PDQ

User Manual

Quick Access to

VIRGA Database Files

Programmer: Rod Price

Author/Editor: Michael Morgan

Text Files: PDQ.TXT, PDQ.OUT

Design File Document No. 4.22 Rev Ø

office 213

TABLE OF CONTENTS

1 INTRODUCTION ..... 1 - 1
2 THE PURPOSE OF PDQ ..... 2 - 1
3 INTERFACE GUIDE ..... 3 - 1
3.1 Overview ..... 3 - 1
3.2 Schema Dependency ..... 3 - 3
3.3 The General Interactive Facts ..... 3 - 4
3.3.1 Global Pointers ..... 3 - 5
3.4 SEED Commands in PDQ ..... 3 - 6
3.4.1 ALLOCATE ..... 3 - 6
3.4.2 DEALLOCATE ..... 3 - 6
3.4.3 HELP ..... 3 - 7
3.4.4 EXIT ..... 3 - 7
3.4.5 CINPUT ..... 3 - 8
3.4.6 OUTPUT ..... 3 - 9
3.4.6.1 Varieties of Output Redirection ..... 3 - 10
3.4.6.2 Redirection through CMD ports ..... 3 - 11
3.4.6.3 Redirection of single commands ..... 3 - 11
3.4.7 COPY_MEMORY ..... 3 - 12
3.4.8 READ_MEMORY ..... 3 - 13
3.4.9 WRITE_MEMORY ..... 3 - 14
3.4.10 FILL_MEMORY ..... 3 - 16
3.4.11 EVALUATE ..... 3 - 17
3.4.12 Pointer Expressions ..... 3 - 18
3.4.13 FILE_STATUS ..... 3 - 19
3.4.14 FILE_NUM ..... 3 - 19
3.4.15 SET_FILE_DESC ..... 3 - 20
3.4.16 UNLOCK_FILE ..... 3 - 21
3.4.17 ROOT_STATUS ..... 3 - 22
3.4.18 MOD_INVENTORY ..... 3 - 23
3.5 Purely PDQ Commands ..... 3 - 24
3.5.1 BUILDER ..... 3 - 24
3.5.2 PFRAME ..... 3 - 26
3.5.3 DTYPE ..... 3 - 27
3.5.4 DALL ..... 3 - 28
3.5.5 FRAME_TYPES ..... 3 - 29
3.5.6 FLIP ..... 3 - 30
3.6 VIRGA Procedures as PDQ Commands ..... 3 - 31
3.6.1 CREATE ..... 3 - 31
3.6.2 OPEN ..... 3 - 32
3.6.3 CLOSE ..... 3 - 32
3.6.4 FLUSH ..... 3 - 33
3.6.5 FFIND ..... 3 - 34
3.6.6 FADD ..... 3 - 34
3.6.7 FUPDATE ..... 3 - 35
3.6.8 FGROW ..... 3 - 35
3.6.9 FDELETE ..... 3 - 36
3.6.10 FTDELETE_BY_TYPE ..... 3 - 36
3.6.11 FIRST_FRAME ..... 3 - 37
3.6.12 NEXT_FRAME ..... 3 - 37
3.6.13 MGET_MODULES ..... 3 - 38
3.6.14 FALLOC ..... 3 - 39

3.6.15 MALLOC ..... 3 - 39
3.6.16 MDISPOSE ..... 3 - 40
3.6.17 MTDISPOSE ..... 3 - 40
3.6.18 DIRTY_MODULE ..... 3 - 41
3.6.19 CLEAR_DIRTY_MOD ..... 3 - 41
3.6.20 LFRAME_LOCK ..... 3 - 42
3.6.21 UFRAME_UNLOCK ..... 3 - 42
3.6.22 LMODULE_LOCK ..... 3 - 43
3.6.23 UMODULE_UNLOCK ..... 3 - 43
3.6.24 LTMOD_TYPE_LOCK ..... 3 - 44
3.6.25 UTMOD_TYPE_UNLCK ..... 3 - 44
3.6.26 WINDOW ..... 3 - 45
3.6.27 NCREATE_INDEX ..... 3 - 46
3.6.28 NDELETE_INDEX ..... 3 - 46
3.6.29 NLIST_INDEX ..... 3 - 47
3.6.30 NINDEX_FIND ..... 3 - 48
3.6.31 NFIRST ..... 3 - 49
3.6.32 NNEXT ..... 3 - 49
3.6.33 NSFIRST ..... 3 - 50
3.6.34 NSNEXT ..... 3 - 50
4 THE BASIS OF PDQ IN SEED AND VIRGA ..... 4 - 1
5 COMMAND TEXTS ..... 5 - 1
INDEX ..... I - 1

1 INTRODUCTION

PDQ and its PE-1Ø companion program, PEQ, allow a user to interactively access a binary database (if it is a VIRGA file) and read a text version of its contents. This may be done either to check on the actions of subprograms which have calls to VIRGA routines, or to modify data in the database.

A PDQ user can also construct a small database for testing purposes. In the GATEMASTER system, if a programmer wants to use a small file that has several kinds of frame types, or if a routine needs to be tested on an improbable condition (e.g., a field = zero), PDQ may be quicker and easier to use for creating a test database than the GAIL language and BASE processor.

The commands available in PDQ are of three types: SEED routines which access system memory, special PDQ routines for database I/O, and VIRGA routines for database manipulation.

Effective use of PDQ for database access requires a good knowledge of the structure of a VIRGA database. Therefore, the user probably should read the VIRGA User Manual first, and learn about modules and frames before going on. However, using PDQ is also a good way to learn the structure of the database, since the user can actually see the data frames, headers, and fields.

2 THE PURPOSE OF PDQ

Since PDQ is a very powerful tool, unhindered by checks on the validity or safety of a user's input, it is only for use by Daisy personnel. Also, its structure and interfaces with VIRGA, SEED, and database schemas are confidential Daisy knowledge.

PDQ allows the user to put raw data into the database quickly or dump out data frames in order to check the data. In particular, it allows GM-1Ø and PE-1Ø people to rapidly query the database.

PDQ's original use was in creating GM-1Ø databases. GM-1Ø personnel would write a text file full of gate array data, and use the Builder command in PDQ to set up the database.

Since the 1983 advent of the GAIL language and processor, GATEMASTER databases (consisting of macros and gate arrays) are written directly in the GAIL language, and compiled into binary data files. Hence, the database creation and editing aspects of PDQ have greatly declined. However, PDQ is still useful as a programmer's tool. A programmer using VIRGA routines can only check a program or routine's operation if there exists a quick, flexible way of querying the database. PDQ provides the necessary window into the binary database.

3 INTERFACE GUIDE

This section covers the various PDQ user commands.

The PDQ program in itself is invoked by typing the command:

```
CMD> PDQ
```

This loads the PDQ program and initializes a number of global structures.

3.1 Overview

PDQ is a powerful, simple, direct tool. It does not perform checks on input data or on memory boundaries. Error messages are simple and few.

A user on the 8086-based Daisy workstation who uses the WRITE_MEMORY command can easily crash or corrupt the operating system. On the 286-based system, O.S. corruption is also possible, but much less likely. If the O.S. becomes corrupted, serious disk file corruption could occur, so the user should immediately re-boot. This can be a nuisance, but is not serious so long as the novice is experimenting on test files. It is recommended that you make a copy of a test VIRGA file, and experiment with PDQ on the extra file copy so that an original file with good data is always available.

Most PDQ commands are entirely safe. In general, the PDQ and VIRGA commands are safer than the SEED commands which write to

memory.

Once you are in PDQ, before using any of the VIRGA commands, you should open a VIRGA file. For instance, MCA600.BASE is a GATEMASTER database file. If you want to get a frame's data, you must first find that frame (i.e., know the pointer to the beginning of the frame of interest). Most of PDQ (and VIRGA) is pointer-oriented.

PDQ can handle all frame types that are declared in the schema. Its output is automatically formatted, and a heading is added for each data field.

NOTE to new GATEMASTER personnel: a dump of the database through a PDQ command is not in GAIL language. The BASE (gate array) processor reads only GAIL language, not a PDQ dump. Conversely, PDQ can only create a Virga file according to a pre-existing schema (PLM code), and cannot read GAIL.

3.2 Schema Dependency

When PDQ is invoked at the command (CMD) level, it will display its name and version number in the status window of the screen. It will also display the version number of the database schema that PDQ was compiled with. For instance, it might display "V 4.Ø.Ø7" to indicate the GM-1Ø schema that it was compiled with. Please note that the schema version number that is displayed was manually entered by another user: whoever it was that compiled and built PDQ.

The database schema is PLM code in an ELT file that should have a version number in its source listing. Whatever schema was used in compiling PDQ determines PDQ's image of the database. A user who must update data should check the schema number against the schema version number displayed by PDQ and see if the database schema is in fact structured the way that PDQ expects. Otherwise, PDQ might write bytes into memory at a totally wrong location.

3.3 The General Interactive Facts

A user may shorten commands to their shortest unambiguous form. For instance, "HELP" can be abbreviated to 'H'. 'FILE_STATUS' can be abbreviated to 'FILE', since it must be distinguishable from 'FILL_MEMORY'.

SHIFT_CANCEL aborts a command. This only works when PDQ is expecting input. The program will expect input to be from the keyboard unless the user has changed the input port.

In any command that requires one or more parameters, PDQ will print the name of each parameter, and thereby prompt the user for the needed arguments. Displayed values in brackets, e.g., [1024], indicate default values that the user can input by striking the EXECUTE key.

NOTE: all parameter requests and default values reflect a specific PDQ schema: for example, the GM-10 schema.

The description of each command in this Interface Guide includes a list of the parameters that PDQ will request.

If the user wants to avoid the extensive interaction of waiting for each parameter prompt, all the parameters may be entered in a row (separated by blank spaces) following the command name. A user writing a PDQ command file would probably want to use this more compact format.

3.3.1 Global Pointers

PDQ holds onto the most recently used pointer of each particular pointer type: module, frame, global, file_descriptor, or root. This is essential to creating command files, and very useful to know when responding interactively to parameter prompts. To access the last pointer, the user must specify @ (atsign), and then a number, to indicate the pointer-type:

```
file_desc  --  7
frame      --  5
global     --  6
module     --  4
root       --  8
```

In some cases, it is not necessary to specify the pointer type; a simple @ will retrieve the last pointer. The @ character always gets the default value given in brackets. A carriage return may do the same thing, but @ is needed in %SUBMIT files, and when referencing a value on the interactive command stack.

The last data frame ID that was accessed can also be retrieved. When faced with a prompt that requires the frame ID, respond with "@W3Ø", and the last ID will be retrieved, and used as an input.

3.4 SEED Commands in PDQ

PDQ contains commands which belong to SEED, and which are called automatically. For instance, the PDQ command ALLOCATE calls SEED_ALLOCATE.

3.4.1 ALLOCATE

Allocate a memory segment of user-specified size in main system memory. PDQ picks its own address for the segment. It will also udpate a global segment pointer.

The maximum allowed allocation is approximately 655ØØ bytes.

Parameters requested:
1. Number of bytes to allocate [1Ø24]

SEED routine: SEED_ALLOCATE

3.4.2 DEALLOCATE

Deallocate a memory segment. The command knows about the last segment allocated (only once -- the global pointer does not get updated after de-allocation).

Parameters requested:
1. Pointer to memory segment [ptr to last segment allocated]

SEED routine: SEED_DEALLOCATE

3.4.3 HELP

Tell the user what references can be made to recent pointers and ID's, and then list all the commands that PDQ recognizes.

No parameters.

SEED routine: SEED_HELP

3.4.4 EXIT

Exit PDQ and return to the command level.

Changes to PDQ buffers will only be flushed to the disk files if the modules' dirty flags are set. (All Virga routines in PDQ which modify data will set appropriate dirty flags automatically.) Open files will be closed.

No parameters.

SEED Command Processing: The EXIT command causes an immediate exit from PDQ and SEED processing.

3.4.5 CINPUT

CINPUT means Command Input: specifies the input source for PDQ commands.

Set the input port (one of the Daisy standard I/O ports) to a particular device or file. For instance, "CI INFILE" establishes a file called 'INFILE' as the input port. That file will be read automatically, and its commands and text strings will be processed as though they were interactive input. Of course, the default input port is the KEYBOARD.

For information on using the Cinput and Builder commands to read command files, please see the last chapter.

Parameter requested: Pathname to the input port.
(for instance, -DIR1/DIR2 INFILE)

SEED routine: SEED_CINPUT

3.4.6 OUTPUT

Much like the CINPUT command, the OUTPUT command redirects PDQ's output to a different port. Recall that "port" means either a file or a device. The default output port is :SCREEN.

Output redirection is critical to database editing. That is, a user can obtain a text version of the database only by redirecting the output of PFRAME, DTYPE, or DALL to a disk file. After TEC editing of the text file, the user can put the modified data back into the database by using the CINPUT command (and BUILDER).

WARNING: if any of the three dump routines (PFRAME, DTYPE, DALL) reads a corrupted frame, the routine might miss an EOF marker and create bad output continuously.

Upon use of the OUTPUT command, a message will be sent to the previous output port specifying the name of the new output port. For instance, on screen the user may see:

```
REDIRECTING OUTPUT TO PORT:  -FOO/BAR.BAR  PDQ.OUT
```

Output redirection remains in force until another redirection.

Output may be returned to the user's screen by re-specifying the screen as output port:

```
PDQ> OUTPUT :SCREEN
```

3.4.6.1 Varieties of Output Redirection

Redirection may overwrite (or create) a file or append to a file.

To redirect output to a file, use the word OUTPUT followed by a ">" symbol, with the filename (pathname) immediately following. To append data to a file, you must type (on one line) "OUTPUT >>" before the filename.

Examples:

```
PDQ> OUTPUT >:LP                    ~redirect to lineprinter
PDQ> OUTPUT >FOO                    ~redirect to file FOO
PDQ> OUTPUT FOO                     ~same
PDQ> OUTPUT >$BILL FOO              ~redirect over Ethernet
PDQ> OUTPUT >>FOO.ALL               ~append to FOO.ALL
PDQ> OUTPUT >>/USER/DOC FOOBAR      ~append to FOOBAR
PDQ> OUTPUT                         ~Since filename not entered on
  ~(On next line, type name for       same line, cannot use ">>".
   ~new output port: -FOO BAR.OUT)
```

Recall that the Daisy lineprinter interface will spool output. Redirection of output to LP will actually write output to a spool file and nothing will appear on the LP until that spoolfile is closed (which you may force by reinvoking the output command, even with LP argument).

SEED routine: SEED_OUTPUT

3.4.6.2 Redirection through CMD ports

Of course, the Daisy standard for redirection of files may also be used; i.e., you might call PDQ via:

```
CMD> PDQ CI PDQ.CSD TO PDQ.OUT
```

3.4.6.3 Redirection of single commands

There is a also way to temporarily redirect output for the duration of any particular command, without changing the output for later commands. Following any PDQ command name, and before the first (if any) parameter to that command, and on the same input line, you may enter ">" or ">>" followed by a port name.

For example:

```
PDQ> READ_MEMORY >:LP C5Ø4:1Ø 1ØØH
```

will read 1ØØH bytes of memory and write them (in ascii format) to the lineprinter. Printing at the LP will start immediately, handled automatically by the spooler.

Only the output of one command (e.g., READ_MEMORY) will be redirected (to the line printer in this case). The output of subsequent commands will go to the standard output as previously defined.

3.4.7 COPY_MEMORY

Copy bytes from one memory location to another one.

The command will copy the specified number of bytes from a specified source to a specified destination. This is useful for creating special conditions to test programs. The command should not be used to update database frames; to do that, use BUILDER or the Virga commands in PDQ.

The command sets the global pointers @PØ, @P1 to the source and destination addresses.

Parameters:
1. source_pointer
2. destination_pointer
3. length_in_bytes

SEED routine: SEED_COPY (uses MOVB)

3.4.8 READ_MEMORY

Read the specified memory segment and copy it to the standard output port (default is :SCREEN).

This command reads the specified number of bytes starting at the specified address in memory and displays them on the standard output using the specified numerical base and type of output format. The corresponding ASCII values are always displayed.

This command sets default values for pointers in the global PDQ byte array.

Parameters:
1. base (HEX,TEN) [HEX]
2. type (BYTE, WORD, POINTER) [BYTE]
3. source_pointer
4. length_in_bytes

SEED routine: SEED_READ_MEMORY

3.4.9 WRITE_MEMORY

Take memory values from command input and write them into a memory location. This is useful for creating special conditions to test programs.

WARNING: No protection or error checking is provided. You can overwrite anything in system memory, whether code or data. This includes possibly corrupting the operating system.

Syntax:

```
    Write_Memory <source_pointer> {[type] input_value} |

       where:

          {}  means repeat zero or more times.
          []  means that 'type' is optional.
```

```
Parameters:
  1. source_pointer
  2. type (BYTE, WORD, POINTER) [BYTE]
  3. input_value
     (NOTE: input_value must be appropriate for the
      specified type or string:
        a. pointer input is always hex;
        b. byte and word default to decimal;
        c. append trailing H for hex;
        d. a quote mark starts a string.)
  4. |  This character must be used to end the command input.
```

The input Type may be specified anywhere in the input stream, and that specification remains in effect until a new type is specified. Initially, type is BYTE.

Strings are indicated by a leading ' character; trailing ' is ignored, if present.

NOTE: SEED always parses tokens delimited by white space, so blanks may NOT be included in the string.

An example of the WRITE_MEMORY command is provided below:

```
PDQ> WR @P2 48 W 32 'ABC' 1ØH P 2Ø |
```

The command shown will write the 12 bytes, starting at the address of memory last allocated (see globals, above, for @P2 notation). PDQ interprets the input values as follows:

(byte 48) (word 32) (string ABC) (byte 1ØH) (pointer 2ØH)

The following 12 bytes are then written:

```
3Ø 2Ø ØØ 41 42 43 1Ø ØØ 2Ø ØØ ØØ ØØ
```

SEED routine: SEED_WRITE_MEMORY

NOTE: if the user has used this command on database frames in memory and wants the changes to be kept, the user must set the module's dirty flag using DIRTY_MODULE. The module will be flushed to disk upon CLOSE or EXIT.

However, that method is strongly discouraged. To modify the database, a user can more safely use FDELETE and FADD, or FUPDATE, or the %UDPATE_FRAME statement in BUILDER.

3.4.1Ø FILL_MEMORY

Read a string of bytes and copy it repeatedly to specified memory locations. The command will write the specifed data into the specified address for the specified number of bytes. Data will be repeated as often as necessary to complete the required length. Input data values are in the same syntax as WRITE_MEMORY (see above).

Syntax:

```
FILL_MEMORY <address> <length_in_bytes> {[type] value} |
```

For example,

```
PDQ> FILL_M 3CCD:Ø 13 'TESTING' |
```

will write 13 bytes into memory. The results are shown below, in hex first, then in ascii:

```
54 45 53 54 49 4E 47 54 45 53 54 49 4E
T  E  S  T  I  N  G  T  E  S  T  I  N
```

Parameters:
1. source_pointer
2. length_in_bytes
3. type (BYTE, WORD, POINTER) [BYTE]
4. input_value
   (NOTE: input_value must be appropriate for the specified type or string:
   a. pointer input is always hex;
   b. byte and word default to decimal;
   c. append trailing H for hex;
   d. a quote mark starts a string.)
5. | This character must be used to end the command input.

SEED routine: SEED_FILL_MEMORY

3.4.11 EVALUATE

This command simply echoes whatever input value you give it. The echo will specify whether PDQ/SEED thinks the token is the default character (@), the end character (|), a pointer, a word (which is also a valid pointer), or a string. Then, the value of the token will be printed as a pointer (hex), a word (both hex and decimal) and ascii (two bytes worth).

This may be used to examine the values of the globals or to do pointer-arithmetic.

Parameter: the input to be evaluated

SEED routine: SEED_EVALUATE

3.4.12 Pointer Expressions

Anywhere that PDQ expects a pointer, a pointer expression with addition OR subtraction is legal (but not both). That is, [<ptr>+<word_expression>] OR [<ptr>-<word_expression>]. The <word_expression> should be an actual word (or global word or byte). <ptr> may be an actual pointer or global pointer.

For example, the expression 1:54H+11H is interpreted as 1:5FH.

Furthermore, if @P5 means the last frame pointer, and that frame contains the offset of the trace field in the seventeenth byte, then to change the first byte of that trace field to ØAH, do:

```
PDQ> READ @P5 18
PDQ> WRITE @P5+@W16 AH I
```

Global PDQ information can be referenced by using the following expressions:

```
@PØ    = last copy_from pointer
@P1    = last copy_to pointer
@P2    = last allocated memory pointer

@W3Ø   = id from last find_by_id routine

@W, @B   are set by READ_MEMORY.
```

3.4.13 FILE_STATUS

List all files open in PDQ.

An example of the command's output is shown below:

```
Current File_Num = Ø
>>Ø:  MCA6ØØ.BASE.  Root_ptr = 6979:ØH.  Mode = NQA.  2Ø DEC 83
  1:  MCA12ØØ.BASE.  Root_ptr = 6B8F:ØH.  Mode = NQA.  23 SEP 83
  2:  MCA6ØØ.MLIB.  Root_ptr = 6D93:ØH.  Mode = NQA.  2Ø DEC 83
  3:  MCA12ØØ.MLIB.  Root_ptr = 6F97:ØH.  Mode = NQA.  2Ø DEC 83
```

The double arrow (>>) indicates the current file.

No parameters.

SEED routine: SEED_FILE_STATUS

3.4.14 FILE_NUM

Change the current PDQ file number to another number. This allows the user to set (establish) the "current" PDQ file.

The current file is the file upon which all PDQ commands operate. Most commands will return an error status if invoked when there is no current file.

Parameter: a file number

SEED routine: SEED_FILE_NUM

3.4.15 SET_FILE_DESC

Creates a VIRGA file-descriptor, so that a VIRGA file can be created. The descriptor is used as a parameter in the CREATE routine in PDQ.

PDQ selects its own location for the file-descriptor, creates the descriptor, and reports the pointer to it.

It is recommended that the user write the necessary values into a file via TEC, and then read the file into PDQ with the CI (CINPUT) command.

```
Parameters:
  1. module_size [8192]
  2. module_inventory_size (>2) [100]
  3. module_inv_ext_size (>2) [100]
  4. number of frame_types [5]
  5. [for EACH subsequent frame-type...(see parameter 4)]
     a. module_type (>=2)
     b. inv_size [10]
     c. inv_ext_size [5]
```

SEED routine: SEED_SET_FILE_DESC

3.4.16 UNLOCK_FILE

Unlocks a locked VIRGA file. Since VIRGA locks a file only for writing, this means that the data in the file being unlocked is almost certainly corrupt. Probably, the system crashed and some dirty buffers were not flushed to disk.

WARNING: Use this command at your own risk. Unlocking a file probably will add to the bugs already in it.

Parameter: pathname to a file

SEED routine: SEED_UNLOCK_FILE

SEED calls VIRGA routines: GDB_LOCK
GDB_CLOSE

3.4.17 ROOT_STATUS

Dumps VIRGA's internal data structure concerning the current file. Application programs should never need to look at this information from the root. However, the information might occasionally be useful in debugging.

No parameters.

An example of the command's output is shown below:

```
Root status:
 lock=FALSE, type=255, mode=NQA, temp_flag=FALSE, version=1,
    rev=Ø, time=2Ø DEC 83 17:44, root_size=1Ø24,
 module_size=8192,  mod_inv_size=5Ø, mod_inv_ext=1Ø,
    num_frame_types=34,
 mod_mod_id=Ø, mod_marker=1Ø24, mod_mod_size=7168,
    mod_mod_offset=32, mod_inv_ptr=6FD9:2ØH,
 frames_mod_id=Ø, frames_mod_offset=848, frames_ptr=6FD9:35ØH,
 num_modules=21, eof_marker=18Ø224, global_ptr=4936:ØH,
    file_num=3,
 aft=1:ØAH, marker=8192, eof_flag=Ø, path_size=25,
    path_offset=17Ø.
```

SEED routine: SEED_ROOT_STATUS

3.4.18 MOD_INVENTORY

Gives internal VIRGA information on all modules which exist in the file. Application programs should never need to look at this information. However, the information might occasionally be useful in debugging.

No parameters.

An example of the command's output is shown below:

Lists of modules: inv_ptr = 6FD9:2ØH

MOD-ID	TYPE	MARKER	SIZE	FREE-BYTES	BUF-NUM
Ø	1	1Ø24	7168	1Ø71	3
1	4	8192	8192	1Ø1	65535
2	6	16384	8192	7434	65535
3	4	24576	8192	116	65535
4	4	32768	8192	43	65535
5	4	4Ø96Ø	8192	Ø	65535
6	4	49152	8192	27	65535
7	4	57344	8192	131	65535
8	4	65536	8192	65	65535
9	4	73728	8192	117	65535
1Ø	4	8192Ø	8192	45	65535
11	4	9Ø112	8192	1Ø5	65535
12	4	983Ø4	8192	167	65535
13	4	1Ø6496	8192	199	65535
14	4	114688	8192	69	65535
15	4	122888Ø	8192	151	65535
16	4	131Ø72	8192	145	65535
17	4	139264	8192	165	65535
18	4	147456	8192	144	65535
19	4	155648	8192	7587	65535
2Ø	4	1638 4Ø	16384	7933	65535

(ID) (from schema) (byte offset in file) (module size in bytes) (module free space) (current buffer or, module not in system memory. 65535 = FFFFH)

SEED routine: SEED_MOD_INVENTORY

3.5 Purely PDQ Commands

The following commands were written specifically to fulfill the needs of PDQ users.

3.5.1 BUILDER

If you are in the BUILDER command, the text you type will define frames. BUILDER takes text, and builds a binary database file or updates an existing one.

If the input port is the default (KEYBOARD), BUILDER sends a colon (:) prompt to the user. Control-E ends input mode and returns the user to the PDQ command level.

The great advantage of BUILDER is that it works with the PDQ output routines to allow a person to dump out a database, edit its contents, and then push data back into the VIRGA file. Thus, BUILDER is used to re-process (reenter into the database) the output file which is dumped by PFRAME, DTYPE, or DALL.

To process the text file, the user invokes the CINPUT command while in PDQ (or in a SUBMIT file) so that the desired file is immediately read. Usually, the first word in the file is "BUILDER", so that the BUILDER command is invoked by the input file. The dump routines automatically add the word "BUILDER" at the beginning of their output, so that the output file will easily serve as an input file.

If the file contains a %UPDATE_FRAME statement (as any PFRAME dump does), BUILDER will update the frame in PDQ memory and set the dirty flag so the new data will eventually go to disk. The frame may also contain %ADD_FRAME and %SYNTAX_FRAME statements. A %ADD_FRAME statement will call a VIRGA routine to add the input frame to the database. A %SYNTAX_FRAME statement instructs SEED/VIRGA to check only the syntax of the input.

Ordinarily, you should see your group's system integrator to obtain test-worthy VIRGA files, rather than using BUILDER to build them from scratch. Virga files are now built with %SUBMIT command files which run PDQ. For instance, in GM-10 group, VIRGIN.CSD and MLIB.CSD are used to create the desired database files.

However, the files that these submit files create will essentially be empty; that is, they will contain only information internal to VIRGA, not user application data.

BUILDER, on encountering an error, will (1) call sys_check_exception right then, and (2) skip all input until a %END is found, so that PDQ will not read and interpret all the input.

For information on command files, please see the last chapter.

PDQ routine: PDQ_BUILDER

3.5.2 PFRAME

Print frame. PFRAME is used to dump out just one frame. The dump gives ASCII string, numeric, and hex values, with a heading appropriate to each frame-type.

PFRAME, if its output is redirected to a file, is useful for creating a readable text version of a frame. The text data can be edited and returned to the database through PDQ by using the BUILDER command. In fact, the word "BUILDER" is the first part of this command's output, so that the edited file, when used as input, will invoke the BUILDER command.

Bit-encoded data will be output in hex.

Parameters:
1. frame-ptr
2. module_ptr

PDQ routine: PDQ_DUMP_FRAME

3.5.3 DTYPE

DTYPE is used to dump all frames of a given type.

WARNING: this command does not stop until all the frames of the specified type are output. Set OUTPUT to LP or a new filename before invoking this command.

DTYPE, if its output is redirected to a file, is useful for creating a readable text version of all the data frames in a frame type. The text data output can be edited and returned to the database through PDQ by using the BUILDER command. In fact, the word "BUILDER" is the first part of this command's output, so that the edited file, when used as input, will invoke the BUILDER command.

Bit-encoded data will be output in hex.

Parameters:
1. frame_type
2. Do you also want a dump in binary (Y/N) [Y]

PDQ routine: PDQ_DTYPE

3.5.4 DALL

DALL is used to dump the whole data base. The command keeps the user informed about which frame type it is working on.

WARNING: this command dumps all frames in the database without stopping. Set OUTPUT to LP or a new filename before invoking this command.

DALL, if its output is redirected to a file, is useful for creating a readable text version of the entire database. The text data output can be edited and returned to the database through PDQ by using the BUILDER command. In fact, the word "BUILDER" is the first part of this command's output, so that the edited file, when used as input, will invoke the BUILDER command.

Bit-encoded data will be output in hex.

Parameters:
1. Do you also want the dump in binary (Y/N) [Y]

PDQ routine: PDQ_ALL

3.5.5 FRAME_TYPES

The command FRAME_TYPES displays the correlation between all frame type numbers and their names.

No parameters.

An example of the command's output is shown below:

```
FRAME TYPE NUMBERS AND NAMES

    4 NET_FRAME
    6 MACRO_FRAME
   32 MCLASS_FRAME
    5 COMP_FRAME
   11 CELL_FRAME
   12 CELL_TYPE_FRAME
   16 OBSTRUCTION_FRAME
   17 UNDERPASS_FRAME
    7 ROUGH_MISC_FRAME
    8 ROUGH_PROJECTION_FRAME
    9 ROUGH_GRID_CELL_FRAME
   15 BITMAP_FRAME
   1Ø PATHNAME_FRAME
   19 BLOCK_FRAME
   21 NOTE_FRAME
   23 TEXT_FRAME
   14 PROJECTION_FRAME
   18 DELTA_FRAME
   31 PARAM_FRAME
   22 PARAM_TABLE_FRAME
   28 PLACE
   13 GLOBAL_DESC_FRAME
```

PDQ routine: PDQ_FRAME_TYPES

3.5.6 FLIP

Flip a macro: change its orientation by flipping it in the X dimension or flipping it in the Y dimension or both. Save the new macro under a new name.

The macro to be flipped must have a rectangular outline. Its outline will be the same in the flipped macro; only its associated structures (traces, vias, pins, obstructions, recognition window) are flipped.

Parameters:
1. flip-type (MY, MX, MXY)
2. frame_ptr to the macro

The procedure will try to increment the cell-type part of the macro name (the suffix, which is in hex) and use the resulting string as the name of the new macro. If the new name already exists in the database, an error will be returned and the flipped version will not be created.

For instance, the flipped version of macro H28.9 will be named:

```
H28.A   if MY flip    (increment by one)
H28.B   if MX flip    (increment by two)
H28.C   if MXY flip   (increment by three)
```

NOTE: this command only creates valid results within the context of a particular gate array and its conventions concerning cell-types and their names.

PDQ routine: MFL_FLIP

3.6 VIRGA Procedures as PDQ Commands

All VIRGA routines are called through SEED intermediaries. For instance, the CREATE command in PDQ calls SEED_CREATE, and SEED_CREATE calls the VIRGA routine GDB_CREATE.

3.6.1 CREATE

Creates a VIRGA file, given a file pathname. Reports the root pointer for the new file.

Parameter: a pathname

VIRGA routine: GDB_CREATE

3.6.2 OPEN

Open a pre-existing VIRGA file and report a root pointer to the opened file.

This routine may be used several times in order to open multiple files.

NOTE: the user must call OPEN in order to access a database file.

Parameters:
1. filename
2. file access mode (R,W,NQA) [R]
(R means Read, W means write, NQA means No-Questions-Asked)

VIRGA routine: GDB_OPEN

3.6.3 CLOSE

Close the current file.

Modules with dirty flag set will be flushed to disk. (Virga routines which modify data automatically set the dirty flag.)

Parameters:
Asks, "Purge file? (Y/N) [N]". This prompt allows a user to purge a VIRGA file upon closing PDQ access to it.

WARNING: Answering "Y" totally erases the file.

VIRGA routine: GDB_CLOSE

3.6.4 FLUSH

Flush to disk any modules which have their dirty flag set. (Virga routines which modify data will set that flag automatically). This procedure also time-stamps the file and flushes VIRGA-internal root and control modules.

The modules stay in memory; this command only synchronizes the disk files with the contents of the modules in memory.

No parameters.

VIRGA routine: GDB_FLUSH

3.6.5 FFIND

Frame-Find. Find a frame by frame TYPE and ID and return pointers to it.

Parameters:
1. TYPE (the type of frame to search for)
2. ID (the ID of the desired frame)

VIRGA routine: GDB_FIND_FRAME

3.6.6 FADD

Frame-Add. Add a frame by copying from a frame and set the new frame's dirty flag.

VIRGA automatically gives PDQ the following feature: if you add a frame whose ID is not ØFFFFH, VIRGA will honor that request, i.e., it will use that ID (if possible). If the specified frame has an ID = ØFFFFH, then VIRGA will make up its own ID.

Parameter: pointer to the frame to be added.

VIRGA routine: GDB_ADD_FRAME

3.6.7 FUPDATE

Frame Update. Update an existing (target) frame by copying new data from a (source) frame and setting the target frame's dirty flag.

Parameters:
1. module_ptr to the target frame
2. frame_ptr to the target frame
3. frame_ptr to the source frame

VIRGA routine: GDB_UPDATE_FRAME

3.6.8 FGROW

Frame Grow. Increase or decrease the size of a specified frame. (Sets the dirty flag on the modified frame.)

WARNING: Decreasing the size of the frame means that data will be lost.

Parameters:
1. module_ptr
2. frame_ptr
3. size for new frame

VIRGA routine: GDB_GROW_FRAME

3.6.9 FDELETE

Frame Delete. Delete the indicated frame and set the dirty flag on the module which held the frame.

Parameters:
1. module_ptr
2. frame_ptr

VIRGA routine: GDB_DELETE_FRAME

3.6.10 FTDELETE_BY_TYPE

Delete all frames of the specified type. (Set dirty flags on modules modified.)

Always returns success, if valid frame-type number, even if no actual frames of that type.

Parameter: frame_type [4]

VIRGA routine: GDB_DELETE_BY_TYPE

3.6.11 FIRST_FRAME

Report the first frame found while searching the database for a frame of the specified frame-type. Note that the first frame found is not necessarily the frame with the lowest ID in that frame-type.

Parameters:
1. frame ID
2. module_ptr
3. frame_ptr

VIRGA routine: GDB_FIND_FIRST

3.6.12 NEXT_FRAME

Given a reference frame, report the next frame (of the same frame-type) found in the database. Remember that the next frame's ID does not follow from the last one found (frames are not in sorted order).

Parameters:
1. module_ptr
2. frame_ptr

VIRGA routine: GDB_FIND_NEXT

3.6.13 MGET_MODULES

Get and lock all modules of a specified type.

The command reports that frames are locked whether such frames exist or not.

Parameter: frame_type [4]

VIRGA routine: GDB_GET_MODULES_BY_TYPE

3.6.14 FALLOC

Allocate an empty frame in a particular module.

Parameters:
1. module_ptr
2. frame_type [4]
3. frame_size

VIRGA routine: GDB_ALLOCATE_FRAME

3.6.15 MALLOC

Allocate an empty module in memory.

Parameters:
1. module_type [2]
2. module_size (in bytes) [8192]

VIRGA routine: GDB_ALLOCATE_MODULE

3.6.16 MDISPOSE

Dispose of a specified module (remove it from memory).

Parameter: module pointer

VIRGA routine: GDB_DISPOSE_MODULE

3.6.17 MTDISPOSE

Dispose of all modules in memory that are of the specified type.

The command does not give an error if no modules of that type exist.

Parameter: module_type to dispose [2]

VIRGA routine: GDB_DISPOSE_MODULES_BY_TYPE

3.6.18 DIRTY_MODULE

Set the dirty flag on a module in memory, so that it will (eventually) be written to disk.

Parameter: module pointer

VIRGA routine: GDB_SET_DIRTY_MODULE

3.6.19 CLEAR_DIRTY_MOD

Clear the dirty flag on a module in memory (because user mistakenly set a module's dirty flag).

WARNING: this procedure should not be necessary since VIRGA routines will clear a dirty flag after a module is flushed.

Parameter: module pointer

VIRGA routine: GDB_CLEAR_DIRTY_MODULE

3.6.20 LFRAME_LOCK

Lock a specified frame in system memory so that it will not be swapped out by another memory request.

Parameters:
1. module_ptr
2. frame_ptr

VIRGA routine: GDB_LOCK_FRAME

3.6.21 UFRAME_UNLOCK

Unlock a specified frame.

Parameters:
1. module_ptr
2. frame_ptr

VIRGA routine: GDB_UNLOCK_FRAME

3.6.22 LMODULE_LOCK

Lock the specified module in system memory so it will not be swapped out.

Always returns success, if valid pointer.

Parameter: module_ptr

VIRGA routine: GDB_LOCK_MODULE

3.6.23 UMODULE_UNLOCK

Unlock a module.

Always returns success, if valid pointer.

Parameter: module_ptr

VIRGA routine: GDB_UNLOCK_MODULE

3.6.24 LTMOD_TYPE_LOCK

Lock all modules of the specified type that are in memory.

Always returns success, if valid pointer.

Parameter: module_type to be locked [2]

VIRGA routine: GDB_LOCK_MODULES_BY_TYPE

3.6.25 UTMOD_TYPE_UNLCK

Unlock all modules of the specified type.

Always success, if valid pointer.

Parameter: module_type to be unlocked [2]

VIRGA routine: GDB_UNLOCK_MODULES_BY_TYPE

3.6.26 WINDOW

Report the ID's of all data frames (of the specified type) whose recognition areas intersect the specified window. The search window specified can be a rectangle or just one point.

Parameters:
1. frame_type
2. size (in bytes) for the RESULT buffer [1Ø24]
3. X1 co-ordinate
4. Y1 co-ordinate
5. X2 co-ordinate
6. Y2 co-ordinate

For instance:

```
WIN 6 @ 1 1 1 1
```

reports (using a RESULT buffer of default size) all the frames of type 6 which intersect the point, (1,1).

VIRGA routine: GDB_FIND_FRAMES_IN_WINDOW

3.6.27 NCREATE_INDEX

Create a name index for a specified frame-type. This command requires many parameters, which means you might want to use the CINPUT command, and put the parameters in a file. Some command files for creating indexes already exist: see your group's system integrator.

Parameters:
1. frame_type to index [6]
2. frame_type for index frames [26]
3. max name_length [16]
4. name's offset [28]
5. frame offset of NAME_LEN [26]
6. KEY2 offset
7. Extra bytes in symtab [Ø]
8. flags (1 = upper case) [1]
9. initial number of entries [64]
1Ø. incremental number of entries [32]

VIRGA routine: GDB_CREATE_INDEX

3.6.28 NDELETE_INDEX

Delete the name index for a specified frame-type.

Always success, if given a valid frame_type number, even though no name-index exists for that frame_type.

Parameter: frame_type [4]

VIRGA routine: GDB_DELETE_INDEX

3.6.29 NLIST_INDEX

List in text format all of a name index (index is not sorted).

Parameter: frame_type [4]

Below is an example of the command's output:

```
***Name Index for frame_type=6, key2=Ø. Count=18.
  *NAME                NAME_LEN FRAME_ID FLAGS
   H18.B                    5      168    ØH
   H27.A                    5      175    ØH
   M13.A                    5      16Ø    ØH
   HØ2.A                    5      133    ØH
   HØ2.B                    5      134    ØH
   HØ3.A                    5       52    ØH
   O1Ø.D                    5       21    ØH
   O11.4                    5      239    ØH
   O14.4                    5      119    ØH
   IØ3.2                    5       28    ØH
   IØ3.3                    5       29    ØH
   PAD.23                   6      198    ØH
   IØ3.7                    5       31    ØH
   I13.7                    5      1ØØ    ØH
   I14.2                    5      144    ØH
   PAD.22                   6      197    ØH
   PAD.F                    5      178    ØH
   O25.9                    5      395    ØH
```

SEED routines: SEED_LIST_INDEX
SEED_DUMP_INDEX

SEED calls VIRGA routines: GDB_FIND_INDEX_FRAME
GDB_MAX_ID

3.6.30 NINDEX_FIND

Finds an index for the specified frame_type, if the index exists. Reports the module and frame pointers of the found index.

Gives an error if no name index is found.

Parameters:
1. frame_type [4]
2. the frame's KEY2, if desired (e.g., a page ID).

VIRGA routine: GDB_FIND_INDEX_FRAME

3.6.31 NFIRST

Find the first matching name of specified TYPE with specified KEY2. Bring the matching frame into memory and output the exact name found.

Parameters:
1. frame_type which is indexed [4]
2. key2, if required (e.g., page-id)
3. the name to find

An example of the command's output is shown below:

```
Found name "PAD.21", name_len=6, frame_id=196, flags=ØH.
 and found frame at mod_ptr=6FDC:ØH, frame_ptr=6FDC:1421H.
```

VIRGA routine: GDB_FIRST_NAME

3.6.32 NNEXT

Find the next matching name of specified TYPE with specified KEY2. Bring the matching frame into memory and output the exact name found.

Parameters:
1. frame_type which is indexed [4]
2. key2 (e.g., page-id)
3. name to find

An example of the command's output is shown below:

```
Found name "PAD.21", name_len=6, frame_id=196, flags=ØH.
 and found frame at mod_ptr=6FDC:ØH, frame_ptr=6FDC:1421H.
```

VIRGA routine: GDB_NEXT_NAME

3.6.33 NSFIRST

Search multiple files for the first matching name of specified TYPE with specified KEY2. Bring the matching frame into memory and output the exact name found.

Parameters:
1. frame_type which is indexed [4]
2. key2
3. name to find

An example of the command's output is shown below:

```
Found "HØ1.A", name_len=5, frame_id=5Ø, flags=ØH, file=1,
    root_ptr=6AB3:ØH
 and found frame at mod_ptr=7CB9:ØH, frame_ptr=7CB9:21FH.
```

VIRGA routine: GDB_SEARCH_FIRST_NAME

3.6.34 NSNEXT

Search multiple files for the next matching name of specified TYPE with specified KEY2. Bring the matching frame into memory and output the exact name found.

Parameters:
1. frame_type which is indexed [4]
2. key2
3. name to find

An example of the command's output is shown below:

```
Found "HØ1.A", name_len=5, frame_id=5Ø, flags=ØH, file=1,
    root_ptr=6AB3:ØH
 and found frame at mod_ptr=7CB9:ØH, frame_ptr=7CB9:21FH.
```

VIRGA routine: GDB_SEARCH_NEXT_NAME

4 THE BASIS OF PDQ IN SEED AND VIRGA

PDQ is based on SEED and VIRGA.

SEED is an interactive command interface to procedures. SEED provides PDQ with some direct-to-memory procedures and serves as an interface to all of PDQ's Virga routines.

PDQ is also based on VIRGA. Since the point of PDQ is to access a VIRGA-created database, it contains a lot of VIRGA routines that have been turned into SEED procedures. Almost all VIRGA routines are available as commands in PDQ through the SEED_VIRGA interface.

All commands available in PDQ are linked to the GM-1Ø schema. Most commands are linked to the specifics of that database.

For SEED, three documents exist:

1) the SEED User Guide
   (available as a Design-File photocopy only, not on floppy)
2) SDØ1.TXT (REV 1)
3) SDØ2.TXT (REV 2)

For VIRGA, there exists one document: the VIRGA User Manual (Design File Doc. No. 4.21). It is available through the HELP facility. If you must print from floppy, the file is VIRGA.TXT, accompanied by drawing files.

VIRGA routines know about modules and frames in general, but don't know the actual data fields in the schema at all.

PDQ can be used to access any VIRGA database because PDQ's tables, such as those in PDSCH.P86, describe what Virga-internal structures can be expected in a file, and where they will be found. PDQ, because of its tables, knows what offsets to use to reach the application data.

PDSCH.P86 is the source of PDQ's tables (for a specific schema). It gives all the offsets of the fields, relative to their records. This is handy information if you are trying to decipher a hex dump, or write an ICE macro. Remember that the offset in a header is not the offset in the frame, due to the frame_hdr, which is 16 bytes long. If the MACRO_FRAME (for example) was declared without the frame header, the user must add 1ØH to the offset in the header to achieve the offset into the frame. Some PDQ input files contain the headers (or an offset), but some do not.

The file to be accessed contains VIRGA information (in its file descriptor) about what module and frame types it has, where they are, and what frame-types are legal for the PDQ user to work with.

5 COMMAND TEXTS

The advantage of a command text is that it can invoke PDQ, and call the BUILDER command in order to build a database file or add or delete frames. A user can edit the output file from any of the three dump routines (PFRAME, DTYPE, DALL) and use it as input to PDQ.

It is strongly suggested that new PDQ users sit down at a workstation and try using PDQ interactively before creating big command files.

Output redirection is handled uniformly as part of SEED. It is suggested that at first the OUTPUT command not be used. Rather, one should redirect the output on a per command basis by giving the name of the port, preceded by a > (or a >> for append), just after the command name.

Comments are signified by a ~ anywhere in the line, and extend to the end of line. They can be written anywhere in the text.

SEED does not necessarily fold to upper case. It does not fold to upper case when executing the BUILDER command. Thus (due to PDQ sensitivity), the field names must be given in upper case. One can enter lower case names, but generally it is best to give PDQ a file that is totally in upper case.

The BUILDER command can be used to create a whole database through PDQ, but usually you should obtain the correct text

files from your group's system integrator.

Before proceeding with command file usage, the reader should get a copy of TPDQ.TXT (or look at the output of the dump routines). This is a sample of the text file format that PDQ expects. The input can either be typed at the keyboard to PDQ, or else PDQ can read it from a file. To have PDQ take input from a file, use the CINPUT command.

Notice that in each frame in TPDQ.TXT the keyword %UPDATE_FRAME appears. The other two choices are %ADD_FRAME, and %SYNTAX_FRAME. %ADD_FRAME and %UPDATE_FRAME tell PDQ to make the corresponding VIRGA calls. %SYNTAX_FRAME tells PDQ to not make any VIRGA call. That is, all that happens is that the syntax is checked.

All fields must be in the order given in TPDQ.TXT, which follows the order in the schema. The one relaxation on this rule is that fields can be skipped. In this case, F's will be put into the skipped field, unless the field is one whose correct value is computed by PDQ. These will be enumerated below.

To enter several records of the same type, follow the example given in TPDQ.TXT. To not have any record of a certain type, simply do not mention that type of record.

Many fields can be given in column format, though this depends on the database schema that PDQ is reading. Column format is not totally free format. You must put the correct number of

entries in each line.

String lengths are automatically filled in. To be able to dump and read back in a zero length string, string values of fields must be quoted strings. These are not normal quoted strings in that (1) blanks are not allowed and (2) to signify a quote in the middle of a quoted string, use just one quote. This is because tokens are delimited by white space (blanks, eolns, tabs). If a string-valued field is not even mentioned, and neither is its length field, then both of those fields will be filled in with F's, which is NOT what the user wants. So be careful. The automatic filling in of string lengths only happens for strings that are explicitly given in the input.

Many fields are automatically set. Frame_hdr.size, counts, and offsets, are always automatically set, as are frame_hdr.type, and string lengths. The automatic values over-ride user values for these critical fields in order to avoid disastrous consequences. In the GATEMASTER system, the frame_hdr.opcode and frame_hdr.hdr_size fields are automatically filled in. No other opcodes or hdr_sizes are automatically filled in.

The input text file should end with a %END statement. If the text was created by one of the PDQ dump routines, it will already have the %END statement that it needs. In any case, whether with a %END, or via an error clean-up, the input port automatically will be reset to :KEYBOARD.

*** INDEX ***

ØFFFFH 3-34
Abbreviated 3-4
ADD_FRAME 3-25, 5-2
ALLOCATE 3-6
BUILDER 3-9, 3-12, 3-15, 3-24, 3-25, 3-26, 3-27, 3-28, 5-1
Case 3-11, 3-46, 5-1, 5-2, 5-3
CINPUT 3-8, 3-9, 3-2Ø, 3-24, 3-46, 5-2
CLEAR_DIRTY_MOD 3-41
CLOSE 3-15, 3-32
Column 5-2
Comments 5-1
COPY_MEMORY 3-12
CREATE 3-2Ø, 3-31
DALL 3-9, 3-24, 3-28, 5-1
DEALLOCATE 3-6
DIRTY_MODULE 3-15, 3-41
DTYPE 3-9, 3-24, 3-27, 5-1
ELT 3-3
EVALUATE 3-17
EXIT 3-7, 3-15
FADD 3-15, 3-34
FALLOC 3-39
FDELETE 3-15, 3-36
FFIND 3-34
FGROW 3-35
FILE_NUM 3-19
FILE_STATUS 3-4, 3-19
FILL_MEMORY 3-4, 3-16
FIRST_FRAME 3-37
FLIP 3-3Ø
FLUSH 3-33
Fold 5-1
Format 3-4, 3-11, 3-13, 3-47, 5-2
FRAME_TYPES 3-29
FTDELETE_BY_TYPE 3-36
FUPDATE 3-15, 3-35
GAIL 1-1, 2-1, 3-2
Hdr_sizes 5-3
HELP 3-4, 3-7, 4-1
ID 3-5, 3-23, 3-34, 3-37, 3-48
INTERFACE 3-1
INTRODUCTION 1-1
LFRAME_LOCK 3-42
LMODULE_LOCK 3-43
LTMOD_TYPE_LOCK 3-44
MALLOC 3-39
MDISPOSE 3-4Ø
MGET_MODULES 3-38
MOD_INVENTORY 3-23
MTDISPOSE 3-4Ø

NCREATE_INDEX 3-46
NDELETE_INDEX 3-46
NEXT_FRAME 3-37
NFIRST 3-49
NINDEX_FIND 3-48
NLIST_INDEX 3-47
NNEXT 3-49
NSFIRST 3-50
NSNEXT 3-50
Opcodes 5-3
OPEN 3-32
Original 2-1, 3-1
OUTPUT 3-9, 3-10, 3-27, 3-28, 5-1
PFRAME 3-9, 3-24, 3-25, 3-26, 5-1
Pointer-arithmetic 3-17
Pointer-type 3-5
PURPOSE 2-1
READ_MEMORY 3-11, 3-13, 3-18
Redirection 3-9, 3-10, 3-11, 5-1
ROOT_STATUS 3-22
Schema 3-2, 3-3, 3-4, 3-23, 4-1, 4-2, 5-2
SET_FILE_DESC 3-20
SHIFT_CANCEL 3-4
String-valued 5-3
SYNTAX_FRAME 3-25, 5-2
Tables 4-2
Test 1-1, 3-1, 3-12, 3-14
Tokens 3-15, 5-3
UFRAME_UNLOCK 3-42
UMODULE_UNLOCK 3-43
UNLOCK_FILE 3-21
Update 3-3, 3-12, 3-25, 3-35
UPDATE_FRAME 3-25, 5-2
UTMOD_TYPE_UNLCK 3-44
Version 1-1, 3-3, 3-9, 3-26, 3-27, 3-28, 3-30
White 3-15, 5-3
WINDOW 3-45
WRITE_MEMORY 3-1, 3-14, 3-15, 3-16