CONFIDENTIAL

Howard J Cohen
Sept 185

VIRGA
(SAM and GDB)

Virtual Data-file Access
(Integrated Layout Environment)
(GATEMASTER)

V3.0

Author/Programmer: Drew Wade and Harold Rabbie

Technical Writer: Shana Lavatelli and D. Avren

Design File Document No. 4.21 Rev 2

Doc. No. 4.21 Rev 1 May 1985

TABLE OF CONTENTS

1 INTRODUCTION ..... 1 - 1
1.1 Features ..... 1 - 1
1.2 A General Access Tool ..... 1 - 3
1.3 Restrictions ..... 1 - 4
1.4 CHIPMASTER Use of VIRGA ..... 1 - 5
1.5 PDQ, PEQ, or DQL uses of VIRGA ..... 1 - 5
1.6 Most Important Routines ..... 1 - 6
2 VIRGA STRUCTURES ..... 2 - 1
2.1 VIRGA File Structure ..... 2 - 1
2.1.1 Module and Frame Types ..... 2 - 3
2.1.2 Reserved Control Modules and Frames ..... 2 - 3
2.1.3 Restriction ..... 2 - 4
2.1.4 Required PLM Structures ..... 2 - 5
2.1.4.1 PLM File Description ..... 2 - 5
2.1.4.2 PLM Module Description ..... 2 - 7
2.1.4.3 PLM Frame Description ..... 2 - 7
2.1.5 Sizes of the File, Modules, and Frames ..... 2 - 10
2.1.5.1 Module Size ..... 2 - 10
2.1.5.2 Frame Size ..... 2 - 10
2.2 VIRGA Cell Libraries ..... 2 - 11
2.3 VIRGA Box Structure ..... 2 - 13
2.4 VIRGA Name Index Tables ..... 2 - 14
2.5 VIRGA Inventory Tables ..... 2 - 19
2.5.1 A Drawing of Internal VIRGA Tables ..... 2 - 20
2.5.2 The Module Inventory ..... 2 - 21
2.5.3 The Frame Types Inventory ..... 2 - 22
2.5.4 The Frame Inventory ..... 2 - 23
2.5.5 The Buffer Inventory ..... 2 - 24
3 THEORY OF OPERATION ..... 3 - 1
3.1 VIRGA Files ..... 3 - 1
3.2 Name Indexing ..... 3 - 3
3.3 Inventory Tables ..... 3 - 5
3.4 Buffer Control ..... 3 - 6
3.4.1 Module Swapping ..... 3 - 6
3.4.2 The Buffer Aging Algorithm ..... 3 - 7
3.4.3 Using Pointers and Locks ..... 3 - 7
3.5 Transaction Procedures ..... 3 - 10
3.5.1 SAM_START_TRANSACTION ..... 3 - 10
3.5.2 SAM_UNDO_TRANSACTION ..... 3 - 11
3.5.3 SAM_GET_TRANSACTION_LOG ..... 3 - 11
3.5.4 SAM_FINISH_TRANSACTION ..... 3 - 12
3.5.5 SAM_SET_DIRTY_MODULE ..... 3 - 12
3.5.6 SAM_DIRTY_FRAME ..... 3 - 13
3.5.7 Example Transaction Operations ..... 3 - 13
3.6 SEARCH Procedures ..... 3 - 15
4 VIRGA PROCEDURES ..... 4 - 1
4.1 xxxx.EXT File Contents ..... 4 - 2
4.1.1 Configuration Procedure ..... 4 - 2
4.1.2 File Manipulation Procedures ..... 4 - 2
4.1.3 Data Access Procedures ..... 4 - 2
4.1.4 Name Index Procedures ..... 4 - 3

4.1.5 Buffer Control Procedures ..... 4 - 4
4.1.6 Utility Procedures ..... 4 - 4
4.1.7 Higher Level Procedures ..... 4 - 4
4.2 General Information About Procedures ..... 4 - 6
4.3 A Complete List of VIRGA Procedures ..... 4 - 8
4.3.1 SAM_ADD_FRAME ..... 4 - 9
4.3.2 SAM_ADD_FRAME_TYPE ..... 4 - 11
4.3.3 SAM_ADD_NEW_FRAME_TYPE ..... 4 - 12
4.3.4 SAM_ALLOCATE_FRAME ..... 4 - 13
4.3.5 SAM_ALLOCATE_MODULE ..... 4 - 15
4.3.6 SAM_CLEAR_DIRTY_MODULE ..... 4 - 16
4.3.7 SAM_CLOSE ..... 4 - 17
4.3.8 SAM_CREATE ..... 4 - 18
4.3.9 SAM_CREATE_SHELL_PATH ..... 4 - 19
4.3.10 SAM_CREATE_BOX_INDEX ..... 4 - 21
4.3.11 SAM_CREATE_CELL_DIR ..... 4 - 22
4.3.12 SAM_CREATE_INDEX ..... 4 - 23
4.3.13 SAM_DELETE_BY_TYPE ..... 4 - 25
4.3.14 SAM_DELETE_FRAME ..... 4 - 26
4.3.15 SAM_DELETE_INDEX ..... 4 - 27
4.3.16 SAM_DIRTY_FRAME ..... 4 - 28
4.3.17 SAM_DISPOSE_ALL ..... 4 - 29
4.3.18 SAM_DISPOSE_MODULE ..... 4 - 30
4.3.19 SAM_DISPOSE_MODULES_BY_TYPE ..... 4 - 31
4.3.20 SAM_DUMP_GLOBAL ..... 4 - 32
4.3.21 SAM_FIND_FIRST ..... 4 - 33
4.3.22 SAM_FIND_FIRST_FRAME_IN_BOX ..... 4 - 35
4.3.23 SAM_FIND_FRAME ..... 4 - 37
4.3.24 SAM_FIND_FRAMES_IN_WINDOW ..... 4 - 39
4.3.25 SAM_FIND_INDEX_FRAME ..... 4 - 41
4.3.26 SAM_FIND_NEXT ..... 4 - 43
4.3.27 SAM_FIND_NEXT_FRAME_IN_BOX ..... 4 - 45
4.3.28 SAM_FINISH_TRANSACTION ..... 4 - 47
4.3.29 SAM_FIRST_NAME ..... 4 - 48
4.3.30 SAM_FLUSH ..... 4 - 50
4.3.31 SAM_FLUSH_ALL ..... 4 - 51
4.3.32 SAM_FLUSH_AND_DISPOSE ..... 4 - 52
4.3.33 SAM_GET_AFT ..... 4 - 53
4.3.34 SAM_GET_CONFIG ..... 4 - 55
4.3.35 SAM_GET_MODULES_BY_TYPE ..... 4 - 57
4.3.36 SAM_GET_PATHNAME ..... 4 - 58
4.3.37 SAM_GET_TIMESTAMP ..... 4 - 59
4.3.38 SAM_GET_TRANSACTION_LOG ..... 4 - 60
4.3.39 SAM_GROW_FRAME ..... 4 - 62
4.3.40 SAM_INITIATE ..... 4 - 64
4.3.41 SAM_INQUIRE_FRAME_ID ..... 4 - 65
4.3.42 SAM_LOCK_FRAME ..... 4 - 66
4.3.43 SAM_LOCK_MODULE ..... 4 - 67
4.3.44 SAM_LOCK_MODULES_BY_TYPE ..... 4 - 68
4.3.45 SAM_MAKE_FRAME ..... 4 - 69
4.3.46 SAM_MAX_ID ..... 4 - 70
4.3.47 SAM_MOVE_FRAME ..... 4 - 71
4.3.48 SAM_NEXT_NAME ..... 4 - 73
4.3.49 SAM_NUM_FRAMES ..... 4 - 75

4.3.50 SAM_NUM_LOCKS ..... 4 - 76
4.3.51 SAM_OPEN ..... 4 - 77
4.3.52 SAM_OPEN_SHELL_PATH ..... 4 - 78
4.3.53 SAM_SEARCH_FIRST_NAME ..... 4 - 80
4.3.54 SAM_SEARCH_NEXT_NAME ..... 4 - 83
4.3.55 SAM_SET_DIRTY_MODULE ..... 4 - 85
4.3.56 SAM_SET_INDEX_TYPE ..... 4 - 86
4.3.57 SAM_START_TRANSACTION ..... 4 - 87
4.3.58 SAM_SWITCH_MUX ..... 4 - 88
4.3.59 SAM_TERMINATE ..... 4 - 89
4.3.60 SAM_UNDO_TRANSACTION ..... 4 - 90
4.3.61 SAM_UNLOCK_FRAME ..... 4 - 91
4.3.62 SAM_UNLOCK_MODULE ..... 4 - 92
4.3.63 SAM_UNLOCK_MODULES_BY_TYPE ..... 4 - 93
4.3.64 SAM_UPDATE_FRAME ..... 4 - 94

LIST OF FIGURES

Figure 2 - 1 The VIRGA File Structure ..... 2 - 2
Figure 2 - 2 MAX Modules and Frames ..... 2 - 3
Figure 2 - 3 File Descriptor Table ..... 2 - 5
Figure 2 - 4 File Descriptor Definition ..... 2 - 5
Figure 2 - 5 Frame Descriptor Definition ..... 2 - 6
Figure 2 - 6 Module Header Structure ..... 2 - 7
Figure 2 - 7 ..... 2 - 8
Figure 2 - 8 Cell Libraries File Structure ..... 2 - 11
Figure 2 - 9 Index Base Structure ..... 2 - 14
Figure 2 - 10 Index Header Structures ..... 2 - 14
Figure 2 - 11 Index Descriptor Structure ..... 2 - 15
Figure 2 - 12 Index Descriptor explained ..... 2 - 15
Figure 2 - 13 Index Header Diagram ..... 2 - 17
Figure 2 - 14 Symtab Diagram ..... 2 - 18
Figure 2 - 15 Index Entries Diagram ..... 2 - 18
Figure 2 - 16 VIRGAs Internal Tables ..... 2 - 20
Figure 2 - 17 The Module Inventory for a File ..... 2 - 21
Figure 2 - 18 The Frame Inventory ..... 2 - 23
Figure 2 - 19 The Buffer Inventory ..... 2 - 24
Figure 3 - 1 Meanings of Wild Cards ..... 3 - 15
Figure 4 - 1 All the VIRGA Procedures ..... 4 - 8
Figure 4 - 2 Files Definition Structure ..... 4 - 81

1 INTRODUCTION

VIRGA is a general-purpose data manager, whose capabilities fall between those of a full-fledged database manager and those of a virtual memory manager. VIRGA (VIRtual Gate Array Access routines) was originally written to facilitate storage of and access to GATEMASTER gate array data. The routines now available in VIRGA, the virtual data file access routines, are also referred to as SAM routines. SAM routines documented here are not valid for use with the GATEMASTER database GDB.

VIRGA manages the database information for application programs. For example, when a customer edits xxxx.CELL or xxxx.CLIB files (using the CHIPMASTER Mask Editor, MAX), VIRGA enters the changes into the database. The xxxx.CELL and xxxx.CLIB files contain the geometry and hierarchical reference information, and this information is organized in VIRGA format.

VIRGA format refers to the structure of the contents of a VIRGA file. VIRGA file structure contains modules and frames. VIRGA manipulates the modules and frames (containers of data), not the data itself, unlike a full-fledged database manager.

1.1 Features

VIRGA has the following advantages:

- o high speed (read and write) data access,
- o efficient use of central memory, and
- o efficient use of the hard disk.

For instance, the Polygon Editor program uses VIRGA routines to access data and to control memory buffers.

The main goals for the VIRGA design are:

FUNCTIONALITY—you can read and write data.

EASE OF USE—you can read and write data using a minimal amount of code. Routines are simple to understand.

PERFORMANCE—high execution speed.

The VIRGA design assumes that you are flexible, and therefore may prefer to trade some robustness in favor of performance or ease of use.

Both interactive editor programs and batch programs use VIRGA routines. For instance, in the CHIPMASTER system, not only MAX uses VIRGA; the programs STREAMIN and STREAMOUT also use VIRGA routines to access the database.

NOTE: In this manual, "you" refers to the programmer using the database. The user is the person who employs a program that uses VIRGA. For example, customers use MAX, but you (programmers) use VIRGA routines in programs that manage the data for MAX.

1.2 A General Access Tool

The source code for the VIRGA routines is not concerned with the actual data contained in a VIRGA file. VIRGA knows the structure of a VIRGA database file (described in Chapter 2); VIRGA finds the requested frame by searching through the data frames, or by using an index for quicker access.

You can retrieve data frames in any of four ways:

- o you specify the frame type and frame ID for one frame
- o you specify the frame type and the name of the desired frame
- o sequentially, you specify the frame type (each frame of the specified type is examined)
- o sequentially, you specify the frame type and box (each frame within a specified rectangle is examined)

Because of VIRGA's focus on file structure rather than file data, you can use VIRGA routines to create a database file that is structured efficiently and is easy to access.

VIRGA uses buffers to access the disk and memory. The buffers (described in Section 2.4 below) are normally handled automatically, but you can control buffer management when necessary. The buffer control procedures can be thought of as disk read/write procedures or as memory management procedures. To illustrate, the two approaches are shown below.

(1) You can use VIRGA as a simple file access method (read/write procedures). To read, you simply copy data from a retrieved frame into your workstation; this procedure results in a copy of the desired data from disk. Similarly, a write request can be viewed as writing to disk.

(2) You can also use VIRGA as a virtual memory manager that supplies buckets of memory (modules). If the buckets overflow physical memory, then they are retained on disk until needed again. In this approach, you can either explicitly choose a particular file for virtual memory and save it between invocations, or you can allow VIRGA to choose a unique filename and purge the file when it is no longer needed. In this mode, the modules that you want to keep in memory must be locked.

The detailed contents (data structures) of particular VIRGA database files are described in other documents. For instance, SIDSCHEMA.ELT and SLTSCHEMA.ELT describe the data structures (the schema) of MAX data frames.

1.3 Restrictions

The VIRGA routines are not intended to constitute a full database management system. Many of the features familiar in a database management system, such as backup and recovery, and the ability to search for data according to complex context-sensitive relationships, are not provided here.

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

If a program that calls VIRGA is divided into code overlays, you must ensure that the code for particular database manipulations is available to any overlays that require these manipulations. The simplest approach to handling overlays is to include the routines in the part of the code that is permanently resident in memory.

1.4 CHIPMASTER Use of VIRGA

In the CHIPMASTER system, some VIRGA routines are called indirectly through MPI (MAX Procedural Interface). MPI routines are high level procedures that call several VIRGA routines to access a variety of data structures.

CHIPMASTER database files, i.e., xxxx.CELL and xxxx.CLIB, are all files in VIRGA format.

1.5 PDQ, PEQ, or DQL uses of VIRGA

Another way to access VIRGA files, in addition to calling VIRGA routines in a program, is to use the PDQ, PEQ, or DQL VIRGA interface programs.

PDQ is designed for the LED (GATEMASTER) databases.

PEQ is designed for the MAX (CHIPMASTER) databases.

DQL is designed for the EDS (symbolic layout) databases.

You invoke these programs at the command level simply by typing their names, for example:

```
PEQ {EXECUTE}
```

These programs provide a general programmer-interface to any VIRGA file. Most commands in these programs are simply VIRGA procedures that you can invoke individually at the user level.

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

For more information about how to use PDQ, see the "PDQ User's Manual," Design File Document No. 4.22 Rev. 0 (Feb 1984).

1.6 Most Important Routines

The most important VIRGA routines are:

- o the following nineteen routines for file configuration, file initialization, file manipulation, and data access

```
SAM_GET_CONFIG
SAM_INITIATE
SAM_TERMINATE
SAM_CREATE
SAM_OPEN
SAM_CLOSE
SAM_DUMP_GLOBAL
SAM_FIND_FRAME
SAM_ADD_FRAME
SAM_MAKE_FRAME
SAM_UPDATE_FRAME
SAM_MOVE_FRAME
SAM_GROW_FRAME
SAM_DELETE_FRAME
SAM_DELETE_BY_TYPE
SAM_FIND_FIRST
SAM_FIND_NEXT
SAM_GET_MODULES_BY_TYPE
SAM_DIRTY_FRAME
```

- o the first six of the eight index routines

```
SAM_DELETE_INDEX
SAM_CREATE_INDEX
SAM_CREATE_BOX_INDEX
SAM_FIND_INDEX_FRAME
SAM_FIRST_NAME
SAM_NEXT_NAME
```

The routines listed above provide the basic VIRGA functions that are necessary for average data access needs, database structure housekeeping, and use of VIRGA itself. Additional routines for buffer control, interface, and the search index, and some utilities are available.

2 VIRGA STRUCTURES

This chapter contains a discussion of the following VIRGA structures and their related PLM structures:

- o VIRGA Files
- o Cell Libraries
- o Boxes
- o Name Index Tables
- o Inventory Tables

2.1 VIRGA File Structure

All VIRGA files have the same basic structure as shown in Figure 2 - 1 below. All VIRGA files begin with a root module, which is followed by one or more modules.

The root module contains VIRGA file control information. The whole VIRGA file can be any size which can be contained on one disk.

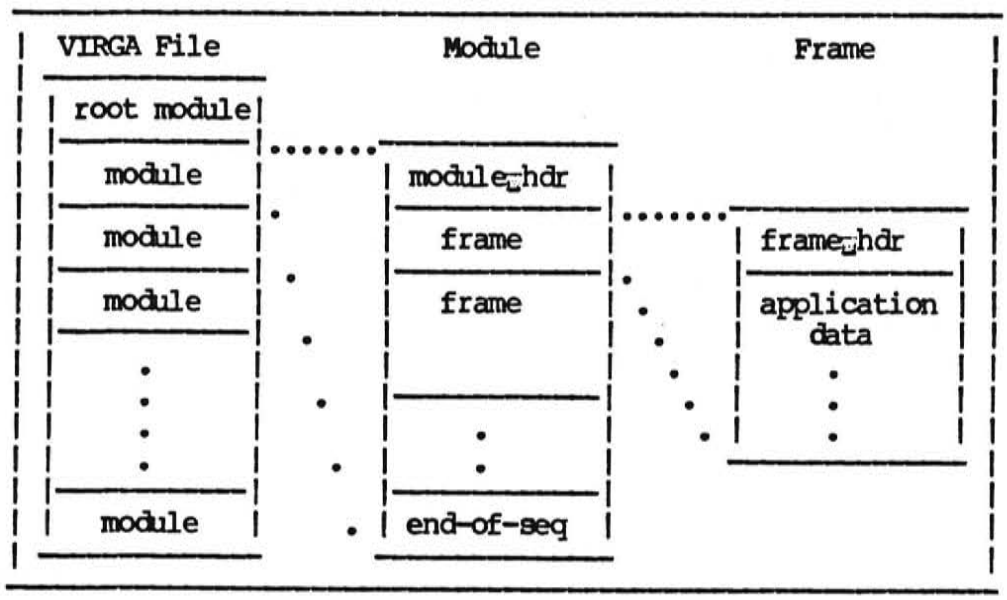


Figure 2 - 1 The VIRGA File Structure

A module contains a module header and frames. Each module has a default size of 8K, but the module size may be set to any multiple of 512 bytes. VIRGA files are usually created so that the Contiguous Allocation Factor (CAF) is equivalent to the module size. The default size makes it easy for VIRGA to read the module from the disk; every disk access call reads or writes a whole module. (See Section 2.1.5.1 for more information about module size.)

A frame contains a frame header and data arranged according to your schema design. A frame can be any size that fits within a single module. If a frame is so large that it cannot fit into the default module size, VIRGA creates a module that is a multiple of the basic module size. Note that the module size must always be less than 64K; the module size cannot be equal to 64K. (See Section 2.1.5.2 for more information about frame size.)

A VIRGA search routine, when called, looks for a frame. The frame is in a module that is either in a memory buffer or on the hard disk. If it is on the hard disk, VIRGA reads the module containing the desired frame into a buffer.
Note that a VIRGA data module is a dynamic database structure that is controlled by VIRGA routines. The meaning of the word "module" here is entirely different from its use in the phrase, "PLM source code module."

2.1.1 Module and Frame Types

There are several types of frames and modules. Typing allows some "sorting" of the data in the file. Different module types contain specified frame types. Each module type can contain one or more frame types, but each frame type can belong to only one module type.

Figure 2 - 2 below shows the module types and respective frame types of the first design for the xxxx.CELL file.

MODULE TYPES:	FRAME TYPES:
reserved for VIRGA	
SAMM_EMPTY	SAMF_EMPTY
SAMM_CONTROL	SAMF_MOD_INV SAMF_FRAMES_INV SAMF_FRAME_INV SAMF_CELL_DIRECTORY
for the CHIPMASTER database	
SIDM_TEMPLATE	SLTF_TEMPLATE SIDF_CELL_INFO
SIDM_INSTANCE	SIDF_INSTANCE
SIDM_GEOMETRY	SIDF_MASK_GEOMETRY SIDF_NON_MASK_GEOMETRY SIDF_COMMENT

Figure 2 - 2 MAX Modules and Frames

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

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

2.1.2 Reserved Control Modules and Frames

The first two module types and first five frame types, as shown in Figure 2 - 2 above, are reserved by VIRGA for its own use.

Module type 0 and frame type 0 signify empty modules/frames. These are place-holders.

Module type 1 contains VIRGA control information. This information consists of four types of frames:

Module inventory
Frame type inventory
Frame inventories
Cell libraries directory

Note that VIRGA uses the root for some control information. Note also that to avoid crossing CAF boundaries, the first module (a control module) is forced to have a size equal to the module-size minus the root-size.

The control frames (the inventories) are reserved for internal VIRGA use, however, you can reference them when accessing portions of a frame or module header.

2.1.3 Restriction

All module and frame types are specified by one-byte integers, thus there can be no more than about 255 module or frame types. There can be a total of about 3500 modules and a total of about 16,000 frames of a given type. However, you must estimate some of these sizes to help VIRGA efficiently direct frame and module access. (See Section 4.1.5 "Buffer Control Procedures.")

2.1.4 Required PLM Structures

VIRGA files have the same basic structure whether they are used for GATEMASTER, CHIPMASTER, CELLMASTER, or BOARDMASTER databases. The specific database schemas are different for each of these applications, but the data organization is the same.

2.1.4.1 PLM File Description

The root structure is defined by the SAM_ROOT_DEF declaration in the file SAMINT.ELT.

The file descriptor definition, shown in Figure 2 - 4 below, describes the specific module and frame structure of a new VIRGA file. SAM_CREATE is the only routine that uses this file description. The root structure in GM_10 is created by BASE, SLIDE, and LED. The root structure in CHIPMASTER is created by STREAMIN and MAX.

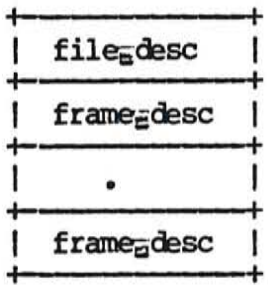


Figure 2 - 3 File Descriptor Table

All fields in the structure must be filled in by the caller. The file descriptor definition has the format shown in Figure 2 - 4 below.

```
SAM_FILE_DESC_DEF LITERALLY
  'MODULE_SIZE      WORD,  /* in bytes */
  MOD_INV_SIZE      WORD,  /* initial # modules allowed */
  MOD_INV_EXT       WORD,  /* extension size */
  NUM_FRAME_TYPES   WORD', /* # of frame_desc's to follow */
```

Figure 2 - 4 File Descriptor Definition

MODULE_SIZE defines both the CAF and the default module size for the file. If the configuration table (specified in the SAM_INITIATE call) has the field CAF = 0, the definition

for the CAF specified by MODULE_SIZE is used. However, if the configuration table has the field CAF <> Ø, the CAF value specified in SAM_INITIATE is used instead.

MOD_INV_SIZE defines the number of entries in the initial module inventory.

MOD_INV_EXT defines the estimated increase in size required when the module inventory overflows. Since all inventories are dynamic, these numbers need not be exact, but are merely helpful hints to get VIRGA started off on the right foot.

NUM_FRAME_TYPES is the number of SAM_FRAME_DESC_DEF's that follow.

To estimate the size, use the following approximations:

MOD_INV_SIZE should be a rough estimate of the expected number of modules. You should specify a size of at least four.

MOD_INV_EXT should be a small number if you want to conserve table space and expect that modules will rarely be added. A larger number should be used if you expect modules to be added often and want to minimize the overhead of dynamically expanding the module inventory.

NUM_FRAME_TYPES should specify the number of types that you want to define in the SAM_FRAME_DESC_DEF blocks. Remember that the first four frame types are reserved for VIRGA itself, so the first frame-descriptor block you supply will specify frame type number 4. If, for example, you need 5 frame types, then NUM_FRAME_TYPES should be 5, and they will describe numbers 4 through 8.

The PLM code for the SAM_FRAME_DESC_DEF is shown in Figure 2 - 5 below.

```
SAM_FRAME_DESC_DEF LITERALLY
   'MOD_TYPE          BYTE,  /* module the frame goes to */
   INV_SIZE           WORD,  /* initial # frames allowed */
   INV_EXT_SIZE       WORD', /* extension size */
```

Figure 2 - 5 Frame Descriptor Definition

Each SAM_FRAME_DESC_DEF block specifies the module type into which the frame type should be placed.

The frame INV_SIZE and INV_EXT_SIZE are similar to those for the module inventory described above except that here the numbers apply to the expected number of frames of this type.

For example, the schema for MAX describes a GEOMETRY frame type. If you expect that there will be relatively few frames of that type, you might specify an inv_size of 10 and an inv_ext_size of 2. However for a large hierarchical cell, you might specify 2500 for the inv_size, and 100 for the inv_ext_size. You should specify an inventory size of at least four.

2.1.4.2 PLM Module Description

All modules must begin with the module header format shown in Figure 2 - 6 below and that is defined in the SAMFIL.ELT file.

```
SAM_MODULE_HDR LITERALLY
   'OPCODE        BYTE,   /* always SAM_SCR_MODULE (06H) */
   HDR_SIZE       WORD,   /* always 10H */
   ID             WORD,   /* module ID */
   SIZE           WORD,   /* size of module */
   TYPE           BYTE,   /* module type */
   MARK           WORD,   /* offset to last deleted or added
                             frame */
   FREE_BYTES     WORD,   /* size of largest empty frame in
                             this module */
   BUSY_FRAMES    WORD,   /* number of frames used */
   DUMMY(2)       BYTE',  /* reserved for future use
                             (must be zero) */
```

Figure 2 - 6 Module Header Structure

2.1.4.3 PLM Frame Description

All frames must begin with the frame header format shown in Figure 2 - 7 below This frame header is defined in the SAMFIL.ELT file.

```
SAM_FRAME_HEADER LITERALLY
   'OPCODE        BYTE, /* always SAM_SCR_FRAME (05H) */
   HDR_SIZE       WORD, /* 1AH for 32-bit; 12H for 16-bit */
   SIZE           WORD, /* frame size */
   ID             WORD, /* frame ID */
   CELL_ID        WORD, /* ID of cell to which frame belongs,
                           if frame is in library */
   TYPE           BYTE, /* frame type */
   SAM_RECTANGLE';

SAM_RECTANGLE LITERALLY  /* 32-bit rectangular */
   '(X1,Y1)      DWORD,    /* recognition area   */
    (X2,Y2)      DWORD',
SAM_MIN_COORD LITERALLY '0',
SAM_MAX_COORD LITERALLY '0FFFF$FFFEH',
```

```
/* To recompile SAM for a 16-bit rectangle, change the */
/* previous rectangle declarations to the following:   */
/*                                                     */
/* SAM_RECTANGLE LITERALLY  /* 16-bit rectangular     */
/*    '(X1,Y1)      WORD,    /* recognition area       */
/*     (X2,Y2)      WORD',                             */
/* SAM_MIN_COORD LITERALLY '0',                        */
/* SAM_MAX_COORD LITERALLY '0FFFEH',                   */
```

Figure 2 - 7 Frame Header Structure

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

The CELL_ID is the ID of the cell library file to which the frame belongs. Otherwise, the CELL_ID is set to SAM_NIL_WORD (0FFFFH).

Every VIRGA frame has a data structure for a geometrical recognition area (RECTANGLE), whether it is needed or not. The recognition area is defined by either a 16-bit or 32-bit rectangle.

You modify the file SAMFIL.ELT as described in the note in

Figure 2 - 7

and in the comment in SAMFIL.ELT, then recompile SAM. GATEMASTER uses 16-bit rectangles; CHIPMASTER uses 32-bit rectangles.

If spatial (geometrical) data is not needed to define the object that the frame represents, the recognition area does not have to have "good" data. However, every frame must begin with the exact header shown, since VIRGA looks for

user-designed application data just below the fields reserved for the recognition rectangle coordinates.

2.1.5 Sizes of the File, Modules, and Frames

The following sections describe the relationship between the sizes of the VIRGA file, its modules, and its frames.

The VIRGA file itself can be any size. It must contain a root, which contains control information, and at least one module.

2.1.5.1 Module Size

Since the module is the block of information read from the disk into a VIRGA buffer, the module's size is determined by the file's CAF (Contiguous Allocation Factor). Thus, the default size for each module is 8K which is equivalent to 1 CAF.

Although you can create a module of any multiple of a block size (512 bytes), performance is enhanced by creating a module size that is an integer multiple of the CAF. For example, since the default size is 8K, then you might specify the larger module to be 16K or 24K.

You can also indirectly create a larger module by creating a frame larger than the module size. When you create a large frame, VIRGA creates a larger module that is a multiple of the default module size (e.g., 16K or 24K).

The larger module size decreases the number of disk accesses required, so performance is enhanced. However, when the size is so large that there is insufficient memory to hold all the modules needed at one time, they are swapped in and out which slows down execution time. Also, if memory is fragmented and VIRGA tries to read a large module, it may be unable to allocate enough memory to hold the module, especially when the module size approaches the size of the memory pool.

Some limited measurements and experience indicate that a reasonable module size is 8K or 16K bytes.

2.1.5.2 Frame Size

Frames can be any size. However, frame size should be at least an order of magnitude smaller than the module size to accomodate memory management within a module. For example, the MAX schema's largest frame is the geometry frame, which should be about 512 bytes.

2.2 VIRGA Cell Libraries

VIRGA can store several databases in one physical file. Each database is called a cell, and the collection of cells is a cell library. MAX uses cell libraries.

A cell library is a VIRGA file with one root and multiple databases. The structure of the cell libraries file is shown in Figure 2 - 8 below.

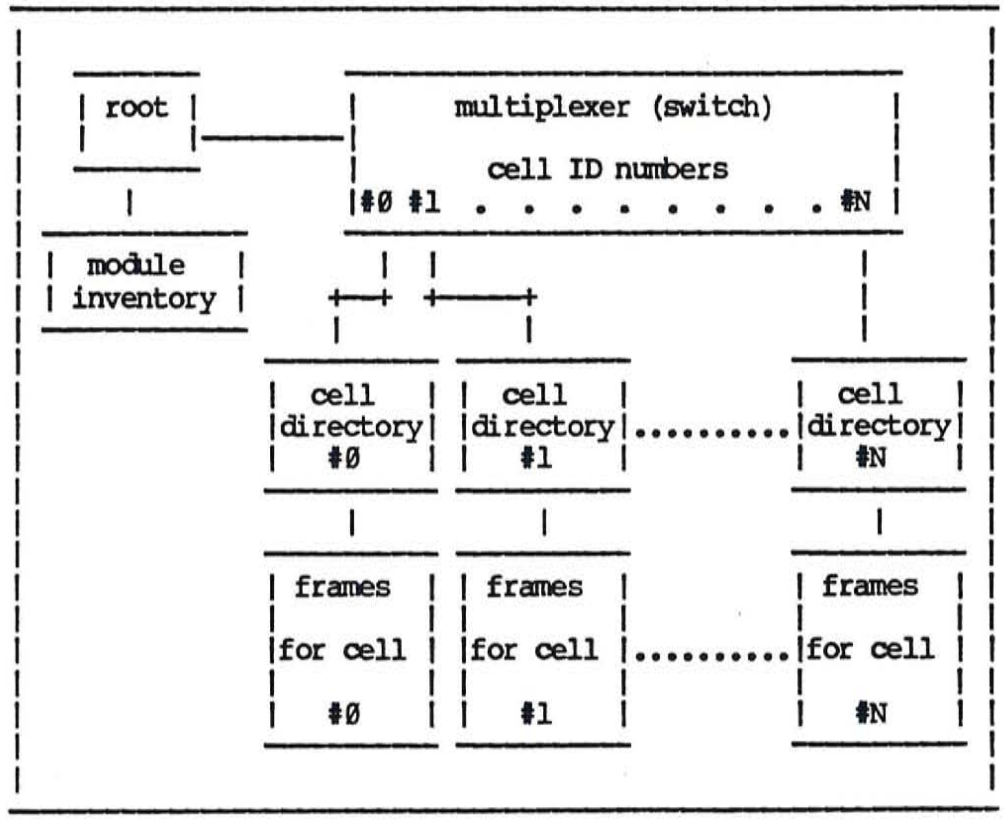


Figure 2 - 8 Cell Libraries File Structure

Cell libraries are generally read-only. To access a frame in a cell library, you switch the library multiplexer to point to the desired cell. You access the multiplexer by calling SAM_SWITCH_MUX and specifying the cell ID of the desired cell. Once you have made this call, you can make any desired calls to VIRGA to read frames as though this were an ordinary VIRGA file. All frames retrieved belong to the cell whose ID was specified in the last call to SAM_SWITCH_MUX.

If you want to access frames from a different cell, you must call SAM_SWITCH_MUX again with the different cell ID for that cell.

A cell library has one module inventory for the whole cell library, but it has a frame types inventory and a set of frame inventories for each cell. These inventories are stored in a special cell directory frame.

To add a cell to a library, you must call SAM_CREATE_CELL_DIR to create a cell directory frame. Then you add the data frames to the database.

Except for this process of adding a cell to a library, it is not possible to write into a cell library.

2.3 VIRGA Box Structure

Sometimes you might want to access data in a file based on the geometric location of the data in a two-dimensional space (such as the surface of a chip). You may specify to VIRGA that certain frame types represent geometric data, for example, in the MAX database. That is, you declare the mask geometry and non-mask geometry frames. VIRGA then creates a box index which is an array of rectangles. There is one rectangle for each allocated module that represents the bounding box of all data frames stored in that module.

You must set the SAM_RECTANGLE field of each geometric frame before adding it to the database. VIRGA then uses a heuristic algorithm to try to localize the data in each module (see Appendix D "The Bucket Manager"). VIRGA's goal is to ensure that data stored in one particular module corresponds to a particular region of the chip rather than to any location on the chip.

When you make the FIND_FIRST_IN_BOX and FIND_NEXT_IN_BOX calls, VIRGA uses the box index to rapidly reject modules whose bounding boxes fall outside the retrieval box. In this way, data for a particular rectangle on the surface of the chip may be retrieved without reading every module from the disk.

This functionality is especially useful for diplaying a {ZOOM}ed view of the chip or {SELECT}ing an object under the cursor.

2.4 VIRGA Name Index Tables

Each name index always has a frame with ID 0 (zero); it is the base index frame and contains slightly more information than the other index frames.

A name index must begin with a particular header (see SAMNAM.ELT). The index base frame comprises index frame 0.

The base header must begin as shown in Figure 2 - 9 below.

```
    /* index frame 0 continues with this structure */
SAM_INDEX_BASE_HDR  LITERALLY
    'SAM_INDEX_HDR,
    SAM_INDEX_DESC_DEF,
    FRAME_TYPE          BYTE,
    INDEX_FRAME_TYPE    BYTE,
    DUMMY0(14)          BYTE',
```

Figure 2 - 9 Index Base Structure

Each index frame starts with the index header after frame header as shown in Figure 2 - 10 below.

```
SAM_INDEX_HDR_1     LITERALLY
    'INDEX_OPCODE  BYTE,
    INDEX_HDR_SIZE WORD,
    SYM_STRUC_DEF,
    OFFSET_SYMTAB WORD,
    MAX_AVAIL     WORD',

SAM_INDEX_HDR_2     LITERALLY
    'DELTA_ENTRIES WORD,
    BASE_FLAGS    WORD,
    KEY2          WORD,
    OFFSET_BUF    WORD,
    LINK_FORWARD  WORD,
    LINK_BACK     WORD,
    DUMMY(17)     BYTE',

SAM_INDEX_HDR       LITERALLY
    'SAM_INDEX_HDR_1,
    SAM_INDEX_HDR_2',
```

Figure 2 - 10 Index Header Structures

Figure 2 - 11 below shows the CREATE_INDEX routine parameters.

```
SAM_INDEX_DESC_DEF      LITERALLY
    'NAME_LEN           WORD,
    NAME_OFFSET         WORD,
    NAME_LEN_OFFSET     WORD,
    KEY2_OFFSET         WORD,
    EXTRA_BYTES         WORD,
    FLAGS               WORD,
    INIT_ENTRIES        WORD,
    INCR_ENTRIES        WORD',
```

Figure 2 - 11 Index Descriptor Structure

In the parameter descriptions below, "offset" is always a byte-count from the top of the frame (so offset = 0 is the first byte of the frame_hdr, etc.):

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

Figure 2 - 12 Index Descriptor explained

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

If KEY2_OFFSET is set to SAM_NIL_WORD, then VIRGA assumes there is no key2, and the name itself constitutes the full index key. In that case, a "hashing" mechanism distributes the name index across eight index frames. This action speeds up name searches by a factor of eight.

If SAM_UPPER_CASE_BIT is set in the FLAGS word of the

descriptor, names added to the index will be mapped to upper-case. The SAM_CREATE_INDEX routine is invoked with a pointer to the index_desc[riptor] block, which is copied into the base index frame. All other index frames begin with the index_hdr, which contains enough information about the index structure that the base index is not usually needed.

The descriptor defines where in the frame the name, name_len, and key2 are to be found, some flags (of which only the upper_case flag is implemented), the initial size for the index (in number of entries), and the incremental size (number of entries for which space is reserved whenever the index overflows and needs to be enlarged).

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

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

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

Figure 2 - 13 shows the complete index header.

opcode	(skip)
hdr_size	
symtab_ptr	points to start of symtab (after SAM_FIND_INDEX_FRAME)
name_size	size of each name field, byte_count
entry_size	size of each entry, including name, byte_count
first_avail	index into symtab of next entry to be added = number of entries
max_entry	total number of entries for which space is currently reserved
mark	index into symtab of current entry (set by find procedures)
offset_symtab	byte-count from top of frame to start of symtab symtab_ptr = utl_increment_ptr (frame_ptr, frame.offset_symtab);
max_avail	maximum number of entries allowed (due to 56K limit)
delta_entries	number of entries to be added to max_entry when necessary to enlarge symtab
base_flags	flags from base index frame (including upper_case flag)
key2	value of KEY2 for this frame; currently, always same as frame ID
offset_buf	byte-count from top of frame to buffer of size name_size bytes
link_forward	future use: to expand index into more than one frame
link_back	future use: to expand index into more than one frame

Figure 2 - 13 Index Header Diagram

A description of the symtab is shown in Figure 2 - 14 below.

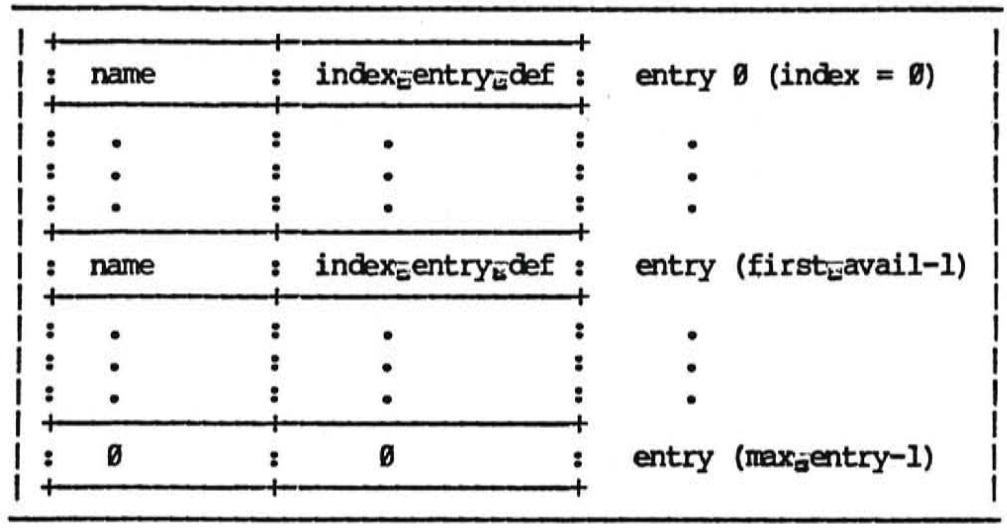


Figure 2 - 14 Symtab Diagram

A map of the index-entry definition is shown in Figure 2 - 15 below.

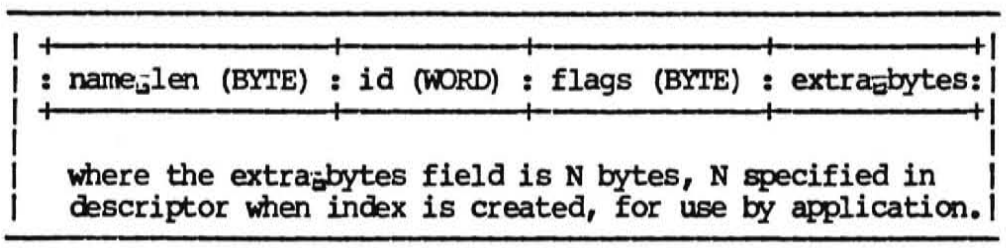


Figure 2 - 15 Index Entries Diagram

The index for a given KEY2 is limited to one frame, which means it is limited to somewhat less than 56K (56K less sizes of module hdr, frame hdr, index hdr, module's end of sequence byte, and perhaps minimum frame size). If there are no extra bytes in the index entry def and the name size is 16, this leaves room for about 3000 index entries (see Chapter 3 "Theory of Operation" for more size discussion).

2.5 VIRGA Inventory Tables

This section is a description of VIRGA's internal inventory tables. You should NOT write any code depending on these structures since a significant advantage of VIRGA is its application-independence.

For each VIRGA file, VIRGA maintains three inventories to speed access to individual modules and frames.

- o Module Inventory—each file has an inventory of all its modules.
- o Frame Inventory—each frame-type has an inventory of its kind of frames and what modules they are in.
- o Frame-Type Inventory—a table exists that has one entry for each frame-type, and has pointers to each frame-type's frame-inventory.

There is one entry in the frame type inventory for each frame type. This type is fixed at the time the file was created. For each frame type, there is an inventory of the frames belonging to that type. The number of entries in the frame inventory varies depending on how many frames have been created and deleted for that particular type.

In addition to the three inventories described above, there is one Global Buffer Table that is shared by all files opened by VIRGA. This Global Buffer Table contains information on all memory buffers used for all files.

The first three inventories are file-resident and permanent. The last inventory (global buffer) is created when VIRGA is initiated and discarded when VIRGA is terminated.

2.5.1 A Drawing of Internal VIRGA Tables

Figure 2 - 16 below shows a drawing of these internal tables and their inter-relationships. This information is not necessary for application use of VIRGA, but is intended for your clarification.

Note that some structures, shown under the words "per file," exist in every VIRGA file. In contrast, the structures, shown under the word "global," only exist while VIRGA is running. Only one set of global structures exists during a VIRGA session, regardless of the number of open files.

Figure 2 - 16 VIRGAs Internal Tables

2.5.2 The Module Inventory

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

the buffer manager (SAMBUFR, SAM_READ_MODULE, and SAM_WRITE_MODULE) uses the module inventory to read/write a module from/to disk. In particular, the module inventory has a disk marker that specifies where each module is located on the disk. The buffer manager uses the module inventory to locate a module: if the module is on the disk, the buffer manager writes the module to the VIRGA buffers.

The SAM_ALLOCATE_FRAME routine uses the module inventory to decide whether or not a module has sufficient space for a new frame without having to read from disk.

The module inventory (per file) contains an entry for (at least) every module in the file. The module inventory structure is shown in Figure 2 - 17 below and each field is discussed below the figure.

type	disk_marker	size	free_bytes	buf_num
(byte)	(dword)	(word)	(word)	(word)
•	•	•	•	•
•	•	•	•	•
•	•	•	•	•
type	disk_marker	size	free_bytes	buf_num

Figure 2 - 17 The Module Inventory for a File

type—is a byte that specifies the module type (ØFFH [SAM_NIL_BYTE] if the module does not exist).

disk_marker—is a dword that specifies the start of the module on disk as a byte offset from the start of the file.

size—is a word that specifies the module size.

free_bytes—is a word that specifies the size of the largest empty frame in the module.

buf_num—is a word that specifies the number of the memory buffer in which the module currently resides (ØFFFFH [SAM_NIL_WORD] if the module is not in memory).

2.5.3 The Frame Types Inventory

For each file there is another table, the Frame Types Inventory, which has one entry for each frame type. The Frame Types Inventory finds the associated frame inventory for a given frame type.

The structure of the Frame Type Inventory is the same as the Module Inventory shown in Figure 2 - 17 above. The fields and their meanings are shown below.

mod_type—is a word that specifies the module type.

inv_size—is a word that specifies the current size of the module inventory.

inv_ext_size—is a word that specifies the size by which it will be expanded when it overflows.

inv_ptr—is a pointer in memory to the frame inventory for that frame type that is initialized when the file is opened.

num_ids—is a word that specifies the total number of frame IDs (of that frame type) allocated so far.

inv_mod_id—is a word that specifies the ID of the module where the frame inventory is located.

inv_mod_offset—is a word that specifies the location of the frame containing this frame type's frame inventory.

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

flags—is a byte containing bit-flags. Currently, SAM_BOX_INDEX_FLAG (01H) indicates that this is a geometric frame type.

min_module_num—is a word that specifies the ID of the first module containing frame of this type.

max_module_num—is a word that specifies the ID of the last module containing frames of this type.

2.5.4 The Frame Inventory

Each frame type has an inventory that contains at least one entry for every frame of that type. Every frame type has this inventory.

The structure of the frame inventory is shown in Figure 2 - 18 below and each field is discussed below the figure.

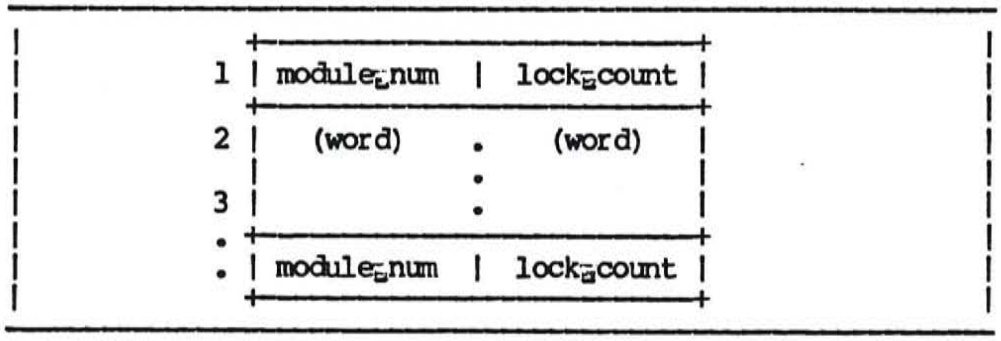


Figure 2 - 18 The Frame Inventory

module_num—is a word that specifies which module contains this frame (0FFFFH [SAM_NIL_WORD] if nomodule is found).

lock_count—is a word that specifies the number of frame locks that have been issued for this frame.

2.5.5 The Buffer Inventory

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

The Buffer Inventory has the structure shown in Figure 2 - 19 below and each field is discussed below the figure.

buf_ptr	file_num	mod_num	LRU counter	lock	status_flags
.	(byte)	(word)	(dword)	(word)	.
.	.	.	.	.	.
.	.	.	.	.	.
.	.	.	.	.	.
buf_ptr	file_num	mod_num	LRU counter	lock	status_flags

Figure 2 - 19 The Buffer Inventory

buf_ptr—points to the in-memory buffer holding the module (NIL if none).

file_num—is a byte that specifies which file is being accessed.

mod_num—is a word that specifies which module of that file occupies the buffer.

LRU counter—is a dword that specifies the buffer replacement algorithm (see Section 2.4.2 "The Buffer Aging Algorithm"). Note that all files contribute to the buffer inventory, and the LRU counter operates over all buffers. This means that a read request for a module in one file might cause a module of a different file to be swapped out.

lock—is a word that counts how many locks have been nested on this module (see the discussion on locking and unlocking in Section 4.1 "Buffer Control Procedures").

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

3 THEORY OF OPERATION

This chapter contains a discussion of how VIRGA operates; in particular, how VIRGA uses the following structures discussed in the previous chapter:

- o VIRGA files
- o name indexing
- o inventory tables
- o buffer control
- o transaction procedures
- o search procedures
- o enhancement considerations

Although the you do not need to know the internal structures of VIRGA to use VIRGA routines, the knowledge might be usefulto understand how VIRGA does its job.

3.1 VIRGA Files

When you create a file, you specify all the frame types and also the module types to which the frame types belong. In this way you have the flexibility to impose the organization desired for a particular application.

The decision about which module and frame types to create is a trade-off between space optimization, and ease and speed of access. Space optimization is a concern both in the amount of memory consumed and in the amount of hard disk space required for the permanent file.

> That is, if every frame were assigned to only one (the same) module type, a minimum of space would be wasted, but it would not be possible to read only one frame type without getting all the other frame types interspersed.
>
> Alternately, if each frame were assigned to a separate module, it would be simple to read only frames of one type, but the modules would be only partly full. In this

situation much space would be wasted.

For example, suppose module type 7 contains only frames of type 12, and module type 9 contains only frames of type 6. If there is only one frame of type 12 and one of type 6, then there are two modules, one of type 9 and one of type 7, each of which contains only a single frame.

The module and frame type numbers are simply one byte positive integers starting at Ø and advancing up to, but not exceeding, 254. You should declare the frame types and module types as literals in your code, so that the assignment may be changed by a recompilation.

You references the control modules as literals:

```
SAM_FIRST_MODULE    LITERALLY   '1',
SAM_FIRST_FRAME     LITERALLY   '3',
```

You should, for example, use SAM_FIRST_FRAME+n and SAM_FIRST_MODULE+n, where n > Ø.

Access routines:

Direct access
Sequential access
Module access

Direct access routines allow you to find the specific frame desired.

The sequential access routines allow you to step through all the frames of a given type. FIND_FIRST locates the "first" frame of the specified type; FIND_NEXT locates the "next" one, if any. The sequence is the order in which the frames exist in the file, which does not have any particular connection. Note that some modules might contain frames of more than one type in no particular order.

Since some other calls (like UPDATE_FRAME) may cause VIRGA to change the order of frames in the file, you should not mix sequential calls with calls that might change the order. If it is necessary to modify frames between the sequential requests, you must find the next frame, keep its ID, modify the previous frame, then make a (direct) FIND_FRAME request for the ID of the "next frame," etc.

Note that any interspersed calls (even for other module types or other files) could cause the module of interest to be swapped out, so the pointers would no longer be valid.

Module access routines allow you to read all the modules of a specific type into memory, thus speeding up the subsequent access to frames in those modules.

3.2 Name Indexing

Name index routines allow you to find the frame by name; the name indexing routines search the index or indices of the appropriate file or files to find the entry for the specified name, and return a pointer to that entry. That entry contains the full, exact name, NAME_LEN, frame ID, and (optionally) extra bytes of application-defined information.

VIRGA allows you to specify that particular types of frames have names. VIRGA automatically maintains a name index for each frame type declared. The name index is simply an array of (name, ID) pairs provided so that a frame with a particular name may be rapidly retrieved.

The name index tables are organized by frame type. That is, you create an index frame type to contain the names and IDs of specified frames of a specified frame type.

If you wish to use an index, you must create the index via SAM_CREATE_INDEX. Later, you may delete the index via SAM_DELETE_INDEX. These routines may be invoked at any time. For instance, you can call SAM_CREATE_INDEX before adding any frames, or you can add frames to a database without an index, invoke CREATE_INDEX, then add more frames. After an index is created, VIRGA will update it automatically.

All the VIRGA requests other than index routines operate successfully whether there is an index or not. If there is a name index, then SAM_ADD_FRAME, SAM_UPDATE_FRAME, and SAM_DELETE_FRAME take the index into account. SAM_ADD_FRAME will add the new frame's name to the index; if the name is a duplicate, it will still be added to the index, but a negative status will be returned. SAM_DELETE_FRAME will remove the frame's name from the index; if the name is not in the index, the frame will still be deleted, but negative status will be returned. SAM_UPDATE_FRAME first checks if the name and key2 (see below) are unchanged; if either has changed, the update request will be refused.

The name index gives you a second method of direct access to frames. The first method is to call SAM_FIND_FRAME, which uses the specified frame ID and frame type to find the frame. The specified frame, if it exists, is always found. A maximum of one disk access may be required to get the module with the frame.

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

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

The general approach to finding a frame by name is to find the index frame type (from the frame types inventory), find the index_frame for desired key2, and then search that index. This involves gaining access to the disk once, and results in the ID of the desired frame. That frame may then be obtained via SAM_FIND_FRAME which could involve gaining access to the disk more than once.

3.3 Inventory Tables

The following description is best understood by examining the look-up process IN the drawing of VIRGA's internal tables shown in Figure 2 - 16 above in Chapter 2.

SAM_FIND_FRAME operates as follows:

The routine starts with root_ptr and locates the root table. There, it gets frames_ptr and finds the Frame-Types Inventory. The programmer-specified frame type is used to index into this table, which yields a pointer to the correct Frame Inventory.

The programmer-specified frame ID is used to index into the frame inventory and find the module ID. Using the root.module_inv_ptr, the routine finds the module inventory, and using the module ID, indexes to find the buffer number.

Using the already known global_ptr (in the root), SAM_FIND_FRAME finds the buffer inventory and uses buf_num to look up buffer_ptr, which points to the necessary module in memory. (Note: if buf_num = SAM_NIL_WORD, the module is not in memory so the routine reads from disk.)

Finally, SAM_FIND_FRAME performs a linear search to find the desired frame within the module in memory.

3.4 Buffer Control

The buffer control procedures allow you to control VIRGA's buffer handling. However, it is quite possible to use VIRGA without ever using any of the buffer control calls. Straightforward use of FIND_FRAME, ADD_FRAME, UPDATE_FRAME, and DELETE_FRAME will take care of everything for you. However, you might want to directly control memory buffering to improve performance or to simplify memory-dependent tasks.

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

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

When data is requested and the module containing that data is not in memory, VIRGA gets a buffer from the system and reads the module from disk into that buffer. VIRGA is allowed a maximum number of buffers (a MAX and VIRGA configuration parameter). When that maximum is reached or there is insufficient memory, no more buffers are allocated.

There is also a configurable total amount of memory that VIRGA will use. This limit is in addition to the limit on the number of buffers, which may be of different sizes.

3.4.1 Module Swapping

When the maximum number of buffers is reached or insufficient memory exists for a new module, one of the previously read modules (the least accessed one) is removed from memory. Its buffer is deallocated (returned to the system), and the buffer for the new module is allocated. If necessary, multiple old buffers are removed.

When a buffer is removed, VIRGA checks its dirty flag. If the dirty flag is set (true), it means the data has been changed and the contents must be written back to the disk. Thus, when a buffer is removed that has its dirty flag set, the module is first written back to disk, then the buffer is removed from memory. The dirty flag is set automatically by the procedures that write data (see Chapter 4), or it can be explicitly set by you.

A module buffer may be locked in memory by an explicit programmer request. A locked buffer cannot be removed from memory except upon a close file or terminate request.

Modules may be removed from memory automatically or by explicit programmer request. They are automatically removed when there is a request for a new buffer and insufficient resources exist (for example, too many buffers or insufficient memory), when you close the file, or when you terminate VIRGA.

3.4.2 The Buffer Aging Algorithm

When a buffer needs to be removed from memory, VIRGA chooses which buffer to remove according to an LRU (Least Recently Used) algorithm. That is, the first buffer removed is the one with the longest "age."

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

In the aging algorithm, when any buffer is accessed using VIRGA, that buffer's age (LRU counter) is reset. One global counter is incremented for each access, and each buffer maintains its own local counter. When a buffer is accessed, its own counter is set to the global counter's value. Then, the effective age of a buffer is given by the global counter less the buffer's local counter. The "oldest" buffer is determined by choosing the one with the smallest local counter.

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

3.4.3 Using Pointers and Locks

FIND routines in VIRGA return pointers to the data that is in the module buffer. The data is not copied out of the buffer.

You should consider this pointer temporary, since the buffer may be swapped by any later VIRGA request. Because of this, you should consider this data as read-only.

Specific write requests guarantee that the data is written, eventually, to the correct place on disk. However, if the calling program uses the pointer and modifies the data in the buffer, the data might not be written to disk unless certain precautions are taken.

To directly modify the data in the buffer and guarantee that it will be written to the disk, you should first declare your intention to modify the data in the frame, and then write in the data. Declaring your intention to modify the frame has two purposes:

(1) It allows the transaction management system to keep the old frame (see Section 3.5).

(2) It sets the dirty flag which ensures that the data will be written to disk before VIRGA allow the buffer to be re-used.

Note that it is not permissible to directly modify any of the data in the SAM_FRAME_HDR. This includes the ID, SIZE, and RECTANGLE fields. To modify these data, use SAM_UPDATE_FRAME, SAM_GROW_FRAME, or SAM_MOVE_FRAME. It is never possible to modify the ID or the NAME of any frame.

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

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

Since the module is the unit that is actually swapped to and from disk, the module is the only unit that is really locked. However, to simulate frame locks, the following approach is used. Each time a frame is locked, the corresponding module's lock counter is incremented. Also, each frame (in the frame inventory) has a lock count which is incremented. An unlock request on a frame will decrement the module counter and decrement the frame's lock count; however, if the frame was not locked, the lock counter will not be decremented (and an error will be returned).

Any module with a non-zero lock counter is considered locked. However, one additional flag is maintained to indicate an actual module lock.

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

There is a simple rule that must be followed when using VIRGA:

> If you have retrieved a frame and it is not locked, then you must assume that the frame will be swapped out by any subsequent call to VIRGA or to any subsystem (such as MEM or SID) that itself calls VIRGA.

This rule is the safest assumption.

The problem is that if you violate this assumption, your program may work in many cases, but there is no guarantee that your program will not crash randomly and unpredictably.

3.5 Transaction Procedures

VIRGA provides a simple transaction management system that allows changes to the frames in a single database to be recorded in a scratch file. When a transaction UNDO is requested, the previous versions of all changed frames are restored to the database, while the modified versions are stored in the scratch file. Saving all the frames allows you to UNDO an UNDO.

A transaction can only affect one database. Calls like OPEN_CELL, CLOSE_CELL, or MAKE_CELL cannot be "UNDOne." You can copy data from one cell to another, however, since you only need to declare a transaction for the database you want to write into. Also note that any changes to in-memory data structures will not be undone automatically - you will have to handle this yourself.

When an UNDO occurs, the old copies of the frames are written back into the database in place of the existing frames. This means that none of the modified frames may be locked at the time an UNDO occurs.

DO NOT LEAVE MODIFIED FRAMES LOCKED DURING AN UNDO!

3.5.1 SAM_START_TRANSACTION

SAM_START_TRANSACTION begins a transaction on a database.

SYNTAX

SAM_START_TRANSACTION (ROOT_PTR)

FUNCTION

There is a configuration item in /PROJECT VIR_CONFIG called TRANSACTION_ENABLE. If TRANSACTION_ENABLE is set to ØFFH (true), a scratch transaction file is created when SAM is initialized. The default setting of TRANSACTION_ENABLE is TRUE.

From the time the transaction is started by SAM_START_TRANSACTION, any call to one of the following routines specifying the same ROOT_PTR will cause the previous state of the affected frame to be stored in the scratch file.

SAM_ADD_FRAME, SAM_MAKE_FRAME, SAM_ALLOCATE_FRAME (creating a fram
SAM_UPDATE_FRAME, SAM_MOVE_FRAME, SAM_GROW_FRAME, SAM_DIRTY_FRAME (modifying a fra

SAM_DELETE_FRAME (deleting a f

NOTE

SAM_START_TRANSACTION will fail if the scratch file does not exist (i.e. TRANSACTION_ENABLE was false).

3.5.2 SAM_UNDO_TRANSACTION

SAM_UNDO_TRANSACTION "undoes" or reverses the effects of changes made to a database.

SYNTAX

```
SAM_UNDO_TRANSACTION (ROOT_PTR)
```

FUNCTION

SAM_UNDO_TRANSACTION exchanges the frames in the scratch file with the frames in the database.

3.5.3 SAM_GET_TRANSACTION_LOG

SAM_GET_TRANSACTION_LOG lists frames that have been affected by the current transaction.

SYNTAX

```
SAM_GET_TRANSACTION_LOG (ROOT_PTR,
                         @NEW_FRAME_COUNT,
                         @NEW_FRAME_LIST_PTR,
                         @UPD_FRAME_COUNT,
                         @UPD_FRAME_LIST_PTR)
```

FUNCTION

If this procedure is executed BEFORE an UNDO, the NEW_FRAME_LIST returned is a list of the frames that have been added to the database by the current transaction; the UPD_FRAME_LIST is a list of the frames that have been modified by the current transaction.

If this procedure is executed AFTER a call to SAM_UNDO_TRANSACTION, the NEW_FRAME_LIST is a list of the frames that were deleted by the transaction and hence have been re-created by the UNDO operation. The UPD_FRAME_LIST is

the list of frames that were modified by the UNDO.

NOTE

Both NEW_FRAME_LIST and UPD_FRAME_LIST are arrays of SAM_FRAME_LIST_DEFs defined in /F0/ILE/ELT SAMFIL.ELT as follows:

```
DECLARE
        SAM_FRAME_LIST_DEF                  LITERALLY
       'TYPE              BYTE,
        SPARE             BYTE,
        ID                WORD';
```

DECLARE NEW_FRAME_LIST BASED NEW_FRAME_LIST_PTR (1) STRUCTURE (SAM_FRAME_LIST_DEF), UPD_FRAME_LIST BASED UPD_FRAME_LIST_PTR (1) STRUCTURE (SAM_FRAME_LIST_DEF),

3.5.4 SAM_FINISH_TRANSACTION

SAM_FINISH_TRANSACTION causes changes to no longer be saved in the scratch file.

SYNTAX

SAM_FINISH_TRANSACTION (ROOT_PTR)

FUNCTION

SAM_FINISH_TRANSACTION terminates recording of current transaction results.

NOTE

You must call FINISH_TRANSACTION before calling START_TRANSACTION on the current or any other database because you can only have one transaction outstanding at a time.

3.5.5 SAM_SET_DIRTY_MODULE

The transaction management system only works properly if you use the routines named above under the section on SAM_START_TRANSACTION for writing into the database. UNDO will not work properly if you "dirty" the frame and then call SAM_SET_DIRTY_MODULE. SAM_SET_DIRTY_MODULE must NOT be used if

you want UNDO to work properly.

DO NOT USE SAM_SET_DIRTY_MODULE DURING A TRANSACTION! USE SAM_DIRTY_FRAME INSTEAD!

3.5.6 SAM_DIRTY_FRAME

A functionality equivalent to that of SAM_SET_DIRTY_MODULE is provided by the new routine SAM_DIRTY_FRAME.

SAM_DIRTY_FRAME must be called BEFORE you modify the frame in order to give the transaction management system a chance to store the frame before you get it dirty. Calling SAM_DIRTY_FRAME after writing into the frame should not be done.

SYNTAX

SAM_DIRTY_FRAME (ROOT_PTR, MODULE_PTR, FRAME_PTR)

3.5.7 Example Transaction Operations

In the following example write operations are performed and undone. The write operation of modifying a frame is performed and undone to revert to an old copy of that frame. The write operation of creating a frame and its complement undo procedure, deletion of that frame is accomplished. Finally, a particular frame produced as the result of some modification is deleted, and the original version of that same frame is found.

```
PROGRAM USER

SAM_START_TRANSACTION
                                        edit existing polygon frame 1
                                        create a new polygon frame 3
                                        delete an old polygon frame 2
                                        select {CANCEL}
/* get back frame 3 (new frame) and frame 1 (modified frame) */
SAM_GET_TRANSACTION_LOG
/* erase frames 1 and 3 from screen */
/* restore database state */
SAM_UNDO_TRANSACTION
/* recover frame 2 (new frame) and frame 1 (modified frame) */
SAM_GET_TRANSACTION_LOG
```

```
/* draw frames 1 and 2 on screen */
DONE
```

3.6 SEARCH Procedures

Almost all VIRGA procedures act on one VIRGA file, which is specified by the file's root_ptr. However, the name SEARCH routines (progrmmer interface described later) act on a list of files.

When searching for a name, you may specify a name with wild cards. The wild cards available are those defined by the Daisy symtab facility as shown in Figure 3 - 1 below.

WILD_CARD		Meaning
*	:	match any string
?	:	match any single character

Figure 3 - 1 Meanings of Wild Cards

The symtab name_match procedure is invoked directly so the results from these VIRGA procedures will be as you expect from symtab.

It would be very simple to expand the code to use full regular expressions (as used in TEC). That has not been done partly because there is no current need for that, and partly because it would degrade performance.

4 VIRGA PROCEDURES

This chapter describes the VIRGA procedures and their parameters; the procedures are listed in the directory /F0/ILE/EXT as the following xxxx.EXT files:

SAMACC.EXT
SAMCELL.EXT
SAMCNFG.EXT
SAMDAT.EXT
SAMFIL.EXT
SAMHIL.EXT
SAMIDX.EXT
SAMNAM.EXT
SAMTRANS.EXT
SAMUTL.EXT

The data structures referred to in this chapter are defined as literals in the directory /F0/ILE/ELT as the following xxxx.ELT files:

SAMFIL.ELT
SAMINT.ELT
SAMNAM.ELT

The VIRGA error file literals are described in the file: /F0/ILE/ELT SAMERR.ELT.

These files define the interface between user application programs written in PL/M and the VIRGA subsystem. To use VIRGA, the user must reference some or all of these files via an $INCLUDE directive. To use VIRGA from a Pascal program, see Appendix E.

4.1 xxxx.EXT File Contents

The following sections describe, in general terms, the use of the procedures contained in each of the xxxx.EXT files.

4.1.1 Configuration Procedure

The VIRGA configuration procedure allows you to initialize your program and returns the values from the configuration parameters for the desired VIRGA file. This configuration procedure is described in SAMCNFG.EXT and named:

SAM_GET_CONFIG

4.1.2 File Manipulation Procedures

The VIRGA file manipulation procedures allow you to initiate and terminate VIRGA; create, open, and close files; and dump out VIRGA's state information. The file manipulation procedures are described in SAMFIL.EXT and listed below:

SAM_INITIATE
SAM_TERMINATE
SAM_CREATE
SAM_CREATE_SHELL_PATH
SAM_OPEN
SAM_OPEN_SHELL_PATH
SAM_CLOSE
SAM_DUMP_GLOBAL

4.1.3 Data Access Procedures

The VIRGA data access procedures allow you to access either frames (which contain data) or modules (which contain frames). The data access procedures are all described in SAMDAT.EXT. There are three ways to access data:

Direct access	(by frames)
Sequential access	(by frames)
Module access	(by modules)

The direct access routines are listed below:

```
SAM FIND FRAME
SAM ADD FRAME
SAM MAKE FRAME
SAM UPDATE FRAME
SAM MOVE FRAME
SAM GROW FRAME
SAM DELETE FRAME
SAM DELETE BY TYPE
SAM DIRTY FRAME
```

The sequential access routines, listed below, are used whenever all members of a given type need to be processed.

```
SAM FIND FIRST
SAM FIND NEXT
```

The module access routine is shown below:

```
SAM GET MODULES BY TYPE
```

4.1.4 Name Index Procedures

The name index procedures allow you to:

- o create or delete a name index for a VIRGA file
- o find a frame in the name index of one VIRGA file or search through the name indexes of all VIRGA files for the specified frame.

The create/delete name index procedures are described in the file SAMIDX.EXT and listed below:

```
SAM CREATE INDEX
SAM DELETE INDEX
SAM CREATE BOX INDEX
```

The find/search name index procedures are described in the file SAMNAM.EXT and listed below:

```
SAM FIND INDEX FRAME
SAM FIRST NAME
SAM NEXT NAME
SAM SEARCH FIRST NAME
SAM SEARCH NEXT NAME
```

4.1.5 Buffer Control Procedures

VIRGA controls the buffers for you; however, the buffer control procedures allow you to free up memory and control the buffers yourself. The buffer control procedures are described in SAMACC.EXT and listed below:

```
SAM FLUSH
SAM FLUSH AND DISPOSE
SAM FLUSH ALL
SAM DISPOSE ALL
SAM ALLOCATE MODULE
SAM ALLOCATE FRAME
SAM DISPOSE MODULE
SAM DISPOSE MODULES BY TYPE
SAM LOCK FRAME
SAM UNLOCK FRAME
SAM LOCK MODULE
SAM UNLOCK MODULE
SAM LOCK MODULES BY TYPE
SAM UNLOCK MODULES BY TYPE
SAM SET DIRTY MODULE
SAM CLEAR DIRTY MODULE
```

4.1.6 Utility Procedures

The VIRGA utility procedures allow you to retrieve various state information from VIRGA's internal data structures. The utility procedures are described in the file SAMUTL.EXT and listed below:

```
SAM GET PATHNAME
SAM MAX ID
SAM INQUIRE FRAME ID
SAM NUM FRAMES
SAM GET TIMESTAMP
SAM GET AFT
SAM ADD FRAME TYPE
SAM ADD NEW FRAME TYPE
SAM NUM LOCKS
```

4.1.7 Higher Level Procedures

The higher level (i.e., smarter) procedures allow you to locate the frames within a specified rectangle (box). The higher level procedures are described in the file SAMHIL.EXT and listed below:

```
SAM FIND FRAMES IN WINDOW
SAM FIND FIRST FRAME IN BOX
SAM FIND NEXT FRAME IN BOX
```

The routine SAM FIND FRAMES IN WINDOW has been provided only to maintain backward compatibility.

4.2 General Information About Procedures

All VIRGA procedures return an INTEGER status. The integer could be a value or an indication of the routine's success. If an error occurs, the procedure returns a negative interger.

Each procedure, when actually invoked in a program, must be preceded by the integer STATUS variable:

```
STATUS =
```

Thus each procedure command listed in this section must be preceded by the STATUS variable as shown:

```
STATUS = SAM_ROUTINE(PARAMETERS)
```

For example:

```
STATUS = SAM_FIND_FRAME (ROOT_PTR, @MODULE_PTR, @FRAME_PTR,
         FRAME_TYPE, FRAME_ID);
```

However, in this document, the "STATUS" return has been dropped for simplicity of reading.

Each procedure contains one or more input parameters, output parameters, or bidirectional parameters.

Input parameters require that you enter a value.

Output parameters return a value to you.

Bidirectional parameters both (1) require that you enter a value and (2) return a different value to you. Bidirectional parameters are used most often in NEXT routines, which are, in turn, used after FIRST routines.

For example, you use SAM_FIND_FIRST to locate the first incidence in a VIRGA file of a frame of a specified frame type. For SAM_FIND_FIRST, you specify as inputs:

- o the pointer to the root (ROOT_PTR) and
- o the number of the frame type (FRAME_TYPE).

SAM_FIND_FIRST locates the first frame it finds of the specified type in the specified file and returns as outputs:

- o the address of the frame (@FRAME_PTR) and
- o the address of the module (@MODULE_PTR).

Then, you use SAM_FIND_NEXT to locate the next incidence in the same VIRGA file of a frame of the same frame type. That

is, for SAM_FIND_NEXT, you specify as inputs:

- o the same (previous) file root (ROOT_PTR),
- o the address of the previously located frame (FRAME_PTR from the SAM_FIND_FIRST output), and
- o the address of the previously located module (MODULE_PTR from the SAM_FIND_FIRST output).

SAM_FIND_NEXT locates the next frame it finds of the specified frame type and returns as outputs:

- o the address of the next frame (the same FRAME_PTR) and
- o the address of the next module (the same MODULE_PTR).

Note that the output and bidirectional procedure parameters are always preceded by the "at sign" (@). However, in the procedures listed in the xxxx.EXT files, these same parameters do not contain the "at sign" ane are followed by "_PTR". For example:

@FRAME_PTR	is equivalent to	FRAME_PTR_PTR
@MAX_ID	is equivalent to	MAX_ID_PTR

The following parameter descriptions are valid for all access routines:

ROOT_PTR is the pointer returned from the OPEN or CREATE requests; it specifies the root of the VIRGA file to be accessed.

@MODULE_PTR returns a pointer to the address of the module in which the desired frame resides.

@FRAME_PTR returns a pointer to the address of the desired frame.

FRAME_TYPE is the number of the desired frame type.

FRAME_ID is the identification number of the desired frame.

For example, if a frame is added, the new frame's name and ID are added to the name index for that frame type.

4.3 A Complete List of VIRGA Procedures

All the VIRGA procedures available to the user are described on the following pages; they are listed in alphabetical order as shown in Figure 4 - 1 below.

SAM_ADD_FRAME	SAM_GET_AFT
SAM_ADD_FRAME_TYPE	SAM_GET_CONFIG
SAM_ADD_NEW_FRAME_TYPE	SAM_GET_MODULES_BY_TYPE
SAM_ALLOCATE_FRAME	SAM_GET_PATHNAME
SAM_ALLOCATE_MODULE	SAM_GET_TIMESTAMP
SAM_CLEAR_DIRTY_MODULE	SAM_GET_TRANSACTION_LOG
SAM_CLOSE	SAM_GROW_FRAME
SAM_CREATE	SAM_INITIATE
SAM_CREATE_BOX_INDEX	SAM_INQUIRE_FRAME_ID
SAM_CREATE_CELL_DIR	SAM_LOCK_FRAME
SAM_CREATE_INDEX	SAM_LOCK_MODULE
SAM_CREATE_SHELL_PATH	SAM_LOCK_MODULES_BY_TYPE
SAM_DELETE_BY_TYPE	SAM_MAKE_FRAME
SAM_DELETE_FRAME	SAM_MAX_ID
SAM_DELETE_INDEX	SAM_MOVE_FRAME
SAM_DIRTY_FRAME	SAM_NEXT_FRAME
SAM_DISPOSE_ALL	SAM_NUM_FRAMES
SAM_DISPOSE_MODULE	SAM_NUM_LOCKS
SAM_DISPOSE_MODULES_BY_TYPE	SAM_OPEN
SAM_DUMP_GLOBAL	SAM_OPEN_SHELL_PATH
SAM_FIND_FIRST	SAM_SEARCH_FIRST_NAME
SAM_FIND_FIRST_FRAME_IN_BOX	SAM_SEARCH_NEXT_NAME
SAM_FIND_FRAME	SAM_SET_DIRTY_MODULE
SAM_FIND_FRAMES_IN_WINDOW	SAM_SET_INDEX_TYPE
SAM_FIND_INDEX_FRAME	SAM_START_TRANSACTION
SAM_FIND_NEXT	SAM_SWITCH_MUX
SAM_FIND_NEXT_FRAME_IN_BOX	SAM_TERMINATE
SAM_FINISH_TRANSACTION	SAM_UNDO_TRANSACTION
SAM_FIRST_NAME	SAM_UNLOCK_FRAME
SAM_FLUSH	SAM_UNLOCK_MODULE
SAM_FLUSH_ALL	SAM_UNLOCK_MODULES_BY_TYPE
SAM_FLUSH_AND_DISPOSE	SAM_UPDATE_FRAME

Figure 4 - 1 All the VIRGA Procedures

4.3.1 SAM_ADD_FRAME

SAM_ADD_FRAME adds a new frame that contains new data to the file.

SYNTAX

```
SAM_ADD_FRAME (ROOT_PTR,
               @MODULE_PTR,
               @FRAME_PTR);
```

"ROOT_PTR" is the pointer returned from OPEN or CREATE requests; it specifies the file to be accessed.

"@MODULE_PTR" returns a pointer to the address of the module that contains the frame that has been added.

"@FRAME_PTR" (bidirectional) both inputs the address of the user-prepared data and returns a pointer to the address of the frame after it has been added to the file.

FUNCTION

This routine adds a new frame that contains new data to the file. From the location of the frame pointed to, the procedure will extract the frame type, find a suitable module, allocate space for the frame, and copy the frame into its new location. If the frame cannot be allocated (e.g., invalid frame type), the pointers will be set to nil and a negative status will be returned.

If the database FRAME.ID field in the SAM_FRAME_HDR is SAM_NIL_WORD (ØFFFFH), VIRGA will allocate a unique ID for the frame. Otherwise, the routine tries to use the specified ID. If that ID is already in use, the call will fail with a negative status.

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

The routine will add the frame name to an existing index; if the name is a duplicate, it will be added anyway, but a negative status (SAM_ERR_DUPLICATE_NAME) will be returned.

NOTE

IDs which are automatically allocated by VIRGA are re-used; e.g., if you add a frame and VIRGA gives it ID 93, add another (94), delete frame 93, and then add another, that last frame will be given ID 93.

The SAM_ADD_FRAME routine can, like SAM_ALLOCATE_FRAME, create a frame of a specified type and size. However, SAM_ADD_FRAME puts the frame wherever VIRGA thinks best (and will copy the specified data into it), while SAM_ALLOCATE_FRAME puts the frame in the module specified by the user.

EXAMPLE

```
DECLARE
    FRAME_COPY STRUCTURE (SAM_FRAME_HDR, USER_DATA_DEF);
    FRAME_HDR BASED PTR STRUCTURE (SAM_FRAME_HDR);
FRAME_COPY.OPCODE = SAM_SCR_FRAME;
FRAME_COPY.HDR_SIZE = SIZE (FRAME_HDR);
FRAME_COPY.SIZE = SIZE (FRAME_COPY);
FRAME_COPY.ID = SAM_NIL_WORD; /* let VIRGA allocate ID */
FRAME_COPY.TYPE = SIDF_FOOBAR;
FRAME_COPY.CELL_ID = SAM_NIL_WORD;
CALL MORB (RECTANGLE_PTR, @FRAME_COPY.X1, SIZE (RECTANGLE));
FRAME_PTR = @FRAME_COPY;
STATUS = SAM_ADD_FRAME (ROO_PTR, @MODULE_PTR, @FRAME_PTR);
IF STATUS C = SYS_FAILURE THEN GRUMBLE;
```

4.3.2 SAM_ADD_FRAME_TYPE

SAM_ADD_FRAME_TYPE adds a specified frame type to the file.

SYNTAX

```
SAM_ADD_FRAME_TYPE (ROOT_PTR,
                    FRAME_TYPE,
                    FRAME_DESC_PTR);
```

"ROOT_PTR" is the pointer returned from OPEN or CREATE requests; it specifies the file to be accessed.

"FRAME_TYPE" specifies the number of the frame type to be added.

"FRAME_DESC_PTR" points to the frame descriptor structure in SAMFIL.ELT.

FUNCTION

This routine adds a specified frame type to the file. The frame type added must be one already creates as an unused frame type. No frame of this type must currently exist. The data from the frame description is used to initialize the frame types inventory entry.

NOTE

The difference between SAM_ADD_FRAME_TYPE and SAM_ADD_NEW_FRAME_TYPE is that SAM_ADD_FRAME_TYPE assumes that an empty frame inventory already exists. SAM_ADD_NEW_FRAME_TYPE creates a new frame inventory.

EXAMPLE

```
DECLARE FRAME_DESC STRUCTURE (SAM_FRAME_DESC_DEF);
FRAME_DESC.MOD_TYPE = SIDM_MUMBLE;
FRAME_DESC.INV_EXT = 10;
FRAME_DESC.INV_SIZE = 20;
STATUS = SAM_ADD_FRAME_TYPE
(ROOT_PTR, SIDF_SPARE1, @FRAME_DESC);
IF STATUS C = SYS_FAILURE THEN GRUMBLE;
```

4.3.3 SAM_ADD_NEW_FRAME_TYPE

SAM_ADD_NEW_FRAME_TYPE adds a new frame type to the file.

SYNTAX

```
SAM_ADD_NEW_FRAME_TYPE (ROOT_PTR,
                        FRAME_TYPE,
                        FRAME_DESC_PTR);
```

"ROOT_PTR" is the pointer returned from OPEN or CREATE requests; it specifies the file to be accessed.

"FRAME_TYPE" specifies the number of the frame type to be added.

"FRAME_DESC_PTR" points to the frame descriptor structure in SAMFIL.ELT.

FUNCTION

This routine allows a frame type to be defined for a spare frame type. The number of spare frame types was defined when the file was created (from CONFIG.SPARE_TYPES). This routine will use one of these defined spare types.

NOTE

The difference between SAM_ADD_FRAME_TYPE and SAM_ADD_NEW_FRAME_TYPE is that SAM_ADD_FRAME_TYPE assumes that an empty frame inventory already exists. SAM_ADD_NEW_FRAME_TYPE creates a new frame inventory.

EXAMPLE

```
DECLARE FRAME_DESC STRUCTURE (SAM_FRAME_DESC_DEF);
FRAME_DESC.MOD_TYPE = SIDM_MUMBLE;
FRAME_DESC.INV_EXT = 10;
FRAME_DESC.INV_SIZE = 20;
STATUS = SAM_ADD_NEW_FRAME_TYPE
(ROOT_PTR, SIDF_SPARE1, @FRAME_DESC);
IF STATUS C = SYS_FAILURE THEN GRUMBLE;
```

4.3.4 SAM_ALLOCATE_FRAME

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

SYNTAX

```
SAM_ALLOCATE_FRAME (ROOT_PTR,
                    MODULE_PTR,
                    @FRAME_PTR,
                    FRAME_TYPE,
                    FRAME_SIZE);
```

"ROOT_PTR" is the pointer returned from OPEN or CREATE requests; it specifies the file to be accessed.

"MODULE_PTR" points to the module to which the frame will be allocated.

"@FRAME_PTR" returns a pointer to the address of the frame.

"FRAME_TYPE" specifies the number of the frame type to be added.

"FRAME_SIZE" specifies the size of the new frame.

FUNCTION

This routine puts the frame wherever the user specified. This routine helps the user control memory, but does not immediately affect buffer handling.

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

NOTE

The SAM_ADD_FRAME routine can, like SAM_ALLOCATE_FRAME, create a frame of a specified type and size. However, SAM_ADD_FRAME puts the frame wherever VIRGA thinks best (and will copy the specified data into it), while SAM_ALLOCATE_FRAME puts the frame in the module specified by the user.

EXAMPLE

```
STATUS = SAM_FIND_FRAME (ROOT_PTR, @MODULE_PTR, @FRAME_PTR,
         SIDF_FOO, 27);
IF STATUS C = SYS_FAILURE THEN GRUMBLE;
    /* find frame 27 */

STATUS = SAM_ALLOCATE_FRAME (ROOT_PTR, MODULE_PTR,
         @FRAME_PTR, SIDF_FOO, 500);
IF STATUS C = SYS_FAILURE THEN GRUMBLE;
    /* try and allocate a new frame in the same module as */
    /* frame # 27 */
```

4.3.5 SAM_ALLOCATE_MODULE

SAM_ALLOCATE_MODULE creates a new module of the specified type and size.

SYNTAX

```
SAM_ALLOCATE_MODULE (ROOT_PTR,
                     @MODULE_PTR,
                     MODULE_TYPE,
                     MODULE_SIZE);
```

"ROOT_PTR" is the pointer returned from OPEN or CREATE requests; it specifies the file to be accessed.

"@MODULE_PTR" returns a pointer to the address of the module allocated in memory.

"MODULE_TYPE" specifies the number of the desired module type.

"MODULE_SIZE" specifies the desired module size.

FUNCTION

This routine creates a new module of the specified type and size. It helps a user control memory, but does not directly affect existing buffers.

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

EXAMPLE

```
STATUS = SAM_ALLOCATE_MODULE (ROOT_PTR,
                              @MODULE_PTR,
                              XYZ_SAME_MODULE_TYPE,
                              3*8192);
STATUS = SAM_ALLOCATE_FRAME (ROOT_PTR,
                             MODULE_PTR,
                             @FRAME_PTR,
                             FRAME_TYPE,
                             FRAME_SIZE);
```

4.3.6 SAM_CLEAR_DIRTY_MODULE

SAM_CLEAR_DIRTY_MODULE clears the dirty flag on a module.

SYNTAX

```
SAM_CLEAR_DIRTY_MODULE (ROOT_PTR,
                        MODULE_PTR);
```

"ROOT_PTR" is the pointer returned from OPEN or CREATE requests; it specifies the file to be accessed.

"MODULE_PTR" points to the address of the module whose dirty flag is to be cleared.

FUNCTION

This routine clears the dirty flag on a module so that it will NOT be written to disk when its buffer is deallocated.

NOTE

It is dangerous to clear the dirty flag because other updates, which might have been made to other frames within that module, will be lost.

EXAMPLE

No example is given because, although this routine is provided for completeness, it should never be called by application programs.

4.3.7 SAM_CLOSE

SAM_CLOSE makes a specified file unavailable for access by VIRGA until the user again calls SAM_OPEN.

SYNTAX

```
SAM_CLOSE (ROOT_PTR,
           PURGE_FLAG);
```

"ROOT_PTR" is the pointer returned from OPEN or CREATE requests; it specifies the file to be accessed.

"PURGE_FLAG" sets a flag. If set to 1 (TRUE), VIRGA deletes the file; if set to 0 (FALSE), the file is only closed.

FUNCTION

This routine makes a specified file unavailable for access by VIRGA until the user again calls SAM_OPEN.

All VIRGA files MUST be closed using this call.

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

NOTE

Purge_flag is a BOOLEAN data type; see SYSCOM.ELT.

EXAMPLE

```
DECLARE PURGE_FLAG BOOLEAN;
STATUS = SAM_OPEN (ROOT_PTR, GLOBAL_PTR, PATH_NAME_PTR,
                   FSYS_WRITE_MODE);
IF STATUS C = SYS_FAILURE THEN GRUMBLE;
STATUS = PERFORM_DATABASE_UPDATE (ROOT_PTR);
PURGE_FLAG = (STATUS <= SYS_FAILURE);
    /* if update failed, delete file */
STATUS = SAM_CLOSE (ROOT_PTR, PURGE_FLAG);
IF STATUS <= SYS_FAILURE THEN GRUMBLE;
```

4.3.8 SAM_CREATE

SAM_CREATE creates a VIRGA file with user-specified module and frame types.

SYNTAX

```
SAM_CREATE (@ROOT_PTR,
            GLOBAL_PTR,
            PATH_NAME_PTR,
            FILE_DESC_PTR);
```

"@ROOT_PTR" returns a pointer to the created file. (This is the pointer that gets passed as the ROOT_PTR in many VIRGA procedures.)

"GLOBAL_PTR" points to the structure returned from the SAM_INITIATE (@GLOBAL_PTR).

"PATH_NAME_PTR" point either to an existing pathname structure or is NIL (Ø). VIRGA uses the specified pathname for the file (pathname includes filename). Otherwise, if it is NIL, VIRGA creates a unique name in the temporary directory /TEMP.

"FILE_DESC_PTR" points to the file description structure described in Section 2.x above and in SAMFIL.ELT.

FUNCTION

This routine creates a VIRGA file with user-specified module and frame types.

If a file already exists by that name, then no file is created and VIRGA returns an error.

EXAMPLE

The literal:

```
SAM_FIRST_FRAME LITERALLY '3',
```

must be used when defining your frame type literals. For example, you may use:

```
SAMF_FOO_FRAME LITERALLY 'SAM_FIRST_FRAME+1'
```

4.3.9 SAM_CREATE_SHELL_PATH

SAM_CREATE_SHELL_PATH converts a shell style null-terminated (string) pathname pointer to a file descriptor and calls SAM_CREATE to create a VIRGA file with user-specified module and frame types.

SYNTAX

```
SAM_CREATE_SHELL_PATH (@ROOT_PTR,
                       GLOBAL_PTR,
                       SHELL_PATH_PTR,
                       SHELL_PATH_LEN,
                       FILE_DESC_PTR);
```

"@ROOT_PTR" returns a pointer to the created file. (This is the pointer that gets passed as the ROOT_PTR in many VIRGA procedures.)

"GLOBAL_PTR" points to the structure returned from the SAM_INITIATE (@GLOBAL_PTR).

"SHELL_PATH_PTR" points to a null-terminated string SHELL pathname. This may be the pointer returned by a previous call SHL_GET_ARGV or SHL_GET_UNARGV call. (See example below.)

"SHELL_PATH_LENGTH" is the length of the string SHELL pathname pointed to by SHELL_PATH_PTR.

"FILE_DESC_PTR" points to the file description structure described in Section 2 above and in SAMFIL.ELT.

FUNCTION

The specified pathname is converted to a file descriptor, SAM_CREATE is called, memory allocated for the pathname conversion is deleted, and a status is passed back from SAM_CREATE. SAM_CREATE creates a VIRGA file with user-specified module and frame types.

If a file with the specified name already exists, no file is created and VIRGA returns an error.

EXAMPLE

```
STATUS = SHL_GET_UNARGV (1, @SHELL_PATH_PTR);
SHELL_PATH_LEN = UTL_STRLEN (SHELL_PATH_PTR);
```

```
SAM_CREATE_SHELL_PATH (@ROOT_PTR,
                       GLOBAL_PTR,
                       SHELL_PATH_PTR,
                       SHELL_PATH_LEN,
                       FILE_DESC_PTR);
```

4.3.10 SAM_CREATE_BOX_INDEX

SAM_CREATE_BOX_INDEX creates an index of the frames located within a specified rectangle (BOX).

SYNTAX

```
SAM_CREATE_BOX_INDEX (ROOT_PTR,
                      BOX_INDEX_FRAME_TYPE,
                      FRAME_TYPE_LIST_PTR,
                      FRAME_TYPE_LIST_COUNT);
```

"ROOT_PTR" is the pointer returned from OPEN or CREATE requests; it specifies the file to be accessed.

"BOX_INDEX_FRAME_TYPE" specifies the number of the desired frame type.

"FRAME_TYPE_LIST_PTR" points to the list of frame types (bytes).

"FRAME_TYPE_LIST_COUNT" specifies the number of frame types in the list.

FUNCTION

This routine creates an index of the bounding boxes of the modules in the file. Only those frame whose types are mentioned in the FRAME_TYPE_LIST contribute to the box index. A box index greatly speeds up geometrical access to the disk.

EXAMPLE

```
DECLARE SIDF_BOX_INDEX LITERALLY 'SAM_FIRST_FRAME+11',
        SIDF_MASK_GEOMETRY LITERALLY 'SAM_FIRST_FRAME+3',
        SIDF_NON_MASK_GEOMETRY LITERALLY 'SAM_FIRST_FRAME+4';
STATUS = SAM_CREATE_BOX_INDEX (ROOT_PTR, SIDF_BOX_INDEX,
         @(SIDF_MASK_GEOMETRY, SIDF_NON_MASK_GEOMETRY), 2);
```

4.3.11 SAM_CREATE_CELL_DIR

SAM_CREATE_CELL_DIR creates a cell directory for a cell in a library and returns the cell ID of the cell.

SYNTAX

```
SAM_CREATE_CELL_DIR (SRC_ROOT_PTR,
                     DST_ROOT_PTR,
                     CELL_NAME_PTR,
                     CELL_NAME_SIZE,
                     @CELL_ID);
```

"SRC_ROOT_PTR" specifies the source VIRGA file to be included in the cell library.

"DST_ROOT_PTR" specifies the library file into which the cell directory is created.

"CELL_NAME_PTR" points to the name of the cell to be created.

"CELL_NAME_SIZE" specifies the size of the cell name.

"@CELL_ID" (bidirectional) returns the ID number of the cell into which the file will be placed.

FUNCTION

This routine uses the fram types and frame inventories of the source file to create a new cell directory in a library file. This cell directory is used by the library manager (SID) before cataloging a cell into the library.

If the CELL_ID is set to SAM_NIL_WORD before the call, VIRGA will allocate the CELL_ID for the new cell; otherwise the ID specified will be used. If the ID is already in use by another cell, an error is returned.

EXAMPLE

```
CELL_ID : SAM_NIL_WORD;
STATUS = SAM_CREATE_CELL_DIR (FREE_CELL_ROOT_PTR,
         LIBRARY_ROOT_PTR, @TEMPLATE_FRAME.CELL_NAME,
         TEMPLATE_FRAME.CELL_NAME_SIZE, @CELL_ID);
```

4.3.12 SAM_CREATE_INDEX

SAM_CREATE_INDEX creates a name index for a specified frame type.

SYNTAX

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

"ROOT_PTR" is the pointer returned from OPEN or CREATE requests; it specifies the file to be accessed.

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

"INDEX_FRAME_TYPE" specifies the frame type of the index itself. This is the only time the user needs to know this type; after this call, VIRGA maps the frame types automatically. This frame type must be included in those specified as legal when the file was created.

"DESC_PTR" points to the index description structure, SAM_INDEX_DESC_DEF, described in Section 2.x and in SAMNAM.ELT.

FUNCTION

This routine creates a name index for a specified frame type.

If an index of the specified frame type already exists, the routine deletes it, then recreates it. The names of all frames of the specified frame type are added to the index so that the index is up-to-date when this routine is complete.

NOTE

The fields in SAM_INDEX_DESC_DEF (see SAMNAM.ELT) must have been initialized by the user before invoking SAM_CREATE_INDEX.

EXAMPLE

```
DECLARE INDEX_DESC STRUCTURE (SAM_INDEX_DESC_DEF);
        INDEX_DESC.NAME_OFFSET = OFFSET OF
        (@TEMPLATE.CELL_NAME) - OFFSET OF (@TEMPLATE);
        INDEX_DESC.NAME_LEN_OFFSET = OFFSET OF
        (@TEMPLATE.CELL_NAME_SIZE) - OFFSET OF (@TEMPLATE);
        INDEX_DESC.INIT_ENTRIES = 20;
        INDEX_DESC.INCR_ENTRIES = 10;
STATUS = SAM_CREATE_INDEX (ROOT_PTR, SLTF_TEMPLATE,
         SIDF_TEMPLATE_INDEX, @INDEX_DESC);
IF STATUS <= SYS_FAILURE THEN GRUMBLE;
```

4.3.13 SAM_DELETE_BY_TYPE

SAM_DELETE_BY_TYPE removes all frames of the specified frame type.

SYNTAX

```
SAM_DELETE_BY_TYPE (ROOT_PTR,
                    FRAME_TYPE);
```

"ROOT_PTR" is the pointer returned from OPEN or CREATE requests; it specifies the file to be accessed.

"FRAME_TYPE" specifies the number of the desired frame type to be removed.

FUNCTION

This routine deletes all frame of the specified type.

EXAMPLE

```
STATUS = SAM_DELETE_BY_TYPE (ROOT_PTR, SIDF_INSTANCE);
   /* delete all instance frame */
```

4.3.14 SAM_DELETE_FRAME

SAM_DELETE_FRAME removes a frame from the file.

SYNTAX

```
SAM_DELETE_FRAME (ROOT_PTR,
                  MODULE_PTR,
                  FRAME_PTR);
```

"ROOT_PTR" is the pointer returned from OPEN or CREATE requests; it specifies the file to be accessed.

"MODULE_PTR" points to the module to be removed.

"FRAME_PTR" points to the frame to be removed.

FUNCTION

This routine removes a specified frame from the specified file. It also removes the name of the specified frame from the frame type's name index, if it exists. If the name is not in the index, the frame is still removed, but a negative status is returned.

The routine always sets the dirty flag on the module that contains the frame to be removed so that the module will eventually be written to disk.

NOTE

Other data frames might be moved by this routine to condense the database.

EXAMPLE

```
STATUS = SAM_FIND_FRAME (ROOT_PTR, @MODULE_PTR, @FRAME_PTR,
         SIDF_FOO, 27);
IF STATUS <= SYS_FAILURE THEN GRUMBLE;
    /* find frame # 27 */

STATUS = SAM_DELETE_FRAME (ROOT_PTR, MODULE_PTR, FRAME_PTR);
IF STATUS <= SYS_FAILURE THEN GRUMBLE;
    /* delete frame # 27 */
```

4.3.15 SAM_DELETE_INDEX

SAM_DELETE_INDEX removes the index of the specified frame type.

SYNTAX

```
SAM_DELETE_INDEX (ROOT_PTR,
                  FRAME_TYPE);
```

"ROOT_PTR" is the pointer returned from OPEN or CREATE requests; it specifies the file to be accessed.

"FRAME_TYPE" specifies the number of the desired frame type whose names are indexed (not the type of the index frame itself).

FUNCTION

This routine removes the index of the specified frame type. All the relevant index frames are deleted and the index type byte in the frame inventory of the specified frame type is set to zero, indicating that there is no name index for that frame type.

The actual frames of the specified frame type are left untouched.

If there is no index for the specified frame type, then a the status, SAM_STAT_NO_INDEX (positive), is returned.

EXAMPLE

```
STATUS = SAM_DELETE_INDEX (ROOT_PTR, SLTF_TEMPLATE);
IF STATUS <= SYS_FAILURE THEN GRUMBLE;
    /* delete name index for template frames */
```

4.3.16 SAM_DIRTY_FRAME

SAM_DIRTY_FRAME indicates an intention to modify the specified frame.

SYNTAX

```
SAM_DIRTY_FRAME (ROOT_PTR,
                 MODULE_PTR,
                 FRAME_PTR);
```

"ROOT_PTR" is the pointer returned from OPEN or CREATE requests; it specifies the file to be accessed.

"MODULE_PTR" points to the desired module.

"FRAME_PTR" points to the frame to be modified.

FUNCTION

This routine sets the dirty flag in the specified module so that it will eventually be written to disk.

You use this routine before you change anything in the frame.

NOTE

This routine should be used to set the dirty flag if you intend to use transaction calls; SAM_SET_DIRTY_MODULE does not work with the SAM_UNDO_TRANSACTION routine.

You cannot use this method to modify any of the fields in the SAM_FRAME_HDR. You must use SAM_GROW_FRAME, SAM_MOVE_FRAME, or SAM_UPDATE_FRAME to modify fields in the SAM_FRAME_HDR.

EXAMPLE

```
STATUS = SAM_DIRTY_FRAME (ROOT_PTR, MODULE_PTR, FRAME_PTR);
IF STATUS <= SYS_FAILURE THEN GRUMBLE;
FRAME.USER_DATA_FIELD = SOMETHING_NEW;
```

4.3.17 SAM_DISPOSE_ALL

SAM_DISPOSE_ALL clears all the unlocked modules from the VIRGA buffers.

SYNTAX

SAM_DISPOSE_ALL (GLOBAL_PTR)

"GLOBAL_PTR" points to the structure returned frm SAM_INITIATE.

FUNCTION

This routine clears all the unlocked modules from the buffers to free up memory space. It acts on ALL VIRGA files, not just one.

EXAMPLE

```
STATUS = SAM_DISPOSE_ALL (GLOBAL_PTR)
            .
            .
            .
/* load big task */
            .
            .
            .
```

4.3.18 SAM_DISPOSE_MODULE

SAM_DISPOSE_MODULE removes the specified module and deallocates that buffer space.

SYNTAX

```
SAM_DISPOSE_MODULE (ROOT_PTR,
                    MODULE_PTR);
```

"ROOT_PTR" is the pointer returned from OPEN or CREATE requests; it specifies the file to be accessed.

"MODULE_PTR" points to the module to be removed.

FUNCTION

This routine removes the specified module and deallocates that buffer space to free up memory for another purpose.

If the module has its dirty flag set, it is written to disk before it is removed.

NOTE

The module is not removed from the disk file, only from memory.

If the specified module is locked or contains any locked frames, it will NOT be disposed; an error message will be returned.

EXAMPLE

```
STATUS = SAM_DISPOSE_MODULE (ROOT_PTR, MODULE_PTR);
/*  done with this module */
```

4.3.19 SAM DISPOSE MODULES BY TYPE

SAM DISPOSE MODULES BY TYPE removes modules of the specified type and deallocates that buffer space.

SYNTAX

```
SAM DISPOSE MODULES BY TYPE (ROOT PTR,
                             MODULE TYPE);
```

"ROOT PTR" is the pointer returned from OPEN or CREATE requests; it specifies the file to be accessed.

"MODULE TYPE" specifies the desired module type of the modules to be removed. If MODULE TYPE = SAM NIL BYTE (ØFFH), then all types will be disposed except for type SAMM CONTROL (which contains VIRGA's inventories).

FUNCTION

This routine removes modules of the specified type and deallocates that buffer space to free up memory for another purpose.

If any of the specified modules have a dirty flag set, they will be written to disk.

NOTE

No error occurs if modules of the specified type do not exist.

Modules are not removed from the disk file, only from memory.

If any of the modules of the specified type are locked, they will NOT be disposed.

EXAMPLE

```
STATUS = SAM DISPOSE MODULES BY TYPE (ROOT PTR, SIDM INSTANCE);
/* finished processing instances */
```

4.3.20 SAM_DUMP_GLOBAL

SAM_DUMP_GLOBAL prints some of VIRGA's internal data structures to the specified device or file.

SYNTAX

```
SAM_DUMP_GLOBAL (GLOBAL_PTR,
                 OUTPUT_ID);
```

"GLOBAL_PTR" points to the structure returned by SAM_INITIATE.

"OUTPUT_ID" specifies the output device or file ID.

FUNCTION

This routine prints out the SAM global data structure, the configuration table, the root data structure of each open file, and a table of allocated buffers showing which module occupies each buffer. This information is useful for debugging applications that use VIRGA.

EXAMPLE

```
/* debugging command */
STATUS = SAM_DUMP_GLOBAL (ENVIRON.SAM_PTR, OUTPUT_ID)
```

4.3.21 SAM_FIND_FIRST

SAM_FIND_FIRST returns a pointer to the address of the first frame found of a specified frame type.

SYNTAX

```
SAM_FIND_FIRST (ROOT_PTR,
                @MODULE_PTR,
                @FRAME_PTR,
                FRAME_TYPE);
```

"ROOT_PTR" is the pointer returned from OPEN or CREATE requests; it specifies the file to be accessed.

"@MODULE_PTR" returns a pointer of the module of the first frame found.

"@FRAME_PTR" returns a pointer of the frame found.

"FRAME_TYPE" specifies the number of the desired frame type.

FUNCTION

This routine returns a pointer to the first frame found of a specified frame type.

If a frame cannot be found of the specified frame type, the returned pointers will be set to NIL and a negative status will be returned (SAM_NO_SUCH_FRAME).

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

NOTE

The frames are found in physical order in the file. This order bears no relation to the order in which the frame were created or the order of their IDs.

EXAMPLE

```
STATUS = SAM_FIND_FIRST (ROOT_PTR, @MODULE_PTR, @FRAME_PTR,
         FRAME_TYPE);
DO WHILE STATUS = SYS_SUCCESS;
    CALL PROCESS (FRAME_PTR);
    STATUS = SAM_FIND_NEXT (ROOT_PTR, @MODULE_PTR,
             @FRAME_PTR);
END;    */ do while */
IF STATUS <> SAM_NO_SUCH_FRAME THEN GRUMBLE;
```

4.3.22 SAM_FIND_FIRST_FRAME_IN_BOX

SAM_FIND_FIRST_FRAME_IN_BOX returns pointers to the addresses of the frame and module that contain the first frame located within the specified rectangle.

SYNTAX

```
SAM_FIND_FIRST_FRAME_IN_BOX (ROOT_PTR,
                             BOX_PTR,
                             FRAME_TYPE,
                             @MODULE_PTR,
                             @FRAME_PTR,
```

"ROOT_PTR" is the pointer returned from OPEN or CREATE requests; it specifies the file to be accessed.

"BOX_PTR" points to SAM_RECTANGLE (defined in SAMFIL.ELT).

"FRAME_TYPE" specifies the number of the desired frame type.

"@MODULE_PTR" returns a pointer to the module that contains the first frame found.

"@FRAME_PTR" returns a pointer to the first frame found in the rectangle.

FUNCTION

This routine returns pointers to the frame and module that contain the first frame located within the specified rectangle.

Within a specified rectangle on the screen may be several frames that describe various geometric objects. You might want to find the first frame that describes a component within the rectangle. One frame type number is linked to component frames.

When the routine completes, the status messages indicate whether the returned frames are:

- o completely within the rectangle (SYS_SUCCESS),
- o only partly contained within the rectangle (SAM_STAT_PARTIALLY_IN_BOX), or
- o not found (no more frames within the rectangle) (SAM_NO_SUCH_FRAME).

NOTE

The frames are found in physical order in the file. This order bears no relation to the order in which the frame were created or the order of their IDs.

EXAMPLE

```
STATUS = SAM_FIND_FIRST_FRAME_IN_BOX (ROOT_PTR, BOX_PTR,
         SIDF_GEOMETRY, @MODULE_PTR, @FRAME_PTR);
DO WHILE STATUS = SYS_SUCCESS;
    CALL PROCESS (FRAME_PTR);
    STATUS = SAM_FIND_NEXT_FRAME_IN_BOX (ROOT_PTR, BOX_PTR,
             @MODULE_PTR, @FRAME_PTR);
END;   /* do while */
IF STATUS <> SAM_NO_SUCH_FRAME THEN GRUMBLE;
```

4.3.23 SAM_FIND_FRAME

SAM_FIND_FRAME finds the desired frame by ID.

SYNTAX

```
SAM_FIND_FRAME (ROOT_PTR,
                @MODULE_PTR,
                @FRAME_PTR,
                FRAME_TYPE,
                FRAME_ID);
```

"ROOT_PTR" is the pointer returned from OPEN or CREATE requests; it specifies the file to be accessed.

"@MODULE_PTR" returns a pointer of the module that contains the desired frame.

"@FRAME_PTR" returns a pointer of the desired frame.

"FRAME_TYPE" specifies the number of the desired frame type.

"FRAME_ID" specifies the identification number of the desired frame.

FUNCTION

This routine finds the desired frame, given a frame type and frame ID. This routine uses an inventory to achieve direct access to the frame's module (maximum one disk access). Then, it performs a linear search within that module to find the specified frame from the frame type and frame ID given.

Upon return, @MODULE_PTR and @FRAME_PTR point to the module and frame which were found. If the frame could not be found, the pointers will be set to nil, and a negative status will be returned.

NOTE

If necessary, the module containing the frame will be read from disk.

SAM_FIND_FRAME doesn't know or care about name indexes.

FIND (and similar procedures such as NEXT) return a pointer directly to the module buffer. This should be considered a TEMPORARY pointer only. Any later request could invalidate

that pointer, either because the module was swapped out or because the frame was moved.

Also, the frame pointer must not be used to modify the contents of the frame directly unless you set the lock and dirty flags. See Section 4.1.5 "Buffer Control Procedures" for lock and dirty flag requests.

EXAMPLE

```
STATUS = SAM_FIND_FRAME (ROOT_PTR, @MODULE_PTR,
         @TEMPLATE_FRAME_PTR, SLTF_TEMPLATE, 0);
    /* find template frame #0 */
```

4.3.24 SAM_FIND_FRAMES_IN_WINDOW

SAM_FIND_FRAMES_IN_WINDOW locates all frames of a specified type that are contained within a rectangle.

SYNTAX

```
SAM_FIND_FRAMES_IN_WINDOW (ROOT_PTR,
                           RECTANGLE_PTR,
                           FRAME_TYPE,
                           RESULT_SIZE,
                           RESULT_PTR);
```

"ROOT_PTR" is the pointer returned from OPEN or CREATE requests; it specifies the file to be accessed.

"RECTANGLE_PTR" points to SAM_RECTANGLE (defined in SAMFIL.ELT).

"FRAME_TYPE" specifies the number of the desired frame type of the frames to be found.

"RESULT_SIZE" is size in bytes of the table reserved for the list of frames found. If the size specified in this parameter is exceeded, the call will terminate unsuccessfully.

"RESULT_PTR" points to an array of WORDS to contain the IDs of the frames found. The first word in this array holds the number of IDs found.

FUNCTION

This routine locates all frames of a specified type that are contained within a rectangle. This call is provided for backward compatibility with GM-10.

All frames of the specified type are checked to see if the frame intersects the specified rectangle (where boundaries are considered to be included both in the recognition area and in the rectangle). If there is an intersection (any overlap at all between recognition area and the defined window), the frame's ID will be saved an placed in the RESULT table.

NOTE

A problem with this routine is that you must determine the arrpoximate number of frames you expect to find within the rectangle. Thus, this routine has been replaced by the

following two routines:

SAM FIND FIRST FRAME IN BOX
SAM FIND NEXT FRAME IN BOX

4.3.25 SAM_FIND_INDEX_FRAME

SAM_FIND_INDEX_FRAME maps the frame type to the index frame type, locates the frame with ID = KEY2, invokes SAM_FIND_FRAME to find the frame, and sets the frame's INDEX_HDR.SYMTAB_PTR appropriately.

SYNTAX

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

"ROOT_PTR" is the pointer returned from OPEN or CREATE requests; it specifies the number of the file to be accessed.

"@MOD_PTR" returns a pointer to the address of the module that contains the desired frame.

"@FRAME_PTR" returns a pointer to the address of the desired frame.

"FRAME_TYPE" specifies the number of the frame type of the frames that were indexed.

"KEY2" specifies the key2 for the desired index; if there is no key2 for the index, it should be set to SAM_NIL_WORD.

FUNCTION

This routine maps the frame type to the index frame type, locates the frame with ID = key2 (or = Ø if KEY2 = SAM_NIL_WORD), invokes SAM_FIND_FRAME to find the frame, and sets the frame's INDEX_HDR.SYMTAB_PTR appropriately.

To control searching, the user can lock the index into memory (using SAM_LOCK_FRAME) and perform an explicit search of the symtab. Or, the user can directly set the INDEX_HEADER.MARK field, using the FIRST_NAME/NEXT_NAME routines.

The user may also invoke this procedure to directly obtain the symtab of names for a given frame type and key2.

NOTE

This procedure is not really necessary. The user may directly invoke first_name, next_name, etc.

EXAMPLE

```
STATUS = SAM_FIND_INDEX_FRAME (ROOT_PTR, @MOD_PTR, @FRAME_PTR, SLTF_TEMPLATE, 0);
IF STATUS = SYS_FAILURE THEN RETURN ERROR;
STATUS = SAM_LOCK_FRAME (ROOT_PTR, MOD_PTR, FRAME_PTR);
/* now access all names with KEY2 = 0 */
```

4.3.26 SAM_FIND_NEXT

SAM_FIND_NEXT finds the next frame given the frame type from SAM_FIND_FIRST and the pointers returned from the first frame found.

SYNTAX

```
SAM_FIND_NEXT (ROOT_PTR,
               @MODULE_PTR,
               @FRAME_PTR);
```

"ROOT_PTR" is the pointer returned from OPEN or CREATE requests; it specifies the number of the same file accessed by SAM_FIND_FIRST.

"@MODULE_PTR" (bidirectional) both inputs the address of the previously located module (@MODULE_PTR from the SAM_FIND_FIRST output) and returns a pointer to the address of the next module.

"@FRAME_PTR" (bidirectional) both inputs the address of the previously located frame (@FRAME_PTR from the SAM_FIND_FIRST output) and returns a pointer to the address of the next frame.

FUNCTION

This routine finds the next frame given the frame type from SAM_FIND_FIRST and the pointers returned from the first frame found.

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

NOTE

None applicable.

EXAMPLE

```
STATUS = SAM_FIND_FIRST (ROOT_PTR, @MODULE_PTR, @FRAME_PTR,
         FRAME_TYPE);
DO WHILE STATUS = SYS_SUCCESS;
    CALL PROCESS (FRAME_PTR);
    STATUS = SAM_FIND_NEXT (ROOT_PTR, @MODULE_PTR,
             @FRAME_PTR);
END;    */ do while */
IF STATUS <> SAM_NO_SUCH_FRAME THEN GRUMBLE;
```

4.3.27 SAM_FIND_NEXT_FRAME_IN_BOX

SAM_FIND_NEXT_FRAME_IN_BOX returns pointers to the addresses of the frame and module that contain the next frame located within a specified rectangle.

SYNTAX

```
SAM_FIND_NEXT_FRAME_IN_BOX (ROOT_PTR,
                            BOX_PTR,
                            @MODULE_PTR,
                            @FRAME_PTR);
```

"ROOT_PTR" is the pointer returned from OPEN or CREATE requests; it specifies the number of the file to be accessed.

"BOX_PTR" is the pointer to rectangle definition, SAM_RECTANGLE, in file SAMFIL.ELT.

"@MODULE_PTR" (bidirectional) both inputs the address of the previously located module (@MODULE_PTR from the SAM_FIND_FIRST_FRAME_IN_BOX output) and returns a pointer to the address of the next module that contains the next frame found in the rectangle.

"@FRAME_PTR" (bidirectional) both inputs the address of the previously located frame (@FRAME_PTR from the SAM_FIND_FIRST_FRAME_IN_BOX output) and returns a pointer to the address of the next frame that contains the next frame found in the rectangle.

FUNCTION

This routine returns pointers to the addresses of the frame and module that contain the next frame located within a specified rectangle.

Within a specified rectangle on the screen may be several frames that describe various geometric objects. For example, you might want to find the first frame that describes a component within the rectangle. One frame type number is linked to component frames.

All frames of the specified type are checked to see if the frame intersects the specified rectangle (where boundaries are considered to be included both in the recognition area and in the rectangle). If there is an intersection (any overlap at all between recognition area and the defined window), the frame's address is returned.

When the routine completes, the status messages indicate whether the returned frames are:

- o completely within the rectangle (SYS_SUCCESS),
- o only partly contained within the rectangle (SAM_STAT_PARTIALLY_IN_BOX), or
- o not found (no more frames within the rectangle) (SAM_NO_SUCH_FRAME).

NOTE

None applicable.

EXAMPLE

```
STATUS = SAM_FIND_NEXT_FRAME_IN_BOX (ROOT_PTR, @MODULE_PTR, @FRAME_PTR,
         FRAME_TYPE);
DO WHILE STATUS = SYS_SUCCESS;
    CALL PROCESS (FRAME_PTR);
    STATUS = SAM_FIND_NEXT (ROOT_PTR, @MODULE_PTR,
             @FRAME_PTR);
END;   */ do while */
IF STATUS <> SAM_NO_SUCH_FRAME THEN GRUMBLE;
```

4.3.28 SAM_FINISH_TRANSACTION

SAM_FINISH_TRANSACTION specifies the end of a group of calls that form a "transaction."

SYNTAX

```
SAM_FINISH_TRANSACTION (ROOT_PTR);
```

"ROOT_PTR" is the returned pointer from the SAM database root.

FUNCTION

SAM_FINISH_TRANSACTION allows you to signal the end of a series of updates to the database that comprise a transaction. After SAM_START_TRANSACTION has been called and before SAM_FINISH_TRANSACTION is called, you may undo the effects of the current transaction by calling SAM_UNDO_TRANSACTION.

NOTE

Only one transaction in one database may be pending at a time.

EXAMPLE

```
/* get new command */
/* validate new command */
STATUS = SAM_FINISH_TRANSACTION (ROOT_PTR);
/* disallow any more undoing of previously issued commands */
STATUS = SAN_START_TRANSACTION (ROOT_PTR);
/* create new transaction log for this command */
```

4.3.29 SAM_FIRST_NAME

SAM_FIRST_NAME finds the first name in a name index which matches a specified name and has the specified KEY2.

SYNTAX

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

"ROOT_PTR" is the pointer returned from OPEN or CREATE requests; it specifies the number of the file to be accessed.

"FRAME_TYPE" specifies the number of the desired frame type which has been indexed.

"NAME_LEN" specifies the length of the name to be found.

"NAME_PTR" specifies the pointer to the name to be found.

"KEY2" specifies the value of the KEY2 for the desired frame.

"@INDEX_NAME_PTR" returns a pointer to the address of the name found in the index.

"@INDEX_INFO_PTR" returns a pointer to the index entry definition, SAM_INDEX_ENTRY_DEF, which contains the length of the name and the corresponding frame's ID.

FUNCTION

This routine finds the first name in a name index which matches a specified name and has the specified KEY2 (SAM_NIL_WORD if none). If there is no KEY2 for that name index, it should be set to SAM_NIL_WORD.

The name to be found may contain wildcards (see Section x.x "Name Index Procedures" above), as supported by Daisy symtabs.

Here, "first" means that it starts at the beginning of the name index (for specified key2). The name_index is not ordered in any meaningful way.

The routine returns information about the exact name found, the length of the name, and the ID of the frame with that name.

VIRGA will reset the mark field in the symtab to sym_initial_mark (set the mark field to point to the found entry).

A SYMTAB is a symbol table structure whose definition is found in SYMTAB.ELT. This symbol table structure basically associates values with names. The SYMTAB subsystem provides the ability to add entries to and delete entries from the symbol table, and, given a name in the table, retrieve entries associated with that name.

NOTE

If the name index is created with KEY2_OFFSET = SAM_NIL_WORD, the name index is scattered across multiple index frames. To find all the name matches, the caller must repeat the search for every index frame. A null value for KEY2 is therefore not allowed for SAM_FIRST_NAME.

EXAMPLE

```
STATUS = SAM_FIRST_NAME (ROOT_PTR, SLTF_TEMPLATE, 5,
                         @('BOZO*'), @INDEX_NAME_PTR,
                         @INDEX_INFO_PTR);
DO WHILE STATUS = SYS_SUCCESS;
   ID = INDEX_INFO.ID
/* process this frame */
STATUS = SAM_NEXT_NAME (ROOT_PTR, SLTF_TEMPLATE, 5, @('BOZO*'), @INDEX_NAME_PTR, @INDEX
END;
```

4.3.30 SAM_FLUSH

SAM_FLUSH writes any dirty modules to disk, clears their dirty flags, and flushes the root of a specified file.

SYNTAX

SAM_FLUSH (ROOT_PTR);

"ROOT_PTR" is the pointer returned from OPEN or CREATE requests; it specifies the number of the file to be accessed.

FUNCTION

This routine writes any dirty modules to disk, clears their dirty flags, and flushes the root of a specified file. The updating effect is exactly as if the file had been closed and then re-opened, but with less overhead.

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

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

NOTE

All modules that were in memory will remain there; i.e., the buffers are not cleared.

EXAMPLE

```
/* do heavy database processing */
STATUS = SAM_FLUSH (ROOT_PTR);
/* protect against crash */
```

4.3.31 SAM_FLUSH_ALL

SAM_FLUSH_ALL writes any dirty modules to disk, clears their dirty flags, and flushes the roots of ALL VIRGA files.

SYNTAX

```
SAM_FLUSH_ALL (SAM_PTR);
```

"SAM_PTR" points to the list of all VIRGA files (SAM_GLOBAL_FILE_DEF in SAMINT.ELT).

FUNCTION

This routine writes any dirty modules to disk, clears their dirty flags, and flushes the roots of ALL VIRGA files.

NOTE

All modules that were in memory will remain there; i.e., the buffers are not cleared.

EXAMPLE

```
/* do heavy database processing on multiple files */
STATUS = SAM_FLUSH_ALL (ENVIRON_SAM_PTR);
/* update disk image */
```

4.3.32 SAM_FLUSH_AND_DISPOSE

SAM_FLUSH_AND_DISPOSE writes any dirty modules to disk, clears their dirty flags, and deallocates the buffers.

SYNTAX

```
SAM_FLUSH_AND_DISPOSE (SAM_GLOBAL_PTR);
```

"SAM_GLOBAL_PTR" points to the list of all VIRGA files (SAM_GLOBAL_FILE_DEF in SAMINT.ELT).

FUNCTION

This routine writes any dirty modules to disk, clears their dirty flags, and deallocates the buffers. Thus, memory is freed for other uses.

If a module in a buffer is dirty, it is written to disk. Then the buffers are cleared and the memory is returned from SAM_INITIATE to the system.

NOTE

Modules are not removed from the disk file, only from memory.

Locked modules are not cleared from the buffers.

EXAMPLE

```
/* after accessessing database, get memory back */
STATUS = SAM_FLUSH_AND_DISPOSE (ENVIRON.SAM_PTR);
```

4.3.33 SAM_GET_AFT

SAM_GET_AFT gets the AFT (Active File Table) number for a file, some other pieces of information about the file, and a work area.

SYNTAX

```
SAM_GET_AFT (ROOT_PTR,
             @AFT_NUMBER,
             @WORK_AREA_PTR,
             @MOD_SIZE,
             @EOF_MARKER);
```

"ROOT_PTR" is the pointer returned from OPEN or CREATE requests; it specifies the number of the file to be accessed.

"AFT_NUMBER" returns the AFT file number for the specified file.

"WORK_AREA_PTR" returns a pointer to the address of the work area that could be used for file system operations, however, this could be dangerous (see NOTE below).

"MOD_SIZE" returns the the WORD-valued module size for the file.

"EOF_MARKER" returns the DWORD-valued byte offset to the end-of-file (equivalent to the size of the file in bytes).

FUNCTION

This routine gets the AFT (Active File Table) number for a file, some other pieces of information about the file, and a work area. The AFT number is the 32 bit number that identifies the file in the Active File Table.

Generally, with the work area and information on the file, various kinds of direct file system actions could be performed on the file (see file system documentation), but this could be detrimental to VIRGA.

NOTE

Direct file system operations with the file open through VIRGA is likely to screw up VIRGA.

Note that the caller must allocate space for both the AFT_NUMBER and the EOF_MARKER; both require 4 bytes.

EXAMPLE

```
STATUS = SAM_GET_AFT (ROOT_PTR, @AFT_NUMBER, @WORK_AREA_PTR,
                      @MOD_SIZE, @EOF_MARKER);
STATUS = OFO_VAR_WRITE (@('FILE SIZE IS %d'), 15, 1,
                        @(4, @EOF_MARKER));
/* print size of file */
```

4.3.34 SAM_GET_CONFIG

SAM_GET_CONFIG sets the values for the VIRGA configuration table from either the SAM_CONFIG_DEF or the configuration parameters set in the file VIR_CONFIG under /PROJECT.

SYNTAX

SAM_GET_CONFIG (@CONFIG_PTR);

"@CONFIG_PTR" returns a pointer to the structure SAM_CONFIG_DEF.

FUNCTION

This routine first sets the default values for the VIRGA configuration table from the configuration definition, SAM_CONFIG_DEF, in SAMINT.ELT. This routine looks either in /PROJECT PROFILE or else in the project file whose name is specified in the command line for the keyword $VIRGA_CONFIGURATION_REF$. This profile file entry points to the VIRGA configuration file (conventionally named /PROJECT/VIR_CONFIG). This configuration file is set up as a SYMTAB file in order to control certain parameters of VIRGA's behavior.

These parameters are used:

```
@CONFIG.PROTECT_BITS
@CONFIG.MAX_FILE_SIZE
@CONFIG.ROOT_SIZE
@CONFIG.MAX_FILES                  /* max number of files */
@CONFIG.MAX_BUFS                   /* max number of buffers */
@CONFIG.MAX_MEMORY                 /* max total size of buffers */
@CONFIG.SPARE_TYPES                /* spare frame types */
@CONFIG.CAF                        /* caf:  use module size */
@CONFIG.TRANSACTION_ENABLE
@CONFIG.SPARE
@CONFIG.MIN_EMPTY_FRAME_SIZE
@CONFIG.MAX_BUCKET_DIMENSION       /*max module bucket dimension*/
@CONFIG.MIN_BUCKET_DIMENSION       /*min module bucket dimension*/
@CONFIG.MAX_PERCENT_GROWTH
@CONFIG.MIN_PERCENT_SHRINK
```

You may use TEC on VIR_CONFIG to reset the values of an existing VIR_CONFIG disk file or to change the default values of the parameters listed above, if you wish. See the file SAMCNFG.P86 to find the default values.

NOTE

This routine must be called before SAM_INITIATE. SAM_GET_CONFIG is used only for stand-alone programs. Programs that operate within the ILE environment use a different mechanism.

EXAMPLE
None available.

4.3.35 SAM GET MODULES BY TYPE

SAM GET MODULES BY TYPE loads the modules of a given type into memory and locks them there.

SYNTAX

```
SAM GET MODULES BY TYPE (ROOT PTR,
                         MODULE TYPE);
```

"ROOT PTR" is the pointer returned from OPEN or CREATE requests; it specifies the number of the file to be accessed.

"MODULE TYPE" specifies the number of the desired module type.

FUNCTION

This routine loads the modules of a given type into memory and locks them there. If MODULE TYPE = SAM NIL BYTE (= ØFFH), all modules of all types will be read and locked in memory. If there is insufficient memory space in which to read in all modules, VIRGA will get as many as possible and then return a negative status.

Locking the modules in memory ensures that later requests for frames within those modules will not have to go to disk (unless the module has subsequently been swapped out) See Section 4.1.5 "Buffer Control Procedures" above for lock requests.

NOTE

None applicable.

EXAMPLE

```
STATUS = SAM GET MODULES BY TYPE (ROOT PTR, SIDM GEOMETRY);
/* get all geometry frames into memory */
```

4.3.36 SAM_GET_PATHNAME

SAM_GET_PATHNAME returns a pointer to the full pathname for a specified file.

SYNTAX

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

"ROOT_PTR" is the pointer returned from OPEN or CREATE requests; it specifies the number of the file to be accessed.

"@PATHNAME_PTR" returns a pointer to the pathname in the SAM_CREATE or SAM_OPEN procedures. The structure for the pathname definition is in ILE_PATH_NAME_DEF in ILE_GLOBAL.ELT.

FUNCTION

This routine returns a pointer to the full pathname for a specified file in MAESTRO V4.0J and earlier.

NOTE

The pathname is READ_ONLY! Do NOT modify it! Rather, if you need to modify it, copy it to your own buffer, and then modify your own copy.

EXAMPLE

```
STATUS = SAM_GET_PATHNAME (ROOT_PTR, @PATHNAME_PTR);
STATUS = OFO_VAR_WRITE ('Filename is %f', 15, 1,
                        @(0, PATHNAME_PTR);
/* print name of database */
```

4.3.37 SAM_GET_TIMESTAMP

SAM_GET_TIMESTAMP returns a the timestamp for a specified file.

SYNTAX

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

"ROOT_PTR" is the pointer returned from OPEN or CREATE requests; it specifies the number of the file to be accessed.

"@TIMESTAMP" returns a pointer to the system timestamp in Daisy standard format. It requires 6 bytes, which the caller must allocate (see UTILP3 for routines to translate timestamp to ASCII string, etc.).

FUNCTION

This routine returns a the timestamp for a specified file. The timestamp shows the date the file was last written, which may indicate the date the file was last flushed.

NOTE

None applicable.

EXAMPLE

```
STATUS = SAM_GET_TIMESTAMP (ROOT_PTR, @TIME_STAMP);
/* find out when last file was writtem */
```

4.3.38 SAM_GET_TRANSACTION_LOG

SAM_GET_TRANSACTION_LOG works with SAM_UNDO_TRANSACTION to restore a file to its state prior to the SAM_START_TRANSACTION call.

SYNTAX

```
SAM_GET_TRANSACTION_LOG (ROOT_PTR,
                         @NEW_FRAME_COUNT,
                         @NEW_FRAME_LIST_PTR,
                         @UPD_FRAME_COUNT,
                         @UPD_FRAME_LIST_PTR);
```

"ROOT_PTR" is the pointer returned from OPEN or CREATE requests; it specifies the number of the file to be accessed.

"@NEW_FRAME_COUNT" returns the number of added (new) frames.

"@NEW_FRAME_LIST_PTR" returns a list of types and IDs to added (new) frames.

"@UPD_FRAME_COUNT" specifies the number of modified frames.

"@UPD_FRAME_LIST_PTR" returns a list of types and IDs to modified frames.

FUNCTION

This routine works with SAM_UNDO_TRANSACTION to restore a file to its state prior to the SAM_START_TRANSACTION call.

Both lists, returned by @NEW_FRAME_LIST_PTR and @UPD_FRAME_LIST_PTR, are arrays of SAM_FRAME_LIST_DEFs defined in SAMFIL.ELT.

NOTE

None applicable.

EXAMPLE

See Section 3.5 for an explanation of transaction procedures.

4.3.39 SAM_GROW_FRAME

SAM_GROW_FRAME changes (increases or decreases) the size of a frame.

SYNTAX

```
SAM_GROW_FRAME (ROOT_PTR,
                @MOD_PTR,
                @FRAME_PTR,
                NEW_SIZE);
```

"ROOT_PTR" is the pointer returned from OPEN or CREATE requests; it specifies the number of the file to be accessed.

"@MOD_PTR" (bidirectional) both inputs the address of the original module and returns the address of the (possibly new) module that contains the changed frame.

"@FRAME_PTR" (bidirectional) both inputs the address of the original frame and returns the address of the changed frame.

"NEW_SIZE" specifies the WORD-value of the byte-count of the new frame. The new frame's FRAME_HEADER.SIZE field will be updated to the new value.

FUNCTION

This routine changes (increases or decreases) the size of a frame.

- o If size is increased, the new data area is not initialized.
- o If size is decreased, data is lost from the area that was decreased.

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

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

NOTE

You cannot call SAM_GROW_FRAME on a locked frame, since the growing process requires moving the frame.

NEVER write anything into a SAM_FRAME_HDR of a frame already in the database. You MUST use SAM_GROW_FRAME, SAM_MOVE_FRAME, or SAM_UPDATE_FRAME to modify the header for you.

EXAMPLE

```
STATUS = SAM_GROW_FRAME (ROOT_PTR, @MOD_PTR, @FRAME_PTR,
                         FRAME_SIZE+GROW_QUANTUM);
/* make frame bigger */
```

4.3.40 SAM_INITIATE

SAM_INITIATE allocates the buffer control structures and initializes the global data tables.

SYNTAX

```
SAM_INITIATE (@GLOBAL_PTR,
              CONFIG_PTR);
```

"@GLOBAL_PTR" returns a pointer to the global buffer definition in SAM_GLOBAL_DEF in SAMINT.ELT. This pointer must be retained by the caller, never modified, and passed, eventually, to the termination procedure.

"CONFIG_PTR" points to the configuration definition in SAM_CONFIG_DEF in SAMINT.ELT which has already been initalized by SAM_GET_CONFIG.

FUNCTION

This routine allocates the buffer control structures and initializes the global data tables.

This routine must be called following the GET_CONFIG call, which has initialized VIRGA with all the required configuration fields.

SAM_INITIATE needs to be called only once, no matter how many files are opened subsequently.

NOTE

None applicable.

EXAMPLE

```
STATUS = SAM_GET_CONFIG (@CONFIG_PTR);
STATUS = SAM_INITIATE (@SAM_GLOBAL_PTR, CONFIG_PTR)
```

4.3.41 SAM_INQUIRE_FRAME_ID

SAM_INQUIRE_FRAME_ID returns the next available frame ID number.

SYNTAX

```
SAM_INQUIRE_FRAME_ID (ROOT_PTR,
                      FRAME_TYPE,
                      @FRAME_ID);
```

"ROOT_PTR" is the pointer returned from OPEN or CREATE requests; it specifies the number of the file to be accessed.

"FRAME_TYPE" specifies the number of the desired frame type.

"@FRAME_ID" returns the next available identification number for a frame (a WORD-value).

FUNCTION

This routine returns the number of the next the number of the next frame that would be allocated if SAM_MAKE_FRAME or SAM_ADD_FRAME were to be called. It is used primarily to synthesize unique names for frames automatically.

NOTE

None applicable.

EXAMPLE

```
STATUS = SAM_INQUIRE_FRAME_ID (ROOT_PTR, SIDF_INSTANCE,
                               @FRAME_ID);
STATUS = FOO_MAKE_INSTANCE_FRAME (FRAME_ID,
                                  @INSTANCE_FRAME_PTR);
STATUS = SAM_ADD_FRAME (ROOT_PTR, @MODULE_PTR,
                        @INSTANCE_FRAME_PTR);
```

4.3.42 SAM_LOCK_FRAME

SAM_LOCK_FRAME locks a specified frame that is already in memory.

SYNTAX

```
SAM_LOCK_FRAME (ROOT_PTR,
                MODULE_PTR,
                FRAME_PTR);
```

"ROOT_PTR" is the pointer returned from OPEN or CREATE requests; it specifies the number of the file to be accessed.

"MODULE_PTR" points to the memory address of the module that contains the frame to be locked.

"FRAME_PTR" points to the memory address of the frame to be locked.

FUNCTION

This routine locks the specified frame in memory so that VIRGA will not swap it out if more memory is needed. You need to lock frames that you need to retain in memory for read and write procedures.

If the frame is already locked, its lock count is simply incremented. The frame will not be physically unlocked unless its lock count is zero.

NOTE

None applicable.

EXAMPLE

```
STATUS = SAM_FIND_FIRST (ROOT_PTR, SLTF_TEMPLATE,
                         @MOD_PTR, @FRAME_PTR);
STATUS = SAM_LOCK_FRAME (ROOT_PTR, MODULE_PTR,
                        FRAME_PTR);
/* fix frame in memory */
```

4.3.43 SAM_LOCK_MODULE

SAM_LOCK_MODULE locks a specified module that is already in memory.

SYNTAX

```
SAM_LOCK_MODULE (ROOT_PTR,
                 MODULE_PTR);
```

"ROOT_PTR" is the pointer returned from OPEN or CREATE requests; it specifies the number of the file to be accessed.

"MODULE_PTR" points to the address of the module to be locked.

FUNCTION

This routine locks a specified module that is already in memory so that VIRGA will not swap it out if more memory is needed.

An error is returned whether or not the module is already locked in memory.

As distinct from frame locks, a module lock cannot be nested.

NOTE

None applicable.

EXAMPLE

```
STATUS = SAM_LOCK_MODULE (ROOT_PTR, MODULE_PTR);
/* lock whole module into memory */
```

4.3.44 SAM LOCK MODULES BY TYPE

SAM LOCK MODULES BY TYPE locks all modules of the specified type that are already in memory.

SYNTAX

```
SAM LOCK MODULES BY TYPE (ROOT PTR,
                          MODULE TYPE);
```

"ROOT PTR" is the pointer returned from OPEN or CREATE requests; it specifies the number of the file to be accessed.

"MODULE TYPE" specifies the number of the desired module type.

FUNCTION

This routine locks all modules of the specified type that are already in memory. If MODULE TYPE = SAM NIL BYTE (ØFFH), this means ALL module types are to be locked (i.e., all modules in memory will be locked).

This procedure forces the specified type of module to remain in memory until unlocked.

NOTE

None applicable.

EXAMPLE

```
STATUS = SAM GET MODULE BY TYPE (ROOT PTR, SIDF INSTANCE);
STATUS = SAM LOCK MODULES BY TYPE (ROOT PTR, SIDF INSTANCE);
/* get all instances */
```

4.3.45 SAM_MAKE_FRAME

SAM_MAKE_FRAME adds a frame to the file but reads only part of the data into the frame.

SYNTAX

```
SAM_MAKE_FRAME (ROOT_PTR,
                @MODULE_PTR,
                @FRAME_PTR,
                DATA_SIZE);
```

"ROOT_PTR" is the pointer returned from OPEN or CREATE requests; it specifies the number of the file to be accessed.

"@MODULE_PTR" returns a pointer to the address of the module that contains the new frame.

"@FRAME_PTR" (bidirectional) both inputs the address of the original frame and returns a pointer to the address of the new frame. The frame size must be set to the size of the frame to be created.

"DATA_SIZE" specifies the size of the data area to be copied into the database.

FUNCTION

This routine adds a frame to the file but reads only part of the data into the frame. SAM_MAKE_FRAME is similar to SAM_ADD_FRAME but with a slight difference: you use SAM_MAKE_FRAME when you do not yet know all the data you want to put into the frame.

NOTE

None applicable.

EXAMPLE

```
FRAME.SIZE = SIZE (FRAME)+ OBJECT_COUNT * SIZE (OBJECT);
STATUS = SAM_MAKE_FRAME (ROOT_PTR, @MODULE_PTR, @FRAME_PTR,
                         SIZE (FRAME));
```

4.3.46 SAM_MAX_ID

SAM_MAX_ID finds the maximum frame ID number ever allocated in a particular frame type in a file.

SYNTAX

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

"ROOT_PTR" is the pointer returned from OPEN or CREATE requests; it specifies the number of the file to be accessed.

"FRAME_TYPE" specifies the number of the desired frame types to be scanned.

"@MAX_ID" returns the maximum frame ID number found.

FUNCTION

This routine finds the maximum frame ID number ever allocated in a particular frame type in a file.

If SAM_ADD_FRAME was called with FRAME.ID = SAM_NIL_WORD (so that VIRGA allocated the ID automatically), and frames 7, 8 and 9 (all of the same type) were thus allocated, and then frames 8 and 9 were deleted, the maximum ID found will be 9. If no ID has been allocated, MAX_ID = SAM_NIL_WORD.

NOTE

None applicable

EXAMPLE

```
STATUS = SAM_MAX_ID (ROOT_PTR, FRAME_TYPE, @MAX_ID);
IF MAX_ID <> SAM_NIL_WORD
   THEN
        DO I = 0 TO MAX_ID;
                .
                .
                .
        END;
```

4.3.47 SAM_MOVE_FRAME

SAM_MOVE_FRAME allows you to change (increase or decrease) the frame size and update the contents in one routine.

SYNTAX

```
SAM_MOVE_FRAME (ROOT_PTR,
                @MODULE_PTR,
                @FRAME_PTR,
                NEW_FRAME_HDR_PTR);
```

"ROOT_PTR" is the pointer returned from OPEN or CREATE requests; it specifies the number of the file to be accessed.

"@MODULE_PTR" (bidirectional) both inputs the address of the original module and returns a pointer to the address of the moved module.

"@FRAME_PTR" (bidirectional) both inputs the address of the original frame and returns a pointer to the address of the moved frame.

"NEW_FRAME_HDR_PTR" specifies a pointer to the frame header description, SAM_FRAME_HDR, in SAMFIL.ELT.

FUNCTION

SAM_MOVE_FRAME provides the attributes of SAM_GROW_FRAME and SAM_UPDATE_FRAME together without the need to make a private copy of the frame to be updated.

You probably want to change the SIZE and RECTANGLE (bounding box) of the frame.

To move a frame, follow this procedure:

- Retrieve the frame you want to move.
- Copy the old frame header to a scratch frame header structure.
- Change the SIZE and RECTANGLE fields in the new (copied) frame header.
- Unlock the frame, if necessary.
- Call SAM_MOVE_FRAME.

On return, the FRAME_PTR points to a frame of the new size and bounding box (RECTANGLE).

NOTE

If you requested a larger frame (the SIZE in the new frame header is bigger than the SIZE of the original frame), the new space is uninitialized. Extra space allocated when the frame is grown contains indterminate data. NEVER write anything into a SAM_FRAME_HDR of a frame already in the database. You MUST use SAM_GROW_FRAME, SAM_MOVE_FRAME, or SAM_UPDATE_FRAME to modify the header for you.

EXAMPLE

```
STATUS = SAM_FIND_FRAME (ROOT_PTR, @MODULE_PTR, @FRAME_PTR);
CALL MOVB (FRAME_PTR, @FRAME_HDR_COPY, SIZE(FRAME_HDR_COPY));
FRAME_HDR_COPY.SIZE = NEW_SIZE;
CALL MOVB (@NEW_RECTANGLE, @FRAME_HDR_COPY.X1, SIZE
          (NEW_RECTANGLE));
STATUS = SAM_MOVE_FRAME (ROOT_PTR, @MODULE_PTR, @FRAME_PTR,
                         @FRAME_HDR_COPY);
```

4.3.48 SAM_NEXT_NAME

SAM_NEXT_NAME finds the next name in the index that matches the specified input name and has the specified value of KEY2.

SYNTAX

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

"ROOT_PTR" is the pointer returned from OPEN or CREATE requests; it specifies the number of the file to be accessed.

"FRAME_TYPE" specifies the number of the frame type which has been indexed.

"NAME_LEN" specifies the length of the name to be found.

"NAME_PTR" specifies the pointer to the name to be found.

"KEY2" specifies the value of the KEY2 for the desired frame.

"@INDEX_NAME_PTR" (bidirectional) both inputs the address of the first name found in the index (the @INDEX_NAME_PTR from SAM_FIRST_NAME) and returns a pointer to the address of the next name found in the index.

"@INDEX_INFO_PTR" (bidirectional) both inputs the address of the index entry definition of the first name found in the index (the @INDEX_INFO_PTR from SAM_FIRST_NAME) and returns a pointer to the index entry definition, SAM_INDEX_ENTRY_DEF, in SAMNAM.ELT.

FUNCTION

This routine finds the next name in the index that matches the specified input name and has the specified value of KEY2.

The routine is the same as SAM_FIRST_NAME, but it searches the name index starting at the entry whose index is (MARK+1) (see Figure 2 - 13 and Figure 2 - 14 for the "Index Header Diagram" and Symtab Diagram" respectively). The caller may, if desired,

directly set the MARK field, thereby determining the search starting point.

Following an initial SAM_FIRST_FRAME call, successive SAM_NEXT_NAME calls will find all names which match the specified input name. When no more names are found, VIRGA returns SAM_NO_SUCH_FRAME.

NOTE

This first/next pair has the same limitation that SAM_FIND_FIRST/SAM_FIND_NEXT has: namely, any intervening VIRGA calls may swap out the index frame and invalidate the next_name function. If this is not desired, the user may do one of three things:

(1) lock the index frame after finding it with SAM_FIND_INDEX_FRAME,

(2) write the application code so that no VIRGA calls intervene between first_name and next_name calls,

(3) remember the MARK field after each first/next call, and reset it before the succeeding next_name call.

If the name index is created with KEY2_OFFSET = SAM_NIL_WORD, the name index is scattered across multiple index frames. To find all the name matches, the caller must repeat the search for every index frame. A null value for KEY2 is therefore not allowed for SAM_NEXT_NAME.

EXAMPLE

```
STATUS = SAM_FIRST_NAME (ROOT_PTR, SLTF_TEMPLATE, 5,
                         @('BOZO*'), @INDEX_NAME_PTR,
                         @INDEX_INFO_PTR);
DO WHILE STATUS = SYS_SUCCESS;
   ID = INDEX_INFO.ID
/* process this frame */
STATUS = SAM_NEXT_NAME (ROOT_PTR, SLTF_TEMPLATE, 5,
                        @('BOZO*'), @INDEX_NAME_PTR,
                        @INDEX_INFO_PTR);
END;
```

4.3.49 SAM_NUM_FRAMES

SAM_NUM_FRAMES reports the total number of frame ID's that exist for a frame type in one file.

SYNTAX

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

"ROOT_PTR" is the pointer returned from OPEN or CREATE requests; it specifies the number of the file to be accessed.

"FRAME_TYPE" specifies the number of the desired frame type to be scanned.

"@NUM" returns the number of frame ID's (WORD-value) in existence for the specified frame type in this file.

FUNCTION

This routine reports the total number of frame ID's that exist for a frame type in one file.

If no frames have been deleted and if no frames have been allocated with ID's out of sequence (by user-specifed IDs), this number will equal MAX_ID+1, where MAX_ID is the value returned by SAM_MAX_ID.

NOTE

None applicable.

EXAMPLE

```
STATUS = SAM_NUM_FRAMES (ROOT_PTR, SLTF_TEMPLATE,
                         @TEMPLATE_COUNT);
/* number of template frames in database */
```

4.3.50 SAM_NUM_LOCKS

SAM_NUM_LOCKS reports the number of locked frames (or modules).

SYNTAX

```
SAM_NUM_LOCKS (ROOT_PTR,
               @NUM);
```

"ROOT_PTR" is the pointer returned from OPEN or CREATE requests; it specifies the number of the file to be accessed.

"@NUM" returns the number (WORD-value) of frame ID's currently locked.

FUNCTION

This routine reports the number of locked frames, or modules since locking a frame actually locks the entire module.

The SID library manager uses this call to determine whether a library is currently in use by any of the clients of SID.

NOTE

None applicable.

EXAMPLE

```
STATUS = SAM_NUM_LOCKS (ROOT_PTR, @NUMBER_LOCKS);
IF NUMBER_LOCKS = 0
   THEN
/* not in use, database may be closed */
```

4.3.51 SAM_OPEN

SAM_OPEN makes a pre-existing VIRGA file available for access.

SYNTAX

```
SAM_OPEN (@ROOT_PTR,
          GLOBAL_PTR,
          PATH_NAME_PTR,
          MODE);
```

"@ROOT_PTR" returns a pointer to the name of the file to be accessed. This pointer is created by VIRGA as part of VIRGA's internal management system. It must be saved, never modified, and passed to VIRGA for each and every request concerning this file.

"GLOBAL_PTR" specifies the address to the global buffers returned from @GLOBAL_PTR in a previous SAM_INITIATE request.

"PATH_NAME_PTR" specifies the full pathname to the file returned by @ROOT_PTR above.

"MODE" specifies read, write, or other modes defined by the Daisy file system.

FUNCTION

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

EXAMPLE

```
STATUS = SAM_OPEN (@ROOT_PTR, ENVIRON.SAM_PTR, PATH_NAME_PTR, FSYS_WRITE_MODE);
```

4.3.52 SAM_OPEN_SHELL_PATH

SAM_OPEN_SHELL_PATH converts a SHELL style null-terminated (string) pathname into a file descriptor and calls SAM_OPEN to make the indicated pre-existing VIRGA file available for access.

SYNTAX

```
SAM_OPEN_SHELL_PATH (@ROOT_PTR,
                     GLOBAL_PTR,
                     SHELL_PATH_PTR,
                     SHELL_PATH_LEN,
                     MODE);
```

"@ROOT_PTR" returns a pointer to the name of the file to be accessed. This pointer is created by VIRGA as part of VIRGA's internal management system. It must be saved, never modified, and passed to VIRGA for each and every request concerning this file.

"GLOBAL_PTR" specifies the address to the global buffers returned from @GLOBAL_PTR in a previous SAM_INITIATE request.

"SHELL_PATH_POINTER" points to a null-terminated string SHELL pathname. This may be the pointer returned by a previous SHL_GET_ARGV or SHL_GET_UNARGV call. (See example below.)

"SHELL_PATH_LEN" length of SHELL (string) pathname pointed to by SHELL_PATH_PTR.

"MODE" specifies read, write, or other modes defined by the Daisy file system such as fsys_write_mode.

FUNCTION

SAM_OPEN_SHELL_PATH converts a SHELL style (string) pathname to a file descriptor, calls SAM_OPEN, and calls SYS_DELETE_SEGMENT to delete memory allocated by the pathname conversion call. Status is passed back from SAM_OPEN. SAM_OPEN makes a pre-existing VIRGA file available for access. If the file does not exist, no action will be taken, and an error will be returned.

EXAMPLE

```
STATUS = SHL_GET_UNARGV (1, @SHELL_PATH_POINTER);
```

```
SHELL_PATH_LEN = UTL_STRLEN (SHELL_PATH_PTR);
                      .
                      .
                      .
STATUS = SAM_OPEN_SHELL_PATH (@ROOT_PTR,
                              GLOBAL_PTR,
                              SHELL_PATH_PTR,
                              SHELL_PATH_LEN,
                              MODE);
```

4.3.53 SAM_SEARCH_FIRST_NAME

SAM_SEARCH_FIRST_NAME searches a list of files (rather than just one file) for the first name in a name index which matches a specified frame type and has the specified KEY2.

SYNTAX

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

"FILES_PTR" specifies the array of root pointers to VIRGA files contained in the structure, SAM_FILES_DEF, in SAMFIL.ELT and which the caller initializes.

"FRAME_TYPE" specifies the number of the desired frame type which has been indexed.

"NAME_LEN" specifies the length of the name to be found.

"NAME_PTR" specifies the pointer to the name to be found.

"KEY2" specifies the value of the KEY2 for the desired frame.

"@INDEX_NAME_PTR" returns a pointer to the address of the name found in the index.

"@INDEX_INFO_PTR" returns a pointer to the index entry definition, SAM_INDEX_ENTRY_DEF, which contains the length of the name and the corresponding frame's ID.

FUNCTION

SAM_SEARCH_FIRST_NAME searches a list of files (rather than just one file) for the first name in a name index which matches a specified frame type and has the specified KEY2 (SAM_NIL_WORD if none). SAM_SEARCH_FIRST_NAME performs the same function as SAM_FIRST_NAME except it searches all files instead of one file.

The list of VIRGA files is specified by SAM_FILES_DEF (see SAMFIL.ELT) shown in Figure 4 - 2 below.

```
SAM$FILES$DEF   LITERALLY
     'NUM$FILES   WORD,
     MARK         WORD,
     FILES$PTR    POINTER', /* to array of root ptrs */
```

Figure 4 - 2 Files Definition Structure

The first field specifies the total number of files, while the last is a pointer to an array of root$ptrs, one for each file.

The mark field is an index into this array which can be set by the user or by the SEARCH routines to indicate which file to use or which file contained the found name. It is similar to the MARK field in the sym$structure, but differs in that it is initialized to zero (rather than sym$initial$mark = FFFFH). Also, to allow SAM$SEARCH$NEXT$NAME to continue searching within the same file in which the previous name was found, searches begin with the file to which mark points, rather than the file after that one. That is, the sym$struc.mark indicates one BEFORE the next entry to scan, while the files$def.mark indicates EXACTLY the next file to be searched.

At the start of the call, the routine will set MARK (the files$def.mark) to 0 (first file).

Upon return, VIRGA has set MARK to the index of the file in which the name was found. If any of the root$ptrs is zero, VIRGA will simply ignore that file and continue on to next file.

KEY2 may be set to SAMNILWORD if there is no KEY2 for that name index.

The procedure starts at the top of the index for the first file and searches indexes until it either finds the name or returns an error. The routine will re-start at the beginning of the index for each file as it steps down the file list.

The list is specified by the SAM$FILES$DEF structure, which the caller must initialize. After the call, the files$def.mark will be set to the index of the file in which the name was found.

NOTE

This procedure allows searching through a set of libraries for a named frame.

EXAMPLE

No known example.

4.3.54 SAM_SEARCH_NEXT_NAME

SAM_SEARCH_NEXT_NAME searches a list of files (rather than just one file) for the next name in a name index which matches the specified name and has the specified KEY2 (SAM_NIL_WORD if none).

SYNTAX

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

"FILES_PTR" specifies the array of root pointers to VIRGA files contained in the structure, SAM_FILES_DEF, in SAMFIL.ELT and which the caller initializes.

"FRAME_TYPE" specifies the number of the desired frame type which has been indexed.

"NAME_LEN" specifies the length of the name to be found.

"NAME_PTR" specifies the pointer to the name to be found.

"KEY2" specifies the value of the KEY2 for the desired frame.

"@INDEX_NAME_PTR" (bidirectional) both inputs the address of the first name found in the index (the @INDEX_NAME_PTR from SAM_SEARCH_FIRST_NAME) and returns a pointer to the address of the next name found in the index.

"@INDEX_INFO_PTR" (bidirectional) both inputs the address of the index entry definition of the first name found in the index (the @INDEX_INFO_PTR from SAM_SEARCH_FIRST_NAME) and returns a pointer to the index entry definition, SAM_INDEX_ENTRY_DEF, in SAMNAM.ELT.

FUNCTION

SAM_SEARCH_NEXT_NAME operates just like SAM_SEARCH_FIRST_NAME, except that it continues where the last SAM_SEARCH_FIRST_NAME or SAM_SEARCH_NEXT_NAME left off.

Another way to view this routine is that SAM_SEARCH_NEXT_NAME is just like the SAM_NEXT_NAME routine except that it searches a list of files.

The effect of SAM_SEARCH_FIRST_NAME followed by successive SAM_SEARCH_NEXT_NAME requests is to find all matching names in all files.

The procedure starts at the first file and searches indexes until it either finds the next name or returns an error. The routine will re-start at the beginning of the index for each file as it steps down the file list.

The list is specified by the SAM_FILES_DEF structure, which the caller must initialize. After the call, the files_def.mark will be set to the index of the file in which the name was found.

NOTE

None applicable.

EXAMPLE

No known example.

4.3.55 SAM_SET_DIRTY_MODULE

SAM_SET_DIRTY_MODULE sets the dirty flag on the specified module so that it will (eventually) be written back to disk.

SYNTAX

```
SAM_SET_DIRTY_MODULE (ROOT_PTR,
                      MODULE_PTR);
```

"ROOT_PTR" is the pointer returned from OPEN or CREATE requests; it specifies the number of the file to be accessed.

"MODULE_PTR" points to the address of the module to be (eventually) written to disk.

FUNCTION

This routine sets the dirty flag on the specified module so that it will (eventually) be written back to disk.

SAM_SET_DIRTY_MODULE is provided to give the user direct control of the write-back of a module.

NOTE

SAM_SET_DIRTY_MODULE does not work with the SAM_UNDO_TRANSACTION routine; instead, use SAM_DIRTY_FRAME to set the dirty flag if you intend to use transaction calls.

EXAMPLE

```
STATUS = SAM_FIND_FRAME (ROOT_PTR, @MODULE_PTR, @FRAME_PTR);
FRAME.DATA_FIELD = NEW_VALUE;
STATUS = SAM_SET_DIRTY_MODULE (ROOT_PTR, MODULE_PTR);
```

4.3.56 SAM_SET_INDEX_TYPE

SAM_SET_INDEX_TYPE temporarily changes the index type for frame name IDs.

SYNTAX

```
SAM_SET_INDEX_TYPE (ROOT_PTR,
                    FRAME_TYPE,
                    INDEX_TYPE);
```

"ROOT_PTR" is the pointer returned from OPEN or CREATE requests; it specifies the number of the file to be accessed.

"FRAME_TYPE" specifies the number of the desired frame type

"INDEX_TYPE" specifies the index frame type specified when the index was created with SAM_CREATE_INDEX.

FUNCTION

SAM_SET_INDEX_TYPE temporarily changes the index type for frame name IDs.

NOTE

This routine was a temporary solution used for the library catalog procedure.

EXAMPLE

Do not use this routine.

4.3.57 SAM_START_TRANSACTION

SAM_START_TRANSACTION specifies the beginning of a transaction sequence of calls that must be made as a group.

SYNTAX

SAM_START_TRANSACTION (ROOT_PTR);

"ROOT_PTR" specifies a pointer to the root of the SAM database.

FUNCTION

SAM_START_TRANSACTION allows you to specify the beginning of a series of updates to the database that comprise a transaction. SAM_FINISH_TRANSACTION allows you to signal the end of a series of updates to the database that comprise a transaction. After SAM_START_TRANSACTION has been called and before SAM_FINISH_TRANSACTION is called, you may undo the effects of the current transaction by calling SAM_UNDO_TRANSACTION.

NOTE

Only one transaction in one database may be pending at a time.

EXAMPLE

```
/* get new command */
/* validate new command */
STATUS = SAM_FINISH_TRANSACTION (ROOT_PTR);
/* disallow any more undoing of previously issued commands */
STATUS = SAM_START_TRANSACTION (ROOT_PTR);
/* create new transaction log for this command */
```

4.3.58 SAM_SWITCH_MUX

SAM_SWITCH_MUX allows you to switch between VIRGA files, identified by cell ID, in cell libraries.

SYNTAX

```
SAM_SWITCH_MUX (CELL_ID,
                ROOT_PTR);
```

"CELL_ID" specifies the cell ID number for the file returned from @CELL_ID in SAM_CREATE_CELL_DIR.

"ROOT_PTR" is the pointer returned from OPEN or CREATE requests; it specifies the number of the file to be accessed.

FUNCTION

This routine allows you to switch between VIRGA files, identified by cell ID, in cell libraries. SAM_SWITCH_MUX acts as a multiplexer between the cells (i.e., VIRGA files) in the cell libraries.

NOTE

If CELL_ID is specified as SAM_NIL_WORD, the file is treated as a non-library file.

EXAMPLE

```
STATUS = SAM_SWITCH_MUX (SAM_NIL_WORD, ROOT_PTR);
/* find cell directory frame */
STATUS = SAM_SWITCH_MUX (CELL_DIRECTORY.ID, ROOT_PTR);
/* switch to library cell */
```

4.3.59 SAM_TERMINATE

SAM_TERMINATE deallocates all global structures for a file.

SYNTAX

SAM_TERMINATE (GLOBAL_PTR);

"GLOBAL_PTR" specifies the global structure returned by SAM_INITIATE; it must be supplied to this routine.

FUNCTION

This routine deallocates all global structures for a file. This routine must be called before the program which initiated VIRGA terminates. Call it only once. It will search for any files that the user has failed to close, and close them.

NOTE

None applicable.

EXAMPLE

```
STATUS = SAM_TERMINATE (ENVIRON.SAM_PTR);
```

4.3.60 SAM_UNDO_TRANSACTION

SAM_UNDO_TRANSACTION reverses the action of a group of calls that form a "transaction."

SYNTAX

```
SAM_UNDO_TRANSACTION (ROOT_PTR);
```

"ROOT_PTR" points to the SAM database root.

FUNCTION

SAM_UNDO_TRANSACTION allows you to reverse the action of a group of calls started after a SAM_START_TRANSACTION call and revert the status of the database back to what it was before the SAM_START_TRANSACTION call. See Section 3.5 for information about how transactions are used.

NOTE

Performing a second 'UNDO' will restore the database to the state it was in before the first 'UNDO,' so that UNDO toggles the effect of the updates in the current transaction.

EXAMPLE

```
STATUS = SAM_GET_TRANSACTION_LOG (ROOT_PTR,
                                  @NEW_FRAME_COUNT,
                                  @NEW_FRAME_LIST_PTR,
                                  @UPD_FRAME_COUNT,
                                  @UPD_FRAME_LIST_PTR);
/* now erase object from screen */
STATUS = SAM_UNDO_TRANSACTION (ROOT_PTR);
/* get back old frames */
STATUS = SAM_GET_TRANSACTION_LOG (ROOT_PTR,
                                  @NEW_FRAME_COUNT,
                                  @NEW_FRAME_LIST_PTR,
                                  @UPD_FRAME_COUNT,
                                  @UPD_FRAME_LIST_PTR);
/* now display old objects on the screen */
```

4.3.61 SAM_UNLOCK_FRAME

SAM_UNLOCK_FRAME unlocks a frame that was previously locked in memory.

SYNTAX

```
SAM_UNLOCK_FRAME (ROOT_PTR,
                  MODULE_PTR,
                  FRAME_PTR);
```

"ROOT_PTR" is the pointer returned from OPEN or CREATE requests; it specifies the number of the file to be accessed.

"MODULE_PTR" points to the address of the module that contains the locked frame.

"FRAME_PTR" points to the address of the locked frame.

FUNCTION

This routine unlocks a frame that was previously locked in memory.

The lock count is decremented. If the lock count becomes zero, the frame is physically unlocked. If the lock count was zero, an error is returned.

NOTE

None applicable.

EXAMPLE

```
STATUS = SAM_LOCK_FRAME (ROOT_PTR, MODULE_PTR, FRAME_PTR);
/* now the frame is protected from being swapped out */
STATUS = SAM_UNLOCK_FRAME (ROOT_PTR, MODULE_PTR, FRAME_PTR);
/* now the frame may be swapped out */
```

4.3.62 SAM_UNLOCK_MODULE

SAM_UNLOCK_MODULE unlocks a module that was previously locked in memory.

SYNTAX

```
SAM_UNLOCK_MODULE (ROOT_PTR,
                   MODULE_PTR);
```

"ROOT_PTR" is the pointer returned from OPEN or CREATE requests; it specifies the number of the file to be accessed.

"MODULE_PTR" points to the address of the locked module.

FUNCTION

This routine unlocks a module that was previously locked in memory.

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

NOTE

None applicable.

EXAMPLE

```
STATUS = SAM_LOCK_FRAME (ROOT_PTR, MODULE_PTR, FRAME_PTR);
/* now the frame is protected from being swapped out */
STATUS = SAM_UNLOCK_FRAME (ROOT_PTR, MODULE_PTR, FRAME_PTR);
/* now the frame may be swapped out */
```

4.3.63 SAM_UNLOCK_MODULES_BY_TYPE

SAM_UNLOCK_MODULES_BY_TYPE unlocks all previously locked modules (in memory) of the specified type.

SYNTAX

```
SAM_UNLOCK_MODULES_BY_TYPE (ROOT_PTR,
                            MODULE_TYPE);
```

"ROOT_PTR" is the pointer returned from OPEN or CREATE requests; it specifies the number of the file to be accessed.

"MODULE_TYPE" specifies the number of the desired module type.

FUNCTION

This routine unlocks all previously locked modules (in memory) of the specified type.

This procedure allows the specified type of module to be flushed to disk.

If MODULE_TYPE = SAM_NIL_BYTE (ØFFH), this means ALL types, except that type = nil will not unlock VIRGA's internal inventories (module_type = SAMM_CONTROL).

NOTE

None applicable.

EXAMPLE

```
STATUS = SAM_LOCK_MODULES_BY_TYPE (ROOT_PTR, SIDM_GEOMETRY);
                    .
                    .
                    .
STATUS = SAM_UNLOCK_MODULES_BY_TYPE (ROOT_PTR, SIDM_GEOMETRY);
```

4.3.64 SAM_UPDATE_FRAME

SAM_UPDATE_FRAME overwrites (and possibly reallocates) an existing frame in the file.

SYNTAX

```
SAM_UPDATE_FRAME (ROOT_PTR,
                  @MODULE_PTR,
                  @FRAME_PTR,
                  NEW_FRAME_PTR);
```

"ROOT_PTR" is the pointer returned from OPEN or CREATE requests; it specifies the number of the file to be accessed.

"@MODULE_PTR" (bidirectional) both inputs the address of the previous module and returns a pointer to the address of the newly updated module.

"@FRAME_PTR" (bidirectional) both inputs the address of the previous frame and returns a pointer to the address of the newly updated frame.

"NEW_FRAME_PTR" specifies the address of the copy of the frame to be updated.

FUNCTION

This routine overwrites (and possibly reallocates) an existing frame in the file. The routine will allocate a new frame if necessary.

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

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

NOTE

NEVER write anything into a SAM_FRAME_HDR of a frame already in the database. You MUST use SAM_GROW_FRAME, SAM_MOVE_FRAME, or SAM_UPDATE_FRAME to modify the header for you.

EXAMPLE

```
STATUS = SAM_FIND_FRAME (ROOT_PTR, @MODULE_PTR, @FRAME_PTR,
                         FRAME_TYPE, FRAME_ID);
CALL MOVB (FRAME_PTR, @FRAME_COPY, FRAME.SIZE);
FRAME_COPY.SIZE = NEW_SIZE;
FRAME_COPY.STUFF = NEW_STUFF;
STATUS = SAM_UPDATE_FRAME (ROOT_PTR, MODULE_PTR, @FRAME_PTR,
                           @FRAME_COPY);
```

```
GDB_DELETE_BY_TYPE (ROOT_PTR,
                    FRAME_TYPE);

GDB_DELETE_FRAME (ROOT_PTR,
                  MODULE_PTR,
                  FRAME_PTR);

GDB_DELETE_INDEX (ROOT_PTR,
                  FRAME_TYPE);

GDB_DISPOSE_MODULE (ROOT_PTR,
                    MODULE_PTR);

GDB_DISPOSE_MODULES_BY_TYPE (ROOT_PTR,
                             MODULE_TYPE);

GDB_FIND_FIRST (ROOT_PTR,
                @MODULE_PTR,
                @FRAME_PTR,
                FRAME_TYPE);

GDB_FIND_FRAME (ROOT_PTR,
                @MODULE_PTR,
                @FRAME_PTR,
                FRAME_TYPE,
                FRAME_ID);

GDB_FIND_FRAMES_IN_WINDOW (ROOT_PTR,
                           RECTANGLE_PTR,
                           FRAME_TYPE,
                           RESULT_SIZE,
                           RESULT_PTR);

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

GDB_FIND_NEXT (ROOT_PTR,
               @MODULE_PTR, /* bidirectional */
               @FRAME_PTR); /* bidirectional */

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

GDB_FLUSH (ROOT_PTR);
```

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

GDB_GET_CONFIG (@CONFIG_PTR);
```

This routine now returns only the configuration file.

```
GDB_GET_MODULES_BY_TYPE (ROOT_PTR,
                         MODULE_TYPE);

GDB_GET_PATHNAME (ROOT_PTR,
                  @PATHNAME_PTR);

GDB_GET_TIMESTAMP (ROOT_PTR,
                   @TIMESTAMP);

GDB_GROW_FRAME (ROOT_PTR,
                @MOD_PTR,   /* bidirectional */
                @FRAME_PTR, /* bidirectional */
                NEW_SIZE);

GDB_INITIATE (@GLOBAL_PTR,
              CONFIG_PTR);

GDB_LOCK_FRAME (ROOT_PTR,
                MODULE_PTR,
                FRAME_PTR);

GDB_LOCK_MODULE (ROOT_PTR,
                 MODULE_PTR);

GDB_LOCK_MODULES_BY_TYPE (ROOT_PTR,
                          MODULE_TYPE);

GDB_MAX_ID (ROOT_PTR,
            FRAME_TYPE,
            @MAX_ID);

GDB_NEXT_NAME (ROOT_PTR,
               FRAME_TYPE,
               NAME_LEN,
               NAME_PTR,
               KEY2,
               @INDEX_NAME_PTR,  /* bidirectional */
               @INDEX_INFO_PTR); /* bidirectional */

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

```
GDB_OPEN (@ROOT_PTR,
          GLOBAL_PTR,
          PATH_NAME_PTR,
          MODE);

GDB_OPEN_SHELL_PATH (@ROOT_PTR,
                     GLOBAL_PTR,
                     SHELL_PATH_PTR,
                     SHELL_PATH_LEN,
                     MODE);
```

This routine converts a shell pathname (a string) to a file pathname (file descriptor) and calls GDB_OPEN.

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

GDB_SEARCH_NEXT_NAME (FILES_PTR,
                      FRAME_TYPE,
                      NAME_LEN,
                      NAME_PTR,
                      KEY2,
                      @INDEX_NAME_PTR,  /* bidirectional */
                      @INDEX_INFO_PTR); /* bidirectional */

GDB_SET_DIRTY_MODULE (ROOT_PTR,
                      MODULE_PTR);

GDB_TERMINATE (GLOBAL_PTR
               @NUM_FILES);
```

If the termination is successful, the number of files that were closed is returned. A negative status is returned if an error occurs.

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

GDB_UNLOCK_MODULE (ROOT_PTR,
                   MODULE_PTR);

GDB_UNLOCK_MODULES_BY_TYPE (ROOT_PTR,
                            MODULE_TYPE);

GDB_UPDATE_FRAME (ROOT_PTR,
                  @MODULE_PTR,   /* bidirectional */
                  @FRAME_PTR,    /* bidirectional */
                  NEW_FRAME_PTR);
```

B SUMMARY OF PUBLIC MODULES

The VIRGA (SAM) xxxx.ELT and xxxx.EXT files are listed below:

xxxx.ELTs:

SAMERR.ELT—error (status) literals

SAMFIL.ELT—all constants and structures except name-index

SAMINT.ELT—internal to VIRGA; do NOT use or change this file.

SAMNAM.ELT—name-index-related constants and structures

xxxx.EXTs:

SAMACC.EXT—buffer control procedures (flush, flush_and_dispose, flush_all, dispose_all, allocate_module, allocate_frame, dispose_module, dispose_module_by_type, lock_frame, unlock_frame, lock_module, unlock_module, lock_modules_by_type, unlock_modules_by_type, set_dirty_module, clear_dirty_module)

SAMCELL.EXT—interface between library manager (SID) and VIRGA (SAM)

SAMCNFG.EXT—configuration initialization (get_config)

SAMDAT.EXT—data access procedures (find_frame, add_frame, make_frame, update_frame, move_frame, grow_frame, delete_frame, delete_by_type, find_first, find_next, get_modules_by_type, dirty_frame)

SAMDUMP.EXT—dumps out SAM (VIRGA) tables (dump_global)

SAMFIL.EXT—file manipulation procedures (initiate, terminate, create, open, close, dump_global)

SAMHIL.EXT—higher level procedures (find_frames_in_window, find_first_frame_in_box, find_next_frame_in_box)

SAMIDX.EXT—create/delete name index (delete_index,

create_index, create_box_index)

SAMNAM.EXT—find/search name index (find_index_frame, first_name, next_name, search_first_name, search_next_name)

SAMTRANS.EXT—interface to VIRGA transaction management system

SAMUTIL.EXT—utility procedures (get_pathname, max_id, inquire_frame_id, num_frames, get_timestamp, get_aft, add_frame_type, add_new_frame_type, num_locks)

B.1 The Source Code Modules and Their Functions

The following files may be found in .../SAM/P86 on integration systems. Each file contains a mixture of public VIRGA procedures and internal procedures.

SAMINIT—Initiate/Terminate VIRGA operation

SAMYINCFG—Set Configuration Table, assuming that the Symtab for it already exists.

SAMCNFG—Create and Initialize Configuration Table

SAMHILR—The Higher Level Routines: SAM_FIND_FRAMES_IN_WINDOW, SAM_FIND_FIRST_FRAME_IN_BOX, SAM_FIND_NEXT_FRAME_IN_BOX.

SAMNIDX—Name Index manipulation routines

SAMNAME—Find and match names in name index

SAMNCRT—Create/Delete a Name Index

SAMFRAM—Find/Add/Update/Delete frames

SAMLOCK—Lock/Unlock frames and modules

SAMDRTY—Set/Clear the dirty flag on a frame or module

SAMNEXT—Find first or next frame

SAMUTL—A collection of utility routines

SAMFIL—Create/Open/Close VIRGA files

SAMEXP—Expand inventory frames and modules

SAMALOC—Allocate/De-allocate frames and modules

SAMMOD—Intra-module memory manager

SAMBUFR—Buffer Manager: Read/Write/Swap/Flush/Get modules.

B.2 Source Code Module Inter-Connections

The figure below shows a drawing of how the source modules, which carry the VIRGA procedures, are related to one another.

Figure 1 The Source Module Inter-Connections

C CAVEATS

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

C.1 Find/Next Utilities Return a TEMPORARY Pointer

(See Section 4.1.5 "Buffer Control Procedures" and Section 4.1.3 "Data Access Procedures.")

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

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

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

The rule is:

> Consider the pointer temporary. If you need to retain the data, copy it.

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

Therefore, the best policy is to:

(1) Lock the frame (or module).

(2) Work with the data in the frame.

(3) Unlock the frame (or module).

C.2 Use Read-Only for NEXT Routines

The NEXT routines are provided for read-only use; that is, for scanning through all the nets. They will NOT work if the NEXT requests are interleaved with requests for the same frame type. The frame_ptr returned points directly to a frame within the module and should be considered temporary.

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

The following is an example of the problem created by using FIND routines and the temporary pointer.

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

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

There are various ways around the temporary pointer problem. For example, you could make the next request immediately, make a copy of the frame ID, and then work on the current frame. After updating the current frame, make a direct (find_frame) request for the next and loop. Alternatively, you could sequence through the frames in order of their IDs, rather than by their physical order. However, this is slower than using the FIRST/NEXT combination.

C.3 Disk Update Depends on Pointer and Flag

(See Section 4.1.5 "Buffer Control Procedures.")

With the pointer returned from FIND, NEXT, etc. routines, the user can modify the data to which it points. However, this must be done ONLY with caution.

- First, since that pointer is temporary (see caveat above), it is the user's responsibility to ensure that in fact the pointer is still valid (perhaps by locking the frame or module).

- Second, it is the user's responsibility to set the dirty module flag.

C.4 Need to Periodically Call SAM_FLUSH

While an application program in executing, various parts of the database are being modified. Some of those changes will be updated on disk, others won't be. If the system crashes, the file will be left partially updated, resulting in a possibly invalid database. This is very undesirable!

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

For ILE users, the command table may be used to set up for flushing the database automatically after specific commands. If the database was fully flushed at the time a crash occurred, the UNLOCK command of DQL or PEQ could be used to recover. If the database was only partially flushed, any recovery attempt on a crashed database would probably make things worse.

C.5 Locking is Incremented

Locking is incremented. You may lock a frame any number of times. The frame becomes physically unlocked only when the number of unlock operations equals the number of lock operations.

C.6 Toggling of Search Direction

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

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

D THE BUCKET MANAGER

The problem of finding the best frame in which to allocate a geometric object and finding the best module in which to allocate a frame are essentially similar. A subsystem called the bucket manager determines how to allocate new data to a bucket. The heart of the bucket manager is a discriminant function that calculates whether to create a new bucket or use an existing bucket.

```
BUM_EVALUATE_BUCKET: PROCEDURE (DATA_BOX_PTR,
                                BUCKET_BOX_PTR,
                                DATA_SIZE,
                                BUCKET_SIZE,
                                MAX_BUCKET_SIZE,
                                ALLOC_PARAMS_PTR,
                                NEW_BUCKET_BOX_PTR
                                ) INTEGER;
```

where:

DATA_BOX_PTR is the bounding box of data to be added.
BUCKET_BOX_PTR is the current bounding box of the bucket being considered.
DATA_SIZE is the number of bytes of data to be added.
BUCKET_SIZE is the number of bytes of space available in the bucket.
MAX_BUCKET_SIZE is the size of largest bucket to be created.
ALLOC_PARAMS_PTR is the structure containing parameters of the algorithm.
NEW_BUCKET_BOX_PTR is the returned bucket bounding box.

This routine returns SYS_SUCCESS if data fits into existing bucket,
BUM_STAT_GROW_BUCKET if data should be added, but the bucket bounding box will have to grow,
BUM_ERR_NO_ROOM if data cannot fit in the bucket, or
BUM_ERR_TOO_BIG if data should not be put in the bucket because the bounding box would become too big.

BUM_EVALUATE_BUCKET is a black-box function that operates on its input data and produces the answer yes or no.

D.1 Allocating Frames to Modules

BUM_EVALUATE_BUCKET is called inside SAM_ALLOCATE_FRAME_ANYWHERE when a frame of geometric type is to be allocated to a module. The arguments are

DATA_BOX	=	FRAME.BOUNDING_BOX
BUCKET_BOX	=	MODULE_INV.BOUNDING_BOX
DATA_SIZE	=	FRAME.SIZE
BUCKET_SIZE	=	MODULE.FREE_BYTES
MAX_BUCKET_SIZE	=	MODULE.SIZE
ALLOC_PARAMS	=	ROOT.ALLOC_PARAMS

All modules of the appropriate type, starting from FRAME_TYPE_INV.MIN_MODULE_NUM and continuing up to FRAME_TYPE_INV.MAX_MODULE_NUM are evaluated by BUM_EVALUATE_BUCKET.

If any module is found with BUM_EVALUATE_BUCKET = SYS_SUCCESS, the frame is added to that module. If a module is found with BUM_EVALUATE_BUCKET = BUM_STAT_GROW_BUCKET, the frame is added to that module and MODULE_INV.BOUNDING_BOX is updated to NEW_BUCKET_BOX. If BUM_EVALUATE_BUCKET is neither equal to SYS_SUCCESS nor BUM_STAT_GROW_BUCKET, a new module is created and the frame is added to that.

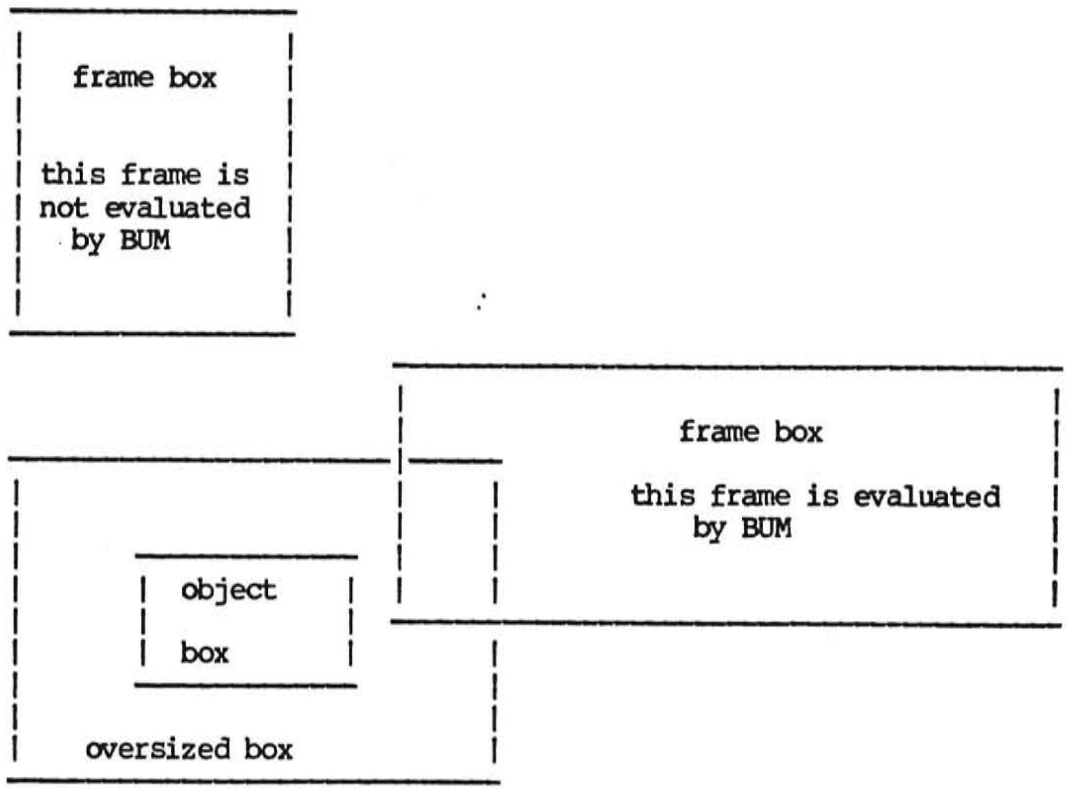
D.2 Allocating Objects to Frames

BUM_EVALUATE_BUCKET is called by BUM_FIND_BEST_FRAME (another part of the bucket manager) when a geometric object and its parameters are to be added to a geometric frame. The arguments are:

DATA_BOX	=	OBJECT.BOUNDING_BOX (including parameters)
BUCKET_BOX	=	FRAME.BOUNDING_BOX
DATA_SIZE	=	OBJECT.SIZE (including parameters)
BUCKET_SIZE	=	QED_CONFIG.MAX_FRAME_SIZE-FRAME.SIZE
MAX_BUCKET_SIZE	=	QED_CONFIG.MAX_FRAME_SIZE (optimum frame size = 1024)
ALLOC_PARAMS	=	QED_CONFIG.ALLOC_PARAMS

The question remains of how BUM_FIND_BEST_FRAME selects which frames are to be evaluated. The method is used to FIND_[FIRST/NEXT]_FRAME_IN_BOX to find all existing frames which are candidates.

The box used is QED_CONFIG.OVERSIZE_PARAM times bigger than the object bounding box itself.



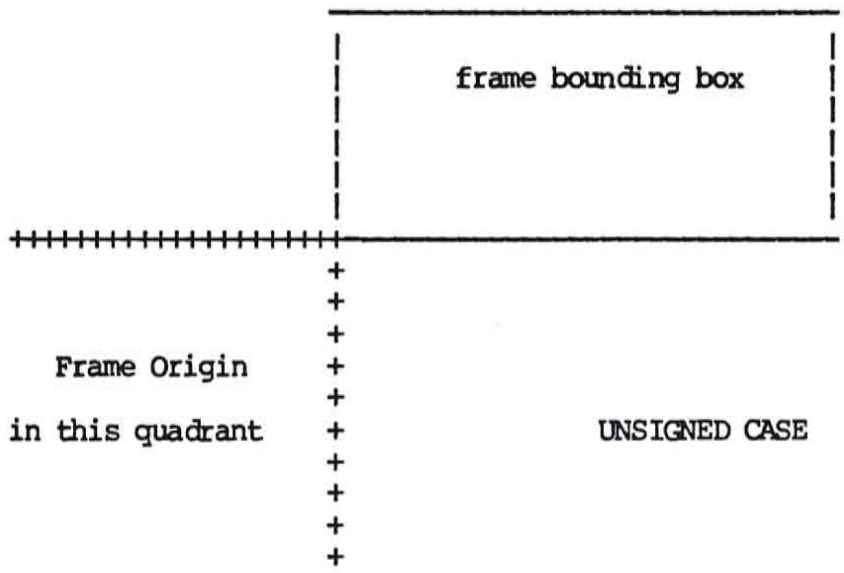
D.3 Algorithm for Signed Coordinate System

All frames that partially or completely intersect the oversized box are evaluated by BUM_EVALUATE_BUCKET.

If any frame is found with BUM_EVALUATE_BUCKET = SYS_SUCCESS, the object is added to that frame unless a frame is found with BUM_EVALUATE_BUCKET = BUM_STAT_GROW_BUCKET, then the object is added to that frame and FRAME.BOUNDING_BOX is updated to NEW_BUCKET_BOX. Otherwise a new frame is created and the object is added to this new frame.

D.4 Algorithm for Unsigned Coordinate System

The above algorithm for a signed coordinate system does not work as is when coordinates are expressed in an unsigned number system. This is because objects in a frame are limited to being in the positive quadrant with respect to the frame origin. In a signed coordinate system, objects may be in any quadrant.



This means that when BUM_EVALUATE_BUCKET returns BUM_STAT_GROW_BUCKET for a particular frame, it may not be possible to add the object to that frame, because the object may extend either to the left or below the origin of that frame. In the signed case, this is not a problem, a frame may grow isotropically (in any direction). In the unsigned case, frames have a preferential direction of growth, namely up and to the right.

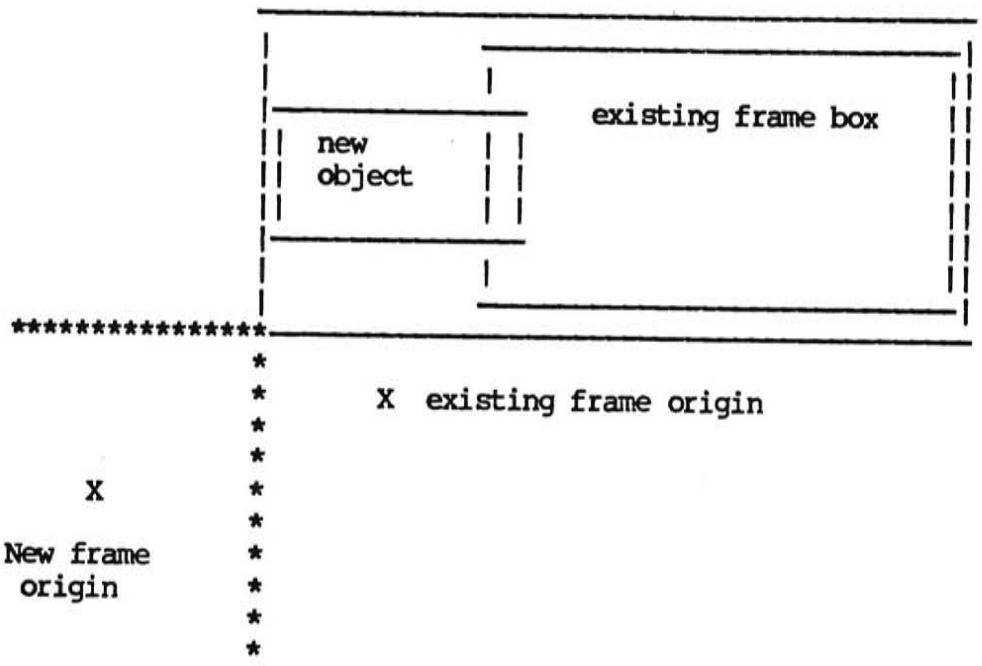
At this point, a decision has to be made among three options:

1. Keep looking for another frame with BUM_STAT_GROW_BUCKET, but with an origin that permits the object to be added.

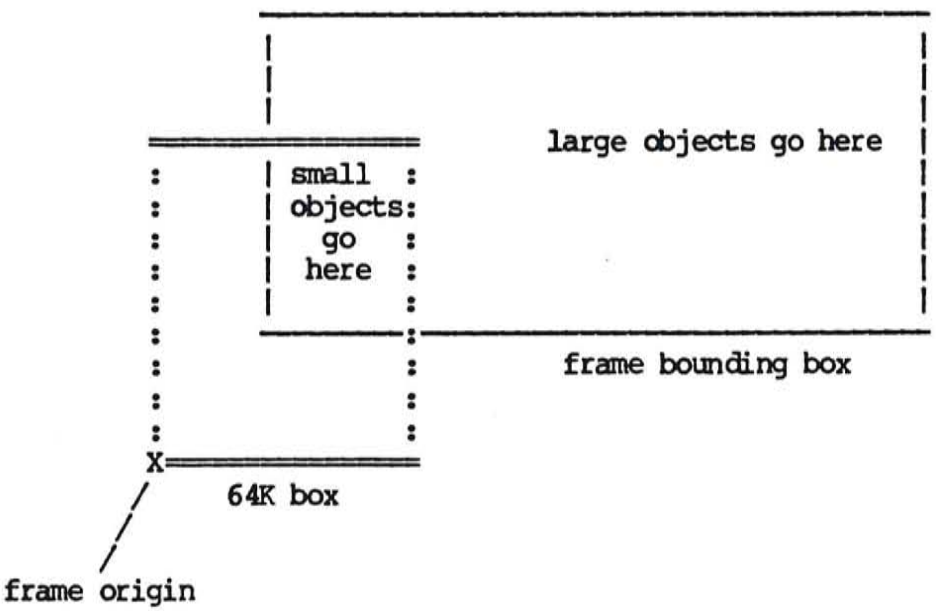
2. Re-origin all the objects in the frame so that the new object may be added.

3. Create a new frame.

In cases (2) and (3) above, the decision then has to be made as to where to place the new frame origin. If the goal is to maximise the number of small objects now in the frame, the best choice is the bottom left of the bounding box. If the goal is to maximise the number of small objects likely to be in the frame in the future, then the origin should be placed somewhere to the left and below the lower left corner of the bounding box.



The operation of giving a geometric frame a new origin involves scanning every point coordinate in the frame, adding in the old origin and subtracting out the new origin, using LARGE arithmetic. If all the coordinates in an object are less than 64K, then the object is stored as a SMALL object.



The question of how to choose a frame origin in the UNSIGNED case is currently unresolved.

D.5 Frame Origins in a Signed Coordinate System

In a signed coordinate system, the frame origin can be chosen anywhere, and all objects can still be represented. The only issue to be considered is the number of small versus large objects. A simple algorithm, such as "choose the center of the bounding box" can be used, at least until a better one can be found.

D.6 Comparison of UNSIGNED versus SIGNED coordinates

The question may then be asked: "Why use unsigned coordinates?" The problem is that PL/M does not provide a signed 32-bit data type. In either case, multiplication and division will have to be handled specially. For example, to evaluate expressions such as

```
(x1 - x2 ) * (x3 - x4 )
```

either a procedure call will be made, or code to handle the cases of x1 < x2 or x3 < x4 will be written in line.

Addition and subtraction in either case is done using two's complement arithmetic, so that is not a problem.

The only remaining case is comparison. PL/M provides an unsigned DWORD comparison.

IF (X < Y) compares unsigned coordinates correctly.

IF (X - Y < 0) however, does not work (this is a signed arithmetic expression).

If we use signed coordinates, then comparison of negative numbers cannot use the native DWORD compare. The above comparison would have to be written:

```
IF ( HIGH( X - Y ) AND 8000H ) <> 0
```

D.6.1 Object Code at Optimize (3)

```
DECLARE ( X, Y ) DWORD;
```

```
IF (X < Y)                    IF (HIGH (X - Y) AND 8000H) <> 0

MOV     AX,X                  MOV     AX,X
MOV     DX,X+2H               MOV     DX,X+2H
CMP     DX,Y+2H               SUB     AX,Y
JNZ     $+6H                  SBB     DX,Y+2H
CMP     AX,Y                  TEST    DX,8000H
JAE     @2                    JZ      @1
```

These are approximately equivalent in size and speed.

D.6.2 Summary of Comparison

	UNSIGNED	SIGNED
Choosing frame origin	TBD	easy
Growing a frame	anisotropic	isotropic
Arithmetic comparisons	easy	hard
Native PL/M data type	yes	no