in the Macro frame.
Literally, it is:

TYPE,
LAYER_MASK,
RELATIVE_RECTANGLE_DEF (with INTEGER coords.)

7.9 Macro Cell List Def.

(GDB_CELL_LIST_DEF)

This structure describes which base array cells are needed at the placement site for the macro. Every macro MUST have a cell-list.

The origin cell is now first in the cell-list. The record for the origin cell tells which cell-type is the only legal placement. In schema version 4.Ø5 and earlier it was not listed.

The programs LED, PLACE, and PIM will look at the macro cell-list.

TYPE BYTE,
The cell-type that this macro-cell requires for placement.

FEATURES WORD,
Features which this macro-cell requires in the placement (which the macro requests of the cell array).

(X,Y) BYTE,
This is set to (Ø,Ø) if it's an origin cell. If it's an extension cell, the coordinates give the cell location in cell-units, relative to the origin cell (in signed-bytes: e.g, ØFEH means -2). The macro's location is specified in the component frame, not here.

The features required in the origin cell come from the component, whereas the features required in all extension cells come from the macro cell-list entries.

REQUIRED_ORIENTATION BYTE;
This field is not yet used.

7.1Ø Macro Logical Equivalents Def.

(GDB_LOG_EQUIV_DEF)

An array of these records holds data on logically equivalent pins.

Each record defines a set of logically equivalent pins. The record has a count and then an array of pin id's which are logically equivalent.

COUNT WORD,
Tells how many pin_ids follow.

TYPE WORD,
Not currently used. For future bit-encoding.

PIN_ID (1) WORD;
This is an array of pin id's (WORD).
For example:

1
2
3
4
5

Where one count/type/array record ends, the next record starts. The word-valued COUNT at the start of each record tells the size of its PIN_ID array so that you can skip from record to record.

8 The MACRO_CLASS Frame

All macro-class information comes from GAIL. The frame specifies classes of equivalent macros. Only LED, PLACE, and PIM care about macro-class data.

The macro-class name is held in the one instance of GDB_MCLASS_NAME_DEF that appears in the macro-class header. For example, a macro-class name could be 'NAND2'.

When the LED user hits SELECT on a component, LED finds the right component frame, gets the macro-class name, looks for that macro-class frame, copies the desired MACRO_CLASS frame from a .MBLIB file if necessary, and displays the macro-class name in a menu with its list of macros. The macro frames are not copied into the .LED file until the user hits PLACE.

8.1 Overview

```
/*---------------------- MACRO_CLASS-------------------------*/
/*     +--------------------+   */
/*     |   frame hdr        |   */
/*     +--------------------+   */
/*     |   mclass_hdr       |   */  Holds macro-class name.
/*     +--------------------+   */
/*     |   mclass_name      |   */  List of equivalent macros.
/*     |      :             |   */
/*     |      :             |   */
/*     +--------------------+   */
```

8.2 The Frame Header

```
GDB_FRAME_HDR    LITERALLY
    'OPCODE  BYTE,  set to LSH_FRAME_OPCODE.
    HDR_SIZE WORD,  always the same: 16 bytes.
    SIZE     WORD,  byte size of this particular frame.
    ID       WORD,  frame id within the frame-type.
    TYPE     BYTE,  type is GDBF_MACRO_CLASS.
    GDB_RECTANGLE'; not used -- no geometric info.
                    All 4 words = GDB_NIL_WORD.
```

8.3 The Macro Class Header

(GDB_MCLASS_HDR)

MCLASS_OPCODE BYTE,
Set to LSH_SKIP_OPCODE.

MCLASS_HDR_SIZE WORD,
size of the macro-class header.

VERSION WORD,
Currently, BASE sets this field to zero, and no one (not even the UPDATE command) uses it. BASE does not check this field, or increment it.

MCLASS_TYPE WORD,
Not used. Spare bits for future encoding.

SPARE WORD,
spare.

GDB_MCLASS_NAME_DEF,
This one record holds the name of the macro-class and its size. E.g., 5, 'NAND2'.

NAME_COUNT WORD,
count of macro names in the array below (thus, this field tells how many equivalent macros the macro-class has).

OFFSET_NAME WORD,
offset to the list (array) of macro names.

8.3.1 Macro Class Name Definition

(GDB_MCLASS_NAME_DEF)

This structure holds an array of equivalent macros. Each record of this type holds one macro name. The macro names (come from GAIL input) may be totally arbitrary.

NAME_SIZE WORD,
size of the name in the next field.

NAME(GDB_NAME_LENGTH) BYTE,
This is a macro name. The fact that it is here means that it belongs to the macro-class defined above.

FLAGS WORD;
Not used. Set to Ø.

9 The CELL_ARRAY Frame

Almost all of this is read-only, and set up by the BASE compiler from GAIL text. Only the COMP_ID is written into after set-up. LED writes the ID of the component that is placed at a cell array location.

LSH uses the cell array to paint the correct cell-type outline at the correct location.

LED uses the cell array frame to check on cell locations for all component placement functions. For example, when the user hits PLACE for a component placement, LED must search through the cell array frames to see if one intersects the cursor location. LED will find the relevant cell array, find the relevant cell, go look up that cell-type, etc.

Most programs will simply look at the CELL_TYPE frame to get cell information. However, any programs performing placement (of either cell or pins) need cell array information.

Please see the GAIL manual for more discussion on cell arrays.

9.1 Overview

```
/*---------------------- CELL_ARRAY -------------------------*/
/*    +--------------------+   */
/*    |   frame hdr        |   */
/*    +--------------------+   */
/*    |   cell_hdr         |   */
/*    +--------------------+   */
/*    |   x_labl           |   */
/*    |     :              |   */
/*    |     :              |   */
/*    +--------------------+   */
/*    |   y_labl           |   */
/*    |     :              |   */
/*    |     :              |   */
/*    +--------------------+   */
/*    |   cell             |   */   y varies faster in array
/*    |     :              |   */
/*    |     :              |   */
/*    +--------------------+   */
/*    |   x_list           |   */   not present for step_type
/*    |     :              |   */
/*    |     :              |   */
/*    +--------------------+   */
/*    |   y_list           |   */   not present for step_type
/*    |     :              |   */
/*    |     :              |   */
/*    +--------------------+   */
```

9.2 The Generic Frame Header

```
GDB_FRAME_HDR     LITERALLY
    'OPCODE  BYTE,   set to LSH_FRAME_OPCODE.
    HDR_SIZE WORD,   always the same: 16 bytes.
    SIZE     WORD,   byte size of this particular frame.
    ID       WORD,   frame id within the frame-type.
    TYPE     BYTE,   type is GDBF_CELL_ARRAY.
    GDB_RECTANGLE';  enclosing rectangle for the whole cell
                     array.  Calculated and set by BASE
                     according to MIN and MAX (x,y).
                     PLACE doesn't use this.  PIM does use it.
                     LED will search through these rectangles
                     when the user hits PLACE.  It has to find
                     the cell array that the cursor is in.
```

9.3 The Cell Array Header

(GDB_CELL_HDR) It is not 'cell_array_hdr'.

CELL_OPCODE BYTE,
Set to LSH_CELL_ARRAY_OPCODE.

CELL_HDR_SIZE WORD,
size of the Cell Array Header.

GDB_ARRAY_DEF,
This indicates that the GDB_ARRAY_DEF is used, following the Cell Array Header, to specify the labels, the cell-types, and the coordinate lists. It also gives the X and Y counts (tells how many coordinates in the lists).

X_LABL_COUNT WORD,
count of X labels. Notice that, e.g., for a 10 by 10 cell array, there will only be 10 X-labels, not 100.

OFFSET_X_LABL WORD,
offset to the X-label data.

Y_LABL_COUNT WORD,
count of Y labels. Notice that, e.g., for a 10 by 10 cell array, there will only be 10 Y-labels, not 100.

OFFSET_Y_LABL WORD,
offset to the Y -label data.

FLAGS WORD,
Used to indicate which label-part (X or Y) should come first in the concatenation of each label. Corresponds to the options set in GAIL text.

OFFSET_CELL WORD,
offset to the Cell data (to the GDB_CELL_DEF records).

There is not a "Cell-Count" field following the offset. The total cell-count is specified in the GDB_ARRAY_DEF.

9.3.1 Cell Array Label Def.

(GDB_LABEL_DEF)

This structure holds the X and Y part-labels which are later used to label each cell array location (each socket). Both the X-label-list and the Y-label-list use this same structure.

See the GAIL manual for examples of labels and their formats.

LABL (GDB_SHORT_NAME_LENGTH) BYTE,
This is the label. It is left-justified, null-padded, and no length is given for it.

FORMAT BYTE,
a byte to specify the format the label's on-screen display.

Since GATEMASTER concatenates X and Y part-labels together for each cell, the number of GDB_LABEL_DEF structures for each cell array only needs to be X+Y, not X*Y.

9.3.2 Cell Array Cell Def.

(GDB_CELL_DEF)

These structures hold particular cell-array cells and their characteristics.

TYPE BYTE,
Specifes the cell-type of this cell.

FEATURES WORD,
Specifies particular features that the user has assigned to this cell.

COMP_ID WORD,
Tells what component has been placed in the cell, if any. In the whole gdb_cell_def, only this field is written into after GAIL initialization.

LEGAL_ORIENTATIONS BYTE,
Not used. See the orientations discussion in the chapter on data constants.

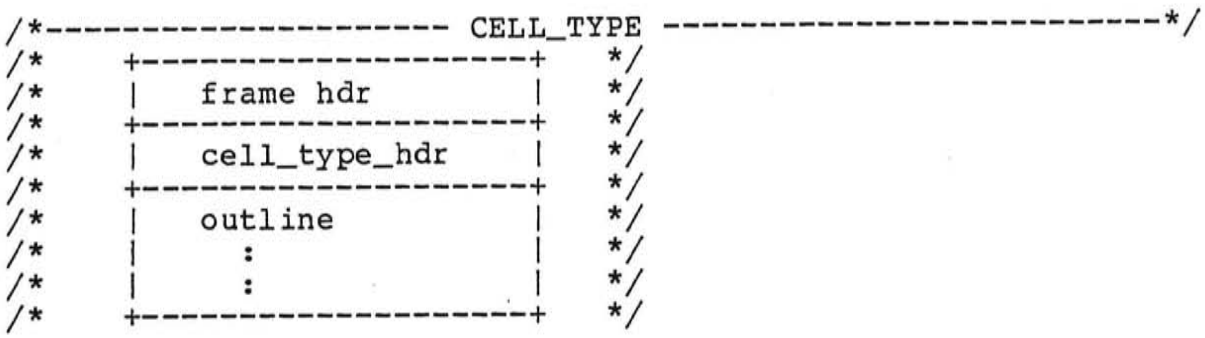
X and Y coordinate lists follow the header if the array type is PROJECTION. In that case, counts and offsets for the X and Y coordinate lists are in the GDB_ARRAY_DEF.

10 The CELL_TYPE Frame

This frame type is used to describe each cell-type and how it relates to particular macros. Every field comes straight from GAIL text. The frame is read-only; nobody writes into it after BASE sets it up.

One surprising thing about usage of cell-type data is that the rough router (RAIN) does not use any of this data. This is because the X and Y blockage data is accounted for by user GAIL text input. BASE must fill in those fields, which eventually affect RAIN's view of things, but RAIN does not presently read cell-type data.

10.1 Overview



10.2 The Generic Frame Header

```
GDB_FRAME_HDR    LITERALLY
   'OPCODE  BYTE,  set to LSH_FRAME_OPCODE.
   HDR_SIZE WORD,  always the same: 16 bytes.
   SIZE     WORD,  byte size of this particular frame.
   ID       WORD,  frame id within the frame-type.
                   (one frame per cell-type).
   TYPE     BYTE,  type is GDBF_CELL_TYPE.
   GDB_RECTANGLE'; not used -- no geometric info.
                   All 4 words = GDB_NIL_WORD.  The LED
                   program looks at cell array rectangles
                   and at the relative_outline def records
                   in the frame below.
```

10.3 The Cell Type Header

(GDB_CELL_TYPE_HDR)

Underpasses and vias come with the cell's macro, not with the cell-type.

CTYPE_OPCODE BYTE,
Set to LSH_CELL_TYPE_OPCODE.

CTYPE_HDR_SIZE WORD,
size of the whole cell-type header.

CTYPE_ATTRIBUTE WORD,
Not used. Just a spare word. This would be named "type", but is called "attribute" to avoid people thinking that this is the cell-type.

PERM_MACRO_ID WORD,
Tells which macro should always go with the cell. BASE sets this field shortly before the end of GAIL text compilation. BASE ensures that the macros receive unique ID's, and that this ID corresponds with the correct macro.

PERM_MACRO_NAME_SIZE WORD,
Size of the name in field below. From GAIL.

PERM_MACRO_NAME(GDB_NAME_LENGTH) BYTE,
Tells the name of the permanent macro. From GAIL. This is not really needed as active data, but BASE needs it for its final pass when it stuffs an ID into the PERM_MACRO_ID field above.

Since permanent macros are always there, LED doesn't care about them. PBM has the only code that looks at this permanent macro data. It has to put obstructions and design rules into the bitmap wherever necessary. PBM reads cells and cell-types, uses the macro ID to get to the macro, and copies macro obstructions into the bitmap.

By contrast with permanent macros, if the base array includes EMPTY macros, LED has got to think about them. They must be unplaced and placed as required for component placement and un-placement.

LED passes this data to BUP to check on, when relevant, though BUP does not know that it is munching on empty-macro data.

EMPTY_MACRO_ID WORD,
This is the frame ID for the macro which goes with the cell-type whenever a cell of this type is empty. LED puts the macro into the cell only when the cell is empty. BASE assigns the frame ID for the macro and puts it here.

EMPTY_MACRO_NAME_SIZE WORD,
size of the name below. Comes from GAIL.

EMPTY_MACRO_NAME(GDB_NAME_LENGTH) BYTE,
Gives the name of the macro that occupies the empty cell (if there is one). Comes from GAIL.

OUTLINE_COUNT WORD,
Count of cell-type outlines.

OFFSET_OUTLINE WORD,
Offset to the outline data.

10.4 The Cell-Type Outline

Cell-type outline data follows the cell-type header. The cell_type_outline is an array of gdb_relative_outline def structures.

LSH uses outline data to paint the cell outline on the screen. LED uses this outline for recognition purposes, not the GDB_RECTANGLE in the frame header.

11 The PROJECTION Frame

This frame holds coordinate data which was input in GAIL. BASE puts the data into the frame. The coordinates come from the GLOBAL_DESCRIPTOR block in the GAIL text. The coordinates are manufacturer's coordinates.

BASE sets up the two PROJECTION frames; the data is read-only thereafter.

BASE reads the .GAIL input and determines which method of coordinate transformation the user has specified (Daisy to manufacturer and vice versa). If the linear method was chosen, the numbers for that go into the database's global descriptor frame. If the projection method was chosen, the numbers are here.

Currently, the number of PROJECTION frames which can exist is zero, one, or two. In the case of two, the X projection frame is distinguished from the Y projection frame by reading the PROJ_TYPE field (listed below).

There are two ways to translate Daisy grid coordinates into the manufacturer's coordinate system: linear and projection. These two methods are fully discussed in the GAIL manual.

If both the X and the Y dimensions are handled by a linear calculation, then no projection frames will exist.

If either dimension is converted to output by use of the projection method, then a projection frame must be used to specify the lists of manufacturer's coordinates which correspond to Daisy grid coordinates.

11.1 Overview

```
/*---------------------- PROJECTION -------------------------*/
/*     +--------------------+    */
/*     |   frame hdr        |    */
/*     +--------------------+    */
/*     |   projection_hdr   |    */
/*     +--------------------+    */
/*     |   projection       |    */
/*     |      :             |    */
/*     |      :             |    */
/*     +--------------------+    */
```

11.2 The Generic Frame Header

```
GDB_FRAME_HDR       LITERALLY
    'OPCODE  BYTE,    set to LSH_FRAME_OPCODE.
    HDR_SIZE WORD,    always the same: 16 bytes.
    SIZE     WORD,    byte size of this particular frame.
    ID       WORD,    frame id within the frame-type.
    TYPE     BYTE,    type is GDBF_PROJECTION.
    GDB_RECTANGLE';   not used -- no geometric info.
                      All 4 words = GDB_NIL_WORD.
```

11.3 The Projection Header

(GDB_PROJECTION_HDR)

PROJ_OPCODE BYTE,
Set to LSH_SKIP_OPCODE.

PROJ_HDR_SIZE WORD,
size of this header.

PROJ_TYPE WORD,
This field tells whether this projection frame is holding x-coordinates or y-coordinates. Both BASE and MAKE use this to distinguish the X and Y coordinate frames. See the Schema Constants chapter for the literals.

PROJ_COUNT WORD,
The count of coordinates in the coordinate list. If this frame is the X-list, the count equals max_x_grid + 1. If this frame is the Y-list, the count equals max_y_grid + 1.

OFFSET_PROJ WORD,
offset to the coordinate data.

11.4 The Projection Def.

(GDB_PROJECTION_DEF)

An array of these structures holds the projection coordinates. "Projection" refers to the process of projecting the Daisy grid system onto the manufacturer's coordinate system.

The projection process involves translating x- or y- grid co-ords to manufacturer's units. The length unit is defined in the global_desc frame.

COORD DWORD;
This is one manufacturer's coordinate. Declared as Dword, but treated as a signed Dword.

Tricky thing: since manufacturer's coordinates can go negative and they can also get very big, this field has to be able to take rather large positive and negative numbers. Therefore, one would like to declare the COORD field as Double Integer. Unfortunately, there is no such data type in PL/M-86.

Therefore, the DWORD type is used, because of its 32-bit size, and the data is interpreted as though it is a big integer. These DWORDS are encoded as 2's complement integers. For example, -1 = ØFFFF FFFFH. There is not a separate sign bit: -1 is NOT equal to Ø8ØØØ ØØØ1H.

This is parallel to the use of signed bytes in the (X,Y) field in the GDB_CELL_LIST_DEF in the MACRO frame.

BASE performs the extra work necessary to convert the GAIL coordinate input into a signed Dword. When converting a signed Dword to outside values or when doing arithmetic with signed Dwords, the programmer must process the signed Dword explicitly and carefully!

12 The OBSTRUCTION Frame

Obstructions come directly from GAIL text (processed by BASE) and they are read-only thereafter. Each OBSTRUCTION block in the GAIL text generates exactly one frame in the database, and each obstruction frame in the database represents just one obstruction array.

The user can have a lot of obstruction arrays. Because of restrictions in GAIL, the user often has to break up a huge obstruction array into a number of smaller arrays. Each obstruction array (each OBSTRUCTION frame in the database) is limited to 5940 total obstruction rectangles when the array has been expanded. See the OBS_COUNT field for more discussion on this limit. Future code will make the limit unnecessary.

Only code which initializes the bitmap will look at obstructions. When LED first generates the .BMP file, it makes a PBM call. A PBM routine puts obstructions into the bitmap file. Even the screen-displayed obstructions are painted from the bitmap data. The Rough Router does not look at OBSTRUCTION frames. It uses the .BMP file to check for obstructions.

Bear in mind that macros have their own data on obstructions: obstructions which come and go with macros. The VISIBILITY command in LED will display all obstructions. All of the following are sources of obstruction data for Visibility:

1. the obstruction frame.
2. permanent and empty macros.
3. obstructions in user-placeable (normal) macros.
4. traces and vias and pins.

12.1 Overview

```
/*---------------------- OBSTRUCTION -------------------------*/
/*     +--------------------+    */
/*     |   frame hdr        |    */
/*     +--------------------+    */
/*     |   obstruction_hdr  |    */
/*     +--------------------+    */
/*     | relative_obstr     |    */
/*     |      :             |    */
/*     |      :             |    */
/*     +--------------------+    */
/*     |   x_list           |    */
/*     |      :             |    */
/*     |      :             |    */
/*     +--------------------+    */
/*     |   y_list           |    */
/*     |      :             |    */
/*     |      :             |    */
```

```
/*    +--------------------+    */
```

12.2 The Generic Frame Header

```
GDB_FRAME_HDR     LITERALLY
    'OPCODE  BYTE,  set to LSH_FRAME_OPCODE.
    HDR_SIZE WORD,  always the same: 16 bytes.
    SIZE     WORD,  byte size of this particular frame.
    ID       WORD,  frame id within the frame-type.
                    Specifies which obstruction frame
                    (and array) this is.
    TYPE     BYTE,  type is GDBF_OBSTRUCTION.
    GDB_RECTANGLE'; enclosing rectangle for the obstruction.
                    array, calculated by the BASE compiler.
                    The rectangle covers all the obstruction
                    areas defined in the array. It does not
                    have to include all the lattice points for
                    the obstructions.
```

12.3 The Obstruction Header

(GDB_OBSTRUCTION_HDR)

Macro frames have additional data for obstructions: their own macro obstructions.

OBS_OPCODE BYTE,
Set to LSH_OBSTRUCTION_OPCODE.

OBS_HDR_SIZE WORD,
size of this obstruction header.

OBS_COUNT WORD,
tells how many obstruction rectangles are to be put down at each lattice point. The obstruction rectangle is defined below. Currently, the BASE compiler checks the expanded count of obstruction lattice points against a size limit. The size limit is as follows: the OBS COUNT * (no. of X lattice points) * (no. of Y lattice points) must be <= 5940. The maximum number of obstructions per array is 5940. Future code will make this limit unnecessary.

OFFSET_OBS WORD,
offset to the obstruction data (to an array of gdb_relative_obstr_def).

GDB_ARRAY_DEF,
This structure gives the lattice of points which position the obstruction arrays. See the Database Conventions chapter for its definition. Also, see the limit on total obstruction lattice points described in the OBS COUNT discussion above. BASE will read the GAIL input, expand the array if the array type is STEP, and store the expanded form of the array here.

SPARE(2) WORD,
spare space.

12.4 The Obstruction Definition

The obstruction definition describes a particular obstruction area (rectangular outline), and its type and layers.

See the GAIL manual for a greater discussion of lattice points and relative rectangles.

12.4.1 Absolute Obstruction Def.

(GDB_OBSTRUCTION_DEF)

This structure is not used. It is only here for backwards compatibility. It MUST match gdb_relative_obstr_def (except word --> integer). The bit patterns for word and integer are only the same for non-negative integers, and the corresponding words with values of half or less than their maximum values (Ø <= x <= 32767). New code should use INTEGER type ONLY.

```
/*  no longer used  */
TYPE            BYTE,
LAYER_MASK      WORD,
GDB_RECTANGLE,
```

12.4.2 Relative Obstruction Def.

(GDB_RELATIVE_OBSTR_DEF)

This structure is used instead of the absolute one. It is also used in the GDB_MACRO_OBSTRUCTION_DEF in the macro frame. If the structure is changed here, that changes it in the macro frame, too.

TYPE BYTE,
Tells whether this obstruction prohibits traces or vias or both.

LAYER_MASK WORD,
Tells whether this obstruction exists on layer one or layer two or both.

GDB_RELATIVE_RECTANGLE_DEF;
Gives the integer coordinates of the obstruction rectangle (in Daisy grid units, relative to each lattice point). The rectangle specifies an area of obstructed grid points.

See the Conventions chapter for a full description of the Relative Rectangle Def.

13 The UNDERPASS Frame

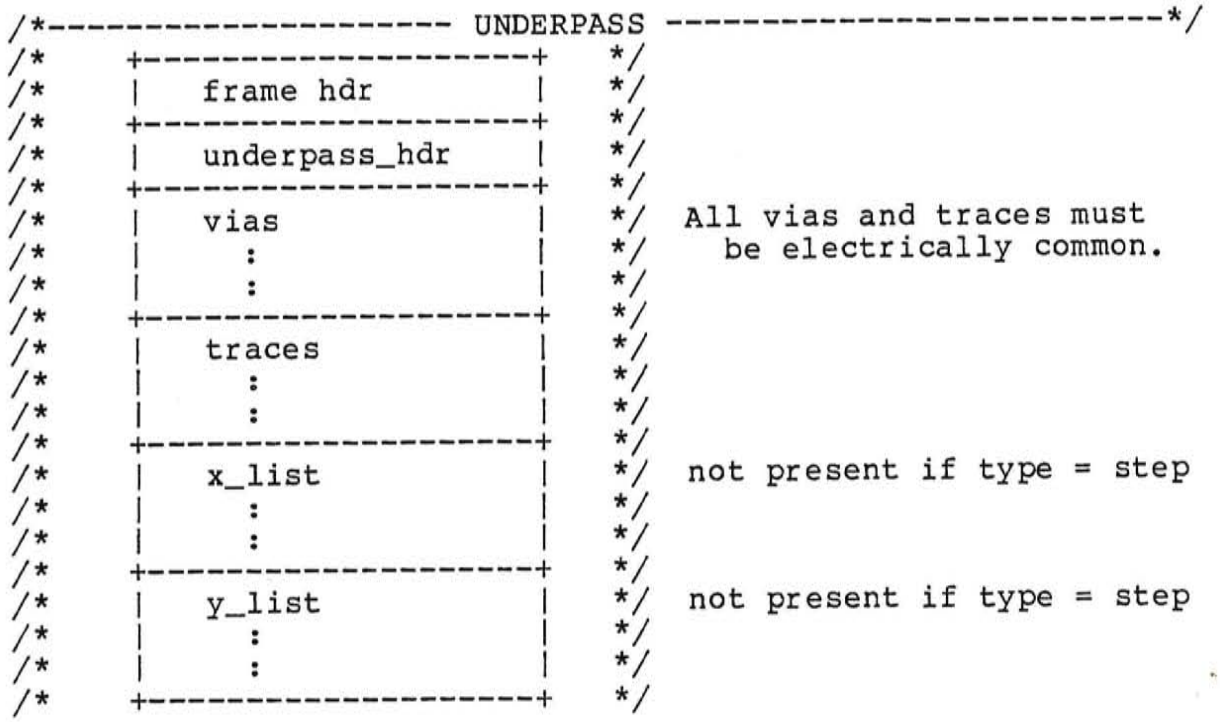
Underpass frames appear in the .BASE files and the .LED file. The underpass coordinate space is much like that for obstructions: there is an X by Y lattice of points, and generic patterns are laid down at each lattice point.

There is one frame for each Underpass array declared in GAIL. All the fields in the frame come straight from GAIL and are read-only thereafter.

The UNDERPASS frame describes its gate array objects in a way similar to the OBSTRUCTION frame. The GDB_ARRAY_DEF in the header describes an array of lattice points where a set of objects will be replicated many times, using X and Y coordinate lists. The coordinate lists define an array of lattice points to be used as origins for a pattern. Via and trace structures define the pattern of wiring to be laid down at each point. Please see the GAIL manual for examples and further discussion.

During construction of the bitmap file, a BUP routine puts underpasses which have been used into that file. When an LED user attempts a component placement, another BUP routine looks at the bitmap and performs two checks on the underpasses: one, it avoids used underpasses; two, it looks for usable underpasses (to touch the macro).

13.1 Overview



13.2 The Generic Frame Header

```
GDB_FRAME_HDR    LITERALLY
    'OPCODE  BYTE,  set to LSH_FRAME_OPCODE.
    HDR_SIZE WORD,  always the same: 16 bytes.
    SIZE     WORD,  the size of this particular frame.
    ID       WORD,  frame id within the frame-type.
                    (uniquely identifies which array).
    TYPE     BYTE,  type is GDBF_UNDERPASS.
    GDB_RECTANGLE'; enclosing rectangle for the underpass
                    array, set by the BASE compiler. The
                    rectangle covers all the underpass traces
                    and vias. It does not have to cover all
                    the lattice points. LED uses the rectangle
                    in the TRACE command the user hits the VIA
                    key. PBM/BUP code does not use this
                    rectangle.
```

13.3 The Underpass Header

(GDB_UNDERPASS_HDR)

UND_OPCODE BYTE,
Set to LSH_UNDERPASS_OPCODE.

UND_HDR_SIZE WORD,
size of the header.

GDB_ARRAY_DEF,
A description of lattice points. Defined in the Database Conventions chapter.

UND_TYPE WORD,
not used.

FLAGS WORD,
not used.

VIA_COUNT WORD,
the number of via records below.

OFFSET_VIA WORD,
offset to the via section.

TRACE_COUNT WORD,
the number of trace records below.

OFFSET_TRACE WORD,
offset to the trace section.

DOWNPAYMENT WORD,
not used. "Downpayment" was supposed to represent how costly it is (to the router) to touch an underpass. However, the router is instead calculating the cost by multiplying the amount of trace and via in the underpass by constants.

SPARE (10) BYTE;
spare space.

13.4 Via and Trace Definitions

Look for the via and trace lists following the Underpass Header. They use the same structure as the macro via and trace definitions.

GDB_MACRO_VIA_DEF structures hold underpass vias.

GDB_MACRO_TRACE_DEF structures hold underpass traces.

The via and trace structures use INTEGER data fields to hold their coordinates. Those coordinates are relative; they must be added to each lattice point. The sum gives the correct set of coordinates for the via and trace locations.

The vias and traces must be electrically common. This means that for a given underpass, there must be a path between any two of these elements.

13.5 X and Y Coordinate Lists

The last items in the frame are X and Y coordinate lists as specified by a GDB_ARRAY_DEF (present only if array type is PROJECTION). Please see the Conventions chapter for the details.

The coordinate lists constitute the grid locations that have a particular underpass. Each underpass trace-via definition is replicated at all the lattice points.

14 The PATHNAME Frame

Each page in the DED schematic has a pathname. The pathname specifies the place of the frame in the design hierarchy, relative to the drawing's root directory.

SLIDE puts one pathname frame into the database for every page that it reads from the circuit drawing. (A component which is moved, in DED, from one page to another is deleted from one page and added to another.)

Each frame holds just one pathname.

The pathname is made of ' {name} / {name} / {name} ', etc. Each of those constituent names must be stored, in order. The series of names is represented as an array. Each name is padded out to GDB_NAME_LENGTH with Ascii nulls. The last name in the series happens to be the terminal file name.

None of this comes from GAIL. All of it is input by SLIDE. There are several programs which must respect these pathnames: PLACE/PIM, QDGET, and MAKE. Currently, PLACE and PIM must care about pathnames because they must create submit files to update the placement in the .LED file. QDGET must respect pathnames, and the future GET program might have to consider pathnames, too. MAKE must send out of the GATEMASTER system the pathname and page-ID information about the original circuit drawing.

Use of the pathname frame: a net frame holds a DED_PAGE_ID. That ID points directly to the relevant pathname frame. For instance, DED_PAGE_ID = 33 means that the full pathname for that net can be found in the pathname frame with ID = 33.

14.1 Overview

```
/*------------------- PATHNAME ------------------------------*/
/*    +--------------------+   */
/*    |   frame hdr        |   */
/*    +--------------------+   */
/*    |   pathname_hdr     |   */
/*    +--------------------+   */
/*    |   pathname_def     |   */
/*    |     :              |   */
/*    |     :              |   */
/*    +--------------------+   */
```

14.2 The Generic Frame Header

```
GDB_FRAME_HDR   LITERALLY
    'OPCODE  BYTE,  set to LSH_FRAME_OPCODE.
    HDR_SIZE WORD,  always the same: 16 bytes.
    SIZE     WORD,  byte size of this particular frame.
    ID       WORD,  frame id within the frame-type.  Each
                    frame of this type holds one pathname.
                    Order of pathnames (order of these frames)
                    is significant!  SLIDE assigns this id,
                    which corresponds exactly to a DED_PAGE_ID
                    that SLIDE received (order of page id's).
    TYPE     BYTE,  type is GDBF_PATHNAME.
    GDB_RECTANGLE'; not used -- no geometric info.
                    All 4 words = GDB_NIL_WORD.
```

14.3 The Pathname Header

(GDB_PATHNAME_HDR)

PATH_OPCODE BYTE,
Set to LSH_SKIP_OPCODE.

PATH_HDR_SIZE WORD,
size of this header.

PATH_TYPE WORD,
Not used. For initialization, SLIDE sets bits for 'valid-info' and 'SLIDE-created'.

NAME_LENGTH WORD,
maximum permissible length of any single name in the pathname list. SLIDE sets this field to GDB_NAME_LENGTH.

PATH_COUNT WORD,
count of the number of names (no. of GDB_PATHNAME_DEF's) which form the one pathname in this frame. (Tells how many names to look for below).

OFFSET_PATH WORD,
offset to the actual pathname data.

LAST_SING_ID WORD,
Carries the last (updated) ID of this drawing page from the SLIDE.SING file for correlations and MAKE output.

SPARE (1Ø) BYTE,
spare space.

Discussion: the SLIDE.SING file (which is produced by DML) describes all the nets and components. However, it doesn't give full pathnames every time. The SLIDE.SING file has a table at its top which specifies pathnames, but in the body of the file, only DED page id's are given (because the numbers are much more compact.) If GM-10 could be assured of the same DML page ID for each drawing page every time the user ran SING, things would be simple. However, for sake of independence, GM-10 must have its own system of correlations.

SLIDE takes a pathname in the SLIDE.SING file and compares it to every pathname in the existing database. If the pathname is not there, SLIDE adds it to the database. It assigns the frame ID without taking into account the page id assigned by SING, and also stores that frame id in the net frame in the field called DED_PAGE_ID. Since SLIDE assigns id's in the order of receiving the pages and VIRGA gives unused frame ID's to SLIDE in the order of unused pathname id's, corresponding id's may often be exactly the same.

The unique frame id assignment means that every unique page gets a unique DED_PAGE_ID which serves to distinguish the net names. Thus, DED_PAGE_ID is a piece of data assigned by GM-10, and it may differ (if DED pages have been erased) from the page ID's that are in the current SLIDE.SING file.

When SLIDE performs updates to a .LED file, it searches the arrays of names, looking for a pathname which matches a given DED pathname. It then updates both the pathname in the .LED file and the DED_PAGE_ID in the net frame.

MAKE must create output which has page id's that match the latest LOGICIAN circuit drawing. Therefore, SLIDE stores the LAST_SING_ID so that MAKE can print it out.

Discussion of the DED_PAGE_ID in the net frame:

Initial Problem: Multiple pages in the DED schematic may have the same name used for different nets. Another page in the schema might have a net named the same.

This field tells the ID of the page that the name came from. The SLIDE.SING file provides the pathname from the root of the design (@ prefix). However, this pathname is so large and awkward that usually you only want to deal with the ID. The Pathname frame holds a table with the full names.

Tricky things:

1. A net need not be on just one page of the schematic. Often nets are on multiple pages. SLIDE must distinguish all nets from each other. For correctness of MAKE output, the PAGE_ID must be carried through exactly. However, within GM-1Ø, the PAGE_ID is just a piece of data that helps us uniquely label the nets that come into the system.

A DML model decides what page the net will fall on. Method: The very first time the DML processor hits the net, that's the page assigned. Therefore, this will be the most upper-left page in the design tree which contains part of the net.

2. If the user goes back and changes a net, then the page that the net is "on" (assigned to) may end up being different. The net will be assigned to the NEW top-left-most page. SLIDE won't know that it is the same net, since SLIDE pays attention to the page id. SLIDE thinks it is a newly added net. This problem is also created when a DED user simply changes the name of a net to something else. SLIDE will think that one net was deleted, and another one was added.

GM-1Ø does NOT want to go through the database and find all the occurences of the old page id. This is a mess and is not easily handled. The end result is that it is unfriendly to the user, since the page id assignment might change, and the layout process will be strained. The user will have to route some nets more than once.

3. A Tricky thing that IS solved in GM1Ø code: what if a whole page is deleted? A new page id will be assigned to lots of nets. For example, if the second page is deleted, the third page will actually be the second page, and so on. In this case, SLIDE might think that the many nets must be deleted and many others must be added.

However, in the pathname frame, this problem is avoided by keeping the correlation between pathnames and frame id's. Therefore, when a pathname changes, the database compensates for this by showing simply a new way of reaching the same page id's. Remember that this frame, the net frame, has only the net's page id. But you can go to the pathname frame, and find the real page pathname.

SLIDE, in incremental update mode, can see the discrepancy between the new pathnames (in the SLIDE.SING file) and the old ones (in the database's pathname frame), and therefore SLIDE can modify the pathname frame in order to reflect the new names. However, it would be too complicated (take too long, and be error-prone) to go through the entire database and change all page id's based on the pathname change.

You would have to iteratively go through and set bits on page

id's while performing the change, indicating whether it was the old one or the new one (e.g., old id 3 or new id 3). Since this is just too crazy, GM1Ø handles it by holding, in the pathname frame, 3 pieces of information that come from SLIDE: pathname, database page id, and the page id coming from SLIDE.

14.4 The Pathname Def.

(GDB_PATHNAME_DEF)

This structure is the basis for an array of names, all of which taken together spell out a pathname.

SIZE WORD,
Gives the size of the name in the NAME field following. Tells how many characters to expect in the name.

NAME (GDB_NAME_LENGTH) BYTE;
This is one name in the pathname.

The pathname structure must be understood: the pathname "@A/B/C 1.DRAW" that SLIDE gets from SING will be held in one pathname frame as "/A/B/C 1.DRAW". SLIDE changes the leading atsign to a slash, breaks the pathname at slash boundaries, and stores the slash, except for the last one.

The Pathname Def. will hold an array of 4 size/name pairs:

(1)	2	/A
(2)	2	/B
(3)	2	/C
(4)	6	1.DRAW

All records except the last will hold names of directory files. The last record will hold the name of a terminal file.

15 The BLOCK Frame

This frame was part of the initial design, but it has not been used yet. It waits for the day when customers are savvy enough to want a group of 4 or 5 component's macros to be moved as a group. Thus, the LED user would indicate a block of components which would serve as a standard component placement that could be used again and again. Each BLOCK frame would hold a user-defined block of macros.

Thus, this frame was designed as a hook into one more level of hierarchy for gate array layout. Essentially, components (macro-classes) and macros offer a little bit of hierarchical view already. The idea was to add another layer: one which would reflect chip-planning. Blocks might correspond to shift-register, I/O buffer, control, etc. However, the CHIPMASTER product is already pursuing hierarchical functionality, so this frame might never be implemented.

15.1 Overview

```
/*--------------------- BLOCK ------------------------------*/
/*    +--------------------+   */
/*    |   frame hdr        |   */
/*    +--------------------+   */
/*    |   block_hdr        |   */
/*    +--------------------+   */
/*    |   outline          |   */
/*    |     :              |   */
/*    |     :              |   */
/*    +--------------------+   */
```

15.2 The Generic Frame Header

```
GDB_FRAME_HDR   LITERALLY
  'OPCODE  BYTE,  set to LSH_FRAME_OPCODE.
  HDR_SIZE WORD,  always the same: 16 bytes.
  SIZE     WORD,  byte size of this particular frame.
  ID       WORD,  frame id within the frame-type.
  TYPE     BYTE,  type is GDBF_BLOCK.
  GDB_RECTANGLE'; this would hold geometric coordinates,
                  if the frame were used.
```

15.3 The Block Header

[Not implemented.]

BLK_OPCODE BYTE,
Set to LSH_BLOCK_OPCODE.

BLK_HDR_SIZE WORD,
size of this header.

GDB_RELATIVE_TEXT_DEF,
Holds the block name and name format.

BLOCK_TYPE WORD, Sixteen bits.

```
NAME_X2             INTEGER,
NAME_Y2             INTEGER,
NAME_FORMAT2        BYTE,
NOTE_ID             WORD,
OUTLINE_COUNT       WORD,
OFFSET_OUTLINE      WORD,
```

15.4 The Block Outline

[Not implemented.]

The block outline would be an array of gdb_outline def.

16 The NOTE Frame

This frame-type holds arbitrary text associated with graphics.

The frame is created dynamically by the LED program and passed to LSH so that it can paint the strings of text on the screen. For example, if LED wants a set of cell names to blink, LED can pass a note frame (which has lots of text and coordinates), and a "blink-it" message to LSH, which will revise the graphic display.

No 'NOTE' information can be found in a .LED disk file. There are no NOTE frames in the database. However, for LED's convenience, the frame's literals are compiled as part of the GM-1Ø system, just as all the GDSCH schema literals are.

Essentially, all the note information is stored in a pile of records that contain (size-of-string,text-string) pairs. Any particular string must be found by serial read, not array access.

16.1 Overview

```
/*---------------------- NOTE --------------------------------*/
/*    +---------------------+   */
/*    |   frame hdr         |   */
/*    +---------------------+   */
/*    |   note_hdr          |   */
/*    +---------------------+   */
/*    |   line_def & text   |   */  variable-length records
/*    |     :               |   */
/*    |     :               |   */
/*    +---------------------+   */
```

16.2 The Generic Frame Header

```
GDB_FRAME_HDR       LITERALLY
    'OPCODE  BYTE,  set to LSH_FRAME_OPCODE.
    HDR_SIZE WORD,  always the same: 16 bytes.
    SIZE     WORD,  byte size of this particular frame.
    ID       WORD,  frame id within the frame-type.
    TYPE     BYTE,  type is GDBF_NOTE.
    GDB_RECTANGLE'; enclosing rectangle for the notes.
                    Used by LSH.
```

16.3 The NOTE Header

(GDB_NOTE_HDR)

NOTE_OPCODE BYTE,
Set to LSH_NOTE_OPCODE.

NOTE_HDR_SIZE WORD,
the size of this header.

LINE_COUNT WORD,
the number of note lines which can be found below.

OFFSET_LINE WORD,
offset from the frame header to the start of the note data.

16.4 The Line Definition

(GDB_LINE_DEF)

This structure is an array of lines of data. Each one-line note gets a GDB_LINE_DEF.

X WORD,
Y WORD,
The grid coordinates of the locating point.

FORMAT BYTE,
Tells how the note should be positioned, relative to the locating point. Also, how the display should look.

LENGTH WORD;
Tells the length of the text string which follows.

Each line_def is followed immediately by its text; enough bytes are allocated in frame for the text, but no more. To get to line_def(n), you must step through first (n-1) line_defs, incrementing the pointer by (size(line def)+line_def.length) for each of those (n-1) line_defs.

17 The TEXT Frame

This frame-type holds large globs of arbitrary text not associated with graphics. The data is never interpreted by GM-1Ø code. The .LED file simply carries the pile of text, in one frame, from the GM-1Ø front end to the back end (MAKE & MKFIX).

The TEXT frame is a quick and dirty GM-1Ø way of getting around the current lack of soft (user-defined) parameters.

This frame is very similar to the note frame in that it holds lines of text. However,

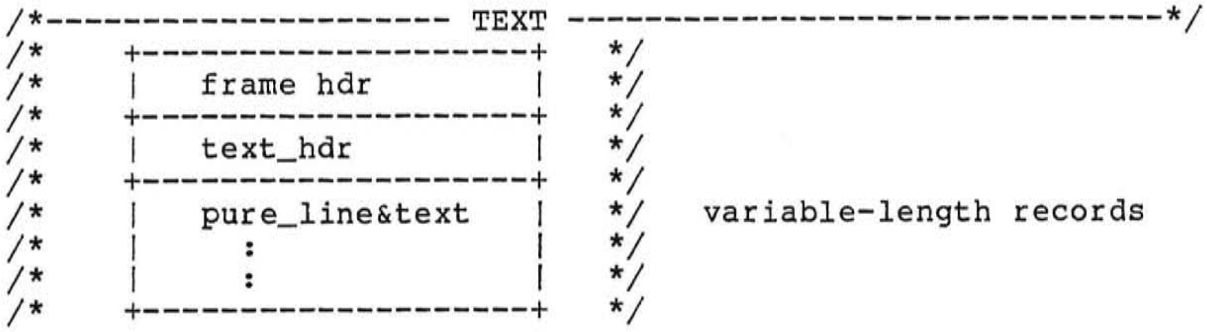
1. This frame holds lines of non-graphic text. Because it is non-graphic, the frame does not have coordinate and 'format' data fields.

2. TEXT frames actually are held in the database (unlike the NOTE frame) since the main point of these frames is to hold lots of .SING text input for MAKE output.

The text data comes to the GM-1Ø system from DED or DML. The data presumably spells out some parameter information, but the GM-1Ø system leaves it uninterpreted. SLIDE puts it into the database. MAKE and MKFIX are the only programs which should look at it (should output the text).

This frame might hold parameter information for thousands of nets. E.g., MAX_CAPACITANCE (= maximum value for a net). DML outputs text which is behavioral language for the desired parameters. The text needs to just sit there inertly until kicked out by MAKE.

17.1 Overview



17.2 The Generic Frame Header

```
GDB_FRAME_HDR     LITERALLY
    'OPCODE  BYTE,   set to LSH_FRAME_OPCODE.
    HDR_SIZE WORD,   always the same: 16 bytes.
    SIZE     WORD,   byte size of this particular frame.
                     Enough bytes are allocated in the
                     frame for the text, but no more.
    ID       WORD,   frame id within the frame-type.
                     Some databases have no text frames.
    TYPE     BYTE,   type is GDBF_TEXT.
    GDB_RECTANGLE'; not used -- no geometric info.
                     All 4 words = GDB_NIL_WORD.
```

17.3 The Text Header

(GDB_TEXT_HDR)

TEXT_OPCODE BYTE,
Set to LSH_SKIP_OPCODE.

TEXT_HDR_SIZE WORD,
size of this header.

TEXT_TYPE WORD,
No longer used. The type of TEXT frame is indicated by the NAME field below.

FLAGS WORD,
SLIDE sets a bit for 'SLIDE_created_text'.

PURE_LINE_COUNT WORD,
count of globs of text.

OFFSET_PURE_LINE WORD,
offset to the start of the line_def data.

NAME_SIZE WORD,
actual size of the name following. E.g., "15", for the example following.

NAME (GDB_NAME_LENGTH) BYTE,
Here is a name which the user knows about; it classifies the text string below. It might be a parameter name. E.g., "MAX_CAPACITANCE".

SPARE (1Ø) BYTE, spare space.

17.4 The Pure Line Def.

(GDB_PURE_LINE_DEF)

An array of these structures holds the textual data: only one line of data per record. A record consists of a length/text pair.

LENGTH WORD;
length of the piece of text which follows this WORD.

Each length word is followed immediately by its text.

To get to pure_line_def(n), you must step through first (n-1) pure_line_defs, incrementing the pointer by (size(pure_line_def) + pure_line_def.length) for each of those (n-1) pure_line_defs.

Enough bytes are allocated in the frame for the text, but no more.

18 The DELTA Frame

This frame does not come from GAIL input. Frames of this type are created and maintained solely by SLIDE and by DELTA_LIST code in LED. (These two are intimately linked.)

The delta-list data and the pin-swap data taken together represent the differences (at any given moment) between the netlist as described in the .LED file and the netlist as given by the SLIDE.SING file. Delta-list records will appear in the DELTA menu in LED (when the user runs LOAD_SET) so that the user can clear up the discrepancies.

If SLIDE discovers that a new component is in the SLIDE.SING file, and therefore it needs to be added to the .LED file, SLIDE will simply add the component (or pin) to the .LED file. A delta-list entry is only necessary when something needs to be deleted by the user: a component or pin or net.

The user's goal in LED interaction is to clear the Delta List. The user must un-place all components and erase all nets listed in the netlist.

Also, if the user tries create an output file before the Delta List has been cleared, MAKE or MKFIX will send a warning to the error port and will refuse to run. Output to the manufacturer must not be allowed while the gate array layout is incomplete or incorrect.

Pin-swap code never creates DELTA frames since it uses PIN_SWAP frames to record the discrepancies it creates.

Organization of information:

Each delta-list frame describes a little, atomic difference:

E.g., "component X1's pin A is to be removed from net S".

and, "component X1's pin B is to be added to net T".

The pair of records means that a DED user altered the wiring to one of the pins on component X1. These are only two types of delta frames. SLIDE can create five types. They are listed below in the DELTA_TYPE field.

The LED user must intelligently alter the gate array layout in order to eliminate the delta list. This corrective action will bring the .LED file netlist into exact correspondence with the latest circuit drawing in the LOGICIAN subsystem.

Examples:

If an old component which must be deleted is already placed, SLIDE adds a delta-list frame which indicates that the user must unplace the component. When the user un-placed the component, the LOAD_SET code will automatically delete the unplaced, obsolete component from the .LED file and update the DELTA menu if it was already loaded.

If a net has two pins on it and it must be deleted, SLIDE adds two delta-list frames which require that the user un-place the two components with those pins, and adds other delta frames which specify net routing to be removed. When the user removes the components and the routing, LED code will delete each pin from the net and delete the net record. Delta-list code will delete the Delta-List records.

If a pin must be moved from one net to another, SLIDE will create two delta-list frames: one for a pin-deletion (from the old net), one for a pin-addition (to the new net). After the user removes routing (if necessary), Delta-List code will carry out the removal and addition operations in that order.

18.1 Overview

```
/*--------------------- DELTA -------------------------------*/
/*    +--------------------+   */
/*    |   frame hdr        |   */
/*    +--------------------+   */
/*    |   delta_hdr        |   */
/*    +--------------------+   */
```

18.2 The Generic Frame Header

```
GDB_FRAME_HDR    LITERALLY
    'OPCODE  BYTE,  set to LSH_FRAME_OPCODE.
    HDR_SIZE WORD,  always the same: 16 bytes.
    SIZE     WORD,  byte size of this particular frame.
    ID       WORD,  frame id within the frame type.
    TYPE     BYTE,  type is GDBF_DELTA.
    GDB_RECTANGLE'; not used -- no geometric info.
                    All 4 words = GDB_NIL_WORD.
```

18.3 The Delta Header

(GDB_DELTA_HDR)

Fields similar to the ones below are used in PIN_SWAP.

DELTA_OPCODE BYTE,
Set to LSH_SKIP_OPCODE.

DELTA_HDR_SIZE WORD,
size of this header.

DELTA_TYPE BYTE,
Tells the type of atomic difference (what action to take). E.g., deleting a component, deleting a net, or deleting a pin. Then, the data below tells you specifically what gate array element needs to be modified. The particular change is specified by a literal; look in GDSCC to see the exact literal. The valid types are: deleted_component, added_net_pin, deleted_net_pin, deleted_net, and unplace_component.

FLAGS WORD,
Not used. 16 bits for future bit-encoding.

COMP_ID WORD,
the Virga frame-ID of the component which is affected. Is NIL if not a component operation.

PIN_ID WORD,
the ID of the affected component pin. Delta-list code must hunt for the matching pin ID in the COMPONENT_PIN_DEF structure. This field is set to NIL if the frame is for a component or net deletion.

OLD_NET_ID WORD,
the ID of the net from which a pin must be deleted.

This field is only used if the frame's delta-type is 'delete pin'. If delta-type is any other type, this field must be NIL.

NEW_NET_ID WORD,
the ID of the net which gets a new pin.

This field is only used if the frame's delta-type is 'add pin'. If delta-type is any other type, this field must be NIL.

FEATURES WORD,
this word holds a bit pattern. The bit pattern is the new set of features that a component in a revised schematic

requires.

There is one particularly obscure type of DELTA frame which must be specially documented: 'unplace_component'. It is a special case (somewhat different from the other four delta types).

SLIDE generates a DELTA_LIST frame of type 'unplace_component' when a component instance in the SLIDE.SING file requires more features than are present in that instance in the .LED file, and that component has been placed in a location which does not have those features.

The critical difference that SLIDE detects is new features on the component in the SLIDE.SING file. The source of the new features in the LOGICIAN system, which end up in the SLIDE.SING file, might be user action in CED, DED parameters, or DML, or a wholesale update of the component library.

The new features might be in just one component instance, or in every instance of that macro-class. In any case, SLIDE must create a DELTA_LIST frame for every component instance which requires review.

Since the incoming component has new features on it, existing placements may now be incorrect. Therefore, those components must be unplaced and then placed again. During the new placement, the component with new features will be checked against the user-selected cell as usual.

Because of this need for review of the correctness of placement, SLIDE will generate DELTA_LIST frames which require that such components be removed. This requires the user to remove the components and then place them again. The whole point is that during the new placement, the component's (new) features will be checked against the cell for correctness.

Only a component instance which requires more features will stimulate SLIDE to add a delta frame. A component which needs fewer features will simply be updated in its COMPONENT frame. Since the component's placements in the layout do not require further checking, no DELTA_LIST frame is needed.

SLIDE does not check anything on macros; BASE deals with discrepancies between macros.

SPARE (2) WORD;
spare space.

19 The PIN_SWAP Frame

Like the DELTA frame, this frame does not come from GAIL input. It is only created by code for the SWAP-PIN command in LED (except that the pin swap is intimately linked to the Delta List).

[The functionality in the following paragraph is not yet implemented.] Some of these frames may be deleted by the DELTA_LIST code. If the user swaps pins, and then swaps them back later, the delta-list code must be smart enough not to add sets of PIN_SWAP frames which will simply cancel each other out. DELTA_LIST handling code will look for contradictory (opposite) PIN_SWAP frames, and delete them in order to kill off some of the pin swaps that will be necessary.

DELTA_LIST frames hold information about GM-10 changes made necessary by changes in the schematic. Delta-list stuff just says, "change the .LED file". PIN_SWAP frames also hold information about discrepancies between schematic and the .LED file, but the discrepancies were caused by the user in LED. You don't want PIN_SWAP information in the DELTA LIST frame because it represents changes ALREADY PERFORMED by the LED user. However, the pin-swap information must be held ready for SLIDE to look at, since SLIDE must get the information that, "YES, these pins and nets are really supposed to be different from the LOGICIAN schematic information".

Therefore, all discrepancies between the LOGICIAN schematic and the GATEMASTER layout will be represented in the union of the DELTA_LIST data and the PIN_SWAP data.

Fundamentally, a pin swap is an interchange of pins between two or more nets on the same component. Pin swaps are possible because a component may often possess pins which are logically equivalent. A user may desire a pin swap because of congestion in the layout. The congestion might be relieved if two pins could be interchanged such that new routing could be in parallel.

Each pin swap involves two pins. The full pin-swap operation requires at most four PIN_SWAP frames (when both pins were being used already). If only one pin is on a net (not two), it requires only two PIN_SWAP frames to record the change. The most complicated case, where a pin which is being swapped has already been moved in a previous swap, may require three PIN_SWAP frames.

At least one pin must be on a net. Otherwise, the user should just be routing as desired (if both pins are free).

The Pin Swap code will carry out two operations when the pins

are exchanged in LED:

1. change the actual pin records in the COMPONENT PIN_DEF in the COMPONENT frame, and the NET frame records.

2. add records to the PIN_SWAP frame to record the exact additions and deletions that were carried out, so that SLIDE knows the discrepancies from a pin swap. The existence of PIN_SWAP records also indicates that the user needs to make changes to the DED schematic.

DELTA_LIST frames will record the changes that need to be made to the layout; PIN_SWAP frames will record changes that need to be made to the schematic.

If the user goes back into DED and swaps the pins there, the next SLIDE run will find that some records in the PIN_SWAP frame are no longer necessary. Therefore, since the schematic and the layout are in correspondence, it will delete the obsolete PIN_SWAP records. Handling engineering changes is one of GM-1Ø's best features.

MAKE does not look at the PIN_SWAP frame. It just outputs the database as received. GET [not yet implemented], when reading a .GET file, will generate PIN_SWAP and DELTA records if it detects any pin swaps performed by the outside software.

19.1 Overview

```
/*---------------------- PIN SWAP ----------------------------*/
/*     +--------------------+    */
/*     |   frame hdr        |    */
/*     +--------------------+    */
/*     |   pin_swap_hdr     |    */
/*     +--------------------+    */
```

19.2 The Generic Frame Header

```
GDB_FRAME_HDR     LITERALLY
    'OPCODE  BYTE,  set to LSH_FRAME_OPCODE.
    HDR_SIZE WORD,  always the same: 16 bytes.
    SIZE     WORD,  byte size of this particular frame.
    ID       WORD,  frame id within the frame-type.
                    (There are many of these frames.)
    TYPE     BYTE,  type is GDBF_PIN_SWAP.
    GDB_RECTANGLE'; not used -- no geometric info.
                    All 4 words = GDB_NIL_WORD.
```

19.3 The Pin Swap Header

(GDB_PIN_SWAP_HDR)

Most of these fields are extremely similar to those in the Delta-List frame.

PIN_SWAP_OPCODE BYTE,
Set to LSH_SKIP_OPCODE.

PIN_SWAP_HDR_SIZE WORD,
the size of the Pin Swap Header.

PIN_SWAP_TYPE BYTE,
Tells the type of atomic change, required by a pin swap, which has already been carried out in the .LED file. The constants from GDSCC are:
GDBC_SWAP_LED_DELETED_NET_PIN
GDBC_SWAP_LED_ADDED_NET_PIN

FLAGS WORD,
Not used. 16 bits for future bit-encoding.

COMP_ID WORD,
this is the Virga ID of a COMPONENT frame which has been changed in LED, but whose component needs to be changed in DED.

PIN_ID WORD,
the id of the pin in the COMPONENT_PIN_DEF in the component frame specified above.

OLD_NET_ID WORD,
tells the ID of the net in this .LED file where the pin used to be; the component-pin on the indicated net needs to be removed in DED. If the field equals GDB_NIL_WORD, the pin was formerly unconnected to any net.

NEW_NET_ID WORD,
tells the ID of the net in this .LED file where the pin is now (after the swap); the component-pin on the indicated net needs to be added in DED. If the field equals GDB_NIL_WORD, disconnect the pin from all routing.

SPARE (3) WORD;
spare space.

20 The PARAM Frame

This frame is not currently used, but it might be used in the future. All of this frame and the PARAM_TABLE frame will be subject to further design review.

Param frames are NOT set up by GAIL. When parameters are implemented, SLIDE will create the PARAM frames, initialize them, and set up the correct values.

The frame would only exist in a .LED file, since individual components and nets are the entities which would have PARAM frames associated with them.

There might also be one global PARAM frame. One PARAM frame can hold parameters assigned to the whole chip. The GLOBAL_DESC frame already has a PARAM_ID field in the Global Header. [However, current SLIDE and LED cannot read or assign a global parameter.] That global PARAM_ID, when parameters are implemented, and if the user inputs a global parameter, would point to the parameter frame holding the relevant data.

The basic idea behind this frame is that it would be helpful to have a third way of handling parameters, besides the current TEXT frame (one big pile of data), and hard-coded parameters (for example, CRITICALITY has its own niche in the schema). The third way can be called "soft" parameters. The advantage of soft parameters is that they would be accessible by the code (like the hard parameters), yet the parameter type can be defined by the user (as in the TEXT frames).

Suppose the LED user wants to put in a component parameter which LED does not recognize (thus, it is probably not a standard, commonly-used parameter). When code is written for the PARAM frame, it will allow the user to assign parameters to objects even when that parameter type is not known by GATEMASTER. And, that data will be binary-readable for fast access, rather than being an arbitrary Ascii mess as in the TEXT frame. Application programs will be able to skip from item to item more rapidly. By contrast to PARAM frames, TEXT frames are just carried through to MAKE output. They are not interpreted by GM-10 code.

At present, the user may put soft data into the NOTE and TEXT frames, but this data is not so effectively linked with a gate array object. In the PARAM frame, every parameter is owned by an object.

Another advantage of this frame is that all soft parameters corresponding to the same object can be lumped together within the same parameter frame, rather than just dumped in a pile in the TEXT frame. Thus, soft parameters might be organized for read and write.

Question: parameters can be found in the DED schematic, in the SLIDE.SING file, and in the .LED file. How do these inter-relate? Answer: SLIDE would put all SLIDE.SING parameter values into .LED unchanged, except for Criticality (divides it by 10).

DED parameters will be carried into the SLIDE.SING file and SLIDE will pick them up and put them into the .LED file. The problem is, how to handle the difference between hard and soft parameters? See the Design Spec by Yosi Kliger on Soft Parameters for more discussion.

20.1 Overview

```
/*---------------------- PARAM -------------------------------*/
/*    +--------------------+   */
/*    |   frame hdr        |   */
/*    +--------------------+   */
/*    |   param_hdr        |   */
/*    +--------------------+   */
/*    |   param            |   */  params are NOT array:
/*    |      :             |   */  they are variable-length.
/*    |      :             |   */
/*    +--------------------+   */
```

20.2 The Generic Frame Header

```
GDB_FRAME_HDR    LITERALLY
    'OPCODE  BYTE,  set to LSH_FRAME_OPCODE.
    HDR_SIZE WORD,  always the same: 16 bytes.
    SIZE     WORD,  byte size of this particular frame.
    ID       WORD,  frame id within the frame-type.
                    This is the "PARAM_ID" which the component
                    frame header has in it.  It allows the
                    component to index this particular PARAM
                    frame.
    TYPE     BYTE,  type is GDBF_PARAM.
    GDB_RECTANGLE'; not used -- no geometric info.
                    All 4 words = GDB_NIL_WORD.
```

20.3 The Param Header

(GDB_PARAM_HDR)

[This frame is not actually implemented.]

PARAM_OPCODE BYTE,
Set to LSH_SKIP_OPCODE.

PARAM_HDR_SIZE WORD,
size of this header.

PARAM_TYPE BYTE,
Not used. Available.

OWNER_TYPE BYTE,
tells whether the parameter belongs to a component or net.

OWNER_ID WORD,
tells which net or component owns this parameter. (Backward pointer). This can only work if the field above disambiguates the frame-type of the owner: i.e, net frame 13 versus component frame 13.

PREVIOUS PARAM_ID WORD,
works with next_param_id, below. This is a frame ID; it points back to a previous frame, for which this frame is only a continuation.

This field would only be needed if there were so many parameters that not all of the ones assigned to a gate array object (net or component) could be held in one parameter frame. (!)

NEXT_PARAM_ID WORD,
works with previous_param_id, above. This is a frame ID; it points forward to the next parameter frame needed to continue listing the many parameters assigned to one object. This field is not likely to be used - see the caveat above.

PARAM_COUNT WORD,
count of parameters in the data section. Parameters are of varying length. The PARAM_SIZE field must be read successively to skip through the parameter records.

OFFSET_PARAM WORD,
offset from the frame header to the start of the data.

SPARE (2) WORD,
spare space.

20.4 The Param Def.

(GDB_PARAM_DEF)

PARAM_SIZE WORD,
the size of the parameter value below. The PARAM_SIZE field must be read successively to skip through the parameter records.

TYPE WORD,
not used. spare for bit-encoding, if needed.

PIN_ID WORD,
this id specifies which pin the parameter is on. This is NIL (all FF's) if the parameter belongs to the component itself or to a net.

PARAM_NUMBER WORD,
this number tells which kind of parameter this is. E.g., this is parameter type "3". The param-table gives the real name of the parameter type.

PARAM_VALUE_LENGTH WORD,
the length of the VALUE array below.

PARAM_VALUE (1) BYTE,
this is the actual parameter value! (in Ascii). E.g., "1234.678", or "1010". It can be as long as needed since the field is a variable-length string. Take caution: the PARAM_VALUE_LENGTH field above is measured as WORD.

21 The PARAM_TABLE Frame

This frame is not read or written into, but it might be used in the future. All of this frame and the PARAM frame may be subject to further design review.

Presently, the TEXT frame holds uninterpreted parameter data.

When used, there will be just one PARAM_TABLE frame. It will be in the .LED file.

The PARAM_TABLE is intended to tell programs more about the kinds (types) of parameters held in the PARAM frames. In particular, it is a table of user-input types of (soft) parameters. However, it can also hold information on hard parameters (parameter-types known to GM-1Ø). In particular, this frame can tell SLIDE which parameters are hard (known) parameters.

The idea behind this frame is to take all the code that deals with hard parameters and make it table-driven. value. (see OWNER_OFFSET field in PARAM_TABLE_DEF below. When this is implemented, BASE will initialize the table. SLIDE would update it.

A PARAM frame would have values associated with a parameter, e.g., number 3. The PARAM_TABLE would then give the textual name of the parameter type. You can tell from the table that parameter 3 contains the MAX_CAPACITANCE value allowed on a net. The name string is held in the table.

See the discussion above, in the PARAM frame.

This table means that a program going through the parameters in the PARAM frame would not have to perform text comparisons to see what parameters were related. Furthermore, the param-table tells the data-type of the parameter value: is the parameter binary or Ascii or BCD, etc.

There will be just one PARAM_TABLE frame. SLIDE will have to put in one record for every new kind of parameter that it reads from the SLIDE.SING file.

Access: SLIDE would initialize this frame, and update it when necessary. MAKE would read it and print it out. Conceivably, utility routines written for customers would read some of it.

Organization of information:

(1) all hard-parameter information goes into four kinds of data areas: component, net, pin, and pin-block. A hard parameter can not just float independently; it must be associated with one of these object types.

Thus, hard-parameter information is sorted by owner type. This helps access and prevents the possible problem of two or more hard-parameter names being assigned which are identical globally, but not within one object type.

(2) all soft-parameter information goes into the last data area: the soft-parameter chunk of data (unsorted).

Soft parameters in the PARAM_TABLE frame would help SLIDE read the parameter block in the SLIDE.SING file. SLIDE could know what types of parameters it has already seen and assigned a number to, so that it doesn't have to compare text or perform continous look-ups.

21.1 Overview

```
/*---------------------- PARAM_TABLE--------------------------*/
/*     +---------------------+    */
/*     |   frame hdr         |    */
/*     +---------------------+    */
/*     |   param_tbl_hdr     |    */
/*     +---------------------+    */
/*     |   param_tbl_def     |    */
/*     |       :             |    */
/*     |       :             |    */
/*     +---------------------+    */
```

21.2 The Generic Frame Header

```
GDB_FRAME_HDR    LITERALLY
    'OPCODE  BYTE,  set to LSH_FRAME_OPCODE.
    HDR_SIZE WORD,  always the same: 16 bytes.
    SIZE     WORD,  byte size of this particular frame.
    ID       WORD,  frame id within the frame-type.
                    (Just one frame, at present.)
    TYPE     BYTE,  type is GDBF_PARAM_TABLE.
    GDB_RECTANGLE'; not used -- no geometric info.
                    All 4 words = GDB_NIL_WORD.
```

21.3 The Param Table Header

(GDB_PARAM_TABLE_HDR) [None of this is implemented.]

PARAM_TABLE_OPCODE BYTE,
Set to LSH_SKIP_OPCODE.

PARAM_TABLE_HDR_SIZE WORD,
size of this header.

PARAM_TABLE_FLAGS WORD,
Not used.

COMP_COUNT WORD,
count of parameters associated with components.

OFFSET_COMP WORD,
offset to the component-parameter information.

NET_COUNT WORD,
count of parameters associated with nets.

OFFSET_NET WORD,
offset to the net-parameter information.

PIN COUNT WORD,
count of parameters associated with pins.

OFFSET_PIN WORD,
offset to pin-parameter information.

PIN_BLOCK_COUNT WORD,
count of parameters associated with pin-blocks.

OFFSET_PIN_BLOCK WORD,
offset to parameter data associated with pin-blocks.

SOFT_PARA_COUNT WORD,
count of soft parameters.

OFFSET_SOFT_PARA WORD,
offset to soft-parameter information.

LAST_NUM_USED WORD,
last parameter number used. This allows SLIDE to know immediately what parameter number should be assigned to a new type of parameter.

SPARE (10) WORD,
spare space.

21.4 The Param Table Def.

(GDB_PARAM_TABLE_DEF)

NAME_LENGTH WORD,
length of the following name.

NAME (GDB_NAME_LENGTH) BYTE,
This is the textual name of the parameter. E.g., "MAX_CAPACITANCE".

DATA_TYPE BYTE,
Tells the type of data to expect in the parameter-value array. E.g., byte, word, integer, dword, or string. This lets a program know how to read, interpret, and print out a parameter's value.

TYPE WORD,
16 bits for bit-encoding, if needed.

DEFAULT_FLAG BYTE,
A bit here would say if there is a default value or not.

DEFAULT_VALUE DWORD,
This is the default value for a hard parameter if it is needed somewhere but nobody assigned an actual value to it. The value can be very big if necessary.

OWNER_OFFSET WORD,
The offset from the top of a structure DEF down to the data item of interest. The owner offset is only valid if the relevant structure is not a pin_param_block. This field is only applicable for hard parameters (already hard-coded in the GM-1Ø schema).

The offset allows a program reading a hard parameter value (whether default or assigned) to rapidly write the value into the appropriate place in the .LED file. (This is dangerous! This field supposes that someone knows to change this if another frame is changed.)

PARAM_NUMBER WORD,
This number correlates the records here with the Param Def. records in the PARAM frame. There is only one parameter number for each different type of parameter. Thus, there probably won't be very many of these.

The field only makes sense for soft parameters. If you know a parameter's number and want to find its table entry, look for the matching number here.

22 The DRV Frame

This is the frame to record Design Rule Violations. It is not currently implemented. No GM-1Ø code has to deal with it yet.

For R&D purposes, there is a program which will print out the information which would otherwise be going into this frame. This is for Daisy-internal debugging purposes only. (The program only writes to an Ascii disk file. It tells what kind of violation occurred at what coordinate location.)

Current GATEMASTER design and claims are based on the rule that no design rule violations are allowed in the database. Thus, at present, the database may always be assumed to be correct: none of its data will create an illegal chip layout. So long as BUP code can do its job perfectly, there will be no design rule violations in the database. The bitmap is constructed directly from .LED information, with no further checking. It is assumed that BUP code already got to it once, upon data entry.

However, with the complexity of GATEMASTER increasing, particularly with the start of batch-mode channel routing, it will not be smart to rely on the perfection of BUP design-rule checking code. Bitmap data will probably need to be checked as it is read in the construction of the bitmap files. We need to set the corrupt flag in the database so that MAKE can check on it, and refuse to create output if there are violations.

But why the DRV frame? The reason is that we want to be able to log certain d.r.v.'s without shutting down the whole layout system. The DRV frame will allow GM-1Ø code to write down a design rule violation, set a global "database-corrupt" flag, and exit gracefully.

There may be a lot of DRV frames. There will be one for every component involved in a design rule violation, and maybe one for each net involved in the d.r.v. There could be multiple d.r.v.'s per object.

It's obvious how a net may have design rule violations, since the traces and vias in the net may conflict with any one of lots of gate array objects. However, d.r.v.'s associated with a component may be more mysterious. Yet, there are many ways that a placed component could cause design rule violations: the macro traces could conflict with something else, such as a neighboring macro's traces or user-routing; the placement might put objects on obstructed areas; or, the placement could be on top of another component or part of a component.

Another reason for the DRV frame is the need to find violations between different nets during batch input. If net A has already been added to the database, and the BUP code finds

a conflict in adding a net B, then it will add net B, but also add a record to the DRV frame which indicates the illegality of net B.

Notice that charging the d.r.v. to B is an arbitrary act. Even though the user may delete net A or net B, the DRV frame will not be changed until the bitmap is re-initialized. At that time, the DRV frame and the bitmap file are both wiped clean, and the BUP code will review all of the layout for correctness. BUP will then fill the DRV frame with a fresh set of here-and-now design rule violations.

Who accesses this kind of frame? The frame is set and cleared only by Bitmap initialization code. When building the bitmap in Checking Mode, it will erase all the entries in the frame, and then re-create entries (log any violations) as needed while it builds the new bitmap.

Presumably, LED (or LSH) will be enhanced so that the user can see DRV's displayed on-screen in reverse video. Also, the user would like DRV report. A program could write to a log file, and a user command would report the DRV log.

22.1 Overview

```
/*------------- DRV ----- (DESIGN_RULE_VIOLATION) ----------*/
/*  +----------------+  */
/*  |   frame_hdr    |  */
/*  +----------------+  */
/*  |   drv_hdr      |  */
/*  +----------------+  */
/*  |   net/comp id  |  */
/*  +----------------+  */
/*  | drv grid points|  */
/*  |       :        |  */
/*  +----------------   */
```

22.2 The Generic Frame Header

```
GDB_FRAME_HDR    LITERALLY
    'OPCODE  BYTE,  set to LSH_FRAME_OPCODE.
    HDR_SIZE WORD,  always the same: 16 bytes.
    SIZE     WORD,  byte size of this particular frame.
    ID       WORD,  frame id within the frame-type.
                    One frame is created for each
                    component or net which has Design
                    Rule Violations.
    TYPE     BYTE,  type is GDBF_DRV.
    GDB_RECTANGLE'; not used -- no geometric info.
                    All 4 words = GDB_NIL_WORD.
```

22.3 The DRV Header

(GDB_DRV_HDR)

DRV_OPCODE BYTE,
Set to LSH_SKIP_OPCODE.

DRV_HDR_SIZE WORD,
the size of this header.

DRV_TYPE WORD,
16 bits for future bit encoding.

DRV_COMP_ID WORD,
gives the ID of the component which caused the design rule violation (points to a particular COMPONENT frame). This field is set to NIL if a net caused the d.r.v.

DRV_NET_ID WORD,
gives the ID of the net which caused the design rule violation (points to a particular NET frame). Is set to NIL if a component caused the d.r.v.

POINTS_COUNT WORD,
count of POINTS_DEF structures below (equivalent to the number of points involved in the design rule violation.)

OFFSET_POINTS WORD,
offset from the frame header to the POINTS_DEF data.

DRV_MACRO_ID WORD,
specifies which macro is involved in the design rule violation, if any.
(This data item is not entirely necessary, since you could use the DRV_COMP_ID to go into the offending Component frame, and find there the ID of the macro that was used to place the component.)

22.4 The DRV Points Def.

(GDB_POINTS_DEF)

An array of these structures holds all the Daisy grid points which comprise one design rule violation. The number of points per violation may be one, or many.

For instance, if a trace is placed that is too close to another (parallel) trace (or on top of it), that will produce a lot of violation points for the one incorrect placement or routing.

LAYER WORD,
On which layer(s) did the d.r.v. occur? This field tells all.

DRV_X WORD,
The X coordinate (daisy grid) of one point in the violation set.

DRV_Y WORD;
The Y coordinate (daisy grid) of one point in the violation set.

23 The BITMAP_DESC Frame

There is only one of these frames (currently).

This is a descriptor of what the bitmap looks like. This frame only tells you where the bitmaps are on the chip, and what the cost of moving through them is. This frame does not contain any of the actual bitmap data such as obstructions. Real bitmap data (data on each gridpoint) is in the BITMAP frame in the .BMP file. It's handled by BUP code.

PBM builds detail bitmap rectangles according to the bitmap descriptor it finds here. PBM does not use only this frame; it also looks in the GLOBAL frame to get layer information which affects vias and underpasses.

The frame encodes how the chip is divided up into bitmap rectangles and where the rectangles are. All its information comes from GAIL's BITMAP frame.

The same bitmap data is always stored in two different ways. One way is the X and Y lists in the GDB_ARRAY_DEF which appears in this frame right after the Bitmap-Descriptor Header. This "two-dimensional" array format comes more or less immediately from the GAIL text, since GAIL requires that bitmap rectangles be specified that way.

The Step form of gdb_array_def should never be found in the bitmap_desc frame. If the BASE compiler finds STEP in its GAIL input, it converts the input array into the Projection type of gdb_array_def. It puts the grid-point array into the database in an expanded form.

The other way to represent bitmaps (the fully expanded form) is also in this frame, in the BITMAP_DESC_DEF. That's an array of BITMAP_DESC_DEF structures. Each record is is simply a bitmap rectangle ID and a GDB_RECTANGLE. The id tells which bitmap, and the rectangle outline is given explicitly. This is a lower level (more detailed) level of description, therefore, than the list method. We call this the flattened bitmap description, since it is a list of rectangles rather than being a dual array of rectangle coordinate lists, as in the GDB_ARRAY_DEF.

Originally, the "flat" BITMAP_DESC_DEF method was used, since at that time, the bitmap rectangles did not need to be regular, though they did need to be rectangles and needed to fill the chip. However, to simplify GAIL encoding and to speed BASE processing and later searches, a more regular structure was adopted. The previous flexibility in bitmap rectangles does not seem to be necessary.

At the present time, each of the two areas which give the

BITMAP data is used by some amount of code. Some old PBM/BUP code still expects bitmap data to be represented the old way: as a flattened list. Therefore, both of the data structures must remain in this frame for a while. BASE, when it reads the GAIL data, will set up only the GDB_ARRAY_DEF set of information. (This is the way that it should be over the long-term.) Later, when PBM code tries to find the bitmap data in the BITMAP_DESC_DEF, it will probably find nothing (unless this is a very old database). When PBM finds nothing there, it goes ahead and transforms the data from the GDB_ARRAY_DEF section, puts it into the BITMAP_DESC_DEF the old-fashioned way, and then prepares the bitmap file. Then, when old LED code is called, LED will find the flattened data that it wants.

Essentially, then, this is a read-only frame. The BITMAP frame gets its information later than the input from BASE, but its data is exactly equivalent.

All code which deals with detail bitmaps will read this frame: d.r.v.-checkers, net-route, and PBM code. When LED needs to paint bitmap outlines in the Detail View, it compares data in this frame with pin and rough-trace data in the NET frame in order to paint only those detail bitmaps which are touched by rough layout.

Also, the code which paints obstructions on screen will look in here to get the bitmap description, and then go to .BMP for the obstruction data.

23.1 Overview

```
/*-------------------- BITMAP_DESC ------------------------*/
/*   +--------------------+   */
/*   |   frame hdr        |   */
/*   +--------------------+   */
/*   |   bitmap_desc_hdr  |   */  item hdr has delta costs
/*   |                    |   */
/*   |    (delta costs)   |   */
/*   +--------------------+   */
/*   |   x_list           |   */
/*   |     :              |   */   X and Y lists are
/*   +--------------------+   */    always present.
/*   |   y_list           |   */
/*   |     :              |   */
/*   +--------------------+   */
/*   |  bitmap_desc_def   |   */
/*   |     :              |   */
/*   +--------------------+   */
```

23.2 The Generic Frame Header

```
GDB_FRAME_HDR    LITERALLY
    'OPCODE  BYTE,  set to LSH_FRAME_OPCODE.
    HDR_SIZE WORD,  always the same: 16 bytes.
    SIZE     WORD,  byte size of this particular frame.
    ID       WORD,  frame id within the frame-type.
                    (Just one frame, at present.)
    TYPE     BYTE,  type is GDBF_BITMAP_DESC.
    GDB_RECTANGLE'; enclosing rectangle of the bitmap,
                    since more than one Bitmap_Desc
                    array might be allowed in the future.
                    Currently, this is not very useful.
```

23.3 The Bitmap Descriptor Header

(GDB_BITMAP_HDR)

BITMAP_OPCODE BYTE,
Set to LSH_OBSTRUCTION_BMP_OPCODE.

BITMAP_HDR_SIZE WORD,
size of this Bitmap Descriptor Header.

GRID_SIZE BYTE,
bytes per grid-point entry in bitmap (per layer). Currently, BASE sets it to four (bytes per layer). GAR code reads it and checks against the GAR system data constant. If they are not equal, GAR quits.

DESC_COUNT WORD,
count of BITMAP_DESC_DEF structures below. (same as number of bitmap rectangles.)

OFFSET_DESC WORD,
offset to the Bitmap_Desc_Def list of rectangle data.

BITMAP_MOD_SIZE WORD,
Not used. Was intended to be the maximum (default) module size of the bitmaps. Instead, BASE checks that the total number of grid points in a bitmap, including the halo grids, is less than or equal to 2030. Thus, PBM receives frames that are less than 8KB. However, PBM checks checks the size of the frames it receives, too.

SPARE (10) BYTE,
spare space.

GDB_ARRAY_DEF and GDB_BITMAP_DELTA_COSTS_DEF are parts of the header, but are described below.

23.4 The Coordinate Lists

(GDB_ARRAY_DEF)

Here's the array def. It has the count and offset pairs necessary to get the arrays of X and Y coordinate information. The Gdb_array_def literal is discussed in the Conventions chapter.

Only the projection type of array is used in this frame since the array of lattice points here is always stored in expanded form, not using the step method.

23.5 The Bitmap Delta Costs Def.

(GDB_BITMAP_DELTA_COSTS_DEF)

This structure can handle up to three layers.

The Delta Costs Def. gives the cost of any particular routing direction. The four (N,S,E,W) directions refer to directions on the chip. UP and DOWN give the cost (how expensive is it?) of moving either up or down between layers.

Each direction gets an array of (3) bytes because GM-1Ø might eventually have to handle three layers.

```
NORTH_DELTA_COST (3) BYTE,
SOUTH_DELTA_COST (3) BYTE,
EAST_DELTA_COST  (3) BYTE,
WEST_DELTA_COST  (3) BYTE,
UP_DELTA_COST    (3) BYTE,
DOWN_DELTA_COST  (3) BYTE,
```

Currently, the GAIL user has the freedom to declare different costs between two opposite directions: i.e., N/S, E/W, Up/Down. However, there is no sensible reason to do so since the detail net router may start a trace (for example, a North-South trace) from either of the two pins.

Currently, GATEMASTER only offers two routing layers. Therefore, only two of the bytes are used for each directional cost. The first byte gives the cost of moving in that direction on layer 1; the second byte gives the cost of moving in that direction on layer 2.

23.6 The Bitmap Descriptor Def.

(GDB_BITMAP_DESC_DEF)

The Bitmap descriptor holds a flattened representation of the bitmap (a list of rectangles). The flattened list will become obsolete when all of the PBM/BUP code has been updated so that PBM/BUP uses the Gdb_Array_Def coordinate lists. You can see the coordinate lists higher up in the frame.

ID WORD,
The ID of a bitmap rectangle. (tells which bitmap). These records must start with ID = Ø and proceed consecutively through ID = (NUM_BITMAPS -1).

GDB_RECTANGLE;
Where the rectangle is on the chip, in Daisy grid coordinates. X1 must be <= X2 and Y1 must be <= Y2. Gdb_rectangle is defined in the Conventions chapter.

Notice that when this flattened-list data structure becomes obsolete and gets removed from the .LED schema, the list of IDs will go away too. This means that programs reading the bitmaps must maintain an implicit mapping of read-order onto ID's. PBM/BUP/DROUTE starts scanning at lower left, Y varies faster.

24 The ROUGH Frames

The ROUGH_ frames hold most of the data relating to the rough router. The NET and GLOBAL_DESC frames are others which have relevant data. After the Rough Router creates the .RAIN file, it will modify and maintain that file also.

The CELL_ROWS and CHANNELS frames contain Rough View routing information which is similar in structure to the ROUGH_DATA frames. However, data in the CELL_ROWS and CHANNELS frames is only for the batch auto-router.

BAR code uses the ROUGH frames to define global cells for the Batch Rough Router. However, BAR constructs its routing cells using references to lower-left boundaries, unlike the upper-right boundaries the Rough Router uses!

A future release of BASE will allow the GAIL user to specify cell rows and channels on the chip. Thus, some duplicate input might occur, between the BITMAP frame input and the ROW and CHANNEL input.

The BITMAP information in GAIL is not enough to let a program derive the cell-row and channel information, but possibly, a program could do the reverse: derive the detail and rough bitmaps from the GAIL cell-channel and cell-row information put in by the user. Presently, though, it is easier to put in the information separately. There may also be the advantage of flexibility if the row and channel data is input explicitly.

The detail router could handle non-regular structures, but the BASE compiler currently makes the user stick to a lattice structure, and coordinate projection. BASE takes the bitmap data from the GAIL BITMAP frame, and puts the same lattice information into both the ROUGH_PROJECTION frame and the BITMAP_DESC_DEF array.

The "rough data" in the following frames describes the rough bitmap (for the rough routing program). The detail bitmap (for detail routing) is described in the BITMAP frame in the .BMP file. Just as the ROUGH_PROJECTION and ROUGH_GRID_CELL frames, taken together, define the rough bitmap, the BITMAP_DESC frame together with the .BMP file defines the detail bitmap.

All rough data in the three ROUGH_ frames is initialized by BASE with data from GAIL input. The Rough Router copies the ROUGH_GRID_CELL frames into the .RAIN file. Only those frame copies are modified by the Rough Router. The ones in the .LED file are read-only.

Remember that if a user in the rough view invokes a routing command, only the rough router operates. If a user in the detail view invokes a routing command, the rough router may be

called first, and then the detail router operates.

24.1 Overview

The ROUGH_DATA module holds three frame-types. It is not a frame in itself.

```
/*---------------------- ROUGH_DATA -------------------------*/
    /* module types */
    GDBM_ROUGH_MISC          LITERALLY 'GDBM_ROUGH_DATA',
    GDBM_ROUGH_PROJECTION    LITERALLY 'GDBM_ROUGH_DATA',
    GDBM_ROUGH_GRID_CELL     LITERALLY 'GDBM_ROUGH_DATA',

    /* frame types */
    GDBF_ROUGH_MISC          LITERALLY 'GDB_FIRST_FRAME+0004',
    GDBF_ROUGH_PROJECTION    LITERALLY 'GDB_FIRST_FRAME+0005',
    GDBF_ROUGH_GRID_CELL     LITERALLY 'GDB_FIRST_FRAME+0006',
```

Each of the three frame types is 'contained in' one module type.

25 The ROUGH MISC Frame

There are two tricky things about the data structures for the rough router. First, PLM does not conveniently allow for two dimensional arrays, so a one dimensional array (in the ROUGH_GRID_CELLS frame) is used. Second, there is a lot of Rough Bitmap data here, and the whole array might not fit in one frame. Thus, the design incorporates an accounting system that helps the user find a desired rough grid cell.

In particular, this ROUGH_MISC frame serves to correlate the coordinate information in ROUGH_PROJECTION with detailed information in the ROUGH_GRID_CELLS frame.

There is one procedure in the Rough Router code which, given a request for rough bitmap data, reads this frame and gets the requested data out of the mess of the other two kinds of frames: ROUGH_GRID_CELL and ROUGH_PROJECTION.

The procedure gets an (x,y) pair in rough-grid-cells (not Daisy grids), and it returns a pointer to the correct GDB_ROUGH GRID_CELL_DEF structure in a ROUGH_GRID_CELL frame. Thus, the procedure figures out what GRID_CELL frame to go into, and where to go in that frame's array of records.

The code locks all the ROUGH_DATA modules and frames into memory with VIRGAS's help, and then dispenses with VIRGA thereafter. All the pointers are maintained in an array, for quick access.

The frame has a header only, since it serves to point into the other two ROUGH_ frames.

25.1 Overview

```
/*--------------------- ROUGH MISC -------------------------*/
   /*  +--------------------------+   */
   /*  +   frame hdr              +   */
   /*  +--------------------------+   */
   /*  +   rough misc hdr         +   */
   /*  +--------------------------+   */
```

25.2 The Generic Frame Header

```
GDB_FRAME_HDR    LITERALLY
    'OPCODE  BYTE,  set to LSH_FRAME_OPCODE.
    HDR_SIZE WORD,  always the same: 16 bytes.
    SIZE     WORD,  byte size of this particular frame.
    ID       WORD,  frame id within the frame-type.
                    The ID ought to be zero, but it
                    shouldn't matter.  No code should
                    be reading it, since there is only
                    one frame to look at!
                    (Get the first frame of this type.)
    TYPE     BYTE,  type is GDBF_ROUGH_MISC.
    GDB_RECTANGLE'; not used --  no geometric info.
                    All 4 words = GDB_NIL_WORD.
```

25.3 The Rough Miscellaneous Header

This header is the only significant data in this frame. The data gives information about the other two kinds of frames which have data on the rough bitmap.

The BASE compiler sets up all the fields. Currently, BASE can put all the rows of rough grid cells into one frame only. This means that BASE is limited in the size of Rough Bitmap it can accept. In the future, BASE will incorporate the same frame accounting system that the Rough Router code already has so that multiple-frame Rough Bitmaps can be handled.

ROUGH MISC_OPCODE BYTE,
Set to LSH_SKIP_OPCODE.

ROUGH_MISC_HDR_SIZE WORD,
size of this header.

NUM_ROWS_PER_FRAME WORD,
tells the number of grid-cell rows put into each grid_cell frame. Arbitrarily, the Rough Router's primary direction is horizontal (sweeps across in bands). This is a regular number: it's the same for every grid_cell frame except the last one. The Rough Router rules over the assignment of grid cells to frames. [BASE currently puts all the rows into one frame.]

NUM_GRID_CELL_FRAMES WORD,
tells the total number of ROUGH_GRID_CELL frames.

NUM_GRID_CELLS_PER_FRAME WORD,
Not used. Supposed to tell the number of grid cells included in each grid cell frame. Constant across all such frames except the last one. Determined by the Rough Router. Approximately 5600 grid cells can fit in one

rough_grid_cell frame.

PREFERRED_DIRECTION WORD,
Not used.

STATUS WORD,
16 bits for future bit-encoding.

The last four fields are not used:

```
DISTANCE_COEFF_1     WORD,
DISTANCE_COEFF_2     WORD,
CAP_UTL_COEFF_1      WORD,
CAP_UTL_COEFF_2      WORD,
```

26 The ROUGH_PROJECTION Frame

This frame holds the X and Y coordinate lists which describe where the Rough Routing rectangles will fall on the chip. Each coordinate list determines the grid-lines which "chop" the chip into rough routing areas. By contrast, it is the ROUGH_GRID_CELL frames which hold the actual data about each rough bitmap rectangle's capacity and utilization.

There is one frame for the X coordinates and one frame for the Y coordinates. Those coordinates are entered in GAIL text, and are input here by BASE. One tricky thing is that we are in discrete geometry: thus, a grid-line which divides rectangles must end up in one routing rectangle, not in two.

The GAIL user determines the rough bitmap with coordinates which specify lower-left hand corners of bitmap rectangles. HOWEVER, coordinates in this frame have been translated, so as to feed into the Rough Router code. Thus, coordinates here specify the upper right corners of bitmap rectangles (each rectangle is to the lower left of a point.) This re-arrangement helps the Rough Router find the edges of the chip as it scans right and up.

The Rough Router knows about the left and lower edges of the rough bitmap area (the whole chip), but those edges are not represented in the database. The router acts as if they were set to Ø (zero) capacity. This trick effectively hardens those boundaries, since no routing is permitted over a boundary with zero capacity.

26.1 Overview

```
/*---------------------- ROUGH PROJECTION ------------------*/
   /*  +--------------------------+   */
   /*  +   frame hdr              +   */
   /*  +--------------------------+   */
   /*  +   rough projection hdr   +   */
   /*  +--------------------------+   */
   /*  +   rough projection       +   */
   /*  +     :                    +   */
   /*  +--------------------------+   */
```

26.2 The Generic Frame Header

```
GDB_FRAME_HDR       LITERALLY
    'OPCODE  BYTE,    Set to LSH_FRAME_OPCODE.
    HDR_SIZE WORD,    always the same: 16 bytes.
    SIZE     WORD,    byte size of this particular frame.
    ID       WORD,    frame id within the frame-type.
                      (two frames only: one each
                       for X and Y lists.)
    TYPE     BYTE,    type is GDBF_ROUGH_PROJECTION.
    GDB_RECTANGLE';   Not used.  All 4 words = GDB_NIL_WORD.
```

26.3 The Rough Projection Header

(GDB_ROUGH_PROJECTION_HDR)

ROUGH_PROJ_OPCODE BYTE,
Set to LSH_SKIP_OPCODE.

ROUGH_PROJ_HDR_SIZE WORD,
size of this header.

X_OR_Y BYTE,
tells whether this frame is an X or Y coordinate list.
RAIN (the Rough Router) uses this.

PROJ_COUNT WORD,
count of projection coordinates in the list.

OFFSET_PROJ WORD,
offset from the frame header to the coordinate list.

26.4 The Rough Projection Def.

(GDB_ROUGH_PROJECTION_DEF)

There's an array of these structures.

The choice of scanning direction was an arbitrary design decision. Now, however, it is hard-coded: scanning direction is right and up, X varies faster.

Given the currently hard-coded direction, the Rough Router labels the lower-left chip corner as (Ø,Ø). The upper-right corner point is then very useful since it tells the Rough Router when it has reached the end-of-chip.

BOUNDARY_COORD WORD,
Tells one coordinate (Daisy grid) for the right-upper corner of a Rough Grid cell. Two of these are needed for each such corner point. The point is part of the cell.

CENTER_LINE WORD,
This is a Daisy grid coordinate (absolute) which specifies the center of the cell. The Rough Router uses the center-line to center its routing in a rough grid cell. The center-line is not necessarily in the exact center of the bitmap rectangle. The GAIL user inputs this data.

BOUNDARY_CROSS_DISTANCE BYTE;
Represents the grid distance between adjacent centerlines (right or up from this cell). It is derived (calculated programatically). It tells the distance from the centerline specified in the field above to the centerline of the rectangle either right or up from this one. The field is used but it is not really necessary since the cross distance could be calculated from the data above.

27 The ROUGH_GRID_CELL Frame

This frame holds the capacity and utilization data for all the rough bitmap rectangles (cells). By contrast, it is the ROUGH_PROJECTION frame which tells how this whole system of rough routing cells relates to the actual chip coordinates.

Each grid cell is just one element in an array of cells. Each cell has capacity and utilization data. However, to the detail router, each of these cells is a chunk of chip which is full of points.

27.1 Overview

```
/*----------------- ROUGH GRID CELL -----------------------*/
   /*  +--------------------------+   */
   /*  +   frame hdr              +   */
   /*  +--------------------------+   */
   /*  +   rough grid cell hdr    +   */
   /*  +--------------------------+   */
   /*  +   rough grid cell        +   */
   /*  +     :                    +   */
   /*  +     :                    +   */
   /*  +--------------------------+   */
```

27.2 The Generic Frame Header

```
GDB_FRAME_HDR   LITERALLY
   'OPCODE  BYTE,     Set to LSH_FRAME_OPCODE.
   HDR_SIZE WORD,     always the same: 16 bytes.
   SIZE     WORD,     byte size of this particular frame.
   ID       WORD,     frame id within the frame-type.
                      (Will be as many frames as needed
                       to cram the grid cell data into.
                       At present, that means only one.)
   TYPE     BYTE,     type is GDBF_ROUGH_GRID_CELL.
   GDB_RECTANGLE';    not used.  All 4 words = GDB_NIL_WORD.
```

27.3 The Rough Grid Cell Header

(GDB_ROUGH_GRID_CELL_HDR)

ROUGH_GRID_OPCODE BYTE,
Set to LSH_SKIP_OPCODE.

ROUGH_GRID_HDR_SIZE WORD,
size of this header.

GRID_CELL_FRAME_NUM WORD,
No longer used. BASE sets it to zero. The Rough Router now uses the frame id.

GRID_CELL_COUNT WORD,
Count of grid-cells in the array in this frame. At present, approximately 5600 cells can fit in one frame. The rough bitmap for a current AMI chip requires only about 1000 rough grid cells.

OFFSET_GRID_CELL WORD,
offset from the frame header to the data area.

27.4 The Grid Cell Def.

(GDB_ROUGH_GRID_CELL_DEF)

There is an array of these structures. Each record holds the data for one grid cell.

The order of the list corresponds to the Rough Router's scanning order: start lower-left, X varies faster.

RIGHT_CAPACITY BYTE,
capacity of the cell's right boundary, in traces. Set by BASE from GAIL input.

UP_CAPACITY BYTE,
capacity of the cell's upper boundary, in traces. Set by BASE from GAIL input.

RIGHT_UTILIZATION BYTE,
Tells how many of the cell's right boundary tracks have been used up. Updated by the Rough Router.

UP_UTILIZATION BYTE,
Tells how many of the cell's upper boundary tracks have been used up. Updated by the Rough Router.

More explanation on capacities and utilization inputs can be found in the GAIL manual.

The next three fields are just scratch fields which are used dynamically by the Rough Router code. They don't take up much space, and the whole array of scratch space (corresponding to each cell) is a convenient structure for the Router to use in its wave-front expansion and processing.

COUNTER WORD,
Temporary count of how far the wavefront expansion has made it from the originator pin. The units of this 'count' are grids, more or less.

SUBNET INTEGER,
Temporary, tells which wavefront (from which pin or pins) is being tracked.

FLAGS_Ø BYTE,
Temporary flags on pins in the routing algorithm. E.g., marks a pin as either source or target for wavefront expansion. Also marks a piece of routing as originator so as to distinguish it from a zero in the COUNTER field.

28 The PLACE Frame

See the GAIL manual to start understanding this, and read the GATEMASTER manual (chapter on Automatic Placers and Routers) if you need the nitty-gritty details.

BASE reads GAIL text and initializes this frame. PLACE is the only program to access this frame thereafter.

Two copies of this frame will appear in the .LED file. One is read-only, after being set by BASE to correspond to the GAIL inputs. The other one can be modified by PLACE during each PLACE session, so that the user can establish a common parameter environment for a set of placement runs on the same .LED file.

28.1 Overview

```
/*------------------------- PLACE --------------------------*/
/*  +----------------+  */
/*  |   frame_hdr    |  */
/*  +----------------+  */
/*  |   place_hdr    |  */
/*  +----------------+  */
/*  |  class_strings |  */
/*  |       :        |  */
/*  +----------------+  */
```

28.2 The Generic Frame Header

```
GDB_FRAME_HDR       LITERALLY
    'OPCODE  BYTE,    set to LSH_FRAME_OPCODE.
    HDR_SIZE WORD,    always the same: 16 bytes.
    SIZE     WORD,    byte size of this particular frame.
    ID       WORD,    frame id within the frame-type.
    TYPE     BYTE,    type is GDBF_AUTO_PLACER.
    GDB_RECTANGLE';   not used -- no geometric info.
                      All 4 words = GDB_NIL_WORD.
```

28.3 The Place Frame Header

(GDB_PLACE_HDR)

For information on the parameters below, see the file, PLACE.ELT, or the PLACE Design Specification or the GATEMASTER Manual, or the GAIL manual.

```
    PLACE_OPCODE BYTE,
    Set to LSH_SKIP_OPCODE.

    PLACE_HDR_SIZE WORD,
    size of this header.


    GAIN                    REAL,
    DSMAX1                  REAL,
    DSMAX2                  REAL,
    FADE_CYCLE              WORD,
    FADEX                   REAL,

    FCI_X                   REAL,
    FCI_Y                   REAL,
    FCR_X                   REAL,
    FCR_Y                   REAL,
    FCE_X                   REAL,
    FCE_Y                   REAL,
    FCG_X                   REAL,

    FCG_Y                   REAL,
    FCIO_X                  REAL,
    FCIO_Y                  REAL,

    FBACK_X                 REAL,
    FBACK_Y                 REAL,
    DREP                    REAL,
    MODE                    WORD,
    CLASS_STRING_COUNT      WORD,
    OFFSET_CLASS_STRING     WORD,
    SPARE(20)               BYTE,
      /* plan ahead */
```

28.4 The Class String Def.

(GDB_CLASS_STRING_DEF)

Each record in this structure consists of a class-string-def and a glob of text following. Thus, the records are variable in length. You have to read LENGTH fields and hop over records to move through the pile.

Each class string is stored like text in the TEXT frames. Namely, a word-valued LENGTH field indicates the number of bytes in the following string. You must step through the first (N-1) strings to get to the Nth string.

FLAGS WORD,
flags.

SPARE WORD,
spare space.

LENGTH WORD;
the length (bytes) of the glob of text which follows. The records are variable length!

A glob of text follows each CLASS_STRING_DEF (completes each record).

29 The CELL_ROW_DESC Frame

There is just one of these frames.

29.1 Overview

```
/*-------------- CELL_ROW_DESC -----------------------------*/
/*    +--------------------+   */
/*    |   frame hdr        |   */
/*    +--------------------+   */
/*    |   cell_row_desc_hdr|   */
/*    +--------------------+   */
/*    |   cell_row         |   */
/*    |     :              |   */
/*    |     :              |   */
/*    +--------------------+   */
```

29.2 The Generic Frame Header

```
GDB_FRAME_HDR       LITERALLY
    'OPCODE  BYTE,    set to LSH_FRAME_OPCODE.
    HDR_SIZE WORD,    always the same: 16 bytes.
    SIZE     WORD,    byte size of this particular frame.
    ID       WORD,    frame id within the frame-type.
    TYPE     BYTE,    type is GDBF_CELL_ROW_DESC.
    GDB_RECTANGLE';   not used.  All 4 words = GDB_NIL_WORD.
```

29.3 The Cell Row Descriptor Header

(GDB_CELL_ROW_DESC_HDR)

CELL_ROW_DESC_OPCODE BYTE,
Set to LSH_SKIP_OPCODE.

CELL_ROW_HDR_SIZE WORD,
size of this header.

ALIGNMENT_BYTE BYTE,
A garbage byte. It's here just to align the count and offset fields with a WORD boundary. The alignment speeds up most CPU operations.

CELL_ROW_COUNT WORD,
count of cell_row records below.

OFFSET_CELL_ROW WORD,
offset to the cell_row data.

29.4 The Cell Row Definition

(GDB_CELL_ROW_DEF)

Roughly speaking, one might find 4 to 40 of these cell_row_def records in the database.

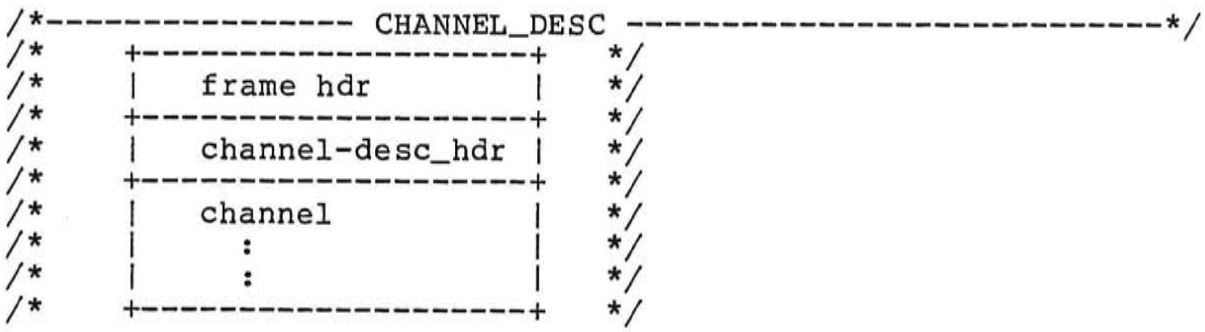
TYPE WORD,
flags - miscellaneous information.

(X1,Y1,
X2,Y2) WORD;
A pair of lattice points which define the band of cells in the cell-row (detail-view coordinates; lower-left, upper-right).

30 The CHANNEL_DESC Frame

There is just one of these frames.

30.1 Overview



30.2 The Generic Frame Header

```
GDB_FRAME_HDR       LITERALLY
    'OPCODE  BYTE,    set to LSH_FRAME_OPCODE.
    HDR_SIZE WORD,    always the same: 16 bytes.
    SIZE     WORD,    byte size of this particular frame.
    ID       WORD,    frame id within the frame-type.
    TYPE     BYTE,    type is GDBF_CHANNEL_DESC.
    GDB_RECTANGLE';   not used.  All 4 words = GDB_NIL_WORD.
```

30.3 The Channel Descriptor Header

(GDB_CHANNEL_DESC_HDR)

CHANNEL_OPCODE BYTE,
Set to LSH_SKIP_OPCODE.

CHANNEL_HDR_SIZE WORD,
size of this header.

ALIGNMENT_BYTE BYTE,
A garbage byte. It's here just to align the coordinates below with a word boundary. The alignment speeds up most CPU operations.

(X1_GLOBAL,Y1_GLOBAL,
X2_GLOBAL,Y2_GLOBAL) WORD,
Coordinates of the enclosing rectangle for the family of channels in the middle part of the chip. (Rough coordinates; lower-left, upper-right.)

CENTRAL_CHANNEL_TYPE BYTE,

the direction of slicing of this family of channels: 'V' - vertical, 'H' - horizontal.

CHANNEL_ROUTING_LAYER WORD,
the layer for routing along the channel.

CELL_ROW_ROUTING_LAYER WORD,
the layer for routing across the channel and across the cell row.

CHANNEL_COUNT WORD,
Count of channel records below.

OFFSET_CHANNEL WORD,
Offset to the channel data.

30.4 The Channel Definition

(GDB_CHANNEL_DEF)

This definition covers just channels which are at the chip edges and not in the central area.

Each channel encloses grid points.

There would be at most four of these records.

TYPE WORD,
Bit-encoded. Six bits are currently used. Two bits indicate the horizontal or vertical slicing of this channel. The other four bits tell what side of the chip is the location of the channel: the north, east, west, or south side of the chip.

(X1,Y1,
X2,Y2) WORD;
Global coordinates of the channel (lower-left, upper-right).

31 Appendix: Index Frames

The one type of frame that can be found in a VIRGA-type database file that is not documented in the annotated schema ~~below~~ is the INDEX frame-type.

The main thing to know about name indexes is that you might see them in the database, BASE provides a few of them, and VIRGA maintains them automatically. Since they are automatic and already described in the VIRGA manual, they are not documented in this manual.

Name indexes are simply frames of type 'index'. This means that their internal structure is organized around pairs of names and pointers. The list of names and pointers to other frames helps speed up access time.

Since VIRGA provides whatever name information is needed, and maintains index frames automatically, the application programmer does not need to know the internal structure of index frames. No application program effort is required other than a call to the VIRGA routines which tell VIRGA to create an index or look through one.

A program may tell VIRGA to create an index for a frame-type which currently has no entries. VIRGA knows enough to create an empty index frame, and then add to it and maintain the index when the relevant frames enter the database. Thus, the BASE compiler, when processing GAIL text for either a .BASE file or a .MLIB file, will create a macro index frame. Thus, macro names will be automatically indexed from then on.

The .MLIB file holds user-placeable macros. The .BASE file, at creation-time, holds empty-cell and permanent macros. SLIDE will turn the .BASE file into a .LED file. After that, LED will copy user macros whenever they are used, from .MLIB into the .LED file.

All the three types of macros which may be found in the .LED file are indexed by the same macro index frame. Also, all macros are classified by MACRO_CLASS frames, and the macro-class frames themselves are indexed by a macro-class index.

Currently, the BASE program will create two indexes in the .BASE file: the macro name index and the macro-class index. SLIDE will carry these into the .LED file. The indexes speed up a search for a specific frame of either of those two types.

Besides these "built-in" indexes for macro names and macro-class names, a person using PDQ to look at an R&D database might easily see other indexes. For testing purposes, programmers may generate name indexes for other gate array

objects in order to obtain a specific correlation between frame order and name entries. For example, components and nets are often indexed. A person who wants to index certain frames for testing purposes should check with the group's system integrator. There are PDQ input files already available for creating test indexes.

For information on creating and reading index frames, please see the VIRGA manual.

32 Appendix: LSH Constants

The data structures and comments on this page are from LSHDEF.ELT.

```
/*****************************************************************
*   LSHDEF.ELT -- Definitions for the LED portion of the screen handler  *
*                    Suzanne Jacobs, December 9, 1983                    *
*                                                                        *
*If you reference any LSH opcodes, include SCDISP.ELT in your source code*
*If you reference maximum valid cell_type or macro_id, include GDFIL.ELT.*
*If you reference LSH_BMP_HDR, include GDSCH.ELT.                        *
*****************************************************************/


    LSH_FRAME_OPCODE                  LITERALLY 'SCR_MAX_OPCODE+1',
    LSH_NET_OPCODE                    LITERALLY 'SCR_MAX_OPCODE+2',
    LSH_BLOCK_OPCODE                  LITERALLY 'SCR_MAX_OPCODE+3',
    LSH_COMPONENT_OPCODE              LITERALLY 'SCR_MAX_OPCODE+4',
    LSH_CELL_ARRAY_OPCODE             LITERALLY 'SCR_MAX_OPCODE+5',
    LSH_NOTE_OPCODE                   LITERALLY 'SCR_MAX_OPCODE+6',
    LSH_CELL_TYPE_OPCODE              LITERALLY 'SCR_MAX_OPCODE+7',
    LSH_MACRO_OPCODE                  LITERALLY 'SCR_MAX_OPCODE+8',
    LSH_OBSTRUCTION_OPCODE            LITERALLY 'SCR_MAX_OPCODE+9',
    LSH_ARBITRARY_RECTANGLES_OPCODE   LITERALLY 'SCR_MAX_OPCODE+10',
    LSH_UNDERPASS_OPCODE              LITERALLY 'SCR_MAX_OPCODE+11',
    LSH_OBSTRUCTION_BMP_OPCODE        LITERALLY 'SCR_MAX_OPCODE+12',
    LSH_SKIP_OPCODE                   LITERALLY 'SCR_SKIP',

    LSH_TEXT_VERTICAL_MASK            LITERALLY '1000$0000B',
    LSH_TEXT_SHIFT_MASK               LITERALLY '0000$1000B',
    LSH_TEXT_UNDERSCORE_MASK          LITERALLY '0000$0100B',
    LSH_TEXT_CENTERED_MASK            LITERALLY '0000$0010B',
    LSH_TEXT_RIGHT_JUSTIFIED_MASK     LITERALLY '0000$0001B',
    LSH_TEXT_REQUIRED_ZEROS           LITERALLY '0111$0000B',
    LSH_TEXT_BAD_COMBINATION          LITERALLY '0000$0011B',
```

An arbitrary rectangles frame is created only for LED--LSH communications; it does not appear in the disk database. The frame consists of the header defined below and a bunch of GDB_RECTANGLE_DEFS that define the rectangles. At the moment, this frame is used by the Manual Editor to display rough traces in the detail view and by the Plot task to display the plotting boun- daries.

```
DECLARE
     LSH_ARBITRARY_RECTANGLES_HDR LITERALLY
    'ARB_RECTANGLES_OPCODE    BYTE,
     ARB_RECTANGLES_HDR_SIZE WORD,
     ARB_RECTANGLES_TYPE     BYTE,
     ARB_RECTANGLES_COUNT    WORD,
     OFFSET_ARB_RECTANGLES   WORD',
```

The data structures and comments on this page are from LSHMSG.ELT.

```
/*****************************************************************
*LSHMSG.ELT -- MTOS messages to the LED portion of the screen handler*
*                  Suzanne Jacobs, August 9, 1983                 *
*****************************************************************/

  OBSTRUCTION_MASK      WORD',
     /* this field exists so that LED can find out the status    */
     /* of the obstruction display and initialize the Visibility */
     /* menu appropriately.  It is sort of kludgy.  Note that    */
     /* LED has code such that when it requests a virtual clear  */
     /* screen, it resets the mask to zero.  I'm not at all sure */
     /* that this is a "clean" implementation but we must make   */
     /* yet another sacrifice on the altar of timely releases.   */
```

The "priority_table_ptr" points to a vector of bytes which give the priorities for the various "brushes" (classes of symbols) in the font for GM-1Ø. Note that some brushes must be given the same priority and are so grouped. The order of the table is given below:

Blank (a single character, normally the lowest priority)

Block (used for drawing blocks)

Cell (used for drawing cells)

Component (used for drawing macros associated with placed components)

Text (the ASCII character set less the control codes)

Layer Ø \
Layer 1 | (these three are a group: the traces on layers Ø and 1)
Complex /

Floating Via \
Fixed Via / (these two are a group: symbols used to draw vias)

Layer Ø Pin \
Layer 1 Pin | (these three are a group: the pins on different layers)
Both Pin /

Invalid (all unused slots inside the GM-1Ø font)

33 Appendix: Size Estimates

These estimates are approximate only. We calculate the size of the major components of the data base for two different hypothetical gate arrays. The first is roughly the smallest gate array of interest; the second is roughly the largest we expect to handle. The smaller one is comparable to (perhaps one-half of) the Motorola Macro_cell. The larger one is roughly 100 times larger. [ESTIMATES ARE FOR VERSION 5.0.02]

	Equiv. Gates	Cells	Nets	Grid
Smallest	1K	100	200	200x200
Largest	10K	10K	10K	2Kx2K

Assume one component per cell. Assume the average rough net contains 6 traces, and the average detail net contains 4 pins, 16 traces, and 12 vias. This results in the following estimates for a single net.

NET:	Bytes
Net_Pin	40
Rough	48
Detail	176
Via	84
Total	382

In the interest of simplicity, and considering the approximate nature of these numbers, we round the single numbers; e.g., the above calculation gives a net size of 382 bytes, which we round to 400.

Here are some of the estimates. For a macro, we assume the graphics takes about 50 Bytes, there are about 12 pins, no routes, 6 traces, 2 obstructions, and one cell_list item, yielding 281 Bytes. Round to 300.

For a component, we assume 6 pins, two equivalences, yielding 111 Bytes. Round to 100.

For a cell array, there is a 23-byte overhead per array, 7 bytes per cell. For the smallest [largest] array, we guess roughly 4 arrays of 10 [1K] I/O cells in addition to the 100 [10K] cells.

Bitmaps have 4 bytes per grid point, and 2 layers, yielding 8 bytes per grid point. Small [large] grid is then 8 * 40K [4M]

equals 320K [32M].

The memory estimate assumes that we have 40K of buffer space for the (virtualized) nets, 20K for the (virtualized) bitmaps, and the remainder is entirely in memory; for the data not considered in detail here, we add 10K in the smallest case and 1M in the largest. Of course, the amount of memory used for buffer space could be increased.

	Single	Smallest Chip	Largest Chip
Net	400	80K	4M
Macro	300	6K	60K
Component	100	10K	1M
Cells	23+7	1.1K	100K
Bitmap	8/grid	240K	32M
Memory		87K	38M

34 Appendix: Future Possibilities

Here we present various thoughts we have considered but not implemented.

1. Pin Names
Pins do not have names (text strings) associated with them. This could be added if and when someone sees a need for it. NOTE: As of V1.Ø.8, pins on macros DO have names, but pins on components do NOT have names.

2. Flip/Rotate Implied Translation
Flipping and rotating macros might include an implied translation. For example, if a (non-square) rectangle is rotated, the rotation may take it off the desired location unless an implied translation is performed. This is similar to asking about what point (or line) the macro is rotated or flipped. Since we are not immediately implementing flipping and rotation, we postpone any decision on this matter until necessary.

3. Cell Array Contents
As of V1.Ø.8, the cell array is (or may be) something like a do-loop or step_and_repeat; see above. A gate array is supposed to be a reasonably regular array of something. That something is the cell. Our schema describes this regularity via the cell array, which has one entry for each cell. You might wonder why we don't simply define a generic cell and then give a step-and-repeat (or do-loop) instruction to spread it over the chip. The reason is that feature bits and component id's are not regular across the cell types.

Actually, we do something similar, in that each cell has a type, and each type has two null macros. Thus, each cell type references all the information in those two macros, and the cell array tells where to copy that information. Introducing the notion of a cell type allows us to handle chips which are almost regular, but with more dimensions to their regularity; e.g., a chip which has two different cells repeated alternately. So, the entry in a cell array is relatively small, but there is nonetheless an entry for every single cell. This allows us to have a slot into which we may insert placement information, to indicate what has been placed at that location.

A possible change would be to include more information with each cell type. For example, the cell type might include underpasses, fixed vias, discrete components, etc. Quite a bit can be done just with macros. For example, permanent and empty-cell macros already carry some information per cell-type.

5. Discretes
Some chips have pre-defined resistors, capacitors, etc., which are available on the chip, but which do not belong to any particular macro; in fact, these discrete components are present whether or not any macros have been placed. We have no explicit provision for these "discretes" in the schema. Perhaps they can be handled with the null macro approach, which we deem desirable since it would avoid complicating the data base with new data types. In any case, we will have to handle these. We postpone the decision on how they will be handled until it is necessary to make that decision and we can see the trade-offs.

6. Input Methods
Ultimately, there could be a base-array editor and a macro editor. Presently, the Gate Array Interface Language serves as input interface to both.

35 Appendix: Old Version Changes

Changes from Version V1.0.2 to current V1.0.08

1. V1.0.08 8/10/82

2. Added Delta frame, for engineering change.

3. Removed fixed_via frame; fixed vias are now in underpass array frame.

4. Added gdb_max_step, for cell_array and underpass_array.

5. Added restriction that rectangle and trace coordinates (structures with fields x1, y1, x2, y2) MUST ALWAYS obey x1 <= x2 and y1 <= y2. (That is, started to enforce this restriction.)

f. Added gdb_relative_rectangle_def, for macro_obstructions, etc.

7. Changed net.status from integer to bit-significant (mask) word.

8. Added net_pin.type byte.

9. Added macro.rough_x_blockage and _y_blockage.

10. Added component.cell_array_id and _x and _y. This is duplicate information to improve performance.

11. Cell_array significantly changed to accommodate three array types: random, projection, and step. Cell_def no longer has (x,y) or type fields. Added x_list, y_list, x_steps, etc.

12. Changed cell_array labl fields to fixed_length.

13. Replaced old underpass (and fixed_via) frames with new underpass (array) frame, which describes an array much like the cell_array.

14. Added rough_misc.preferred_direction and .status.

15. Changed order of bitmap_hdr.desc_count and .offset_desc.

16. Changed note frame to a list of (line_def+text), where text is variable-length.

17. Added global_desc.schema_version_size and .num_updates and .wrap_flags (reserved for use by WRAP program, initialized to zero).

18. Defined gdbc_mask_valid_word; gdbc_horizontal, _vertical, and _diagonal for rough_misc.preferred_direction (et. al.?); gdbc_array_random, _projection, and _step for array.x_type and .y_type; gdbc_mask_x_labl_first for cell_array.flags; gdbc_delta literals for delta.delta_type;

19. Added bits for layer_info.flags: gdbc_mask_vertical says preferred direction is horizontal (else its vertical); gdbc_mask_underpass says layer is for underpasses.

20. Changed interpretation of macro_pin.id to id of subnet internal to macro; many macro_pins may map into one component pin, and all of those pins have the same id as the component pin. To go with this, changed macro_route structure. It now contains the same ID as the corresponding macro_pins as well as a flags field which may contain the bit gdbc_mask_necessary (see above).

21. Added global_desc.rough_x_divisor, ..., overview_y_divisor.

Changes from Version V1.0.1 to V1.0.2

1. V1.0.2 4/13/82

2. Added frame types:
   gdbf_note
   gdbf_block
   gdbf_overview

3. LOWER and UPPER case both allowed in data base. It is the user's responsibility to do any desired case-folding after each and every access to the data.

4. Added gdb_text_def (graphics text) and gdb_outline_def(polygon outline), and gdb_relative_outline_def (polygon with INTEGER rather than WORD coords).

5. In frame_hdr (which is in gdfil.elt, not attached below), the fields ID and SIZE switched places. Re-compile only.

6. Throughout, prefixes were prepended to make hdr names unique from frame_hdr fields; e.g., NET_OPCODE. Also, offset field was added for first list (e.g., net_pin). Also, order of count/offset fields was changed so that future additions will have less impact on existing data bases.

7. Net_hdr: Prefix "NET_"
   Offset_pin added
   order changed

8. Via_def:
   Type: Least-significant-bit = Ø if unused, 1 if used.
   Next bit = Ø if via not fixed; 1 if fixed via.

9. Macro: changed macro_name to gdb_text_def
   added outline
   used macro_obstruction_def, with INTEGER coords
   added macro_vias
   order of count/offset changed
   prefix "MAC_"
   pin: added text_def, for pin name.
   added log_equiv list, count, and offset (used to be on the component)

1Ø. Component: added prefix "COMP_"
   removed log equiv (added to macro)
   pins: all pins appear here, including pins on the macro which are not on the via. This means the max num pins on any macro for this component, so that such physical-only pins may have net_id recorded here. [A placer might dynamically add pins to components, as necessary; for short term, let's assume there are already enough pins.]
   orientation replaces rotation, flip fields
   comp_name changed to text_def
   count/offset order changed
   other order changes
   macro_name changed to macro_class_name for clarity

11. Cell_array: prefix "CELL_"
   changed dimensions(2) to x_count, y_count
   added x_labl and y_labl arrays of strings (labels)
   Note that LABEL is a reserved word in PLM, so we use LABL.

12. Cell_type: prefix "CTYPE_" added
   hdr and def merged into hdr; one cell_type per frame
   added outline
   Note that underpasses and vias now come with cells; they are on the permanent macro for that cell. This will greatly reduce volume of data; e.g. for MITEL, underpasses reduced from 161K to about 6K, including cells.

13. Obstruction: prefix "OBS_"
   Note that obstruction_def has absolute (WORD) coords

14. Underpass: prefix "UND_"
   removed net_id field
   Now, most underpasses appear in cell_type. Any that cannot fit there, go here.
   Note that net_id field is gone completely.

15. Vias: prefix "VIA_"
   removed net_id field

Now, most vias appear in cell_type. Any
that cannot fit there, go here.
Note that net_id field is gone completely.

16. Bitmap_desc: prefix "BITM_"

17. Ded_page_list: Renamed pathname_def
prefix "PATH_"

18. Global_desc: prefix "GLOB_"

19. Block frame (new frame) has id to note associated with it. Other notes are not associated with anything in particular.

20. GDBM_<frame_type> added for every frame. If you ever need to refer to the module type in which a frame type lives, use this literal. If the packing of frames into modules changes, then you need only recompile.

21. Frame_hdr opcode field is SCR_FRAME (defined in gdfil.elt as GDB_SCR_FRAME, and
in some screen handler include file as SCR_FRAME).
SCR_FRAME = 5.
Note that the opcode field in the object's hdr
(e.g., net_opcode) contains the screen_handler's opcode
for that object (e.g., scr_net).

22. Rough data is now defined.

==========
Changes, version V1.Ø.Ø to V1.Ø.1
==========

1. V1.Ø.1 3/Ø3/82

2. Added frame types:
   gdbf_cell_type
   gdbf_projection
   gdbf_global_desc

3. Deleted frame types:
   gdbf_db_info /* see global_desc */
   gdbf_layer_info /* see global_desc */

4. Comp_pin: Deleted comp_id (it's in comp_hdr). Added id (for the pin itself, so no ordering is assumed).

5. Comp: Added rotation, flip fields

6. Macro: Added legal_transforms field. Changed locations (co-ordinates) from WORD to INTEGER. Macro_pin: Added layer_mask field.

7. Net: Changed Attribute to Type Added Criticality Added Status Added Rough_Trace.Type.

8. Layer: Added Type

9. VIRGA now pre-defines its frame types and module types and much of the information that used to be in db_info. See VIRGA functional specification.

1Ø. Global_desc now defines the information which is global to the chip, including the layer_info structures, the range of grid co-ords, the via-adjacency rules, the units, the macro_lib name, the chip_lib name, the chip name. Also included here is a description of all power types, including ground types.

11. Projection array now defines mapping from grid co-ords to actual physical co-ords on the chip.

12. The cell-type structure now defines the null macros for cells.

*** INDEX ***