Megamax, Inc.
DEVELOPMENT SYSTEMS

LASER C

C Language Development System

Atari ST

Laser C 3
Laser Shell and Editor 4
C Compiler 5
Linker 6
Disassembler 7
Archiver/Linker 8
Symbol Namer 9
Make Utility 10
Resource Construction Program 11
Compile and Link 12
Egrep 13
Disk Utilities 14
UNIX Compatible Routines 15
GEM AES 16
VDI 17
BIOS, GEMDOS, XBIOS Routines 18
Line-A Graphics Routines 19
Utility Routines 20
File Formats A
System Globals B
DOS Error Codes C
Key Codes D
Header Files E
Index I

Preface

Megamax, Inc. makes no warranty of any kind in respect to this manual or the software described in this manual. The user assumes any risk as to the quality, performance, and accuracy of this product. In no event will Megamax, Inc. be liable for direct, indirect, incidental, or consequential damages resulting from any defect in the performance or use of this product.

This manual was formated with LaTeX running on an ISI V16S computer and printed on an Apple Laserwriter. The fonts used are "Almost Computer Modern" roman and typewriter.

Megamax C and Laser C are trademarks of Megamax, Inc. UNIX is a trademark of AT&T, Inc. Atari ST is a trademark of Atari Corporation. GEMDOS is a trademark of Digital Research Corp.

Contents

Preface ii

1 Laser C Package 1
- 1.1 Components . . . 1
- 1.2 Update Policy . . . 2
- 1.3 Defective Media Warranty . . . 2

2 Introduction 3
- 2.1 Implementation . . . 3
- 2.2 Hardware Requirements . . . 4
- 2.3 System Setup . . . 4
  - 2.3.1 Single-sided Drive Installation . . . 4
  - 2.3.2 Double-sided Drive Installation . . . 4
  - 2.3.3 Hard Disk Installation . . . 4
- 2.4 Conventions . . . 5
  - 2.4.1 Scroll Bar Usage . . . 6
  - 2.4.2 Selector Usage . . . 7
  - 2.4.3 File Name Conventions . . . 8
- 2.5 Development Steps . . . 8
- 2.6 Sample Session . . . 9

3 Laser C 13
- 3.1 Language . . . 13
  - 3.1.1 Data Types . . . 13
  - 3.1.2 C Preprocessor . . . 13
  - 3.1.3 External Names . . . 14
  - 3.1.4 Enumeration Types . . . 15
  - 3.1.5 Structure Assignment . . . 15

3.1.6 Character Constants 15
3.1.7 Scope of Identifiers 16
3.1.8 Forward Pointer References 16
3.1.9 Assembler 17
3.2 Language Implementation 18
3.2.1 Size of Data Elements 18
3.2.2 Code Generation 18
3.2.3 Switch Statement 19
3.2.4 Function Call Conventions 20
3.2.5 Register Variable Support 21
3.2.6 Assembler 21
3.2.7 Storage Allocation/Initialization 26

4 Laser Shell and Editor **27**
4.1 Shell Startup 27
4.2 Shell Configuration 28
4.2.1 Tools 28
4.2.2 Environment Variables 30
4.2.3 Save Configuration 32
4.2.4 Disk Cache 32
4.3 Text Editor 33
4.3.1 Editor Menu Usage 34
4.3.2 Choosing A File 34
4.3.3 Mouse Usage 35
4.3.4 Scrolling the Window 36
4.3.5 Insertion Point Keys 36
4.3.6 Text Entry 37
4.3.7 Block Operations 37
4.3.8 Editor Options 39
4.3.9 Finding Text 40
4.3.10 Text Marks 42
4.3.11 Rearranging Windows 42
4.3.12 File Information 43
4.4 Running Programs 43
4.4.1 Menu Command Execution 43
4.4.2 Command Line Execution 45
4.5 Disk Operations 47
4.6 Project Management (Make) 48
4.7 Debugging 49
4.8 Menu Summary 51

4.9 Keyboard Summary 52

5 C Compiler 55
5.1 Command Line Usage 55
5.2 Compiler Errors 56
5.3 Memory Usage 56

6 Linker 57
6.1 Command Line Usage 58
6.2 Linker Errors 59
6.3 The Linking Process 60
6.4 Desk Accessory Support 61

7 Disassembler 63
7.1 Command Line Usage 63
7.2 Disassembler Errors 64

8 Archiver/Librarian 65
8.1 Command Line Usage 65
8.2 Random Library 66
8.3 Archiver Errors 66

9 Symbol Namer 69
9.1 Command Line Usage 70
9.2 Namer Errors 70

10 Make Utility 71
10.1 Command Line Usage 72
10.2 A Simple "Makefile" 73
10.3 Makefile Structure 74
10.3.1 Entries 74
10.3.2 Comments 75
10.3.3 Macro Definition 75
10.3.4 Implicit Macros 77
10.3.5 Dynamic Dependency 77
10.3.6 Suffixes Table 77
10.3.7 Transformation Rules 78
10.4 Examples 78

11 Resource Construction Program 81
11.1 Definition of Resource Files 81
11.2 RCP Usage 83
11.2.1 Tree Types 84
11.2.2 Visual Hierarchy 84
11.2.3 Menu usage 85
11.2.4 Mouse usage 85
11.2.5 Resizing 86
11.2.6 Keyboard Usage 86
11.3 Menu Functions 86
11.3.1 Edit Menu 86
11.3.2 File Menu 87
11.3.3 Options Menu 87
11.4 Object Dialogs 88
11.5 The Icon Dialog 89
11.6 The Bit Image 91
11.7 Using RCP as a Resource Editor 91

12 Compile and Link 93
12.1 Command Line Usage 93
12.2 CC Errors 94
12.3 Examples 94

13 Egrep 95
13.1 Command Line Usage 96
13.2 Egrep Errors 97
13.3 Example Searches 97

14 Disk Utilities 99
14.1 LS 99
14.2 CP 100
14.3 MV 101
14.4 RM 101
14.5 RMDIR 102
14.6 MKDIR 102
14.7 CAT 103
14.8 DUMP 103
14.9 SIZE 103

15 UNIX Compatible Routines **105**
15.1 Line Separators 105
15.2 File I/O 106
15.3 I/O Redirection 106
15.4 Device I/O 106
15.5 Memory Allocation 107
15.6 Program Parameters 107
15.7 Summary of Routines 108

16 GEM AES **163**
16.1 Creating a GEM Application 165
16.2 Applications Manager 171
16.3 Event Manager 179
16.4 Form Manager 197
16.5 File Selector Manager 209
16.6 Graphics Manager 213
16.7 Menu Manager 225
16.8 Object Manager 237
16.9 Resource Manager 261
16.10Scrap Manager 271
16.11Shell Manager 275
16.12Window Manager 281

17 VDI **307**
17.1 VDI Examples 310

18 BIOS, GEMDOS, XBIOS Routines **417**
18.1 BIOS Interface 417
18.2 XBIOS Interface 418
18.3 GEMDOS Interface 418
18.4 GEM Run-time Structure 418

19 Line-A Graphics Kernal **531**
19.1 Line-A Graphics Routines 531
19.2 Graphics Modes 532
19.2.1 High-resolution Mode 532
19.2.2 Medium-resolution Mode 532
19.2.3 Low-resolution Mode 533
19.3 Line-A Port 533
19.4 Line-A Data Structures 534

20 Utility Routines **557**

A File Formats **577**
A.1 Laser Object File Format 577
A.2 DRI Object File Format 579
A.3 GEMDOS Application File Format 580

B System Globals **581**

C DOS Error Codes **587**

D Key Codes **589**

E Header Files **593**

Chapter 1

Laser C Package

1.1 Components

The Laser C Development System includes this manual, a warranty card which should be filled out and returned, and three single-sided diskettes, labeled "SYSTEM", "WORK", and "UTILITY". The SYSTEM diskette contains:

- **MEGAMAX** Folder
  - **CCOM.TTP** C compiler
  - **LD.TTP** Linker
  - **LASER.CFG** Configuration file
  - **LASER.RSC** Required by LASER.PRG
  - **MAKE.TTP** Make utility
- **LASER.PRG** Laser development shell (includes editor)

The WORK diskette contains:

- **EXAMPLES** Folder of example programs
- **MEGAMAX** Folder
  - **HEADERS** Folder of C header files
  - **INIT.O** C initialization code
  - **LIBC.A** C function library
- **CC.TTP** Compile and link utility

The UTILITY diskette contains:

AR.TTP	Archiver/Librarian
CAT.TTP	File display utility
CP.TTP	File copy utility
DIS.TTP	Disassembler
EGREP.TTP	Multi-file string search utility
LS.TTP	File list utility
MKDIR.TTP	Create folder utility
MV.TTP	File move utility
NM.TTP	Symbol Table Dumper
RCP.PRG	Resource Construction Program
RCP.RSC	Required by RCP.PRG
RM.TTP	File remove utility
RMDIR.TTP	Folder remove directory
DUMP.TTP	Hex file display utility
SIZE.TTP	File size utility

1.2 Update Policy

Updates to this product, when released, are made available to registered users by sending the original diskettes along with $20 to Megamax. Updates include new disks along with a copy of the new documentation. Please fill out and return the warranty registration card to Megamax. Megamax user services, such as B.B.S. access, update announcements, and the Megamax newsletter depend on the purchaser having done this. The update policy is subject to change without notice.

1.3 Defective Media Warranty

If any physical defects are discovered with the magnetic media within a period of 60 days after purchase, assuming normal use of the diskette, Megamax, Inc. will replace the diskette free of charge. The original diskette must be returned to Megamax, Inc.

Megamax, Inc
Box 851521
Richardson, TX 75085
(214) 987-4931

Chapter 2

Introduction

Laser C is a complete, professional quality C language development system for Atari ST computers. It includes a compiler, a linker, an integrated shell and editor, header files, a library of UNIX compatible routines, and a complete interface to the Atari ST ROM routines. In addition, the package includes an archiver, a resource construction program, a project manager (Make) utility, and a catalogue of example source programs.

The Laser Shell has a multi-window mouse-based text editor, a built-in disk cache, a facility for making system programs RAM resident, and many other special features which aid program development.

2.1 Implementation

The C compiler was implemented according to the book "The C Programming Language" (also known as K&R) written by Brian W. Kernighan and Dennis M. Ritchie. This manual does not attempt to restate the principles of their book, but rather provides the programmer with information about the Laser C implementation. Numerous language extensions have been included which are beyond K&R, such as enumeration types, structure passing and assignment, and in-line assembly of Motorola 68000 instructions.

To make full use of the in-line assembly feature, it will be necessary to obtain the Motorola M68000 Programmer's Reference Manual, published by Prentice-Hall, Inc.

2.2 Hardware Requirements

Laser C may be used on an Atari ST with a single-sided disk drive. However, for the most efficient development, it is suggested that an Atari ST with one megabyte of memory and a double-sided 800K disk drive be used.

IMPORTANT: Make working copies of each of the three original diskettes and put the originals in a safe place.

2.3 System Setup

2.3.1 Single-sided Drive Installation

If a single-sided drive system is to be used it is suggested that the duplicate SYSTEM be inserted to start "**LASER.PRG**", which will load the compiler and linker into RAM. Once loaded, the SYSTEM disk can be replaced with the duplicate WORK disk, which should have sufficient room for development. If a second drive is available, it may be used to run programs on the UTILITY disk, or it may be used for extra storage.

2.3.2 Double-sided Drive Installation

If a double-sided drive system is to be used, copy the contents of the WORK disk to the duplicate SYSTEM disk and use it disk for development. If a second drive is available, it may be used to run programs on the UTILITY disk, or it may be used for extra storage.

2.3.3 Hard Disk Installation

When installing the Laser Development System onto a hard disk it is suggested that a folder called "**MEGAMAX**" be created, into which all files in the "**MEGAMAX**" folders on the SYSTEM and WORK disks be placed. The remaining files on the SYSTEM, WORK, and UTILITY disks can then be placed on the root of the drive. Creation of this folder is *not* required; however, the Laser Shell and the Resource Construction Program will both look for necessary files in this folder if not found in the folder in which the programs reside. Since the Laser Shell uses path names and search paths to locate and run programs, other configurations are possible.

NOTE: The MEGAMAX folder may not be placed inside of another folder, it must remain at the top level of the disk on which it is created.

2.4 Conventions

The following conventions apply to the remainder of this document.

Cursor	The mouse cursor.
Insertion point	The blinking text cursor in an editor window.
Press/type	Keyboard entry is required.
Click	Position the cursor over an item and press the left mouse button.
Double-click	Like click except the mouse button is pressed twice in rapid succession.
Shift-click	Click while holding down the "Shift" key.
Control-click	Click while holding down the "Control" key.
Choose	Make a menu selection by moving the mouse into the menu bar and clicking on an item.
Drag	Indicates that the mouse is moved while the mouse button is held down.
Shift-Drag	A mouse drag in which the "Shift" key is held down (before the drag is begun).
Control-Drag	A mouse drag in which the "Control" key is held down (before the drag is begun).
Select	Often text or other graphical items are selected so that subsequent operations may apply only to the selected items. Selected items are hilighted in some manner. Selected text is often printed in white-on-black. Extended or multiple selections are typically made by Shift-clicking.

2.4.1 Scroll Bar Usage

Scroll bars are typically used to position a window or selector box over text which will not fit entirely within the window or box. The white area of a scroll bar (the thumb) indicates the relative size of the window to the content of the window. Thus, an entirely white scroll bar indicates that the entire contents of the window are visible and that no scrolling is necessary. The thumb may be dragged to directly position the window anywhere over the file. Clicking and holding down the left-hand mouse button on an arrow moves the window over its contents in the indicated direction. Clicking on a grey area moves the window over its contents by a larger amount, typically one-half of the visible content of the window.

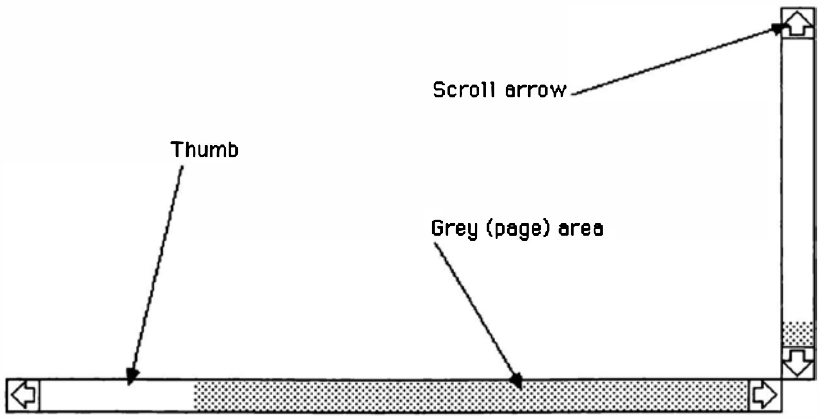


Figure 2.1: Scroll Bar Components

2.4.2 Selector Usage

The Laser Shell often displays lists in what are called "selector boxes". A selector box is an interactive device which allows the display and selection of items (names or strings). If the list of items in a selector box is too long to fit in the box, a scroll bar becomes active with which the list may be scrolled. Some selector boxes allow multiple selections while others allow only one item to be selected at a time. Multiple selections, when allowed, are made by clicking on the first item to be selected and then dragging across other items to be selected, or by shift-clicking on each item to be selected.

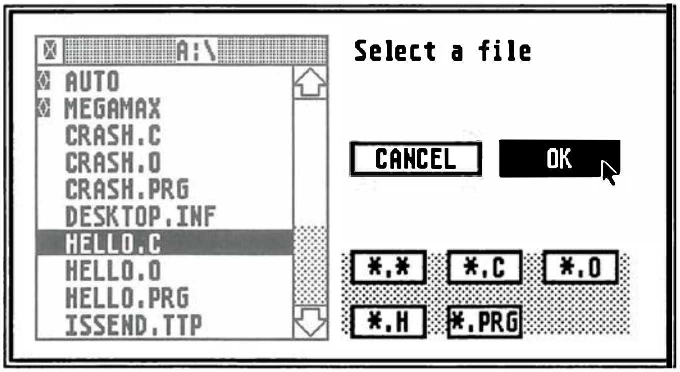


Figure 2.2: Open File Selector Dialog

A special variety of selector box is the "file selector box". A file selector box displays a sorted list of file names and folder names. Folder names are preceded by a graphical character and are sorted to the top of the list. Selections and multiple selections (when allowed) are performed for the standard selector box. A folder is opened by double-clicking on the folder name, and a folder is closed by clicking on the graphical character in the upper left-hand corner of the selector box. Closing the root (the disk drive name) causes a list of available drives to be displayed. Double-clicking on a drive name changes to the root of that drive.

2.4.3 File Name Conventions

A GEMDOS file extension is the optional one to three characters ending a file name, separated from the rest of the file name by a period. File name extensions are typically used to indicate the type of the file. The following file name extensions are recognized by the Laser Shell and the Resource Construction Program:

.ACC	Desk accessory.
.PRG	GEM (graphic) application program.
.TOS	Character based application program.
.TTP	TOS program which takes program parameters.
.CFG	Shell configuration file.
.LNK	Shell linker dialog configuration.
.RSC	GEM resource file.
.DEF	File of resource name definitions (used by "**RCP.PRG**").
.C	C language source file.
.H	C header file.
.O	Object code file.
.A	Archive file.

2.5 Development Steps

In general, the sequence of steps taken to develop a C program, known as the development cycle, are:

Edit	Create the program source with a text editor.
Compile	Run the compiler on the program source. If the program contains errors, as reported by the compiler, return to the **edit** step. If no errors occur, the compiler will output a linkable object file.
Link	Run the linker supplying as input the names of object files (or archives of object files). If the linker reports any errors, return to the **edit** step. Barring any link errors, an executable program will be output by the linker.

Run/Debug Running the executable program may reveal errors. Repeat these steps until the program runs as desired.

The Resource Construction Program may be utilized to create resources for applications. This step is independent of the above process.

The Laser Shell serves as both a source editor, and a utility from which the compiler and other development programs may be run. These programs may alternately be run from the GEM desktop or from a command line shell; however, the Laser Shell has facilities which greatly decrease development time.

2.6 Sample Session

In the following example, a simple C program is created.

- Follow the installation procedure described above. A RAM disk may be used with the Laser Shell, but will not improve performance due to the automatic disk cache within the Shell.

- From the GEM desktop, double-click on the file "**LASER.PRG**". After a few seconds the menu bar will change to the editor's menu bar. An alert box should be presented which proclaims "Loading RAM Resident Programs". If this alert is ***not*** seen, the Shell did not find a configuration file in which case the Shell should be terminated (by choosing **Quit** from the **File** menu). The configuration file, named "**LASER.CFG**" should then be copied to either the same location (drive and folder) as the program "**LASER.PRG**", or to the "**MEGAMAX**" folder, if present.

- When the RAM resident programs are loaded properly, choose **New** from the **File** menu. An empty editor window will appear.

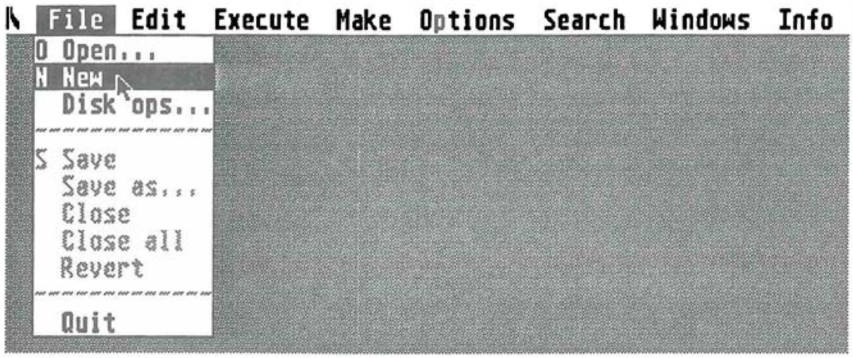


Figure 2.3: New File Item

- Enter the following program ***verbatim*** by typing it into the window. If a typing mistake is made, erase with the "Backspace" key. The insertion point may be moved by clicking on the desired character. Press the "Return" key to create a blank line at the insertion point.

Figure 2.4: Example Program

- Now choose **Save** or **Save as ...** from the **File** menu. When a dialog appears, press the "ESC" key, type "HELLO.C", and press "Return". The file has now been saved to disk and given a name with the "**.C**" extension, which is necessary for the **Run** command.

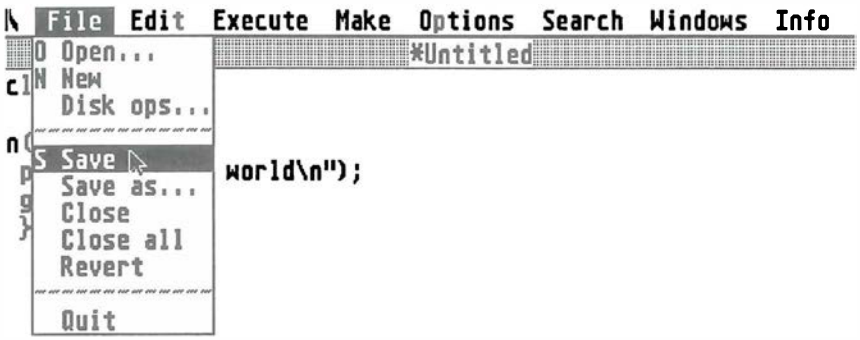


Figure 2.5: Save File Item

- Next, choose **Run** from the **Execute** menu. The Shell will compile, link, and run the program in the front window. If the program contains any errors, correct them and select **Run** again. When run, the program should print directly onto the screen and then wait for a "Return". When the "Return" is typed, the program will terminate and return to the Shell.

Figure 2.6: Run Program Item

- Finally choose **Quit** from the **File** menu. The program "**A.PRG**" should appear on the desktop as a stand-alone application.

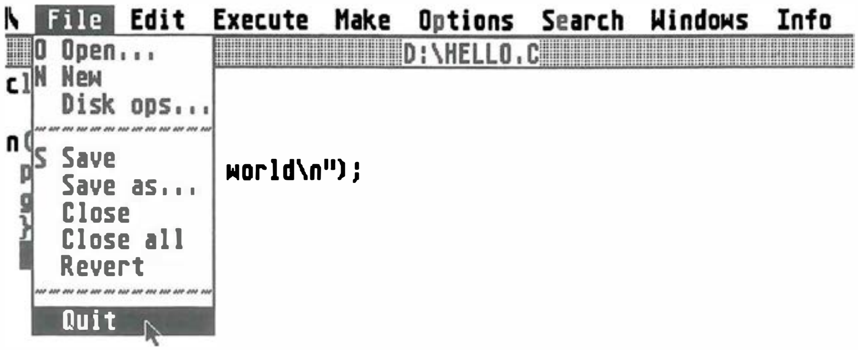


Figure 2.7: Quit Laser Shell Item

Chapter 3

Laser C

3.1 Language

3.1.1 Data Types

The C compiler supports all of the standard scalar types of the C language: **char**, **int**, **short**, **long**, **unsigned**, **float**, and **double**, as well as pointers to all types. Also **unsigned char**, **unsigned long** and **enum** are supported. Bitfields are supported but the fields must only be **unsigned**.

Void Data Type

The keyword **void** is used to tell the compiler that a function does not have a return value. For example

```
void foo()
{
    printf("Hello world\n");
}
```

The **void** type can not be used in an expression.

3.1.2 C Preprocessor

The Laser C preprocessor follows the specification given in K&R. There are some extensions, however.

3

The restriction of having the '#' in the first column of a line has been lifted. There can also be any amount of white space between the '#' and the preprocessor command but the command an the '#' must be on the same line. If there is no command on the same line following a '#' that line is skipped. e.g.

```
#    define TRUE 1
#ifdef TRUE
#
#endif
```

Note that there is white space prior to the **define** and that there is no command on one of the lines with a '#' on it.

Include File Processing

The **#include** feature of the standard C preprocessor allows file names to be given within either double quotes or angle brackets. File names in double quotes make the compiler search the directories in this order:

1. The directory of the source file that contains the include command.
2. Any files given to the compiler with the -I option (see section 5.1).

File names in angle brackets cause the compiler to start searching at step 2 above.

Include files may be nested to a depth of 6 levels, including the main module level. An attempt to nest beyond this maximum (if an include file inadvertently **#included** itself) results in an error message.

3.1.3 External Names

Identifiers (names of variables and functions) may contain up to 255 characters each. As per the standard for the C language, both upper and lower case letters are allowed in identifiers, and are distinct from each other. That is, the names `myvar` and `MyVar` are different. The underscore character (_) is also legitimate within identifiers, as are digits. Identifiers, though, may not begin with a digit. It should be noted that various internal functions, such as floating point routines and support for long integers, have names beginning with an underscore. The programmer should therefore avoid using identifiers which begin with an underscore if possible.

3.1.4 Enumeration Types

Laser C supports enumeration types using 16-bit representation of the enumeration constant. An example of an enumeration type is:

```
enum { apples, oranges, bananas } fruits;

main()
{
    fruits = oranges;
}
```

The values assigned by the compiler start at 0 and are incremented for each identifier. In the example above **fruits** will have the value 1. Enumeration constants can also be assigned values when they are declared. The compiler will use that value to increment from.

```
enum { green = 5, orange, yellow = 10 } colors;

main()
{
    colors = orange;
    colors = yellow;
}
```

In the above example **orange** has the value 6 and **yellow** has the value 10.

3.1.5 Structure Assignment

Laser C supports **structure** and **union** assignment and passing. If x and y are **structures** of type **stype** then the following statements are legal:

```
x = y;               /* contents of y are copied to x */
foo(x);              /* x is passed by value to foo() */
struct stype bar();  /* function returning struct     */
```

GEM routines that have structures as parameters must be passed the address of the structure (using the "&" operator).

3.1.6 Character Constants

The definition of character constants has been extended in Laser C to allow **int** and **long** size as well as **char**. The syntax is a single quote followed by 1, 2 or

3

4 characters and a closing single quote. The resultant type will be a **char**, **int** or **long** respectively. An example of a character constant is:

```
long a = 'ABCD';  /* a will have the value 0x41424344 */
```

3.1.7 Scope of Identifiers

In general, name scoping within the C compiler is as per standard C. One exception to this standard is the treatment of identifiers of structure members. In Laser C, structure member names need not be unique across **struct** boundaries. Therefore it is valid for two different structures to contain members at different relative offsets with identical names. e.g.

```
struct {
    int     number;
    char   *name;    /* name is at offset 2 */
} struct_one;

struct {
    char  *name;     /* name is at offset 0 */
    char  *address;
} struct_two;
```

3.1.8 Forward Pointer References

A problem arises when two structures must refer to each other: the reference in the first structure causes an undefined type error because the second structure hasn't been defined yet. This mutual referencing almost invariable arises with some kind of linked data structure. The C compiler has been extended to allow pointer references to **structs** or **unions** that have not yet been defined. Note that this only works with pointers to structs or unions with a tag name (typedefs will not work). Additional errors will be generated later in the compile if the **struct** or **union** is never defined. e.g.

```
struct node {
    char  *symbol_name;
    struct type_node *type; /* type_node is not defined */
};

struct type_node {
    int type;
};
```

3.1.9 Assembler

The C compiler allows the addition of assembly language code to a C program directly in-line with the C code. The C language has been extended to include the construct:

```
asm {
     . . .
     MC68000 Assembler Instructions
     . . .
}
```

The code within the braces after the keyword `asm` is assembled and included in-line with code generated from surrounding C statements. In-line assembly may appear anywhere in your program; it is not necessary to place it inside a function.

The in-line assembler obviates the need for a separate assembler. General control structure, input/output, and complex data structures can be implemented in C, while certain low-level routines can be coded in assembly language within the same module. The problem of interfacing C functions to assembly language functions and vice-versa is eliminated, because calling sequences can be written in C for functions coded in assembler. Programs can first be developed in C to debug algorithms and to quickly generate a working prototype. Functions which comprise the most time consuming sections of the program (generally less than 10% of the code) can then be re-coded in assembly language. Because of the efficiency of the C code generator, such a hybrid approach yields execution speeds favorably comparable with pure assembly language code while retaining the ease of modification and maintenance of a pure high-level language approach.

However, the use of assembly language decreases readability, exacerbates debugging headaches, and drastically reduces portability. Discretion must be used when considering functions for hand translation. There are some situations where speed is critical, most notably graphics. Such applications frequently involve system or machine dependencies anyway, so portability is not an issue. In such cases, the availability of in-line assembly language is a great benefit.

See section 3.2.6 for the syntax of the assembler.

3

3.2 Language Implementation

3.2.1 Size of Data Elements

The amount of space allocated for each data type (in terms of 8-bit bytes) is as follows:

`char`:	1	`unsigned char`:	1
`unsigned`:	2	`unsigned long`:	4
`short`:	2	`float`:	4
`int`:	2	`double`:	8
`long`:	4	Pointer type:	4
`enum`:	2		

Floating point types are stored in IEEE standard format (although non-IEEE standard routines are used to perform floating point calculations. In particular, the 80-bit temporary is not supported).

Space for variables of type `char` is allocated on the next available byte boundary in memory if the variable is within a struct or union or is of storage class `auto`, or on the next available word boundary if the variable is `extern` or `static`. Space for all other variables, including those of any other storage class as well as arrays, `structs` and `unions`, is always allocated on the next available word boundary, regardless of storage class. Bit fields within `struct`'s are allocated in `unsigned` units, starting from the least significant bit. The maximum size of a string constant is 255 bytes.

3.2.2 Code Generation

The C compiler, including preprocessor, syntax check, and code generation, is one-pass. In other words, all work which needs to be done by the compiler is finished after looking at the contents of the source file once. The compilation process is thus quite fast.

Linkable object code is generated directly by the compiler; there is no assembly post-pass. The compiler performs many processor specific "strength reduction" optimizations, such as using MC68000 "quick" instructions, replacing multiplies by powers of two with shifts, and avoiding intermediate register loads when possible. Simple statements such as increments and assignment operations involving constants frequently generate only one machine instruction. For example, the statement

```
i++;
```

will compile into a single instruction to increment the variable `i`, and the statement

```
i = 50;
```

will compile to a single `MOVE` instruction. The statement

```
*p++ = *q++;
```

where `p` and `q` are register variables, will also compile to a single `MOVE` instruction.

Certain expressions involving constants will be evaluated at compile time. Therefore, the statement

```
i += 5 * ARRAYSIZE;
```

will generate one `ADD` instruction, assuming `ARRAYSIZE` is a constant which was `#defined`.

3.2.3 Switch Statement

The C compiler will generate one of three types of code for a given switch statement, depending on the values of three internal variables. The simplest type of code it can generate is a linear search of the case values. A faster method is a binary search of the case values. The fastest method employed is a jump table. A jump table is always as large as the range (maximum - minimum) of the case values, regardless of the number of cases, and is best used when the range is no greater than twice the number of cases (i.e. when half or less of the space in the table is wasted). To prevent this waste, a search method is used. The choice of linear or binary search is made on the basis of the number of cases. A linear search uses in-line code rather than a function call; however, a binary search can be faster where a large number of cases are used. The compiler can be given options to force it to generate code in any of the three ways (see section 5.1). The `-S` option to the compiler specifies three numbers: `a`, `b` and `c`. The compiler's code generator decides which code to generate by the following method:

```
if ( num_cases < a and case_density <= b )
    generate a jump table
else
    if ( num_cases <  c )
        generate comparison code
    else
        generate a binary search
```

3

Where:

```
case_density = (max_case_val - min_case_val) / num_cases
```

3.2.4 Function Call Conventions

Parameter expressions encountered in function calls are evaluated and then passed to the function on the stack. The parameters are pushed in the reverse of the order given in the parameter list. Reversal of the parameter list is necessary for functions with variable numbers of parameters. Such functions may access lists of parameters as follows:

```
/* Return max of list of ints; n gives list length */
max(n, p);
int n, p;
{
    int *pp, max = -32768;
    for (pp = &p; n; pp++, n--)
        if (max < *pp)
            max = *pp;
    return max;
}
```

The above function *max()* returns the maximum of an arbitrary number of integers. The number of integers is passed as the first parameter, followed by the list of values:

```
m = max(5, i, j, k * 2, 87, f(abc));
```

Note that the pointer variable **pp** is incremented in the for loop of the above function. The pointer will move down through the stack towards higher memory locations retrieving each parameter in turn. Any functions which use this method of obtaining parameters are not necessarily portable to other implementations of C.

Values are returned from functions in processor register D0, or in the case of double values, in the global variable `_fpreg0`. It is the responsibility of the calling environment to remove parameters from the stack after return from a function call. Each function must ensure that any registers used to hold register variable values are saved and then restored when the function terminates.

`Structs` may be passed by value. See section 3.1.5 for details.

3.2.5 Register Variable Support

Each function in a Megamax C program can expect up to eight registers available for register storage class variables. Four data registers are available for integral types (`char`, `short`, `int`, `long`, and `unsigned`), and four address registers are available for pointer variables. Judicious use of register variables can substantially increase execution speed and decrease code size. Data register D7 is allocated first, then D6, D5, and D4. Address register A5 is allocated first, then A4, A3, and A2.

3.2.6 Assembler

Syntax

The syntax for a line of assembly code is:

```
<label :> opcode<size>  <effective address>
```

Anything enclosed with <...> is optional. There can be more than one instruction per line. Opcodes can be in lower or upper case. Register names must be in uppercase. Comments take two forms. There is the standard C type comment /* ... */ or the assembly language comment which starts with a ';' and continues to the end of the line. e.g.

```
asm {
    clr.l   D0  ;   this is a comment
; a line with only a comment
    clr.l   D1  /* this is also a comment */
}
```

The effective address calculation follows the syntax of the Motorola 68000 manual. Addressing modes are not completely orthogonal in the Motorola 68000 instruction set. For complete information on addressing modes and instruction forms, consult a Motorola databook.

The size field can be any one of .B,. W, or .L for byte, word and long sizes respectively. Branch instructions can also have .S for short branches.

There can be zero or more labels on a line. e.g.

```
asm {
    label_1:
    label_2: label_3:   rts
    ; label_1, label_2, label_3 are at the same address
}
```

3

The identifiers CCR, USP and SR are also recognized by the assembler. They stand for the Condition Code Register, the User Stack Pointer and the Status Register respectively.

Note that **#defines** can be used to create simple macros, using the multiple statement per line feature. Within macros, C style comments must be used instead of the normal semicolon-to-end-of-line assembly language comments.

Expressions which give displacement values are restricted in that only one identifier may be involved. A constant expression may be added to or subtracted from this identifier. In such expressions, the identifier must be placed first in the expression; in other words, the statement

```
move   DO, x+2(A6)
```

is legal, but the instruction

```
move   DO, 2+x(A6)
```

is not.

Defaults

If no size specifier is given for an instruction which can operate on more than one size, the assembler defaults to word. If a size specifier is not applicable to a particular instruction, no specifier may be given. All labels default to local code labels unless declared as **extern** previously. This means that all functions called, for example, must be declared or defined previously in C. e.g.

```
extern void foo();

asm {
    foo:            ; foo is global
        bra   end
    end:            ; end is local
        rts
}
```

Branches default to word-sized displacements. A short branch can be forced by using a "**.s**", but no warning message will be given if the necessary displacement is too large for a short branch.

Pseudo Ops

The pseudo ops **DC.B**, **DC.W** and **DC.L** will emit data inline with assembly language. The syntax for a pseudo op is:

```
DC.<size> [constant expression OR string constant], ...
```

The size field can be either .B, .W or .L. There can be any amount of expressions or string constants separated by commas. For example:

```
asm {
    DC.B    "Hello world\0"
    DC.W    5, 10 * 15
    DC.L    0x80000000
}
```

None of the pseudo ops will align data on a word boundry. This means that That the user must ensure that all the data given in a pseudo op ends on an even byte. String constants are not NULL terminated.

Accessing C Variables

External and `static` variables from the C environment are accessed using the name of the variable (Absolute Long Addressing Mode). Auto variables are accessed as a displacement to address register A6 (Address Register Indirect with Displacement Mode). Register variables may be accessed by name from within in-line assemble. The first four non-pointer register variables are placed in data registers; the first four pointer register variables are placed in address registers (see Available Registers below).

```
foo(p)
register int *p;
{
    register int i;

    i = 5;
    asm {
        move.w i,(p)        ; generates move.w D7,(A5)
    }
}
```

Any excess register variables must be accessed relative to A6. The assembler will not report misuse of some variable names.

Functions in the C program can be referred to by name. Arguments are passed to functions on the stack in reverse of the order they are written in C. Values are returned from functions in data register D0, or in global _fpreg0 if the value is `double`.

Available Registers

Registers D0-D3 and A0 and A1 may be used without saving them. Registers D4-D7, A5-A2 are used for register variables, and are allocated in reverse numeric order. Each of these registers not used for a register variable within a function containing in-line assembly language must be saved by the assembly code if modified therein. Register A6 is used to access auto variables and register A7 is used as the stack pointer.

Creating Global Symbols

This section is not for the casual user of the in-line assembly and discusses the use of a construct that is very dangerous. It is almost never needed and should be avoided if at all possible.

The normal functions in C start with a link instruction to make room for local variables and then end with a corresponding unlink instruction. These instructions can be avoided by making a label inside assembly to be called instead of the C function name. An `rts` instruction must also be placed at the end of the routine to avoid the unlink instruction. To indicate that this is an `extern` or `static` symbol it must be declared before it is used as a label. This is done by declaring it as an `extern` or `static` function in C. Remember, by overriding the normal entry point a lot of nice things that C does about parameter passing and setting up local variables is lost. For example

```
extern int sqr();

foo(i)
int i;
{
    asm {
        sqr:                ; sqr is a global function
        move 4(A7), D0      ; load i into D0
        muls D0, D0         ; leave the result in D0
        rts
    }
}
```

Since the `link` is not performed the variable `i` can only be referenced using the stack pointer.

Expansion of `#defined` macros is performed within sections of assembly language, so the programmer is free to rename instructions or registers.

Assembly Language Examples

```
/*
    Program to invert the screen 50 times
*/

#include <osbind.h>
#define LENGTH  80*400
#define N       50

main()
{
    int j;
    char    *screen;

    screen = (char *)Physbase();    /* XBIOS routine */
    for (j = 0; j < N; j++)
        asm {
            move.l  screen(A6), A0
            move    #(LENGTH/4)-1, D0
      loop:
            not.l  (A0)+
            dbf     D0, loop
        }
}

/*
    Function to do a block move from the first pointer to the
    second. The routine moves one char at a time to allow
    odd addresses.  This also shows macro usage
    for assembly language.
*/

#define DEC(x)  subq #1, x

block_move(source, dest, count)
register char *source, *dest;  /* placed in address registers */
register int count;            /* placed in a data register */
{
    asm {
```

3

```
        DEC(count)          ; because dbf counts to -1
                            ; DEC will generate subq #1, count
    lp:
        move.b  (source)+, (dest)+
        dbf     count, lp
    }
}
```

3.2.7 Storage Allocation/Initialization

The compiler places all code into the TEXT segment, all initialized global variables, initialized static variables (both global and local), and string constants into the DATA segment (see File Formats, section A.1). Uninitialized global variables are not allocated by the compiler. Instead, special symbol information is placed into the object file containing the name of the variable and its size (called a common symbol). When the object code is linked, the linker collects all common names and allocates in the BSS segment enough space to accommodate the largest of each common symbol found. Uninitialized static global variables are placed into the BSS segment. Initialization code is required for auto variables so that the initialization may be carried out each time the function is executed.

Chapter 4

Laser Shell and Editor

Introduction

The Laser Shell allows integration of all phases of program development, from editing through debugging. It has a built-in mouse based editor, a dynamic disk cache which buffers disk access, a facility for running the compiler, linker, and other Laser programs directly from RAM, a file operations facility (copy, move, etc.), a project management system (Make), and debugging facilities.

Commands in this chapter are presented according to their functionality, rather than their order on the menu bar. At the end of this chapter a **Menu Summary** briefly presents each menu item in menu-bar order. It is assumed that the documentation conventions discussed in the introduction are understood (see section 2.4).

4.1 Shell Startup

To start the Laser Shell, double-click on the file "**LASER.PRG**" from the GEM desktop. The Laser Shell will only run in high or medium resolution modes (set by choosing **Set Preferences** ... from the GEM desktop's **Options** menu). Once started, the Shell tries to open a file called "**LASER.CFG**", which contains configuration information. This configuration information includes editor settings, environment variables, tool locations and attributes, and RAM residency attributes (discussed below). If found, the file is read, all editor settings are restored, and any RAM resident programs are loaded from disk into RAM residency. The Shell then waits for user interaction.

4

4.2 Shell Configuration

Before the Laser Shell can be used, it must be configured. The configuration file shipped with the Laser C package will suffice for most installations, although other configurations are possible. As mentioned above, the Laser Shell attempts to load a configuration file called "**LASER.CFG**". When attempting to load this file, the Shell looks in two places, loading the first "**LASER.CFG**" found. First in the same folder that "**LASER.PRG**" is in, and then in the "**MEGAMAX**" folder on the same drive as "**LASER.PRG**".

4.2.1 Tools

Tools are programs which are used for development, such as the compiler, the linker, etc. Tool configuration serves three purposes; showing the Shell the names and locations (path names) of its tools, telling the Shell which of the located tools are to be made RAM resident, and specifying how the file I/O of each tool is handled by the disk cache.

The Shell has the ability to preload certain programs and keep them RAM resident until they are run. RAM resident programs are executed directly, with no time wasted loading the program from disk and no memory wasted keeping a duplicate on a RAM disk. This mechanism is ideal for development system programs, such as the compiler and linker, which are typically run many times during the course of program development. Only specially produced programs may be made RAM resident:

AR.TTP	Archiver
CC.TTP	Compile and link utility
CCOM.TTP	C compiler
DIS.TTP	Disassembler
EGREP.TTP	Multiple file regular expression search
LD.TTP	Linker
MAKE.TTP	Make utility
NM.TTP	Symbol table dumper
RCP.TTP	Resource construction program
LS.TTP	All disk utilities mentioned in the chapter Disk Utilities may be made RAM resident.

The **Tool Locate** ... command displays the dialog with which tools are configured.

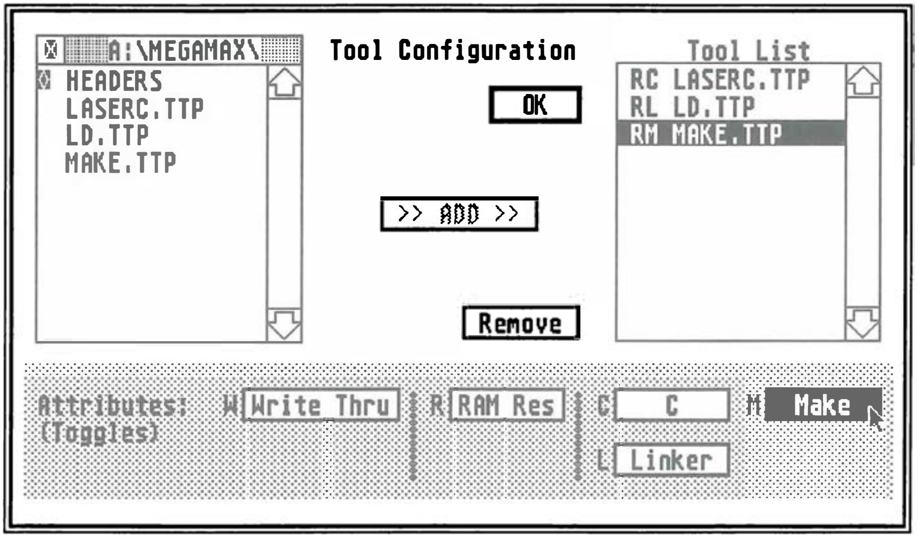


Figure 4.1: Tool Configuration Dialog

There are two selector boxes and some buttons on this dialog. The left selector box is a file selector box and the right selector box (Tool List) is for located tools.

The procedure for performing tool configuration is:

- Using the left selector box, locate the program for each of the tools which is to be used. Multiple selections are allowed in both selector boxes. As a program is located, it should be selected and then added to the "Tool List" by clicking on the ">> ADD >>" button. A minimum tool list should contain a compiler, and a linker. For the Shell's **Make** menu to function, the Make utility should also be added. Other programs which are added will have their names appended to the **Execute** menu. A name may be removed from the tool list by choosing it and clicking on the "Remove" button.

- For each of the predefined tools located, the C compiler, the linker, and the Make utility, select the name in the tool list and click on the corresponding button below the tool list.

- Next, make some or all of the tools RAM resident by selecting the program name in the tool list and clicking on the "RAM Res" button. If using an Atari with 512K bytes of RAM, make only the compiler and linker RAM resident. If using a one megabyte computer, make all three tools RAM resident. Other Laser programs may be made RAM resident if desired (see the list given above). Only the **Flush Resident Progs.** command under the **Options** menu can remove currently resident programs from RAM.

- The "Write Thru" button affects the way a program which is run from the Shell uses the disk cache. A program which has the write through attribute set will write both to the cache and to the disk, where normally output is written only to the cache. The cache is written to disk only when necessary (see section 4.2.4 for more details). To safeguard against possible loss of data, give tools such as the Resource Construction Program and the archiver the write-through attribute.
- Press the "OK" button to close the dialog with its present state. Any RAM resident programs will be loaded at this time.

4.2.2 Environment Variables

Environment variables allow a more general configuration mechanism, and also allow information to be supplied in the Shell and accessed by programs which are run from the Shell. An environment variable is a name to which is assigned a string value. For example, "computer=Atari 520ST" assigns "Atari 520ST" to the environment variable named "computer". The C library function *getenv()* returns the value portion of a named environment variable to a user program.

The Shell and some other development system programs use the following environment variables:

CC	The path name (name and location) of the CC (compile and link) utility. This variable is used by Make to define the $(CC) macro.
CCOM	The path name (name and location) of the C compiler. Used by the Shell to run the compiler. This variable is set by the **Tool Locate** command and does not need to be set manually.
LINKER	The path name of the linker. Used by the Shell to run the linker. Set by the **Tool Locate** command.
MAKE	The path name of the Make utility. Used by the Shell to run Make. Set by the **Tool Locate** command.
CINCLUDE	The location of the folder which contains the C header files. Used by the Shell and the CC utility to pass "-I" options to the compiler.

CINIT The path name of the C initialization code. Used by the Shell and the CC utility pass the name of the initialization code to the linker.

CLIB The path name of the C library. Used by the Shell and the CC utility pass the name of the C library to the linker.

LIBPATH A comma separated list of folders. Used by the linker to find libraries specified as "-L". These folders are searched in order for the library.

PATH A comma separated list of folders. These folders are searched in order by the Shell when executing a program command line style (see section 4.4). A "." may be included in the path to indicate the current directory.

Environment variables may be edited by selecting **Environment Vars** ... from the **Options** menu. A dialog with a single large selector box, some buttons, and a text entry box will appear.

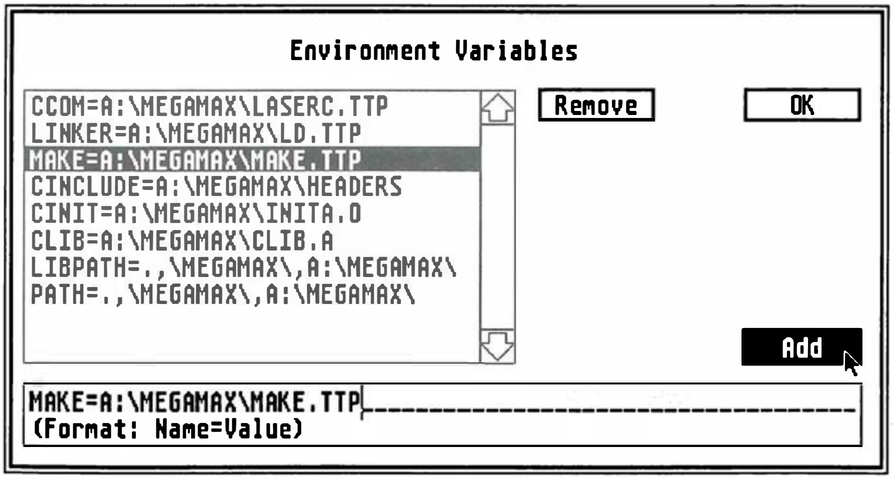


Figure 4.2: Environment Variable Dialog

To add a new environment variable, type it in. Typing appears in the long box at the bottom of the dialog. The format is "name=value". Press the "Return" key or click on the "ADD" button to copy the typed line into the selector box. To edit an existing line, select it with the mouse. It will be copied into the typing line. The "Esc" key will clear the typing line, and the left and

4

right arrow keys move the insertion point within the line. Remember to press the "Return" key or click on the "ADD" button to save any changes. The "Remove" button deletes the line selected in the selector box. Click on "OK" to close the dialog.

If the Shell cannot find its configuration file, these variables will not be set. The distribution disk on which "**LASER.PRG**" is located contains a configuration file with all the variables set to typical values (see the beginning of this section).

4.2.3 Save Configuration

The **Save Configuration** item under the **Options** menu allows the editor settings of the top window, the tool list, and the environment variables to be saved. A file save dialog is issued to allow a choice of file name and file location. The current configuration should always be saved after any changes are made. The **Read Configuration** ... option allows any configuration file to be read. When chosen, file selector dialog is presented. Only folders and "**.CFG**" files are shown. Select the desired configuration file and click on "Open" to resore configuration settings.

4.2.4 Disk Cache

The Laser Shell has a built-in dynamic disk cache which buffers all disk access. Disk files are organized as a series of blocks which contain the actual file data. The disk cache can dramatically decrease file access time by keeping in memory a copy of last read or written blocks, so that subsequent reads are from memory, rather than disk. The Shell's cache is dynamic in that its size will change to fit available RAM. As programs request memory, the cache will flush enough blocks to the disk to accommodate the request. Any program which has the write-through attribute set will write to disk and cache, while programs which are not write-through will write *only* to the cache. All read operations are from the cache, provided the requested blocks are in-cache. The disk image of the file will only be updated to match the cache when the cache is flushed. Automatic flushing of cache blocks is done whenever memory is needed, either by the Shell or a program run from the Shell, or when file I/O overfills the cache with fresh blocks. The blocks are flushed in order of least-recently-used. Quitting the Shell will flush the entire cache.

In addition to automatic flushing, selective flushing may be done through the "Cache Management" dialog, selected from the **Options** menu.

The dialog contains a selector box showing which files have blocks in the

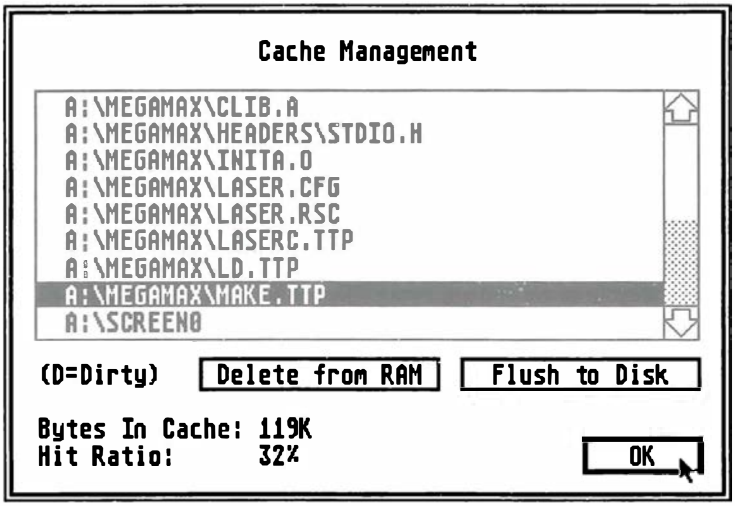


Figure 4.3: Cache Management Dialog

cache. A letter "D" precedes those files which have changed in the cache but have not been updated on the disk. The "Bytes In Cache" item shows how much of the disk is in RAM, and the "Hit Ratio" item shows the percentage cache reads to disk reads. When one or more items in the selector box are selected, the two buttons directly below become active. The "Delete from RAM" button removes the selected file's blocks from RAM without saving them to disk. The "Flush to Disk" button causes the selected blocks to be written to disk.

The **Flush Resident Progs.** command removes all RAM resident programs and deallocates their memory. This command is useful to force a RAM resident program to be reloaded from disk, should it change after being made RAM resident.

4.3 Text Editor

The Shell's editor provides a simple yet powerful means of creating program source or any other text-only files. Editing is performed in up to four individually sized and positioned windows. The mouse is used to position the insertion point and select text for removal or duplication. Most editing functions can be selected from the menu, and many can be initiated by keyboard command as well. The editor also includes some special features to assist with C programming.

4

4.3.1 Editor Menu Usage

Menu commands are used to invoke all editor functions except text selection, scrolling, and the Help function (activated by the **Help** key on the keyboard). If a menu selection is preceded by a single separate character, then that command can also be invoked by pressing that character on the keyboard while holding down the "Control" key. The command operates no differently when selected from the keyboard. For example, a new file may be created (the **New** menu command) by holding down the "Control" key while striking the "N" key.

The current editing situation dictates which editing commands are available. For example, if no files are currently open, none of the **Save** commands of the file menu will be available. Those commands which are available are printed in normal black-on-white lettering. Those commands which are unavailable are printed in gray-on-white lettering. Clicking the mouse on an unavailable command has no effect, other than to cause the selected window to disappear.

The **Info** menu contains some items which are actually information about the current file. Although these items are printed in black-on-white lettering, they can never be selected.

4.3.2 Choosing A File

The File menu contains all operations related to opening existing files, creating new files, and saving work.

Open ... allows a file residing on disk to be loaded into the editor for inspection and modification. When chosen, a dialog which contains a file selector box and some buttons will appear. The file selector box behaves in the usual manner (see Introduction, Conventions). Clicking on the "Open" button causes the editor to open the selected file. Double-clicking on a file name will also open it. The "Cancel" button should be used if the **Open** ... command was issued accidentally. The group of buttons like "*.C" are filters. Clicking on a filter causes the currently displayed list to be filtered and then redisplayed, leaving only those names which match the filter. The "*.*" filter displays all files while the "*.C" filter displays only those files which end with ".C". As a file is opened, a window displaying the file will appear.

New is used when a new file is to be created. An empty window will appear on the screen. The initial title of the window will be "Untitled".

Save causes the contents of the top window to be saved on disk. The name displayed in the title bar of the window is the name of the file which will be saved. If an attempt is made to save a file whose name is "Untitled", the **Save as** ... command will be invoked (see below).

Save as ... is used whenever the contents of a window must be saved to a file whose name is different from the the name in the window title. A dialog which looks much like the one used by the **Open** ... command is presented. The file selector box behaves similarly, with the exception that none of the file names listed are selectable. There is also a typing line into which the new file name may be entered. Pressing the "Esc" key erases the entire typing line. After the file is saved, the name of the window will be changed to the new file name.

Close causes a window to disappear. If changes were made to the window's contents since last saved, an opportunity is given to save the file. The **Close** command can also be activated by clicking on the graphic character in the upper left-hand corner of a window.

Close all performs the **Close** command on all open windows, except for the "STDIO" window.

Revert causes the most recently saved version of the current file to be reloaded. If the file has not yet been saved, the version of the file originally loaded will be used.

4.3.3 Mouse Usage

The mouse is used to manipulate windows, to position the insertion point, and to select text within a window. The position of the mouse on the screen is indicated by the mouse cursor, which is either an arrow, a vertical bar "text" cursor, or a busy bee. When the mouse is over the work area of the top window, the cursor always a "text" cursor.

The left-hand button of the mouse is the only button which has any effect. There are four basic operations available with the mouse button:

Click If the cursor is within the top window, the insertion point is placed on the character nearest the location of the mouse. If the cursor is over an element of a scroll bar, the window is scrolled appropriately. If the cursor is over another window, then that window is made the top window.

Drag The drag can be used to select a range of text. If the initial click is within the top window, text will be selected as the mouse is dragged over it.

The drag can also be used to move and resize the window. If the Drag starts on the gray bar atop the window, it can be dragged to any desired position. If the drag starts within the small box in the lower right corner (the "sizebox"), the lower right corner can be dragged to change the size of the window.

The white areas of the scroll bars can be dragged to reposition the window over the file. For example, the thumb of the vertical scroll bar can be dragged to the very bottom to move the window to the very end of the file.

Double-click If the mouse is initially over an alphabetic or numeric character in the current window, all alphabetic and numeric characters of the "word" surrounding that character are added to the current selection. If the mouse is initially over an open or close parenthesis (one of "(", "{", "[", ")", "}", "]"), all text up to the matching close or open parenthesis (if it exists) is selected.

Shift-click By shift-clicking, text is selected as if a drag had been done from the old insertion point to the cursor position. As an example, all text in a file can be selected by clicking first at the very top line, then dragging the vertical scroll bar thumb all the way to the bottom, and then shift-clicking on the last character of the file.

4.3.4 Scrolling the Window

Scroll bars run along the right side and along the bottom of each window. The vertical scroll bar is used to position the window over lines of text, with the thumb size representing the size of the window relative to the number of lines of text in the window. The horizontal scroll bar always positions the window over columns 1-255, regardless of the actual number of columns of text.

The window may also be scrolled up or down one line at a time, by holding down the "Control" key while pressing the up or down arrows.

4.3.5 Insertion Point Keys

In addition to using the mouse, the insertion point can be repositioned from the keyboard. The four arrow keys move the insertion point in the indicated direction. Shifted left and right arrow keys move the insertion point by words. Control left and right arrows move the insertion point to the beginning and end (respectively) of the current line. The "Home" key positions the insertion point on the first character in the file.

4.3.6 Text Entry

An open window may have text inserted by selecting an insertion point with either the mouse or the arrow keys, and then typing characters on the keyboard. Typing mistakes may be corrected by pressing the "Backspace" key to erase the character to the left of the insertion point, or the "Delete" key to erase the character to the right of the insertion point. "Control-Delete" erases the entire current line and the "Clr" key (shifted "Home" key) erases from the current insertion point to the end of the current line.

The "Return" key inserts a line break into the file at the current insertion point. All text on the current line to the right of the insertion point is redrawn on the next line of the window. If the insertion point was at the end of the line when the "Return" key was struck, the new line will be blank. The "Insert" key inserts a blank line after the current line and the shifted "Insert" key inserts a blank line before the current line.

The "Tab" key inserts a single character into the file, but prints as a field of white space (unless the **Visible tabs** option has been selected). The amount of space allotted a tab depends on the column in which the tab was inserted, and always moves to a column number evenly divisible by the current "tabsize". The tabsize is four spaces when a window is first opened. The tabsize can be changed with the **Options** menu.

If a substantial number of characters must be changed or removed, the text should be selected with by dragging. Once the text has been selected, replacement text can be typed over it, or it may be removed by a "Backspace" or "Delete".

4.3.7 Block Operations

The **Edit** menu contains commands for moving text from one place to another within a window and between windows, reversing mistaken changes, and shifting text horizontally within the window.

Cut copies the current selection to the scrap (a cut/paste buffer), and then erases the selection. Note that **Cut** is different from a "Backspace" or "Delete" where the selected text is removed from the window but is *not* copied into the scrap.

Copy copies the current selection to the scrap but does not remove the selection. If the shift key is held down when a **Cut** or **Copy** command is issued, the selected text will be appended to the scrap, rather than replacing the old buffer contents.

4

Paste adds the contents of the scrap to the current window at the current insertion point. The effect is exactly the same as manually typing the contents of the scrap. Note that if a range of text is selected, the contents of the scrap replace the selection; the selection is deleted just as if the text had been typed over it.

Erase is functionally identical to a "Backspace" or "Delete", except that it works only when text is selected. The selection erased is *not* copied to the scrap.

Undo reverses the effect of the last operation which changed the file or the scrap. The **Undo** command reverses one or more instances of the same type of operation. For example, typing several characters and choosing **Undo** will undo the effect of all characters typed. A new undo sequence is started each time a new type of operation is performed. Types of operations which constitute undo groups are:

- Typing including "Backspace", "Delete", and "Return".
- Blank line insertions (both before and after the current line).
- Clear to end-of-line.
- Line deletions.
- Left and right shift operations on one selection.
- Any **Cut**, **Copy**, **Paste**, **Erase**, **Revert**, or **Undo**.
- Any **Find & change**.

Note that, while repositioning the insertion point does terminate the last operation performed, it *does not* prevent the last changes from being undone.

The **Undo** command may also be issued by pressing the "Undo" key on the keyboard.

Shift left is used to realign text horizontally. The shift affects all lines within the confines of the current selection. If the beginning of the selection is at the very left edge of the file, each entire line is shifted so that the first non-blank character is positioned one tabstop left. If the beginning of the selection is somewhere within a line, then only the characters to the right of that column on each line are shifted left one tabstop. The **Shift right** command shifts selected text to the right instead of to the left.

4.3.8 Editor Options

Under the **Options** menu are commands for modifying the editor environment. These options may be either set (a check mark appears before those options which are set) or not set, and apply only to the current window. Each window remembers its own editor settings.

Tabsize ... is used to set the width of the tab character. When selected, the **Tabsize** ... command issues a dialog box. The new tabsize can be typed in the space provided, and the window will be redrawn reflecting the new size. Note that the tabsize value has no effect upon the actual size of a file; each tab is always just one character.

Autoindent sets a flag indicating that whenever a line break is added to the file, space and tab characters from the previous line, up to the first non-blank character, will be copied to the new line. This feature is very useful when creating indented C source files.

Autosave sets a flag indicating that the file should be saved automatically at an assigned frequency and when a program is run. A dialog is presented in which the interval between saves may be assigned. Clicking on the "off" button removes the **Autosave** feature for the current file. Clicking on the "Make Backup" button allows a backup copy of the file to be saved, in case, after a few autosaves, it is decided that the original version was actually correct. **Autosave** is useful for those whose power supply is unreliable and to ensure that no source changes are lost in the event that a program run from the Shell crashes.

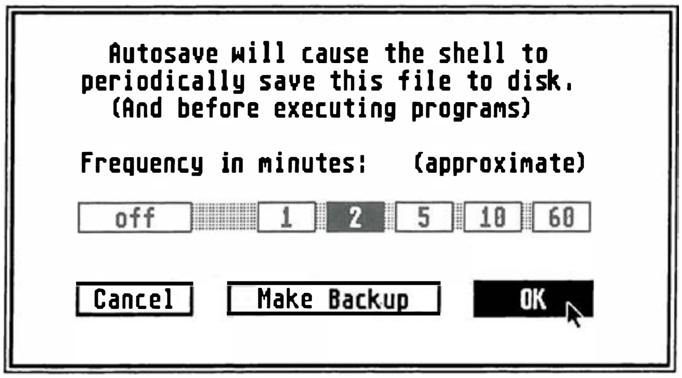


Figure 4.4: Autosave Dialog

4

Note that after a **Save as ...** command, the **Autosave** feature will thereafter save the file to the new file name.

Visible tabs causes the tabsize to be set to one space, and tabs to print as "diamond-in-a-square" characters. **Visible tabs** is sometimes useful when examining files created by some means such that tabs are mixed heavily with spaces. The tabsize cannot be changed while tabs are visible.

Ignore case sets a flag indicating that all searches should be done without regard to the case of alphabetic characters. The **Ignore case** flag also affects the way **Find & Change** works; for more information, see the section which describes searching.

No Undo disables the **Undo** command in the **Edit** menu. Invoking this option will deallocate the undo buffer, which is as large as the file being edited, thus leaving more free memory for programs. This option is automatically set if there is not enough free RAM to accommodate the undo buffer. If the configuration file "**LASER.CFG**" is not found, all flags except **No Undo** are off, and tabs are set to four spaces.

4.3.9 Finding Text

The **Search** menu contains commands used to find and change strings in a file.

Find ... is the basic entry to the searching mechanism. When issued, the **Find ...** command puts a dialog box on the screen (see figure 4.5 below).

In the dialog box, there are two boxes into which strings may be typed. The top box is for the "target" string, which the editor will attempt to locate in the file when one of the **Find** commands is given. The bottom box is for the "replacement" string, which the editor will substitute for an occurrence of the target string if one of the **Find & Change** commands is given. A typing box is chosen by clicking on the desired box, or by pressing the up or down arrow keys. The "Esc" key clears the entire line and the right and left arrow keys can be used to position the insertion point within a line. The search and replace function can be used to delete occurrences of the "target" by leaving the "replacement" box empty.

There are five sub-commands available from the "Find" dialog box:

Forward The editor will search for the target string in the file, starting with the character after the current selection or the insertion

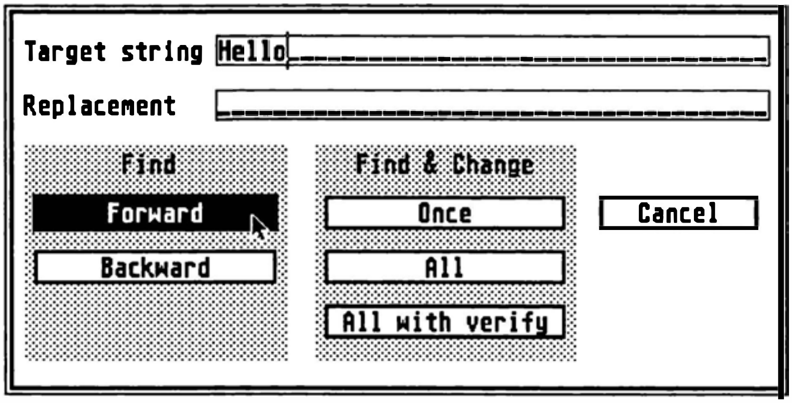


Figure 4.5: Search Dialog

point and on towards the end of the file. If an occurrence of the target string is found, the insertion point position becomes the first character of that occurrence. If no occurrence is found, the current selection or insertion point is not changed.

Backward The editor will search for the target string beginning with the first character of the current selection or the character under the insertion point (unless the target contains only one character, in which case the search begins one character before the current selection) and on towards the beginning of the file. If an occurrence of the target string is found, the insertion point becomes the first character of that occurrence. If no occurrence is found, the current selection or insertion point is not changed.

Once The editor will search for the target string in the file, starting with the character after the current selection or insertion point and on towards the end of the file. If an occurrence of the target string is found, it is replaced by the replacement string. See below for a description of replacement behavior when the **Ignore case** flag is set.

All The above procedure is executed repeatedly until no more occurrences of the target string are found. The entire sequence of changes is considered a single undoable event.

All with verify Similar to **All**, except that each change must be verified before it is made. Each occurrence of the target string is highlighted. If the "Y" key is pressed, the change is made. If the "N" key is pressed, the change is not made. The process repeats until no more occurrences of the target string are found, or until the "Q" key is pressed in response to a verification. The entire sequence of changes is considered a single undoable event.

When the **Ignore case** flag is set, the case (upper or lower) of alphabetic characters is ignored when searching. Therefore, the target "abc" will match "ABC" and "aBc" as well as "abc".

Find next causes the editor to search for the next occurrence of the most recent target string. The search is performed in the same direction as the most recent search.

Change next performs a **Find & Change** once using the most recent target and replacement strings.

Goto line ... is used to move the current insertion point to any line of the file. When issued, the **Goto line** ... command creates a dialog box into which the desired line number is typed. The insertion point is set to the first character on the given line.

4.3.10 Text Marks

The current line of the current window can be marked by holding down the "Shift" key and pressing any function key from "F3" to "F10". Then, while editing on any other line in any other window, pressing the same function key (not shifted) will reposition the insertion point at the beginning of the marked line. Two special marks, "F1" and "F2" are predefined to reposition the insertion point one page up or down, respectively, in the current window.

4.3.11 Rearranging Windows

The **Windows** menu contains commands for arranging currently open windows.

Overlap arranges all open windows so that they overlap one another. The size of each window will be nearly that of the screen. A small part of each inactive window is left visible.

Side-by-side arranges all open windows so that they form columns which fill the screen.

Over/Under causes all open windows to be stacked vertically on the screen.

Show Stdio causes the special "STDIO" window to appear. This window is like any editor window, except that tools, such as the compiler, print messages directly into the window. (see section 4.2.1). If the window is currently visible, the menu will read **Hide Stdio**. Note that the contents of the "STDIO" window may not be saved.

4.3.12 File Information

The **Info** menu contains general information concerning the top window. It also contains a handy reference chart of **C operators**. The operators are listed top-to-bottom in order of precedence.

The information presented in the **Info** menu concerns the size of the current file and the current position in the file.

4.4 Running Programs

Programs may be run from the Shell by one of two methods; either by making a choice from the **Execute** menu, or by typing a command line (a program name followed by program parameters) into a window.

4.4.1 Menu Command Execution

The **Execute** menu has commands for running the compiler, the linker, performing an automatic compile, link, and run, and for running any other program. In addition, the names of user defined tools (see section 4.2) will appear in the **Execute** menu, and may be run by choosing them.

The **Compile** ... and **Link** ... options will not be selectable if they have not been located (see section 4.2). The **Run** item will only be selectable if both the compiler and linker have been located, and the top editor window contains a "**.C**" file.

Compile ... brings up a file selector box like the one used to open a file for editing. This box however will only display files which have an extension of ".C". One or more files may be selected to compile. The usual file selector usage applies. Clicking on the "OK" button runs the C compiler on the selected source files. "CANCEL" aborts the compile. Compiler messages, including syntax errors, are sent to the "STDIO" window, which will automatically appear if hidden.

Pressing "Control-K" on the keyboard will run the compiler on the current window, provided it is named with the "**.C**" extension.

4

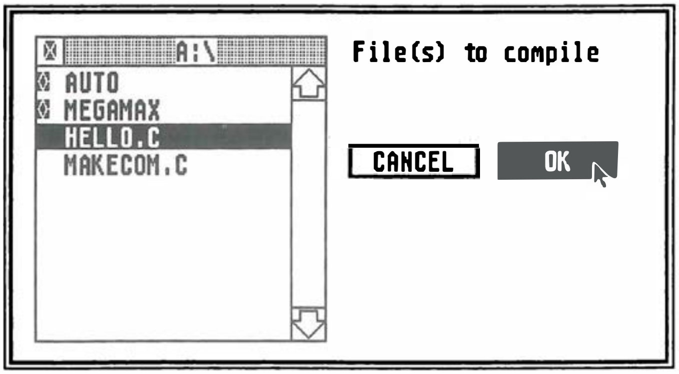


Figure 4.6: Compile Dialog

Link ... invokes the linker dialog, which contains two selector boxes, some buttons, and a typing entry box. The left file selector box is used to locate object files. As each file is located, it should be added to the "To be Linked" selector box by selecting it with the mouse and then pressing the "`>> ADD >>`" button. Inadvertantly added names may be removed by selecting them and pressing the "Remove" button. The name of the linker output file may be changed from the default "**A.PRG**" by pressing the "Esc" key then retyping the new name. Executable programs should have extensions of "**.PRG**", "**.TOS**", "**.TTP**", or "**.ACC**" (see section 2.4.3). Desk accessories are GEM applications which are named with the "**.ACC**" extension. When the computer is started, any "**.ACC**" files which appear on the root are loaded into RAM and started. A desk accessory should call `evnt_multi()` to share processor time with the main application and other desk accessories.

The "Save ..." button allows the list of "To be Linked" files to be saved to disk, so that the next time the Shell is used to link the current program, the component object files will not need to be re-added. The default saved file will be named after the executable, but with an extension of "**.LNK**". The name of the file may be changed, but the "**.LNK**" extension is required. To restore the "To be Linked" list from a "**.LNK**" file, locate and select the file in the left selector box. The "Save ..." button will change to "Load", which when clicked on will perform the restore.

Run checks the dates of the "**.O**" files and the coresponding "**.C**" files used in the last **Link** dialog, and compiles any "**.C**" files which have been changed since their "**.O**" files were produced. Then all "**.O**" files are

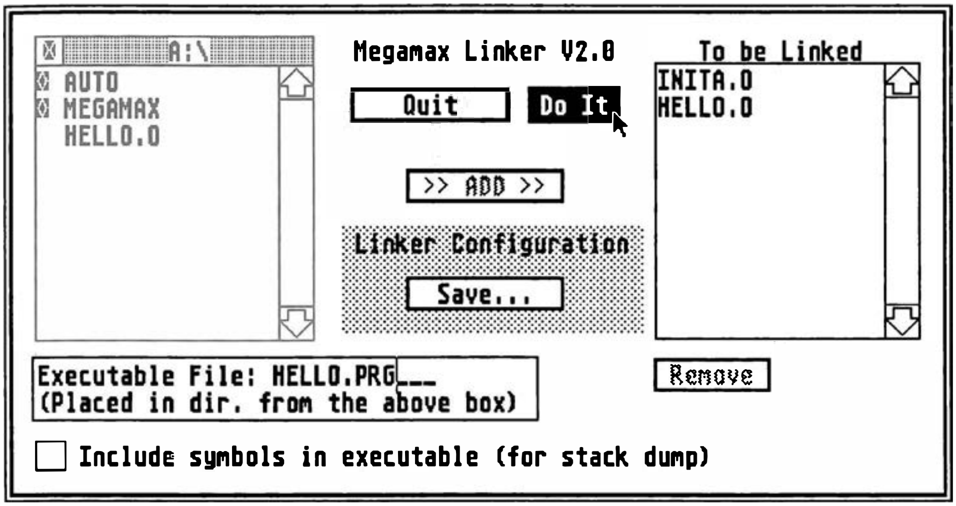


Figure 4.7: Linker Dialog

linked, using the last specified executable name. Finally, the executable program is run. If any compiler or linker errors occur, the run is aborted. If no previous link was performed and the file name in the top window has the "**.C**" extension, the above procedure is applied only to that file. The executable file, in this case, will be the default "**A.PRG**". If the name of the top "**.C**" file was not represented in the last link dialog, the contents of the last link are erased.

4.4.2 Command Line Execution

The alternate method of running programs is similar to that of a command line shell, except that commands need not be retyped. A command line may be typed into any window and executed by pressing the "Enter" key while the entire line is selected. If the command to be executed is in the "STDIO" window, the insertion point may simply be positioned at the end of the command line before pressing the "Enter" key. Multiple commands may be executed by selecting several lines, thus creating a simple batch mechanism. A file of commonly used commands may be written to disk, thus, saving the trouble of retyping them each time the Shell is started. In place of the "Enter" key, "Control Return" may instead typed. Note that while the contents of the "STDIO" window may never be saved, part or all of these contents may be copied to another window which may then be saved.

The Shell supports some built-in commands which may only be executed via command line:

cd Change current working directory (folder). As a convenience, files in the current folder need not be specified with a full path name. A file name with no path is assumed to be in the current directory.

pwd Print the current working directory.

pushd *path*|-*n* Push the named directory onto the directory stack. The Shell maintains a stack of directories, so that when constantly switching among several working directories, the directory names need not be retyped. To add a directory to the stack, type pushd *path*, where *path* is the directory name. The current directory is always the top of the stack. To change to a directory in the stack, type pushd -*n*, where n is the ordinal number in the stack (from the top, starting with zero) of the desired directory. The current directory is swapped with the desired directory in the stack, and the desired directory becomes current. The -*n* option may be omitted to mean pushd -1.

popd Pop the top of the directory stack, changing the current working directory to that of the next item in the stack.

dirs Print the directory stack. The top of the stack is the left-most item.

tos *line* By default, executing a "**.TOS**" program from a command line causes the program's output to go to the "STDIO" window. The tos command followed by a command line will cause a "**.TOS**" program to use the entire screen, as it would if run from the GEM desktop.

rehash Rehash the search path table. It is often desired that programs other than those in the current directory be executed. To save the inconvenience of typing in a full path name, the Shell searches for the program name using the search path given in the environment variable "PATH" (see section 4.2.2). To avoid having to search the disk each time the search path is used, it is searched once, saving the names of all programs in the path. If a program is added to a folder in the search path (other than the current folder), it will not be found until rehash is done.

See the chapter Disk Utilities for the names and usages of the disk-based command line utilities.

4.5 Disk Operations

Files may be copied, moved, deleted, or renamed from the Laser Shell by choosing **Disk ops** ... from the **File Menu**. The "Disk Operations" dialog contains two file selector boxes and some buttons.

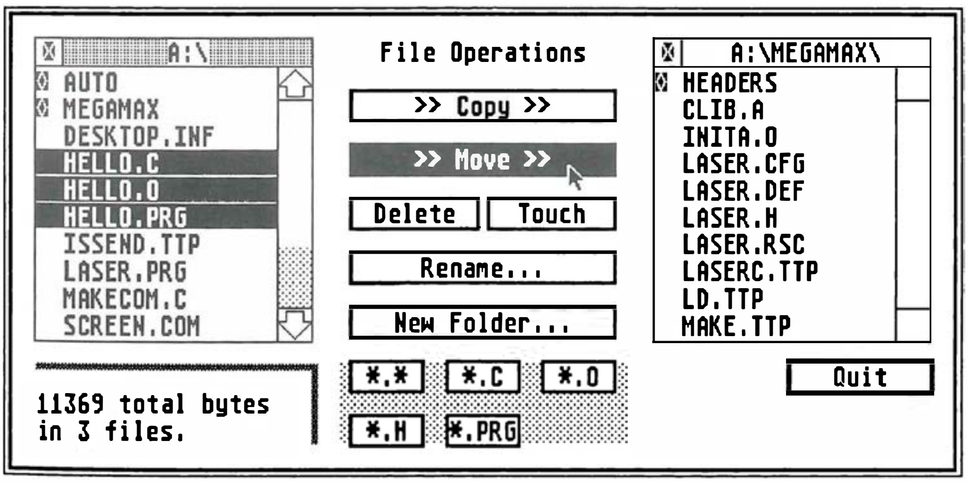


Figure 4.8: File Operations Dialog

Using the "Disk Operations" dialog involves basically two steps; selecting desired folders and/or file names, then clicking on a button to perform the desired file operation. Each of the two file selector boxes may be manipulated independently. Only one file selector box is active at a time, indicated by the grey stripes in the bar at the top. An inactive selector box is activated by clicking the mouse anywhere inside the box. The filter buttons filter the contents of the currently active selector box, exactly as in the file open dialog. The lists may be scrolled with the scroll bars if the list of file names is longer than the space allowed by the selector box. Multiple selections are allowed in both selector boxes.

Once folders and/or files are located, operations may be performed. A file name must be selected for the buttons to become active. The following operations are supported:

Copy Copy the file or files selected in one box to the directory located in the other box. The “>>” or “<<” characters indicate the direction of the copy.

Move Move the file or files selected in one box to the directory located in the other box. The “>>” or “<<” characters indicate the direction of the move.

Delete Delete the selected file or files from the disk.

Touch Touch updates the modification time of the selected file or files. This is useful to force recompilation of a program (see section 4.6 below).

Rename Rename the selected file or files on the disk. The rename will fail if there is an attempt to rename a file to an existing name.

New Folder Allows creation of a new folder. A prompt is supplied to get the name.

Click on “Quit” or pressing the “Return” key to close the dialog box.

4.6 Project Management (Make)

The Laser Shell has an advanced project management facility which uses the separate program Make (see chapter 10),compatible with the UNIX Make utility. Using Make involves creating a text file, called the “Makefile”, which specifies what files are used to create a program (the target), and how those files are converted into that program (the transformation rules). Make checks the modification times of the specified files and performs the transformations necessary to update the target program. The detailed discussion of the Make program should be read before attempting to use the Shell's **Make** command.

The steps involved in using Make from the Shell are:

- Assure that the Make program has been located (see section 4.2). If not located, the **Set Make** ... menu command will be disabled.

- Create, using the editor, the "Makefile". See Make for information on what this file should contain.

- From the **Make** menu, select **Set Make** .... A file selector box is displayed. Locate and select the "Makefile" and click on "OK". The "Makefile" will be processed by Make generating a list of targets. Any syntax errors are reported to the "STDIO" window. If no syntax errors are found, the list of targets is inserted into the **Make** menu.

- Selecting a target reads the "Makefile" and updates the selected target.

4.7 Debugging

Should a user program cause a processor exception (a crash) during execution, a dialog is presented. The dialog explains the type of crash that occurred and allows one of three choices to be made. Clicking on the "Reboot" button resets the machine. Clicking on the "Shell" button cleans up GEM and returns to the Shell. Clicking on the "Dump" button also returns to the Shell, but upon return, a stack dump is performed. The stack dump lists the names of functions in their activation order, from the currently active function down to the first *main()* call. If the program was ***not*** linked with the "Include symbols ..." option, "Unknown" is printed in place of function names. Beside each name, an offset into the code from the function start is printed.

The Shell will ***usually*** be intact when a user program crashes. Certain processor exceptions are more likely to have been caused by a program error which corrupted GEM or some other memory. The dialog will have a message indicating "Probably should reboot" when; a GEM checksum error occurs while returning from a user program, a processor exception occurs in the Shell, or the disk cache has been corrupted. The message "Probably safe to go to the shell" will appear otherwise.

If a program is caught in an infinite loop, it may by stopped by typing "Control-Delete". When typed, a dialog similar to the processor exception dialog is will be presented, except that the "Reboot" button is replaced by "Continue". Clicking on the "Continue" button will close the dialog and resume program execution.

If an external debugger is installed (usually when the computer is started), it may be used from the Shell by choosing **External Debugger** from the

Options menu. When chosen, the external debugger will be invoked as a result of a processor exception, rather than the dialog box discussed above.

The following program will cause a processor exception:

```
/*  Example of a divide-by-zero processor exception
*/
main()
{
    int a = 1;

    a = divide(a, 0);
}

/*  Divide m by n
*/
int divide(m, n)
    int m, n;
{
    return m/n;
}
```

When the above program is compiled, linked (using the "Include symbols ..." option), and then run, it will crash (see figure 4.9 below). Clicking on "Dump" will terminate the program and print the following information into the "STDIO" window.

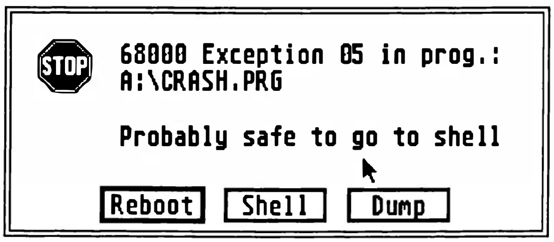


Figure 4.9: Processor Exception Dialog

```
_divide + 0xE
  _main + 0x16
 __main + 0x70
```

The underscores are added to each name by the compiler. The name `__main` is name of the initialization code and the entry point into the program.

4

4.8 Menu Summary

The following summary lists menu items in their actual order in the menu bar and their function.

Fuji	
About ...	Shell version and copyright information.
File	
Open	Open an existing text file.
New	Create a new text file.
Disk Ops... ...	Open the Disk operations dialog.
Save	Save the file which is in the front editor window.
Save as ...	Save under a new name (front window).
Close	Close the front window.
Close all	Close all open file windows.
Revert	Revert to the last saved changes (front window).
Quit	Quit the Laser Shell.
Edit	
Cut	Cut the current text selection to the scrap.
Copy	Copy the current text selection the scrap.
Paste	Paste the contents of the scrap at the insertion point.
Erase	Erase the current selection without affecting the scrap.
Undo	Undo the last editing change.
shift left	Shift the currently selected line left by one tab.
shift right	Shift the currently selected line right by one tab.
Execute	
Compile	Run the C compiler.
Link	Run the Linker.
Run	Compile, link, and run. Based on the last link performed.
Other ...	Run any program.
Make	
"Makefile name"	Currently set "Makefile".
Set Make	Set the name and targets for the current "Makefile".

4

Options	
Tabsize	Set tab width in spaces.
Autoindent	Toggle autoindent.
Autosave	Engage or disengage the autosave feature.
Visible tabs	Make tabs visible or invisible.
Ignore case	Ignore case when searching.
No Undo	Disable the Undo command and free the undo buffer.
Tool Locate ...	Configure compiler, linker, etc.
Environment Vars ...	Set environment variables dialog.
External Debugger	Allow an external debugger to be used from the Shell.
Cache Management ...	RAM cache operations dialog.
Flush Resident Progs.	Flush dirty RAM cache blocks to disk.
Save Configuration	Save the current editor settings, environment.
Search	
Find ...	Search for text dialog.
Find Next	Find last searched text again.
Change next	Change to last replacement text.
Goto line ...	Go to a specific line number.
Windows	
Overlap	Overlap all windows.
Side-by-side	Arrange windows as columnar "tiles."
Over/Under	Arrange windows as horizontal "tiles."
Show/Hide Stdio	Show or hide the special standard I/O window.
Info	
C Operators	Show C operator precedence.

4.9 Keyboard Summary

A list of keyboard commands is presented below:

Backspace	Erase the character left of the insertion point.
Delete	Erase the character right of the insertion point.
Insert	Insert a blank line after the current line.
Shift-Insert	Insert a blank line before the current line.

Control-Delete	Delete the current line.
Up Arrow	Move the cursor up one line.
Down Arrow	Move the cursor down one line.
Left Arrow	Move the insertion point left one character.
Right Arrow	Move the insertion point right one character.
Shift-Up arrow	Scroll up one line.
Shift-Down Arrow	Scroll down one line.
Shift-Left Arrow	Move the insertion point left one word.
Shift-Right Arrow	Move the insertion point right one word.
Control-Right Arrow	Move the insertion point to the end of the current line.
Control-Left Arrow	Move the insertion point to the start of the current line.
Help	Display keyboard help.
Undo	Undo the last change.
Clr	Clear from the insertion point to the end of line.
Home	Move the insertion point to line one, column one.
F1	Move the cursor up one page.
F2	Move the cursor down one page.
Shift-F3-F10	Mark a line in a window.
F3-F10	Position the insertion point on a set mark.

4

5

Chapter 5

C Compiler

Introduction

The C compiler is a fast single pass compiler generating absolute MC68000 machine code. The compiler reads a single C source file and outputs a relocatable object code file (see File Formats, section A.1). The source file must have a file name extension of ".C", and the resultant object file will have the extension ".O".

The compiler will place the object file on the same disk as the source file. For this reason, the programmer should ensure that sufficient space is available on the disk for both the source and the object.

5.1 Command Line Usage

While the compiler may be run from the GEM desktop, it will normally be run from the Laser Shell. The command line syntax is:

```
ccom.ttp [-Dname[=value]] [-Uname] [-Ipath[,path,...]]
         [-S[a[,b[,c]]]] file.c
```

`-Dname[=value]`	Define name with optional value option. This option adds **name** to the compiler's preprocessor symbol table as if, in the source being compiled, the line **#define** name value had been inserted. The value is optional.
`-Uname`	Undefine a predefined name as **#undef name**.

-Ipath[,path,...] Include path option. This option tells the compiler in which directories (folder(s)) it should look to find include files, as in **#include <stdio.h>**. There can be up to 13 paths given and the compiler will look in each directory in the order given. As a default, the compiler uses only the directory that the source file resides.

-S[a[,b[,c]]] Alter the compiler's choice of switch statement code generated. See section 3.2.3 for a description of this option. The default values are; a = 10, b = 2, and c = 12. Omission of any of these parameters leaves the default.

5.2 Compiler Errors

Error messages generated during compilation are reported to the screen, accompanied by the line of source code containing the error. Error messages are of the form:

```
"file-name", line line-number: error message text
```

I/O redirection may be used to save the error messages to a file if desired.

5.3 Memory Usage

The C compiler dynamically allocates memory on an as-needed basis while compiling a program. The only limit on the size of a program is available RAM, thus extremely large programs may be compiled.

Chapter 6

Linker

Introduction

Separate compilation of program source can greatly decrease development time by eliminating the need to recompile all code necessary to create an executable program. For example, the program:

```
main()
{
    printf( "Hello, world\n" );
}
```

calls the function *printf()*. Without separate compilation, the source code for *printf()*, as well as that of any functions which *printf()* may call, would have to be recompiled each time the function *main()* is compiled. Instead, *printf()* can be compiled into object code. The function *main()* need only reference *printf()* by name (known as an **`external reference`**). The task of combining separately compiled object files into a single executable program is performed by the linker. The linker reads multiple object and archive files (collections of object code), resolves external references, and produces an executable program.

The linker must be used even if a program doesn't contain any external references because an object file created by the compiler is not executable.

6

6.1 Command Line Usage

The linker can be run from GEM desktop, the Laser Shell, or a command line shell. Normally, the linker is run from the Laser Shell.

When the linker is run from a command line shell, the usage syntax is:

```
ld.ttp [-G] [-Lx] [-M] [-V] [-Txxx] [-O output] [object...]
     [library...]
```

`-G`	Global symbol include. The GEMDOS executable file will contain symbol entries for function names. If the program run from the Laser Shell terminates abnormally, a stack dump can be printed for debugging purposes. Static functions begin with tilde (˜) while global functions begin with an underscore (_).
`-V`	Verbose option. The linker will print to the standard error the names of object files as they are included by the linker. This option is useful to see which object files are included from an archive.
`-M`	Map option. Setting this option will cause the linker to print names and addresses of globals which are included in the executable program.
`-Lx`	Library name option. The linker will look in the directories given in the LIBPATH environment variable for a library named libx.a. X is a single character. For example -Lc will search the directories for libc.a
`-Txxx`	Text base option. The text base option causes the linker to adjust references within the program as if the program were at hex memory location xxx. Normally, the program is linked as if it were based at location zero, and relocation information is included so that when a program is run, the references may be adjusted for the actual memory location. Setting this option also prevents this relocation information from being included.
`-O output`	Output file name. The linker's output is named output. Without this option the output is named `a.prg`.
`object`	Object files produced by the compiler.

`library` Library of object files. The Laser library of UNIX functions and Atari ST ROM interface routines is named `libc.a`.

Example:

```
ld.ttp \megamax\init.o myprog.o -Lc
```

Link `myprog.o` with the Laser initialization code and the Laser system library. The final program produced is written to `a.prg`. Note that the inclusion of the initialization code, `init.o`, is required as the first file for any application program.

6.2 Linker Errors

Should an error occur during the link, the link is aborted and no output program is written. Error messages and possible causes are:

Usage: ld.ttp ... Either an invalid link option was specified, or no object or library files were given.

File open error: `name`
The object or library file `name` was not found. Check to see that the file name and path name are given correctly and that the file actually exists.

File read error: `name`
Likely a problem with the disk. Try a newly formatted disk.

File write error: `name`
Either the disk onto which the linker output is being written is full, or there is a physical problem with the disk. Check to see that adequate space for the output is available.

Unable to open output file: `name`
Check to see that the disk is not write protected, and that the path given the output, if any, is correct. (May also be the result of a problem with the disk).

File format error: `name`
The named input file is not a Laser or DRI format object or library file or it has been corrupted. (Assure that only object and library files are specified).

6

Undefined symbol(s):

The linker found references to function name(s) or global variable name(s) for which there is no definition. Make sure that the listed globals are actually defined, and that references to library functions are spelled correctly. Note that a leading underscore (_) is added to each global by the compiler and should be ignored by the user.

Duplicate name definition: `name`

The global `name` has been defined in more than one place. Eliminate or rename one of the functions/variables.

No name list: `file`

`File` is missing symbol table information. Object files must have at least one global name to be linked.

No string table: `file`

`File` is missing it's string table, a list of the actual names referred to by the symbol table.

6.3 The Linking Process

The linker examines each argument in the order given. Object code files are *always* included, but libraries are searched by the linker and only the object code modules needed are actually included in the final executable program. The linker is capable of linking object code produced by the Laser C compiler, as well as object produced by the DRI (Digital Research Inc.) C compiler or assembler. The linker will also read library files produced by either the Laser archiver or the DRI archiver.

Since libraries frequently contain many object code modules and may therefore be rather large, a mechanism, known as randomization, has been implemented by which the archiver may be used to add an index of global function and variable names to the beginning of a library. Using this index, the linker can quickly resolve external references, thus greatly speeding the linking process. The index, if it exists, is loaded into memory and searched repeatedly until either no more undefined names need resolving, or a complete pass of the index is made and no additional object code modules are extracted. If the library does not contain this index, the linker will make only one sequential pass of the library, including code modules only if they are needed. Thus, without the index, references must refer to object modules which appear further in the file or in a subsequent file in the command line.

Symbols defined in user specified object files and libraries will override definitions of the same symbol in the libraries provided they are encountered by the linker first. The programmer may make use of this feature by writing his own versions of system library functions (such as *malloc()* for instance) while still using other procedures from the library.

6.4 Desk Accessory Support

Desk accessories are GEM applications which are named with the "**.ACC**" extension. When the computer is started, any "**.ACC**" files which appear on the root are loaded into RAM and started. A desk accessory should call `evnt_multi()` to share processor time with the main application and other desk accessories.

6

Chapter 7

7

Disassembler

Introduction

The disassembler prints the assembly language equivalent of either a Laser format object code file (see File Formats, section A.1), a DRI (Digital Research, Inc.) format object file, or an executable (GEMDOS) format file. If the file contains symbol information, it is used where possible, otherwise actual reference values are printed. Since references internal to an object file are resolved by the compiler, there will be instances where no name is associated with a reference. In these instances, the disassembler attempts to make an educated guess as to the name of a reference and if possible print it rather than just a value. All numeric values are printed in hexadecimal.

7.1 Command Line Usage

When used from a command line shell the usage syntax is:

```
dis.ttp [-N] [-I] [-R] [-Fname] object ...
```

`-N` Suppress reference names. Actual reference values are printed. By default, symbol names are printed when available. Also, addresses are not printed with this option.

`-I` Instruction print. The hex value of each instruction is printed before the instruction is disassembled.

`-R` Relative branches. Normally branch instructions, which specify addresses relative to the program counter, are converted to absolute addresses. This option suppresses the conversion.

`-F`*name* Disassemble the **named** function only.

7.2 Disassembler Errors

Error messages and possible reasons are:

Usage: dis.ttp [-N] [-I] [-R] [-F function] object ...
When run from a command line either an invalid option was specified, or no object or program files were given.

File open error: `name`
The input file `name` was not found. Check to see that the file name and path name are given correctly and that the file actually exists.

Memory full while processing
Memory exhausted. Remove RAM disk.

File format error: `name`
The file `name` is not an object file or program file, or is corrupt.

Example:

```
laserdis.ttp myprog.o
```

Disassembles the object code file `myprog.o`

Chapter 8

Archiver/Librarian

Introduction

The archiver maintains groups of files combined into a single archive file. It is primarily used to maintain libraries (groups of object code files) for use by the linker, but may be used to archive any type of files (including text files). The archiver will maintain both Laser and DRI format archives.

8.1 Command Line Usage

The archiver can be run from the Laser Shell, or the GEM desktop. When used from a command line, the usage syntax is:

```
ar.ttp key[V] [pos] archive [file] [file...]
```

`key` Archiver function key. One of the following keys directs the archiver function:

- `D` Delete from **archive file file...**
- `L` Convert the **archive** into a randomized library.
- `R` Replace/add to **archive** copies of **file file...**
- `RA` Like the `R` key above except the replace/add begins after the component in **archive** named in **pos**. Note that **pos** is *required* with this option.
- `T` List a table of component names in **archive**.

- `W` Write a copy of component `file` in `archive` to the standard output. Normally redirected to a file.
- `X` Extract copys of `file` from `archive`.

`V` Optional verbose. When used with keys **D** (delete), **R** (replace), **RA** (replace after), and **X** (extract), the archiver will print a line which verifies the operation performed. When used with the key **T** (table), The size of each component will be printed after each name.

8

`pos` Used with key `ra` above.

`archive` The archive file upon which the operations are to be performed.

`file` One or more files used depending on the operation performed.

Example:

```
ar.ttp rv megamax\libc.a a.o b.o c.o
```

Replace (or add if they are not already present) in the archive `libc.a` the object files `a.o`, `b.o`, and `c.o`.

Note: If the archive specified on the command line does not exist the archiver will create a new archive with that name.

8.2 Random Library

The **L** (randomize) key converts an archive of object files into a random library so that it can more efficiently be searched by the linker. The archiver performs this randomization by examining the entire library and collecting global function and variable names, along with information about the object modules in which they are defined, and writing a special component into the library named `__.SYMDEF`. The `__.SYMDEF` will always be the first component of the library. It is important to always randomize a newly created library. Once randomized, the archiver will automatically re-randomize any library which is changed.

8.3 Archiver Errors

Error messages printed by the archiver and possible reasons:

Usage: ar.ttp key[V] [pos] archive file [file...]
When run from a command line either an invalid key was specified, or no object or library files were given.

File open error: `name`
The file name was not found. Check to see that the file name and path name are given correctly and that the file actually exists.

File read error: `name`
Likely a problem with the disk. Try a newly formatted disk.

File write error: `name`
Either the disk is full or there is a physical problem with the disk. The archiver writes a temporary file, called "**AR__.TMP**" to the current disk. Check to see that adequate space is available on both the archiver's disk and the disk on which the archive exists.

File create error: `name`
The archiver is unable to create a new archive. Check to see that the disk is not write protected, and that the path given the output, if any, is correct. May also be a disk problem.

Temporary file open error
The archiver is unable to create the temporary file. There is either a problem with the disk or the disk from which the archiver is being run is full. Check to see that adequate space for the temporary file, which will be as large as the archive itself, is available.

File format error: `name`
File given is not an archive file.

Memory allocation error
Memory is exhausted. Remove RAM disk.

Malformed archive (0xXXX)
The archive file is internally corrupt. Make or copy a new one. The hex number given is the address where the archiver expected to find the beginning of a component file but did not.

8

8

Chapter 9

Symbol Namer

9

Introduction

Object files and application files may contain symbolic information in their symbol tables (see File Formats, section A.1). This symbolic information may be printed with the Symbol Namer utility. The nm program is capable of printing tables of Laser format object files, DRI format object files, and GEMDOS format executable files.

Two different formats are used to print the resultant information. If the file is a Laser format object, each symbol is preceded by its value (in hexadecimal) and one of the following letters:

A	Absolute
B	Bss segment
C	Common symbol
D	Data segment
T	Text segment
U	Undefined symbol

If the letter is lower case, the symbol is local. Otherwise the symbol is global.

If a DRI format object or GEMDOS file is printed, each symbol name is followed by its address (in hexadecimal) and one or more of the words: global, external, data_based, text_based, bss_based, equated, or equated_register.

9

9.1 Command Line Usage

When used from a command line shell the usage syntax is:

```
nm.ttp [file]
```

Where `file` is either a Laser format object file, a DRI format object file, or a GEMDOS format executable file which has been linked such that it still has its symbol table.

9.2 Namer Errors

Error messages and possible reasons are:

Usage: nm.ttp file
When run from a command line either an invalid option was specified, or no object or application file was given.

File open error
The input file was not found. Check to see that the file name and path name are given correctly and that the file actually exists.

File format error
The file is not a valid object or application file or program file, or is corrupt.

No name list
The file has no symbol table.

Example:

```
nm.ttp myprog.o
```

Dump the object code file `myprog.o`

Chapter 10

Make Utility

10

Introduction

Programmers often divide large programs into smaller pieces. These smaller units are easier to work with on an individual basis, but tracking the relationships and dependencies among the pieces becomes a time consuming task. As the program is modified, it is difficult to remember which files depend on which others, which files have been modified, and the exact sequence of operations needed to make or test a new version of a program.

The Make utility automates a number of program development activities so that up-to-date versions of programs may be maintained with a minimum of effort.

Make requires that a description file, called the "Makefile" be created which identifies the target files, the dependencies of the targets, and command lines used to create or update the targets. A target is a file, for example a "**.O**" file, which depends on other files, such as a corresponding "**.C**" file.

The information in the "Makefile" enables Make to identify the operations necessary to update and compile a program after modifications have been made.

The basic operation of Make is to:

- Find the name of a specified target file in the "Makefile".
- Ensure that the files upon which the target depends (the dependency files) exist and are up-to-date.
- Update or create the target to incorporate modifications that have been made to the dependency files.

In addition to the information in the "Makefile", Make maintains a table of built-in rules in a special table (called the suffixes table). It uses the information in this table to determine which file name suffixes are applicable, and how to transform those files with specific suffixes into files with other suffixes. For example, an built-in rule is that "**.O**" files are made from "**.C**" files by running the C compiler on the "**.C**" files.

10.1 Command Line Usage

10

Running Make executes command lines in a "Makefile", causing specified target files to be updated or created to reflect changes made to files on which they depend.

Make executes the file with the default name "**MAKEFILE**" unless a different name is specified.

When used from a command line, the syntax for Make is:

```
make [opt] [target] [macro=value] [-Fname...]
```

The following options are available:

`-I` Ignore error codes returned by invoked programs. Alternately, error codes can be ignored using one of two other methods:

- Enter .IGNORE as a false target in the "Makefile".
- Enter "Tab" "-" preceding a command line in the "Makefile".

`-N` No execute mode. Print commands lines, but do not execute them.

`-R` Do not use Make built-in rules specified in the suffixes table. Alternately, the use of the suffixes table can be inhibited by entering `.SUFFIXES`, without a dependency list, as a false target name in the "Makefile".

`-S` Silent mode. Do not print command lines before executing. Alternately, the silent mode may be, using two other methods:

- Enter .SILENT as a false target in the "Makefile".
- Enter "@" as the first character of a command line in the "Makefile".

`-Fname` The name of the "Makefile" to use. In the absence of this option, Make looks for the default names of "Makefile". More than one -f"Makefile" parameter can occur.

`target` The names of one or more target file names separated by a blank space. If target files are not specified in Make, the target(s) specified in the first line of the "Makefile" are updated/created.

`-P` Print all macros and targets.

`-Q` Question up-to-dateness of a target.

`-X` Prints a list of all targets in the "Makefile".

`macro=value` Define a Make macro (see section 10.3.3).

NOTE: All environment variables become defined as macros each time Make is run.

10.2 A Simple "Makefile"

It is not necessary to fully understand Make before it can be used. The following example may be adapted to a particular project by changing the file names used. Note that (tab) means enter a "Tab" character into the "Makefile".

```
# Example Makefile
#
# The target is the application program ''test.ttp'', which
# is created by compiling ''file1.c'', ''file2.c'', ''file3.c'',
# and then linking them with the C initialization code and the C
# library.

test.ttp : file1.o file2.o file3.o
(tab) cc file1.o file2.o file3.o -o test.ttp
```

In the above example, Make knows (by default) that the C compiler utility CC may be used to compile any ".C" file to a corresponding ".O" file. The target "TEST.TTP" depends on the three ".O" files and is created with the CC utility by the command line given after the "Tab". Note that "**CC.TTP**" must be located in the current folder, unless the CC environment variable is defined as the path of CC.

10.3 Makefile Structure

To use Make, a "Makefile" that specifies the target files and the files that depend on them must be created. A "Makefile" contains the following information:

- Entries (targets + dependencies + commands)
- Comments
- Macros

10.3.1 Entries

The entry is the most important part of a "Makefile". It consists of the target file names, their dependencies, and command lines.

There are two types of entries:

- Dependency lines
- Command lines

A dependency line defines the target files and their dependencies (the files that the target depends on). Optionally, a dependency line can contain one or more command lines. If a noncomment line is too long, it can be continued using a backslash. If the last character of a line is a backslash, the backslash, return, and following blanks and tabs are replaced by a single blank space.

The form of a dependency line is:

```
target...:[:][dependent...][;command...]
```

A command line contains a program name followed by program parameters. Command lines must begin with a "Tab". The form of a command line is:

```
(tab)[command...]
```

The items in a "Makefile" entry are described below.

targets The target is the name of one or more target files. These are the files that are to be updated or created. Target names are GEMDOS file names. Multiple target names are separated by blank spaces.

dependent The dependent is the name of one or more files that the target files depend on. Dependent names are also GEMDOS file names. Multiple dependent names are separated by blank spaces.

: A single colon (:) or double colon can be used (::) to separate the targets from the dependencies. A target name can appear on more than one dependency line but all lines that it appears on must be of the same (single or double colon) type.

If a target appears on more than one dependency line and a single colon is used, only one of the dependency lines can have a command sequence associated with it. If the target requires updating, and a command sequence is specified, the command sequence is executed.

If a target appears on more than one dependency line, and a double colon is used, each dependency line can have a command sequence associated with it. If the target requires updating, the associated commands are executed, including built-in rules. The double-colon form is valuable for updating archive-type files.

10

command A command is a program name followed by optional program parameters (any string of characters, excluding a # or carriage return).

Command lines can appear on a dependency line or on the line immediately following a dependency line. If a command appears on the dependency line it is preceded by a semicolon. If a command appears on the line following a dependency line, the command line must begin with a tab.

A line is printed when it is executed unless the `-S` option is used or .SILENT is entered as a false target name in the "Makefile".

Commands returning nonzero status cause Make to terminate, unless the `-i` option is used or .IGNORE is entered as a false target name in the "Makefile".

Some commands return nonzero status inappropriately. For these cases, use the `-i` option or begin the particular command with "Tab" "-" in the "Makefile".

10.3.2 Comments

The pound sign (#) indicates a comment. All characters, from a pound sign to the end of the line, are ignored. Blank lines and lines beginning with # are ignored totally. Comments can appear on dependency lines or command lines.

10.3.3 Macro Definition

Make also provides a simple macro substitution facility for substituting strings in dependency lines and commands.

A macro line contains an equal sign (=) which is not preceded by a colon or a tab. The macro name is the string to the left of the equal sign (trailing blanks and tabs are stripped). The macro is assigned the string of characters to the right of the equal sign (leading blanks and tabs are stripped).

For example, to define a macro named OBJECTS as the object files, file1.o, file2.o and file3.o, enter:

```
OBJECTS = file1.o file2.o file3.o
```

A null string may be assigned as a macro value by leaving the right of the equal sign blank. For example, to assign a null value to the macro named ZIP, enter:

10

```
ZIP =
```

Macros can also be defined in the Make command itself.

A macro is invoked using a dollar sign ($) as shown below:

```
$(macro name) or ${macro name}
```

If the macro name is a single character, the parentheses or braces are optional. Macro names exceeding one character in length, must be enclosed in parentheses () or braces {},as shown.

For example, to invoke a macro named Y, a single-character name, enter either:

```
$Y or $(Y) or ${Y}
```

To invoke a macro named OBJECTS, enter either:

```
$(OBJECTS) or ${OBJECTS}
```

There is also a facility to perform translations when a macro is referenced and evaluated. The general syntax for a macro reference is :

```
$(macro : string1 = string2)
```

This causes each occurrence of string1 to be substituted with string2 in the macro being evaluated, where macro is the name of the macro being evaluated.

Note that all environment variables which are defined as Make is executed, become macro definitions in Make.

10.3.4 Implicit Macros

If a file is generated using one of the built-in transformation rules, the following macros can be used:

`$*`	Name of the file to be made (excluding the suffix)
`$@`	Full name of the file to be made
`$<`	List of the dependencies
`$?`	List of dependencies that are out of date

10.3.5 Dynamic Dependency

10

To use these implicit macros, there is a dynamic dependency parameter referenced by the notation:

```
$$@
```

It has meaning only when it appears on a dependency line. The `$$@` refers to the item(s) to the left of the colon, which is referenced by the `$@` implicit macro.

The following is an example using implicit macros and the dynamic dependency parameter.

`PROGS= s1 s2 s3 s4`
: Defines the macro `PROGS` as the four files `s1-s4`.

`$(PROGS) : @.c`
: Invokes the `PROGS` macro, defining the target file names as `s1`, `s2`, `s3`, and `s4`. Defines their dependencies as C source files (.c) with the same file names: `s1.c`, `s2.c`, `s3.c`, and `s4.c`.

There is also a second form of the dynamic dependency parameter which refers to the file part of `$@`. This form is referenced using the notation `$$(@F)`.

10.3.6 Suffixes Table

As mentioned previously, Make maintains a table of suffixes and built-in transformation rules in suffixes table. This table may be altered with the `.SUFFIXES` directive. For example:

```
# Add the suffixes .o and .c to the suffixes table
.SUFFIXES : .o .c
```

When attempting to determine a transformation for a file which has no explicit target mentioned in the "Makefile", Make uses the suffixes table. Make looks for a file with the desired suffix, and uses the associated transformation rule to create or update the target file.

10.3.7 Transformation Rules

A transformation rule name is the concatenation of the two suffixes. For example, the name of the rule that transforms .c files to .o files is .c.o. For example:

```
# Compile (with CC) a .c file to produce a .o file.
.c.o :
    cc -c $*.c
```

A transformation rule is used only if the user's "Makefile" does not contain an explicit command sequence for these suffixes.

The order of the .SUFFIXES list is significant. Make scans the list from left to right, and uses the first name that has both a file and a rule associated with it. To append new names to the suffix list, the word .SUFFIXES may be entered as a special target in the "Makefile", listing the new suffixes as dependencies. The dependencies will be added to the suffix list.

.SUFFIX	Transformation
.c.o	cc.ttp file.c -c
.p.o	pc.ttp file.p -c

Figure 10.1: Built-in Transformation Rules

For example, to transform a source file into an object(.o) file, Make calls up the appropriate compiler. There are also transformation rules to create library (.a) files from source files.

To delete the built-in suffix table, enter .SUFFIXES as a target, without listing any dependents in the "Makefile". It is necessary to do this to clear the current list if changes in the order of the suffixes is desired.

10.4 Examples

Some example "Makefile"s are described below.

Example 1: For this example, the built-in suffixes table is used.

```
# Example 1
#
prog.ttp : x.o y.o z.o
(tab) cc.ttp x.o, y.o z.o. -o prog.ttp

# ''x.o'' and ''y.o'' depend on the header file ''prog.h''.  They
# will be recompiled if their header is changed.
x.o y.o : prog.h
```

Example 2: This example illustrates the use of macros.

```
# Example 2
#
# Define a transformation rule for creating .o files from .c
# files by compiling them.  While this is predefined as a
# built-in rule, it makes a good example.
#
.SUFFIXES .o .c
.c.o :
(tab) ccom.ttp $*.c

# Define the macro OBJECTS to be the three object files x.o,
# y.o, and z.o.
#
OBJECTS = x.o y.o z.o

# Define the library option (given to ld.ttp) as the C library.
#
LIBES = -lc

# Create prog.ttp (the default target) by linking the updated
# objects.  Uses silent mode on CC.
#
prog.ttp : $(OBJECTS) myarc.a
(tab) @cc $(OBJECTS) $(LIBES) -o prog.ttp

# This target will update and run prog.ttp
#
prog : prog.ttp
(tab) @prog.ttp
```

```
# This target just removes the .o files associated with
# the project.
#
clean : $(OBJECTS)
(tab) rm $(OBJECTS)
```

Chapter 11

Resource Construction Program

11

Introduction

The idea behind resources is that specifications for certain graphical/textual objects may be kept separate from the program which uses them. Thus, items such as menu bars, dialog boxes, and icons may be created and changed independent of the actual program. This not only simplifies coding, but also makes a program "international", since the textual strings can easily be translated into other languages. These object specifications are called resources or object trees, and are stored in a type of file known as a resource file. The Atari's ROM provides routines which use these resource files (see section 16.8). The Resource Construction Program (called "**RCP.PRG**") is used to create and modify resource files.

11.1 Definition of Resource Files

A resource file contains a number of resources stored in the "tree table". A resource (or object tree, the terms are used interchangeably) is a description for either the menu bar or a dialog box. It is composed of a collection of "objects" and their locations on the screen. An object is a basic element that the object manager can display and manipulate. Examples include buttons, strings, editable text, icons and boxes.

Resource files end with the extension ".RCS". The RCP also creates two additional files for each resource file. The ".DEF" file contains some information that the RCP needs that isn't normally part of an ".RCS" file. The ".H" file contains C `#define` commands that relate names given to the various resources and objects to index numbers that are used internally within a resource file. If the ".H" is included in a program that uses the resource file then these names can be used to access the resources instead of the index numbers (which may change if the resource file is later modified).

An example of a resource can be seen in figure 11.1.

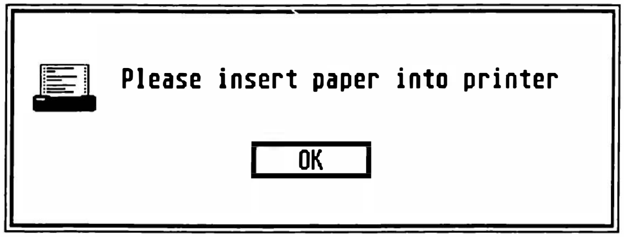


Figure 11.1: "TEST.RSC"

The resource contains three objects: a bit image, a string and a button. The ".H" file looks like this:

```
#define PAPER 1
```

None of the objects have been given names. Here is a program that displays the dialog and waits for the user to press the OK button:

```
#include "TEST.H"
#include <obdefs.h>

main()
{
    OBJECT *paper;
    int x,y,w,h;

    appl_init();
    rsrc_load("TEST.RSC");
    rsrc_gaddr(0, PAPER, &paper);
    form_center(paper, &x, &y, &w, &h);
    objc_draw(paper, 0, 10, x, y, w, h);
    form_do(paper, 0);  /* Wait for OK button */
    appl_exit();
}
```

appl_init and *appl_exit* are required calls for any program using GEM. The resource file is loaded with *rsrc_load*. *rsrc_load* translates the resources from the format used in the ".RSC" to that used by the object manager in memory. *rsrc_gaddr* returns a pointer to a resource given its index number (which was defined as PAPER in "TEST.H"). Object tree pointers are used by the object and form managers. The *form_center* call changes the location of the dialog so that it is centered within the screen. The RCP doesn't set the location of object trees because the resolution of the screen may be different each time a program is executed. The *rsrc_load* call also adjusts the objects within the resource to match the screen resolution.

objc_draw draws the resource on the screen. It will look just as it does in the RCP. This visual correspondence makes creating resources with the RCP easier. *form_do* handles user interaction with the dialog and returns the number of the object that caused the dialog to exit. We are not interested in this number since there is only one way to exit this simple dialog.

In a more complex dialog, particularly one with editable text objects, we would have to manipulate the object tree directly with C. See the object manager section for details on the format of a resource in memory.

11.2 RCP Usage

The screen is divided into two sections: a window containing the resources being edited and a palette of objects that can be added to the window. If the user selects **New** (to create a new resource file) or **Open** (to load an existing one) from the **File** menu the window containing the tree table for the resource file will be displayed. Any changes made will not affect the file until it is saved (by choosing **Save** from the **File** menu).

When the tree table is displayed the palette will contain templates of legal tree types that can be added to the file. The window shows the type and name of all object trees in the resource file. With the tree table displayed one may create, delete, copy or name entire object trees. One may also **Open** a tree which will show the object structure of the tree in the window (which is called an object display). One should close the window (with the close box or by choosing **Close** in the **File** menu) to return to the tree table.

The object display will show the resource as it will appear when drawn by a program.

11.2.1 Tree Types

RCP currently supports four tree types: unknown (when no ".DEF" file is found), free, dialog and menu.

A free tree is the most general type. Any of the other trees can be converted to a free tree by using the **Name** item under **Options** menu. In a free tree, the entire tree is always displayed and objects can be located at any pixel location. Unknown trees are treated the same as free trees so there is really no need to create them. Free trees might look different on the high and medium resolution monitors.

Dialog trees are like free trees except objects are aligned to character cell locations when they are moved. The alignment only happens when an object is moved or **Snap** is selected from the **Option** menu.

Menu trees are very restricted; as a rule, a non-menu tree should not be converted into a menu tree, since there is a good chance that **GEM** will crash.

The following information is useful when manipulating the tree structure of a resource directly with C.

The palette for free, unknown and dialog tree object displays contains the following objects in order:

```
G_BUTTON, G_STRING, G_FTEXT, G_FBOXTEXT, G_IBOX, G_BOX,
G_TEXT, G_BOXCHAR, G_BOXTEXT, G_ICON, G_IMAGE.
```

The palette for menu tree object displays contains the following objects:

```
G_TITLE, G_STRING, G_STRING (gray hyphens), G_BOX.
```

These object type names are defined in the file "OBDEFS.H". See the Object Manager section of the AES documentation for a description of the format of each of these types.

11.2.2 Visual Hierarchy

The object display uses a convention called "visual hierarchy" to make editing resources easier. Assume that one object (say a box) surrounds another object (say a button) on the screen. Visual hierarchy says that any operation performed on the box is also performed on the button. The box is called the parent of the button and the button called a child of the box. There can be multiple children of a parent, and those children may also have children. The relationship can be removed by simply dragging the button outside of the box (the **Flatten** command will also remove it).

11.2.3 Menu usage

Some menu items may not be selectable depending on what is currently being done with RCP. Items which are unavailable are dimmed in the menu bar. Some commonly used items are preceded by a letter. If the "control" key is is held down while simultaneously pressing the letter, then that menu item will be executed.

When selecting **Paste** from the **Edit** menu, the mouse button should be held down so that the object in the clipboard can be placed in its correct place (this is discussed further in the reference section for the edit menu).

11.2.4 Mouse usage

The mouse is used in combination with the keyboard to move, select and resize the objects and trees. See the Conventions section of the Introduction chapter for definitions of mouse usage terms used here.

Clicking over an object in the window will select it with the effect of it being drawn as a negative image. At this point many of the menu functions become available. The selection may be canceled by selecting another object, clicking outside the window, or clicking in the gray region of the window.

Control-clicking an object selects the object's parent. This is useful for selecting a box containing a bunch of buttons for instance or whenever the child of an object overlays its parent.

Double clicking opens an object. This is always the same as clicking and then selecting **Open** from the **File** menu.

Dragging changes an object's location on the screen. If an object is moved such that it is entirely enclosed by another object, then the dragged object is made a child of the enclosing object. Dragging always makes the selected object become the last child of whatever object it is released over. A tree is drawn so that the last child will be drawn last (making it appear on top of any other siblings it may partially overlap). The effect of this is that if two objects overlap, pressing the mouse button on the lower object, holding and then releasing will move it to the top. The main box for a tree (called the root) cannot be dragged.

Shift-dragging makes a copy of the object before dragging. The copy will not have a name (even if the original object did).

Control-dragging or **shift-control-dragging** operates on the object's parent (unless it is the root of the tree).

11.2.5 Resizing

If the mouse is very close to the lower right corner but nevertheless inside an object when dragging occurs, then only the lower right corner of the object's box will be tracked by the mouse. The size of the object will be changed when the mouse button is released. The corner will not be permitted to move outside the parent's box, nor will it be allowed to move into a child's box. Shift-dragging works the same as dragging when re-sizing. Control-dragging is useful when resizing an object which has a child object in the lower right corner, covering the "resize" zone.

11.2.6 Keyboard Usage

With an object selected, the keyboard arrow keys may be used as follows:

arrow moves the selected object one pixel in the direction of the arrow.

shift arrow moves only the lower right corner, resizing the object.

11.3 Menu Functions

The following is a short description of all menu items under each menu title.

11.3.1 Edit Menu

The edit functions operate with a special holding area called the clipboard. The clipboard can hold either a tree or an object along with its name (if it is a tree, then the names of all the objects in the tree are also stored). The value in the clipboard will remain there until it is replaced or RCP is terminated. This is useful for copying resources between two resource files.

Cut The selected object is placed into the clipboard. The object's name will also be placed in the clipboard.

Copy A copy of the selected object is placed into the clipboard. The copy will not have a name.

Paste If object is currently selected, then it is replaced by a copy of what is in the clipboard. If not, then while the mouse button is down the pasted object will be dragged. If the object in the clipboard had a name, then the copy pasted will have that name and the clipboard object will no longer have one. Care must be used when dragging in this case since if the mouse button is released outside the window, the name will be lost.

Erase The selected object and its name (if it has one) is deleted.

11.3.2 File Menu

New Create new tree table window.

Open If no window exists at all, then read a resource file from disk. If the tree table window is displayed and a tree is selected, open the object display window. If the object display window is displayed and an object is selected, open the appropriate dialog box.

Merge Read in new resources from another resource file, but don't delete the current trees.

Close If the tree table window is displayed, delete all trees and close the window (without saving). If the object display window is shown, return to the tree table window.

Save Write the current trees to the file with the same name as the title of the window.

Save as ... Write the current trees to a file to be specified in a dialog box.

Abandon Same as closing from the tree table window.

Quit Terminate the RCP program.

11.3.3 Options Menu

Info Displays some pertinent information about the selected object.

Name ... If a tree is selected, display a dialog which allows the tree type and name to be changed. If an object is selected, display a dialog which allows the object type and name to be changed. Names must be all upper case and are restricted to a length of 8 characters.

Hide Set the HIDETREE flag for the object, the effect of which is to hide it and all its children. The root object cannot be hidden.

Unhide Reset the HIDETREE flag for all children of the selected object, displaying the children.

Sort ... Sorts the children of the selected object, changing the order in which they are displayed. The index numbers of the children are not affected by this operation. The sort can be done either by X-axis or Y-axis coordinate (in ascending order) or by one then the other.

Recreate Forces physical tree structure to match logical structure by performing a preorder traversal. Useful for getting the tab and arrow keys to work correctly with edit fields in a dialog box.

Flatten Rearrange so that children of the selected object becomes siblings of the selected object.

Snap Aligns the selected object to a character cell boundary (this is the default mode for all dragging operations when editing dialog trees).

11.4 Object Dialogs

When an object is opened (by double clicking or by choosing **Open** from the **File** menu) a dialog box will appear which allows certain attributes of the object to be changed depending on its type. There are five such dialogs. The simplest is for a G_BUTTON or G_STRING (see figure 11.2).

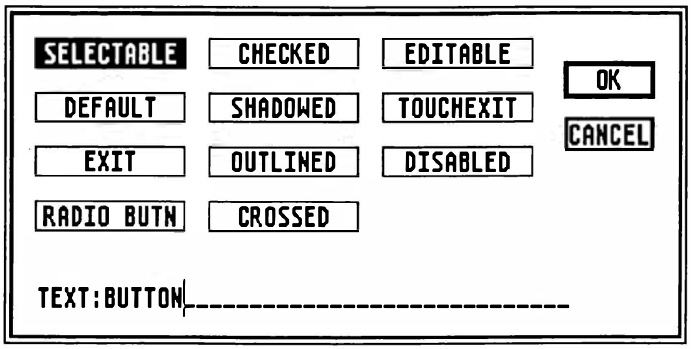
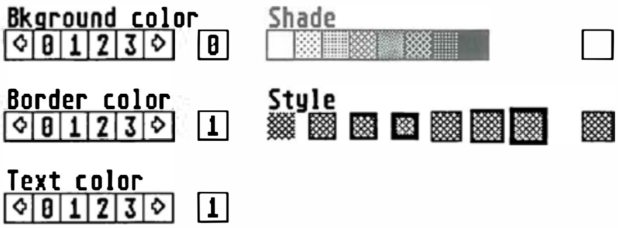


Figure 11.2: Object Dialog

The array of eleven check boxes shows the setting for the useful `ob_state` and `ob_flags` bits. All object dialogs have these eleven boxes. The only other attribute for a button or string is the value of the text. If "CANCEL" is clicked then any changes made to the object will not take effect. Pressing "Return" has the same effect as clicking the "OK" button.

Some objects allow the setting of color, shading and outline characteristics. Those that do will have one or more of the following gizmos:



The color bars have the values 0-9 and A-F listed for a total of 16 colors. Each is a different entry in the color look-up table in the ST (the actual color displayed depends on what is stored in the table, so RCP uses the entry number). By pressing on the arrows the bar can be rotated. The current setting is shown in the character box offset slightly to the right of the bar. The setting can be changed by pressing on a value in the bar. The same is true of the shade and border style bars (except they can't be rotated).

The text objects (`G_TEXT`, `G_FTEXT`, `G_BOXTEXT` and `G_FBOXTEXT`) include a `PTMPLT`, `PVALID` and `PTEXT` field for each of the `TEDINFO` strings. RCP translates the underscore '_' character in the `PTMPLT` field to a tilde '~' for display purposes (the underscore is where the user input will go). The tilde is also used in the `PVALID` and `PTEXT` fields as a place holder so the non-tilde characters line up with the tildes in the `PVALID` string. These place holder tildes are not actually in the strings.

11.5 The Icon Dialog

Icons have two dialog boxes. The first dialog for the icon has the eleven bit boxes, the value of the text field, and the "extra character". In addition, there are two color bars for the icon's foreground and mask images. Clicking on the "Edit Icon" button will display the second dialog box (see figure 11.3). It also causes a "CANCEL" of the first box, except for color settings. The second box operates in three modes chosen by the "Icon", "Text" and "Character" buttons.

The icon mode allows drawing into the bit image of the icon. By clicking the mouse on the enlarged image, the icon may be drawn. Freehand drawing

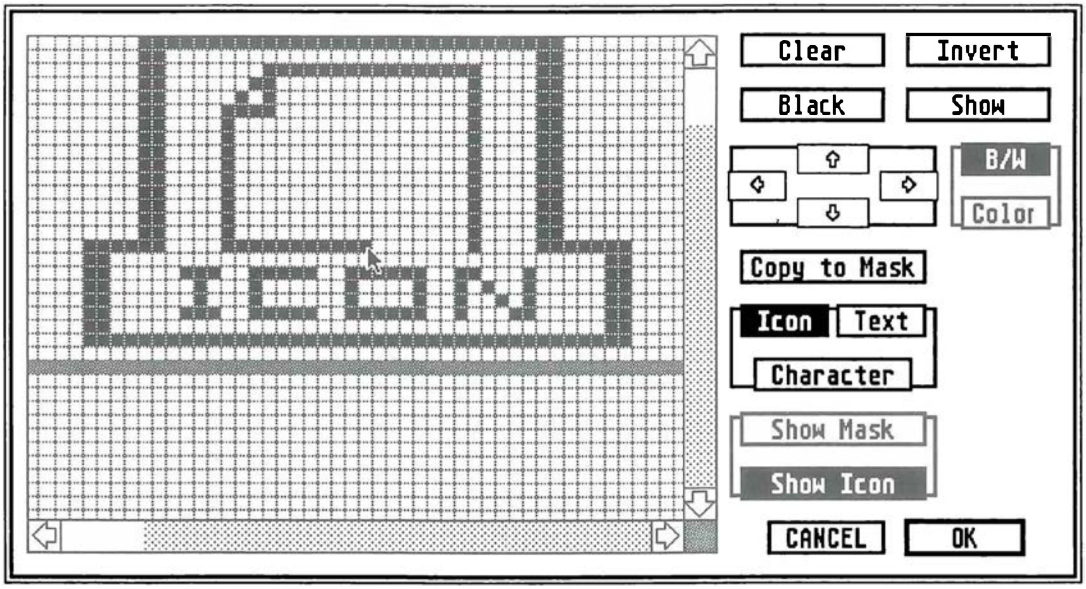


Figure 11.3: Icon Dialog

may be done by holding down the mouse button. The state (set or not set) of a drawn pixel is the opposite of that of the pixel which is first clicked on. The size of the icon is indicated by the gray outline. The Icon may be repositioned by dragging the gray outline. It may be re-sized by dragging the lower right corner of this gray outline. The icon width must always be a multiple of 16 pixels (the re-size code ensures that this is so). If the mouse is clicked in the drawing region for the icon but with the shift key depressed, a selection range may be defined. By dragging the mouse, the selection range may be sized until the mouse button is released. The pixels within the selection range may then be moved by dragging with the mouse the entire box defining the range. By holding down the shift key while moving the selection range, a copy is made of the pixels within the box, rather than a cut. Clicking the mouse outside of the selection range removes it.

The buttons "Black", "Invert", and "Clear" change each pixel within the selection range, or the entire image (either foreground or mask) if no selection is made. "Show" shows quickly what the icon will look like at its normal size. The up, down, right, and left arrows shift the pixels of the icon (or the selection range if any). The icon can be displayed as it will appear in medium resolution by clicking the "Color" button.

You can draw either into the foreground or mask images by clicking on the "Show Icon" and "Show Mask" buttons. The foreground image can be copied to the mask image by clicking "Copy to Mask".

Text mode allows the positioning and sizing of the text string. The icon is displayed in gray and the limits of the text string are displayed in black. The text position can be changed by dragging the black rectangle. It can be re-sized by dragging the lower right pixel. The text string starts out only 2 pixels tall and will need to be resized for the text to show in your icon.

Character mode allows the positioning of the "extra character". It can only be moved since its size is fixed (being only one character).

11.6 The Bit Image

Bit images (type BITBLK) also have two dialog boxes. The first has eleven bit boxes and one color bar for the color of the image's set pixels. Clicking on the "Edit Image" button will display the second dialog box (and causing a "CANCEL" of the first box, except for color setting).

The second, image edit dialog box behaves exactly as does the icon edit dialog, except that some buttons which pertain only to icons have been hidden.

11.7 Using RCP as a Resource Editor

If the source code to the program(s) which use a resource file is not available then extra care must be taken when modifying the resource file. In particular, the index number must not be changed for any object that is used. RCP ensures this is true for all objects except menus. One cannot add, delete or change the order of menu titles and items without recompiling. One may however change the text of the titles and items since this does not affect the structure of the menu tree. RCP will alert the user if changes have been made that require recompilation.

If the ".DEF" file is not available then the names and types of the resources will be unknown. Each resource will have to be opened to determine its type. Menu bars should be changed into menu bar resources prior to making changes.

11

Chapter 12

Compile and Link

12

Introduction

This program simplifies the compile and link process when developing from the GEM desktop or from a shell which does not have a built-in compile and link mechanism. This program runs the compiler (found with the environment variable CCOM) on any ".C" files, and then the linker (found with the environment variable LINKER) on any ".O" files, as well as the ".O" files produced from ".C" files.

The environment variable CINCLUDE is passed to the C compiler. The environment variables CLIB and CINIT are passed to the linker. Any options are passed to the appropriate programs. Unknown options are reported.

12.1 Command Line Usage

When used from a command line shell the usage syntax is:

```
cc.ttp [options] [-C] [file.c ...] [file.o ...] [file.a ...]
```

options Compiler and/or linker options are passed to the appropriate programs.

-C Suppress the link phase.

file.c Multiple C source files to be compiled.

file.o Multiple object files to link.

`file.a` Multiple archive files.

`options` Command line options to either the compiler or to the linker.

12.2 CC Errors

Error messages and possible reasons are:

Unknown option:
When run from a command line an invalid option was specified.

Cannot access: `name`
The named program could not be found. Check that the environment variables LINKER and CCOM are the correct path names for the C compiler and the linker.

12

12.3 Examples

```
cc.ttp hello.c aux.o -o hello
```

This will produce the following command lines:

```
ccom.ttp hello.c -o hello.o -I/MEGAMAX/HEADERS/
ld.ttp -o hello /MEGAMAX/INIT.O hello.o aux.o /MEGAMAX/LIBC.A
```

which will compile `hello.c`, link `hello.o` and `aux.o` with the initialization code specified by CINIT and the library specified by CLIB , outputting a file called “hello”.

Chapter 13

Egrep

Introduction

13

Egrep searches files for patterns that the user specifies. The patterns are in the form of regular expressions. Normally, each line found is copied to the standard output. Egrep patterns are extended regular expressions; it uses a fast deterministic algorithm that sometimes needs exponential space. Lines are limited to 1024 characters; longer lines are truncated.

The file name is shown if there is more than one input file. Care should be taken when using the characters **$ * [^ | ( ) and ** in the expression, as they may also meaningful to the shell. It is safest to enclose the entire expression argument in single quotes (').

Egrep accepts extended regular expressions. A regular expression specifies a set of strings of characters. A member of this set of strings is said to be *matched* by the regular expression. In the following description the term "character" excludes newline:

- A \ followed by a single character other than newline matches that character.
- The character ^ matches the beginning of a line.
- The character **$** matches the end of a line.
- A period (.) matches any character.
- Any other character matches that character.

- A string enclosed in brackets ([ ]) matches any single character from the string. Ranges of ASCII character codes may be abbreviated as in a-z0-9. A] may occur only as the first character of the string. A literal - must be placed where it cannot be mistaken as a range indicator.

- A regular expression followed by an asterisk ($*$) matches a sequence of zero or more matches of the regular expression. A regular expression followed by a plus ($+$) matches a sequence of one or more matches of the regular expression. A regular expression followed by a question mark ? matches a sequence of zero or one matches of the regular expression.

- Two regular expressions concatenated match a match of the first followed by a match of the second.

- Two regular expressions separated by | or newline match either a match for the first or a match for the second.

13

- A regular expression enclosed in parentheses matches a match for the regular expression. The order of precedence of operators at the same parenthesis level is as follows: [] then $*$ + ? then concatenation, then | and newline.

13.1 Command Line Usage

```
egrep.ttp [-C] [-L] [-V] [-N] [-S] pattern [files]
```

-C Matching line count option. This option will make egrep print only how many lines matched the pattern.

-L File listing option. This option will make egrep print out once the file names with matching lines.

-V Print all non matching lines option. This option makes egrep print out all lines that do not match the pattern.

-N Print line number option. This option will make egrep print out the line number of the matching line.

-S Silent option. This option will make egrep print only error messages.

13.2 Egrep Errors

Usage: egrep.ttp [-C] [-L] [-V] [-N] [-S] pattern [files]
No pattern or files were given.

Unable to open: `name`
Egrep can not open the file `name`.

Unknown flag: `flag`
`Flag` is not used by egrep.

Invalid regular expression
Something is wrong with the regular expression.

Unmatched (
A right parenthesis has been left off the expression.

Unmatched)
A right parenthesis has been left off the expression.

Premature end of regular expression
The expression finished before it should have.

Nesting too deep
The nesting of parentheses was to great

Regular expression too big
The expression was to big for egrep to compute

Memory Exhausted
Egrep ran out of memory

13.3 Example Searches

Suppose we have a file with the following line:

```
The Lazy dog jumped over the Cow with 4 big ears.
```

To search for the word "dog" in the file `myfile` use:

```
egrep 'dog' myfile
```

The next search will list all functions in a file:

```
egrep '^([a-zA-Z]|[_])([a-zA-Z0-9]|[_])*[ ]*\(' myfile2.c
```

Using the above search on:

```
main()
{
    int i;
    foo();
}

foo()
{
    int i;

    main();
}
```

would print:

13

```
main()
foo()
```

The regular expression above searches for a beginning of a line with a character or underscore followed by one or more characters, digits or underscores followed by any number of spaces followed by a left parenthesis.

Chapter 14

Disk Utilities

Introduction

This chapter documents the following disk utilities:

LS.TTP	List files
CP.TTP	Copy files
MV.TTP	Move files or directories
RM.TTP	Remove files
RMDIR.TTP	Remove directories
MKDIR.TTP	Make directories
CAT.TTP	Concatenate and print files
DUMP.TTP	Print files in hex
SIZE.TTP	Print size information of object files

14

14.1 LS

LS prints a listing of files and/or directories (folders) and information about them. If a directory (or drive specifier) is given, the name of the directory and a count of files and folders within the directory is printed, followed by a listing of all the files in the directory. If a file name is given, matching file names are listed. If no file/directory names are given, the contents of the current directory is listed. In the absence of a sorting option, names are sorted alphabetically.

Command Line Usage

```
ls.ttp [-L] [-S] [-D] [-K] [files] [directories]
```

Options

`-L` Long listing option. This will list information about the files and directories. The information includes the name, size, date of creation, and date of last modification. LS -L will print out each file on a separate line.

`-S` Sort by size option. This option will sort the listing by size.

`-D` Sort by date option. This option will sort the listing by date of last modification.

`-K` Sort by kind option. This option will sort the listing by the extension. That is, all `.ttp` will be together, all `.c` will be together.

Errors

Unknown option: `option`
An option was given that LS does not recognize.

File not found: `name`
The file or directory name does not exist.

Drive `DRIVE:` **not available.**
The drive is not available

14.2 CP

CP copies files. There are two forms of CP. The first will copy file1 to file2. The second will copy a number of files to a specific directory.

Command Line Usage

```
cp.ttp file1 file2
or
cp.ttp files directory
```

Errors

Can't copy file to itself: `name`
CP was given the same file name to copy to as the source file name.

`name`**: not a directory**
The name of the directory to copy to was invalid.

Can't open: `name`
The file that is to be copied does not exist or there is something wrong with the disk.

14.3 MV

MV moves files. The old copies of the files are removed. The are two forms of MV. The first will move file1 to file2. The second will move a number of files to a specific directory.

Command Line Usage

```
mv.ttp file1 file2
or
mv.ttp files directory
```

Errors

Can't copy file to itself: `name`
MV was given the same file name to move to as the soucre file name.

`name`**: not a directory**
The name of the directory to move to was invalid.

Can't open: `name`
The file that is to be moved does not exist or there is something wrong with the disk.

14.4 RM

RM deletes files.

Command Line Usage

```
rm.ttp [-R] [-F] [-I] file(s)
```

Options

-R Recursive directory delete. This option will cause RM to recursively open any directory and delete all files within the directory.

-F Force option. With this option, no errors are reported.

-I Interactive option. RM will ask for verification to delete each file.

Errors

Usage: rm file ...
There are no options to rm

14

14.5 RMDIR

RMDIR deletes directories. The directory must not contain any files.

Command Line Usage

```
rmdir.ttp directory(s)
```

Errors

No such directory: `name`
The file `name` does not exist or the disk is write protected.

14.6 MKDIR

MKDIR creates directories. If a partial pathname is given MKDIR creates the directory in the current directory.

Command Line Usage

```
mkdir.ttp directory
```

Errors

Can't create directory: `name`
The directory `name` already exists or the disk is write protected.

14.7 CAT

CAT will print files to standard output.

Command Line Usage

```
cat.ttp files
```

Errors

File not found: `name`
The file or directory name does not exist.

14

14.8 DUMP

DUMP will print files in hex to standard output.

Command Line Usage

```
dump.ttp files
```

Errors

File not found: `name`
The file `name` does not exist.

14.9 SIZE

SIZE will print size information for the different segments of object or executable files to standard output.

Command Line Usage

```
size.ttp files
```

Errors

File not found: `name`

The file `name` does not exist.

File format: `name`

The file `name` is not an object or executable file.

Chapter 15

UNIX Compatible Routines

15

Introduction

The functions described in this chapter are compatible with functions by the same names which are available to C programmers using the UNIX operating system. Most of these routines are available in all C implementations; even those on micro-computers without UNIX. Use of these functions will therefore reduce the effort involved in porting a C program to another computer.

Many of the services provided here are also available through BIOS, XBIOS or GEMDOS functions, but these should be avoided if portability is a concern.

15.1 Line Separators

Because of the heritage of the C language, the ASCII line feed character (numerically, 10 decimal) is usually considered to be the line separator character. The ST software considers a carriage return/line feed combination to be the line separator. In order to easily overcome this difference, the Laser C run time library automatically converts carriage return/line feeds to line feeds on input, and converts line feeds to carriage return/line feeds on output to files. This conversion occurs at a very low level within the library routines. Files may be opened in untranslated or binary mode by setting a flag when the open procedure is called. For example `fopen( "FILE.O", "br" );` would open the file "**FILE.O**" for untranslated (binary) read.

15.2 File I/O

Contained in the system library are routines for both buffered and unbuffered input/output to disk files. The buffered routines are those whose names begin with "f", comprising the stream file interface. The unbuffered routines are the low-level *read()* and *write()* routines. Both levels of I/O allow random access to disk files. Along with these routines, the programmer is free to use the BIOS routines for input/output.

Stream I/O

A stream file is a pointer to a `FILE` data structure (declared in the header file "**STDIO.H**"). Each stream is associated with a regular file via a file descriptor (returned by *open* or *creat*). Streams buffer data through the file descriptor so single character I/O is efficient. The buffer size may changed from the default of 512 bytes for added speed by using the *setbuffer* call. Streams are used because of the large number of functions available as compared with the Basic I/O level.

Three streams are open when a program starts: **`stdin`**, **`stdout`** and **`stderr`**. **`stdin`** is open for reading only and is connected to the keyboard (ie. its file descriptor is 0). **`stdout`** and **`stderr`** are open for writing only and are connected to the screen (file descriptor 1).

15.3 I/O Redirection

I/O redirection is a mechanism where **`stdin`** and **`stdout`** are changed from using the keyboard and screen to using files. **`stdin`** is changed by passing '`<INFILE`' on the command line. **`stdout`** can be changed in two ways: '`>OUTFILE`' will open and erase outfile, while '`>>OUTFILE`' will append to an existing outfile. The program does not have to be changed for I/O redirection to work (although it must have the **`argc`** and **`argv`** parameters declared for *main()*).

15.4 Device I/O

All of the system devices are available to the C programmer through the C input/output system. Legal device names are: '`AUX:`', '`PRT:`' and '`CON:`'. For most device input/output, it is wise to use *setbuf()* to prevent buffering on the stream connected to the device.

When using the unbuffered input/output services, the only significant flag in the mode word is the binary (`O_BINARY`) flag. If this flag is set, there will be no special treatment for line separator characters. Note that one cannot *creat()* a device.

BIOS routines may be used to manipulate devices, but they require the file descriptor number. This number is just the *fileno()* (defined in `<stdio.h>`) of the stream or the file number returned by *open()*.

15.5 Memory Allocation

The memory allocation routines *malloc()* and *calloc()* are available to the C programmer. Because of the high space overhead (not to mention the bugs) in memory allocation at the GEMDOS level, these routines allocate memory in 8KB blocks, breaking the blocks up as necessary to satisfy the requests made from the C program. The *free()* routine will coalesce space which is returned and the allocation system will reuse deallocated space; however, memory will not be returned to the GEMDOS routines.

Programs begin execution with 8KB of stack space available. This is plenty of stack for most applications (the C compiler, in fact, uses less than 5KB). The size of the stack may be changed by declaring global variable **_stksize** and initializing that variable to the size of the stack required. Example:

```
long _stksize = 16384L;
```

Note that because pointers are 32 bits long, a C program can use as much memory as is available on the machine through dynamic allocation.

IMPORTANT NOTE: you must make the declaration:

```
extern char *malloc();
```

in your program before you use *malloc* (the same is true for *calloc()*). If you don't do this the compiler will assume *malloc()* returns an **int** (which is only 16 bits wide). The declaration is included in **<stdio.h>**.

15.6 Program Parameters

Program parameters passed from GEM desktop or a shell are available through the **argc** and **argv** program parameters to *main()*:

```
main(argc, argv, envp)
int   argc;
```

15

```
char *argv[];
char *envp[];
```

argc is the number of strings in the **argv** array. **argv[0]** is not defined. If you don't need program parameters, just declare *main()* without any parameters and the linker will not load the code to retrieve them.

envp is a pointer to a NULL terminated list of environment variables from the previous program, and is optional.

15.7 Summary of Routines

Basic I/O Functions

open	open a file	close	close a file
read	read data from file	write	write data to file
lseek	reposition file	isatty	determine file type
creat	create a file (old method, use open)	unlink	delete a file

Stream I/O Functions

fopen	open a stream	freopen	use different file with stream
fdopen	use existing file with stream		
fclose	close a stream	fflush	write buffer to disk
feof	test end of file	ferror	test for error
clearerr	remove error state	fileno	file associated with stream
fread	read data from stream	fwrite	write data to stream
fseek	reposition stream	rewind	reposition stream to front
ftell	report position		
getc	fast read byte	getchar	read byte from "stdin"
fgetc	read byte	getw	read word
gets	read string from "stdin"	fgets	read string
putc	fast write byte	putchar	write byte to "stdout"
fputc	write byte	fputw	write word
puts	write string to "stdout"	fputs	write string
printf	formated write to "stdout"	fprintf	formated write
sprintf	formated "write" to array	sscanf	formated "read" from array
scanf	formated read from "stdin"	fscanf	formated read
setbuf	set buffer (standard size)	setbuffer	set buffer (any size)
setlinebuf	set buffer mode	ungetc	put byte back on "stdin"

Conversion and Classification Functions

atof	ASCII to float	atoi	ASCII to int
atol	ASCII to long	strtol	ASCII (any base) to long
toupper	byte to uppper case	tolower	byte to lower case
_toupper	fast toupper	_tolower	fast tolower
toascii	int to ASCII	isalpha	test for letter

Conversion and Classification Functions, con't.

isupper	test for upper case	islower	test for lower case
isdigit	test for digit	isxdigit	test for base 16 digit
isalnum	test for alphanumeric	isspace	test for white space
ispunct	test for punctuation	isprint	test for printable
iscntrl	test for control char	isascii	test for ASCII

String Functions

strcat	append strings	strncat	append "n" bytes
strcmp	compare strings	strncmp	compare "n" bytes
strcpy	copy string	strncpy	copy "n" bytes
xtrcat	append, but return end	xtrcpy	copy, but return end
xtrncpy	copy "n" bytes, return end	strlen	length of string
index	find byte in string	rindex	find byte from end

Math Functions

abs	absolute value of int	labs	absolute value of long
log	natural logarithm	exp	base e exponential
log10	base 10 logarithm	exp10	base 10 exponential
log2	base 2 logarithm	exp2	base 2 exponential
sin	sine	cos	cosine
tan	tangent	asine	inverse sine
acos	inverse cosine	atan	inverse tangent
sqr	square	sqrt	square root
powerd	raise to power	poweri	raise to integer power
dabs	absolute value of double	dint	integer part of double
mulpower2	fast $n \times 2^k$	lngamma	log of gamma function
fac	factorial	matinv	matrix inversion

Memory Allocation Functions

malloc	allocate memory	lmalloc	allocate lots of memory
calloc	allocate and clear	lcalloc	allocate a lot and clear
realloc	resize allocated memory	lrealloc	resize a lot of memory
free	release memory	alloca	allocate on stack
sbrk	another way to get memory		

Miscellaneous Functions

exit	terminate program	_exit	terminate, but don't clean up
rand	random number	srand	start random sequence
setjmp	non-local label	longjmp	non-local goto
perror	print system error	qsort	quicksort

15

NAME

abs, labs — return integer or long absolute value

SYNOPSIS

```
int abs(i)
    int i;

long labs(l)
    long l;
```

DESCRIPTION

abs and *labs* return the absolute value of the number that is the parameter.

NAME

atof — converts ASCII string to a floating-point number

SYNOPSIS

```
double atof(nptr)
    char *nptr;
```

DESCRIPTION

atof converts a character string pointed to by `nptr` to a double precision floating point number. The first unrecognized character ends the conversion. *atof* recognizes an optional string of white-spaced characters, then an optional sign, then a string of digits optionally containing a decimal point, then an optional E or e followed by an optionally signed integer. If the string begins with an unrecognized character, then a zero is returned.

SEE ALSO

strtol

NAME

bcmp, bcopy, bzero — memory block operations.

SYNOPSIS

```
int bcmp(block1, block2, len)
    char *block1, *block2;
    int   len;

int bcopy(source, destin, len)
    char *source, *destin;
    int   len;

int bzero(block1, len)
    char *block1;
    int   len;
```

DESCRIPTION

These functions perform various operations on blocks of memory.

15

bcmp compares two blocks of memory **block1** and **block2**. The size of the blocks is **len**. A value of 1 is returned if they are identical.

bcopy copies the **source** block of memory to the block of memory pointed to by **destin**. Both blocks are of size **len**.

bzero zeroes the memory pointed to by **block1**. The block is of size **len**.

NAME

close — close a file.

SYNOPSIS

```
int close(fildes)
    int fildes;
```

DESCRIPTION

fildes is a file descriptor obtained from *creat* or *open*.

Close will fail if **fildes** is not a valid, open file descriptor.

DIAGNOSTICS

If successful, a 0 is returned.

If unsuccessful, a -1 is returned and **errno** is set appropriately.

NAME

toupper, tolower, _toupper, _tolower, toascii — convert character

SYNOPSIS

```
#include <ctype.h>

int toupper(c)
    int c;

int tolower(c)
    int c;

int _toupper(c)
    int c;

int _tolower(c)
    int c;

int toascii(c)
    int c;
```

DESCRIPTION

toupper and *tolower* have a range from −1 to 255. If the argument for *toupper* is a lower-case letter, the result is a corresponding upper-case letter. If the argument for *tolower* is an upper-case letter, the result is a corresponding lower-case letter. Arguments other than the ones mentioned are returned unchanged.

Toascii returns the argument with all but the low order 7 bits set to zero.

_toupper and *_tolower* are similar to *toupper* and *tolower* but have smaller domains and are faster. *_toupper* requires a lower-case letter as its argument. *_tolower* requires an upper-case letter as its argument. Undefined results occur if arguments are other than required.

NAME

creat — create a new file or rewrite to an existing one.

SYNOPSIS

```
#include <fcntl.h>

int creat(fname, oflag)
    char *fname
    int   oflag;
```

DESCRIPTION

creat creates a new file or writes to an existing one. If the file exists then the length of the file is reduced to 0.

If successful, the file descriptor is returned and the file is opened for writing. The file pointer is set to the beginning of the file.

oflag may be set to `O_BINARY` to indicate the untranslated mode. No other flag values are allowed here (see *open*).

creat will fail if an OS error occurs.

No process may have more than 20 files open simultaneously.

15

NOTE

This function has been superceeded by open with the O_CREAT flag.

DIAGNOSTICS

If successful, a non-negative integer is returned (the file descriptor).

If unsuccessful, −1 is returned and **errno** is set appropriately.

SEE ALSO

open

NAME

isalpha, isupper, islower, isdigit, isxdigit, isalnum, isspace, ispunct, isprint, iscntrl, isascii — classify characters

SYNOPSIS

```
#include <ctype.h>

int isalpha(c)
    int c;

int isupper(c)
    int c;
```

DESCRIPTION

These macros classify character-coded integer values. A zero is returned for false and a non-zero is returned for true. *isascii* is defined on all integer values, the rest are defined where *isascii* is true and for EOF (−1).

isalpha **c** is a letter

isupper **c** is an upper-case letter

islower **c** is a lower-case letter

isdigit **c** is a digit

isxdigit **c** is a hexadecimal digit

isalnum **c** is alphanumeric

isspace **c** is a space, tab, carriage return, new-line, or form-feed

ispunct **c** is a punctuation character (neither control nor alphanumeric)

isprint **c** is a printing character, 040 (space) through 0176 (tilde)

iscntrl **c** is a delete character (0177) or an ordinary control character (less than 040)

isascii **c** is an ASCII character, code less than 0200

DIAGNOSTICS

If the argument of any of these macros lies outside its domain, the result is undefined.

NAME

execv, execve — Execute a file.

SYNOPSIS

```
int execv(pathname, argv)
    char *pathname, *argv[];

int execve(pathname, argv, envp)
    char *pathname;
    char *argv[], *envp[];
```

DESCRIPTION

execve executes a program from the disk. *execv* calls *execve*, passing the value of the global **environ** for the parameter **envp** (see below).

The parameter **pathname** is a pointer to a string which contains the name of the program to be executed.

The parameter **argv** is an array of character pointers to strings, creating an argument list that is made available to the new program. By convention, at least one argument must be present in this array, and the first element of this array should be the name of the executed program. However, since the Atari operating system does not supply this information the first element is generally NULL.

The parameter **envp** is also an array of character pointers to strings which are not command line arguments, but system environment variables.

When the executed program begins, it is called as follows:

```
main(argc, argv, envp)
    int argc;
    char *argv[]
    char *envp[];
```

where **argc**, the "arg count", is the number of elements in **argv**, and **argv** is the array of character pointers to the arguments themselves.

The parameter **envp** is a pointer to an array of strings which are the environment variables from the calling program. Note that a pointer to this array is also stored in the global variable **extern char **environ**. Each string consists of a name, an "=" sign, and a null-terminated value. The array of pointers is terminated by a null pointer. The Laser Shell passes an environment entry for each global shell variable defined when the program is called.

The result from *execv* and *execve* is the exit code or status of the program. If an error occurs during the launch of the new program, *execv* and *execve* will return the appropriate DOS error code.

NOTE

Since the command line on the Atari is limited to 128 characters, the Laser Shell uses the environment variable `ARGV=` when this limit is exceeded. The value of `ARGV` is a single string containing a space separated list of the arguments past the 128 byte limit. These arguments are added to `argv` by the C initialization code so the program never has to deal with them specially.

SEE ALSO

exit, DOS Error Codes (pg. 587)

NAME

exit, _exit — terminate a process

SYNOPSIS

```
exit(status)
    int status;

_exit(status)
    int status;
```

DESCRIPTION

exit performs some cleanup operations before terminating the program:

- The *onexit* functions are called in the reverse of the order in which they were added.
- All open streams are flushed and closed.
- All remaining file descriptors opened with *open* or *creat* are closed.
- *_exit* is called.

_exit terminates the program immediately without performing any cleanup operations.

`status` is returned to the calling program as the result of the *execv* or *Pexec* call.

SEE ALSO

onexit, *execv*, *Pexec*

NAME

fclose, fflush — close or flush a stream

SYNOPSIS

```
#include <stdio.h>

int fclose(stream)
    FILE *stream;

int fflush(stream)
    FILE *stream;
```

DESCRIPTION

fclose writes any buffered data to disk and closes the stream file. It is called for each open stream by *exit*,

fflush writes any buffered data to disk, but does not close the stream file.

DIAGNOSTICS

If successful, these routines return a 0. If unsuccessful, an EOF is returned.

SEE ALSO

exit, *fopen*

NAME

ferror, feof, clearerr, fileno — stream status inquiries

SYNOPSIS

```
#include <stdio.h>

int feof(stream)
    FILE *stream;

int ferror(stream)
    FILE *stream;

clearerr(stream)
    FILE *stream;

int fileno(stream)
    FILE *stream;
```

DESCRIPTION

feof returns a non-zero when EOF has previously been detected reading the named input stream, otherwise zero is returned.

ferror returns a non-zero when an I/O error has previously occurred reading from or writing to the named stream, otherwise a zero is returned.

clearerr resets the error indicator and EOF indicator to zero on the named stream.

fileno returns the integer file descriptor for the named stream.

NOTE

All these functions are implemented as macros and therefore cannot be declared or redeclared.

SEE ALSO

perror

NAME

fopen, freopen, fdopen — open a stream

SYNOPSIS

```
#include <stdio.h>

FILE *fopen(file_name, type)
    char *file_name, *type;

FILE *freopen(file_name, type, stream)
    char *file_name, *type;
    FILE *stream;

FILE *fdopen(fd, type)
    int fd;
    char *type;
```

DESCRIPTION

fopen opens the file named by `file_name` and associates a stream with it. *fopen* returns a pointer to the `FILE` structure associated with the stream.

freopen substitutes the named file in place of the open stream. The original stream is closed regardless of whether the open succeeds or not. *freopen* returns a pointer to the `FILE` structure associated with stream.

freopen is typically used to attach the pre-opened streams associated with `stdin`, `stdout`, and `stderr` to other files.

fdopen creates a stream from the file descriptor (`fd`) for a file opened with *open* or *creat*.

`file_name` points to a character string that contains the name of the file to be opened.

`type` is a character string with one of the following values:

`r` open for reading

`w` truncate or create for writing

`a` append; open or create for writing at end of file `r+` open for update (reading and writing)

`w+` truncate or create for update

`a+` random open for read or write; pointer will be repositioned to end of file for writing

In addition, any of the above may be preceded by a "`b`" to indicate that line-feed/carriage return combinations are **not** to be translated to line-feeds.

If a file is open for update, both input or output may be attempted on the stream. However, output may not be directly followed by input without an intervening *fseek* or *rewind*, and input may not be directly followed by output without an intervening *fseek*, *rewind*, or an input operation which encounters end-of-file, `EOF`.

Files open for append cannot have information overwritten. All output is appended to the end of file regardless of current pointer position. After output is completed, the pointer is positioned at the end of the file.

DIAGNOSTICS

If unsuccessful, these routines return a NULL pointer.

SEE ALSO

open

NAME

fread, fwrite — binary input/output

SYNOPSIS

```
#include <stdio.h>

int fread(ptr, size, nitems, stream)
    char *ptr;
    int   size, nitems;
    FILE *stream;

int fwrite(ptr, size, nitems, stream)
    char *ptr;
    int   size, nitems;
    FILE *stream;
```

DESCRIPTION

fread places into an array `nitems` of data read from the input stream beginning at `ptr`. The data items are a sequence of bytes of length `size`. Reading is stopped when an error occurs, end-of-file is encountered, or `nitems` of data have been read. *fread* places the pointer, if any, at the byte following the last byte read, if one exists. The contents of the stream are not changed.

fwrite attempts to append `nitems` of data from the array pointed to by `ptr` to the named output stream.

NOTE

fseek or *rewind* must be called before switching between reading and writing on a stream that allows both.

DIAGNOSTICS

Both routines return the number of items written or read. If a non-positive number is given for `nitems`, then a 0 is returned and nothing is read or written.

NAME

fseek, rewind, ftell — reposition a file pointer in a stream

SYNOPSIS

```
#include <stdio.h>

int fseek(stream, offset, ptrname)
    FILE *stream;
    long  offset;
    int   ptrname;

rewind(stream)
    FILE *stream;

long ftell(stream)
    FILE *stream;
```

DESCRIPTION

fseek sets the position of the next input or output operation on the stream. The new position is at the signed distance offset bytes from the beginning, from the current position, or from the end of the file, depending on the value of **`ptrname`** (either 0, 1, or 2 respectively).

rewind is equivalent to **`fseek(stream, 0L, 0)`**, except no value is returned.

ftell returns the the offset of the current byte relative to the beginning of the file associated with the named stream.

fseek and *rewind* undo the effects of *ungetc*.

After *fseek* or *rewind* the next operation to the file may be either input or output.

DIAGNOSTICS

If successful, *fseek* returns a 0.

If unsuccessful, a non-zero is returned. This can occur if *fseek* is attempted on a file not open via *fopen* or if it is used on something other than a file.

SEE ALSO

lseek

NAME

getc, getchar, fgetc, getw — get a character or word from a stream

SYNOPSIS

```
#include <stdio.h>

int getc(stream)
    FILE *stream;

int getchar()

int fgetc(stream)
    FILE *stream;

int getw(stream)
    FILE *stream;
```

DESCRIPTION

getc returns the next byte from the named input stream and positions the pointer ahead one byte in **stream**. *getc* is a macro and cannot be used where a function is required, i.e. a function pointer cannot point to it.

getchar returns the next character from the standard input stream, **stdin**. *getchar* is also a macro.

fgetc performs the same function as *getc*, however it is a true function. It is slower, but takes less space per invocation.

getw returns the next word (integer) from the named input stream. EOF is returned if end-of-file or error is encountered. Since EOF is a valid integer, *feof* or *ferror* will need to be used to check the success of *getw*. The file pointer is positioned at the next word. No special alignment is assumed.

DIAGNOSTICS

EOF is returned when end-of-file or error is encountered.

NAME

getenv — get value of environment variable.

SYNOPSIS

```
char *getenv(envname)
    char *envname;
```

DESCRIPTION

getenv searches the environment variable list (kept in **environ** (see *execv*)) for the name **envname**. The form of an environment variable is **name=value**. If the "name" of the variable is identical to **envname** a pointer to the "value" is returned. If the variable name is not in the environment list a NULL pointer is returned.

SEE ALSO

execv

NAME

gets, fgets — get a string from a stream

SYNOPSIS

```
#include <stdio.h>

char *gets(s)
    char *s;

char *fgets(s, n, stream)
    char *s;
    int   n;
    FILE *stream;
```

DESCRIPTION

gets reads characters from the standard input stream, **stdin**, into the array pointed to by **s**, until an end-of-file or new-line character is encountered. The new-line character is discarded and the string is terminated with a null character.

15

fgets reads characters from the stream into an array pointed to by **s**, until **n** − 1 characters are read, or a new-line character is read and transferred to **s**, or an EOF is encountered. The string is terminated with a null character.

DIAGNOSTICS

If successful, **s** is returned.

If EOF is encountered and no characters have been read, then no characters are transferred to s and a null pointer is returned.

If an error occurs, a null pointer is returned. Attempting to use one of these functions on a file that has not been open for reading will cause an appropriate error.

NAME

isatty — determine file device type.

SYNOPSIS

```
int isatty(fd)
    int fd;
```

DESCRIPTION

isatty determines the type of device that is associated with the file descriptor `fd`. If the device is the keyboard the result of the function is 1.

NAME

lseek — move read/write file pointer

SYNOPSIS

```
long lseek(fildes, offset, whence)
    int  fildes;
    long offset;
    int  whence;
```

DESCRIPTION

lseek sets the file pointer associated with `fildes` according to `whence` as follows:

`whence` = 0 — the pointer is set to offset bytes.

`whence` = 1 — the pointer is set to current position plus offset.

`whence` = 2 — the pointer is set to the file size plus offset.

DIAGNOSTICS

If successful, the pointer position, measured in bytes from the beginning of the file, is returned. 15

If unsuccessful, −1 is returned and `errno` is set appropriately.

lseek will fail and the pointer will remain unchanged if:

- `fildes` is not an open file descriptor.
- `whence` is not 0,1, or 2.
- The resulting pointer position would be negative.

NAME

malloc, lmalloc, calloc, lcalloc, realloc, lrealloc, free, alloca — RAM allocator

SYNOPSIS

```
char *malloc(size)
    unsigned size;

char *lmalloc(size)
    unsigned long size;

char *calloc(nelem, elsize)
    unsigned nelem, elsize;

char *lcalloc(nelem, elsize)
    unsigned long nelem, elsize;

char *realloc(ptr, size)
    char     *ptr;
    unsigned  size;

char *lrealloc(ptr, size)
    char          *ptr;
    unsigned long  size;

free(ptr)
    char *ptr;

char *alloca(size)
    unsigned long size;
```


DESCRIPTION

malloc returns a pointer to a block of at least **size** bytes aligned for any use. Note that the **size** parameter limits the size of the block to 64K.

lmalloc like *malloc* but accepts a long parameter (allowing more than 64K bytes per allocation).

calloc allocates space for an array of **nelem** elements of size **elsize**. The space is initialized to zeros.

lcalloc like *calloc* but accepts long parameters.

realloc changes the size of the block pointed to by **ptr** to **size** bytes and returns a pointer to the (potentially moved) block. Note that the data will remain unchanged and any data defined beyond **size** will be lost.

lrealloc like *realloc* but accepts a long parameter.

free makes space, pointed to by **ptr** and formerly allocated by *malloc*, *lmalloc*, *calloc*, or *lcalloc* available for further allocation. *free* does not affect the contents of the space.

alloca allocates **size** bytes of space in the stack frame of the calling function. This space is temporary and will be automatically released upon the return of the calling function.

NOTE

alloca does not check for stack overflow. The size of the stack is set to the value in **extern long _stksize** when the program starts (default 8K bytes). **_stksize** should be redefined if more space is needed:

```
long _stksize = 10000;

main()
```

15

DIAGNOSTICS

The functions *malloc*, *lmalloc*, *realloc*, *lrealloc*, *calloc* and *lcalloc* will return a null pointer if the memory requested is not available.

SEE ALSO

sbrk

NAME

math — floating point math routines

SYNOPSIS

```
#include <math.h>

double log(x), log10(x), log2(x);
double exp(x), exp10(x), exp2(x);

double sin(x), cos(x), tan(x);
double asin(x), acos(x), atan(x);

double sqr(x), sqrt(x);

double powerd(x, y), poweri(x, a);
double dabs(x);
double dint(x);
double mulpower2(x, k);
double lngamma(x);
double fac(k);
    double x, y;
    int a, k;

double matinv(a, c, n)
    double *a;
    long   *c;
    long   n;
```


DESCRIPTION

These routines implement various mathematical functions. The format of a double precision floating point number is as follows:

- The leftmost bit (63) is the sign for the mantissa.
- The next bit (62) is the sign for the exponent.
- The next 10 bits (61–52) contain the binary exponent which has a bias of `0x3ff` (1023).
- The mantissa, contained in bits 51–0, is preceded by an implied 1-bit (left of the binary point). Therefore, the theoretical precision is $53 \times \log_{10}(2) =$ 15.95 decimal digits.

A zero is represented by all zeros in the floating point variable. The largest possible value for a float variable is contained in the math libary variable **double dcsu**. The value of this variable is `0x7fffffffffffffff`. The value of infinity is represented by the math library variable **double dcin**. It's value is `0xffffffffffffffff`. This value is returned in the instances where a floating point operation exceeded the maximum value of a double floating point number.

The smallest number $x > 0$ is:

$$\begin{aligned} x &= \mathrm{0x0000000000000001} \\ &= (1 + (2^{-52}))(2^{1025}) \\ &= 1.1125369292536009 \times 10^{-308} \end{aligned}$$

If the absolute value of a result is smaller than this number (called underflow), a zero is returned.

log and *exp* are base *e* logarithm and exponential functions.

log10 and *exp10* are base 10 logarithm and exponential functions.

log2 and *exp2* are base 2 logarithm and exponential functions.

sin, *cos*, and *tan* are transcendental functions.

asin, *acos* and *atan* are inverse transcendental functions.

sqr is x^2.

sqrt is $\sqrt{x}$.

powerd is x^y. This is equivalent to `exp2(x * log2(y))`.

poweri is the same as *powerd* but with an integer **a** for y.

dabs is $|x|$.

dint is the integer part of the double that is the parameter. The fractional part is truncated. This is equivalent to

$$\mathrm{sgn}(x) \times \lfloor |x| \rfloor$$

where

$$\mathrm{sgn}(x) = \begin{cases} -1, & \text{if } x < 0; \\ 0, & \text{if } x = 0; \\ 1, & \text{if } x > 0 \end{cases}$$

mulpower2 performs a fast floating point multiplication by 2^k.

lngamma is the natural logarithm of the gamma function if $0 < x < 5.1 \times 10^{305}$. Outside of this range `dcin` (infinity), is returned.

fac is $k!$, where $0 \leq k \leq 170$.

matinv is the matrix inverse of the $\mathtt{n} \times \mathtt{n}$ array **a**. The data in **a** may be stored in either row or column major order (C double dimension arrays are row major). **c** is a vector (one dimensional array) of **longs** used during the computation. *matinv* returns the determinant of **a** as the function result, and the inverse of **a** in **a**. **c** has no meaning after *matinv* finishes. A determinant value of zero indicates failure (**a** is destroyed). Example:

```
#include <math.h>

double e[2][2] = {1, 0, 0, 1};     /* Identity Matrix */

main()
{
    double det;
    long   C[2];

    det = matinv(e, C, 2L);
    printf("The determinant of e is %f\n", det);
}
```


NOTE

All intermediate floating point operations are done in double precision. The transcendental functions use radians.

NAME

onexit — call user defined function upon exit.

SYNOPSIS

```
int onexit(userfunc)
    int (*userfunc)();
```

DESCRIPTION

onexit is used to define user exit functions. These functions will be executed before files are closed by the standard exit function *exit*. The maximum number of exit functions allowed is eight. If the maximum is exceeded the result of the function is 1, `TRUE`.

One of the eight functions is used by the program profiling code.

SEE ALSO

exit

DIAGNOSTICS

1 is returned after the maximum of eight functions are added to the exit list.

NAME

open — open for reading or writing

SYNOPSIS

```
#include <fcntl.h>

int open(fname, oflag)
    char *fname;
    int   oflag;
```

DESCRIPTION

open opens a file for reading and/or writing as specified by the **oflag**. A file descriptor for the file is returned. The parameter **fname** points to a string containing the name of the file. The **oflag** values are constructed by OR-ing flags from the following list:

`O_RDONLY` open for reading only.

`O_WRONLY` open for writing only.

`O_RDWR` open for reading and writing.

`O_CREAT` create file if it does not exist.

`O_TRUNC` truncate size to 0.

`O_BINARY` open in binary (untranslated) mode.

Note that only one of the first three may be used. Upon completion, the file pointer is set to the beginning of the file.

NOTE

No program may have more than 20 file descriptors open simultaneously.

open with `O_CREAT` superceeds the older function *creat*.

DIAGNOSTICS

If successful, the file descriptor is returned.

If unsuccesful, **errno** is set and -1 is returned.

NAME

perror, sys_errlist, sys_nerr — System error messages

SYNOPSIS

```
perror(s)
    char *s;

extern int sys_nerr;
extern char *sys_errlist[];
```

DESCRIPTION

perror writes a short description of the last error that set `errno` onto the standard stream `stderr`. The string `s` is printed first, then a colon, then the message and a new-line. The string `s` is usually the name of the program which called *perror*.

perror should only be called when a function which sets `errno` indicates an error has occurred since `errno` is not cleared upon successful execution.

The messages printed are stored in the array `sys_errlist` and may be indexed by −`errno` (this is ***not*** compatible with UNIX where `errno` is always positive). The number of entries in `sys_errlist` is stored in `sys_nerr`.

NAME

printf, fprintf, sprintf, _fprintf, _sprintf — print formatted output

SYNOPSIS

```
#include <stdio.h>

int printf(format [ , arg ] . . . )
    char *format;

int fprintf(stream, format [ , arg ] . . . )
    FILE *stream;
    char *format;

int sprintf(s, format [ , arg ] . . . )
    char *s, *format;

int _fprintf(stream, format, args)
    FILE *stream;
    char *format, *args;

int _sprintf(s, format, args)
    char *s, *format, *args;
```


DESCRIPTION

printf places output on the standard output stream **stdout**.

fprintf places output on the named output stream.

sprintf places "output", followed by a null character (\0) in consecutive bytes starting at *s; it is the user's responsibility to ensure that enough storage is available.

_sprintf works like *sprintf* except the arguments are retrieved from the pointer **args** (which normally points into the stack).

_fprintf is like *fprintf* except the arguments are retrieved from the pointer **args**.

Each function returns the number of characters transmitted (not including \0 for *sprintf*), or a negative value if an output error was encountered.

Each of these functions converts, formats, and prints its args under control of the format. The format is a character string that contains two types of objects: plain characters, which are simply copied into the output stream, and conversion specifications, each of which results in fetching of zero or more args. The results are undefined if there are insufficient args for the format. If the format is exhausted while args remain, the excess args are simply ignored.

Each conversion specification is introduced by the character `%`. After the `%`, the following appear in sequence:

- An optional flag which modifies the meaning of the conversion specification.

- An optional decimal digit string specifying a minimum field width. If the converted value has fewer characters than the field width, it will be padded on the left (or right, if the left-adjustment flag has been given), with spaces, to the field width. A leading zero indicates zeros should be used instead of spaces.

- A precision which gives the maximum number of characters to be printed from a string, or the number of digits to be printed to the right of the decimal point for float or double.

- An optional `l` specifying that a following d, o, u, or x conversion character applies to a long integer arg.

- A character that indicates the type of conversion to be applied.

15

The only flag character is the minus sign (`-`). When used, the result of the conversion will be left-justified within the field.

A field width or precision may be '*' instead of a digit string. In this case an extra integer argument provides the field width or precision.

The conversion characters and their meanings are:

d,o,u,x The integer arg is converted to signed decimal, unsigned octal, decimal, or hexadecimal notation respectively; the letters `abcdef` are used for x conversion.

`f` The float or double arg is converted to decimal notation in the style:

[-]⟨digits⟩.⟨digits⟩

where the number of digits after the decimal point is equal to the precision specification. If the precision is missing, six (6) digits are output; if the precision is zero (0), no decimal point appears.

e, g The float or double arg is converted to the style:

[-]⟨digit⟩.⟨digits⟩E(+|-)⟨digits⟩

where there is one digit before the decimal point and the number of digits after it is equal to the precision; when the precision is missing, six (6) digits are output; if the precision is zero (0), no decimal point appears.

c The character arg is printed.

s The arg is taken to be a string (character pointer) and characters from the string are printed until a null character (\0) is encountered or the number of characters indicated by the precision specification is reached. If the precision is missing, it will be taken to be infinite, so all characters up to the first null character are printed. A null arg will yield undefined results.

% Print a %; no argument is converted.

In no case does a non-existent or small field width cause truncation of a field; if the result of the conversion is wider than the field width, the field is simply expanded to contain the conversion result. Characters generated by *printf* and *fprintf* are printed as if *putc* had been called.

NOTE

_sprintf and *_fprintf* are not standard UNIX functions.

_sprintf and *_fprintf* allow user defined functions to have the functionality of *printf*. The following example demonstrates:

EXAMPLE

```
int dprintf(format, args)           /* Debug printf */
    char *args;
    char *format;
{
    if (DEBUG) {
        printf("*** DEBUG: ");
        _fprintf(stdout, format, &args);
    }
}
```

NAME

putc, putchar, fputc, putw — put a character or word on a stream

SYNOPSIS

```
#include <stdio.h>

int putc(c, stream)
    char  c;
    FILE *stream;

int putchar(c)
    char c;

int fputc(c, stream)
    char  c;
    FILE *stream;

int putw(w, stream)
    int   w;
    FILE *stream;
```

15

DESCRIPTION

putc writes the character c to the output stream at the current pointer position. putchar(c) is defined as putc(c, stdout). *putc* and *putchar* are both macros.

fputc is similar to *putc* but it is a true function. It is slower but takes less space per invocation.

putw writes the word (integer) w to the output stream at the current pointer position. *putw* does not force even alignment on the file.

DIAGNOSTICS

If successful, the value written is returned.

If unsuccessful, EOF is returned. This can occur if the file is not open for writing or if the output file cannot be grown.

Because EOF is a valid integer, *ferror* should be used to check for error when using *putw*.

NAME

puts, fputs — put a string on a stream

SYNOPSIS

```
#include <stdio.h>

int puts(s)
    char *s;

int fputs(s, stream)
    char *s;
    FILE *stream;
```

DESCRIPTION

puts writes the null-terminated string, pointed to by `s`, to the standard output stream `stdout`. The string is followed by a new-line character.

fputs writes the null-terminated string, pointed to by `s`, to `stream`. The string is not followed by a new-line character.

Neither function writes out the terminating null character.

DIAGNOSTICS

EOF is returned if an error occurs. This will happen if output is attempted to a file not open for writing.

NAME

qsort — a quicker sort.

SYNOPSIS

```
qsort(base, nelem, width, compare)
    char  *base;
    int    nelem, width;
    int   (*compare)();
```

DESCRIPTION

qsort is an implementation of the quicksort algorithm. The first parameter is a pointer to the base of the data. The second parameter **nelem** is the number of elements in the array. The third parameter **width** is the width of each element in bytes. The last parameter **compare** is a pointer to the comparison routine to be called. This user-defined function will be passed two arguments which are pointers to the elements being compared. This routine must return an integer less than, equal to, or greater than 0 accordingly as the first argument is to be considered less than, equal to, or greater than the second.

NOTE

The quicksort algorithm used is recursive.

EXAMPLE

```
#include <stdio.h>

int test(a, b)
int *a, *b;
{
    return *a - *b;
}

main()
{
    int x[100], i;

    for (i=0; i<100; i++)        /* Create some random data */
        x[i] = rand();

    qsort(x, 100, sizeof(int), test);

    for (i=0; i<100; i++)        /* Display sorted result */
        printf("%d ", x[i]);

    puts("Press RETURN to continue"); getchar();
}
```

NAME

rand, srand — simple random-number generator

SYNOPSIS

```
#include <stdio.h>

int rand()

srand(seed)
    long seed;
```

DESCRIPTION

rand uses a multiplicative congruential random-number generator.

srand can be called at any time to reset the random-number generator to a new starting point. The generator is initially seeded with a value of 1.

NOTE

rand and *srand* are both macros defined in `<stdio.h>`.

NAME

read — read from a file

SYNOPSIS

```
int read(fildes, buf, nbyte)
    int      fildes;
    char     *buf;
    unsigned nbyte;
```

DESCRIPTION

read attempts to read **nbytes** bytes from the file associated with **fildes** into the buffer pointed to by **buf**.

fildes is a file descriptor obtained by using an *open* or *creat*.

A value of 0 is returned when an EOF is reached.

read will fail if **fildes** is not a valid file descriptor open for reading or if an operating system error occurs.

If the O_BINARY flag is not set then line-feed/carriage return combinations are translated to line-feeds (except from the keyboard).

DIAGNOSTICS

If successful, a non-negative integer is returned indicating the number of bytes actually read.

If unsuccessful, a -1 is returned and **errno** is set appropriately.

NAME

rename — change the name of a file.

SYNOPSIS

```
int rename(from, to)
    char *from, *to;
```

DESCRIPTION

rename is used to change the existing name of a file on a disk to another name. The `from` parameter is a pointer to the name of the current file on disk. The `to` parameter is a pointer to the new name for the file.

DIAGNOSTICS

If unsuccesful, `errno` is set and −1 is returned.

NAME

sbrk, lsbrk — change data segment space allocation

SYNOPSIS

```
char *sbrk(incr)
    int incr;

char *lsbrk(incr)
    long incr;
```

DESCRIPTION

sbrk requests `incr` bytes of additional memory from the operating system and returns a pointer to the block. The request is limited to 32K since `incr` is a signed integer.

lsbrk is like *sbrk* except a long value is passed allowing for far greater allocations.

Memory allocated by *sbrk* and *lsbrk* may not be returned to the system and remains allocated until the program terminates.

15

NOTE

This is not compatible with UNIX. In particular, blocks returned by sequential calls to *sbrk* or *lsbrk* are not guaranteed to be adjacent in memory. This is due to the memory management scheme employed by the Atari operating system.

DIAGNOSTICS

If successful, *sbrk* returns a pointer to the additional memory.

If unsuccessful, a -1 is returned and `errno` is set appropriately.

SEE ALSO

malloc, *Malloc*

NAME

scanf, fscanf, sscanf — convert formatted input

SYNOPSIS

```
#include <stdio.h>

int scanf(format [  , pointer ] . . . )
    char *format;

int fscanf(stream, format [  , pointer ] . . . )
    FILE *stream;
    char *format;

int sscanf(s, format [  , pointer ] . . . )
    char *s, *format;
```

DESCRIPTION

scanf reads from the standard input stream `stdin`.

fscanf reads from the named input stream.

sscanf reads from the character string `s`.

Each function reads characters, converts them according to a format, and stores the results in its arguments. The arguments consist of a control string format and a set of pointer arguments indicating where the converted input should be stored.

The control string may contain:

- White-space characters (blanks, tabs, and new-lines) which cause input to be read up to the next non white-space character.
- An ordinary character (not `%`), which must match the next character of the input stream.
- Conversion specifications, consisting of the character `%`, an optional assignment suppressing character `*`, an optional numerical maximum field width, an optional `l` indicating the size of the receiving variable, and a conversion code.

A conversion specification directs the conversion of the next input field; the result is placed in the variable pointed to by the corresponding argument, unless assignment suppression was indicated by *. The suppression of assignment provides a way of describing an input field which is to be skipped. An input field is defined as a string of non-white-space characters; it extends to the next inappropriate character or until the field width, if specified, is exhausted.

The conversion code indicates the interpretation of the input field. For a suppressed field, no pointer argument should be given. The following conversion codes are legal:

% a single **%** is expected in the input at this point; no assignment is done.

d a decimal integer is expected; the corresponding argument should be an integer pointer.

h a short decimal integer is expected, the corresponding argument should be a short pointer.

o an octal integer is expected; the corresponding argument should be an integer pointer.

x a hexadecimal integer is expected; the corresponding argument should be an integer pointer.

e,f,g a floating point number is expected; the next field is converted accordingly and stored through the corresponding argument, which should be a pointer to a float. The input format for floating point numbers is an optionally signed string of digits, possibly with a decimal point, followed by an optional exponent field consisting of an **e**, or an **E** followed by an optionally signed integer.

s a character string is expected; the corresponding argument should be a character pointer pointing to an array of characters large enough to accept the string and a terminating \0, which will be added automatically. The input field is terminated by a white-space character.

c a character is expected; the corresponding argument should be a character pointer. The normal skip over white space is suppressed in this case; to read the next non-space character, use **1s**. If a field width is given, the corresponding argument should refer to a character array; the indicated number of characters is read.

The conversion characters `d`, `o`, and `x` may be preceded by `l` to indicate that a pointer to long rather than int is in the argument list. Also, the conversion characters `e`, `f`, and `g` may be preceded by `l` to indicate that a pointer to double rather than to float is in the argument list.

scanf conversion terminates at `EOF`, at the end of the control string, or when an input character conflicts with the control string. In, the latter case, the offending character is left unread in the input stream.

scanf returns the number of successfully matched and assigned input items; this number can be zero in the event of an early conflict between an input character and the control string. If the input ends before the first conflict or conversion, EOF is returned.

DIAGNOSTICS

These functions return EOF on end of input and a short count for missing or illegal data items.

NOTE

Trailing white space (including a new-line) is left unread unless matched in the control string.

NAME

setbuf, setbuffer, setlinebuf — assign buffering to a stream

SYNOPSIS

```
#include <stdio.h>

setbuf(stream, buf)
    FILE *stream;
    char *buf;

setbuffer(stream, buf, bufsize)
    FILE *stream;
    char *buf;
    int   bufsize;

setlinebuf(stream)
    FILE *stream;
```

DESCRIPTION

Three types of buffering are available: unbuffered, block buffered, and line buffered. When an output stream is unbuffered, information appears on the destination file or terminal as soon as written; when it is block buffered many characters are saved up and written as a block; when it is line buffered characters are saved up until a newline is encountered. Normally all files are block buffered.

15

setbuf is used after a stream has been opened but before it is read or written. It causes the character array pointed to by `buf` to be used instead of an automatically allocated buffer. If `buf` is a NULL character pointer then input/output will be completely unbuffered. A constant `BUFSIZ`, defined in the `<stdio.h>` header file, tells how big an array is needed.

```
char buf[BUFSIZE];
```

setbuffer is used to set up a user defined I/O buffer whose size is determined by the parameter `bufsize`. If `buf` is NULL the I/O buffer will be completely unbuffered. Note that this function should only be used after a stream has been opened, but before it has be read or written.

setlinebuf is used to change `stdout` or `stderr` from block buffered or unbuffered to line buffered. Unlike *setbuf* and *setbuffer* it can be used at any time that the file descriptor is active.

NOTE

If the space passed as `buf` cannot be freed (ie. it was not allocated by *malloc*), then the stream must be set to unbuffered before closing.

NAME

setjmp, longjmp — non-local goto

SYNOPSIS

```
#include <stdio.h>

int setjmp(env)
    jmp_buf env;

longjmp(env, val)
    jmp_buf env;
    int     val;
```

DESCRIPTION

These functions are useful for dealing with errors and interrupts encountered in a low-level subroutine of a program.

setjmp saves its stack environment in **env** (whose type, **jmp_buf**, is defined in the **<stdio.h>** header file), for later use by *longjmp*. It returns the value 0.

longjmp restores the environment saved by a call of *setjmp* with the same **env** argument. After *longjmp* is called, program execution continues as if the corresponding call of *setjmp* had just returned the value **val**. *longjmp* cannot cause *setjmp* to return the value 0. If *longjmp* is invoked with a second argument of 0, *setjmp* will return 1. All accessible data have values as of the time *longjmp* was called.

NOTE

If *longjmp* is called when **env** was never primed by a call to *setjmp*, or when the last such call is in a function which has since returned, something bogus will happen.

EXAMPLE

```
#include <stdio.h>

jmp_buf env;

deeply_nested_function()
{
```

15

```
    if ( (p = malloc(30)) == NULL)
        longjmp(env, 1);    /* Out of memory */
    . . .
}

main()
{
    if (setjmp(env))  {
        cleanup();          /* Come back here on fatal error */
        exit(1);
    }
    . . .
}
```

NAME

strcat, strncat, xstrcat, strcmp, strncmp, strcpy, xstrcpy, strncpy, xstrncpy, strlen, index, rindex — string operations

SYNOPSIS

```
#include <string.h>

char *strcat(s1, s2)
char *xtrcat(s1, s2)
char *strncat(s1, s2, n)
int   strcmp(s1, s2)
int   strncmp(s1, s2, n)
char *strcpy(s1, s2)
char *xtrcpy(s1, s2)
char *strncpy (s1, s2, n)
char *xtrncpy(s1, s2, n)
    char *s1, *s2;
    int n;

int  strlen(s)
    char *s;

char *index(s, c)
char *rindex(s, c)
    char *s, c;
```

15

DESCRIPTION

The arguments `s1`, `s2`, and `s` point to strings (arrays of characters terminated by a null character). The functions *strcat*, *xstrcat*, *strncat*, *strcpy*, *xstrcpy*, *strncpy*, and *xstrncpy* all alter `s1`. These functions do not check for overflow of the array pointed to by `s1`.

strcat appends a copy of string `s2` to the end of string `s1` and returns `s1`.

xtrcat appends but returns a pointer to the end of `s1` (pointing at the `null` byte).

strncat appends at most `n` characters.

strcmp compares its arguments and returns an integer less than, equal to, or greater than 0 according as `s1` is lexicographically less than, equal to, or greater than `s2`.

strncmp makes the same comparison but looks at, at most, **n** characters.

strcpy copies string **s2** to **s1**, stopping after the null character has been copied. The result is **s1**.

xtrcpy copies but returns a pointer to the end of **s1**.

strncpy copies exactly **n** characters, truncating **s2** or adding null characters to **s1** if necessary. The result will not be null-terminated if the length of **s2** is **n** or more.

xtrncpy copies like *strncpy*, but returns a pointer to the end of **s1**.

strlen returns the number of characters in **s**, not including the terminating null character.

index (*rindex*) returns a pointer to the first (last) occurrence of **c** in string **s**. NULL is returned if **c** is not in **s**.

NOTE

All of the string functions are declared in the **<string.h>** header file.

NAME

strtol, atol, atoi — convert string to integer

SYNOPSIS

```
long strtol(str, ptr, base)
    char  *str;
    char **ptr;
    int    base;

long atol(str)
    char *str;

int atoi(str)
    char *str;
```

DESCRIPTION

strtol returns as a long integer the value represented by the character string `str`. The string is scanned up to the first character inconsistent with the base. Leading white-space characters are ignored.

If the value of `ptr` is not `(char **)NULL`, a pointer to the character terminating the scan is returned in `*ptr`. If no integer can be formed, `*ptr` is set to `str`, and zero is returned.

If `base` is positive and not greater than 36, it is used as the base for conversion. After an optional leading sign, leading zeros are ignored, and "0x" or "0X" is ignored if base is 16.

Truncation from `long` to `int` can take place upon assignment or by an explicit cast.

atol takes the ASCII representation of a number and converts it into a long integer.

atoi takes the ASCII representation of a number and converts it into an integer.

SEE ALSO

atof

15

NAME

ungetc — push a character back into the input stream

SYNOPSIS

```
#include <stdio.h>

int ungetc(c, stream)
    char  c;
    FILE *stream;
```

DESCRIPTION

ungetc inserts the character `c` into the buffer associated with an input stream. `c` will be returned by the next read from that stream. `c` is returned and the stream is left unchanged.

`c` can be read by *getc*, *getchar*, *fread*, *gets*, *fgets*, *fgetc*, *fscanf*, and *scanf*.

One character pushback is guaranteed provided that something has been read from the stream.

If `c` equals EOF, *ungetc* does nothing to the buffer and returns EOF.

fseek erases all memory of inserted characters.

DIAGNOSTICS

A read must be performed prior to the *ungetc*.

EOF is returned if *ungetc* cannot insert the character.

NAME

unlink — remove a directory entry

SYNOPSIS

```
int unlink(fname)
    char *path;
```

DESCRIPTION

unlink removes the directory entry pointed to by **fname**.

The named file is unlinked unless the operating system returns an error (see **errno**).

DIAGNOSTICS

If successful, a zero is returned.

If unsuccessful, a −1 is returned and **errno** is set to indicate the error.

NAME

write — write on a file

SYNOPSIS

```
int write(fildes, buf, nbyte)
    int       fildes;
    char     *buf;
    unsigned  nbyte;
```

DESCRIPTION

write will write **nbyte** bytes from the buffer pointed to by **buf** to the file associated with the **fildes**.

fildes is a file descriptor obtained from a *creat* or *open*.

Writing begins at the current pointer position and is incremented by the number of bytes actually written after returning from write.

write will fail if an operating system error occurs. The pointer position will remain unchanged in this event.

15

If the O_BINARY flag is not set then line feeds are translated to carriage return/line feed combinations (except to the screen).

DIAGNOSTICS

If successful, the number of bytes actually written is returned.

If unsuccessful, -1 is returned and **errno** is set appropriately.

Chapter 16

GEM AES

Introduction

AES stands for "Application Environment Services". It consists of a series of subroutines that handle displaying, creating and maintaining windows, dialog boxes, menu bars and other "high-level" objects. AES is composed of several parts:

- subroutine libraries divided into various "managers"
- a kernel with limited multi-tasking support
- a desk accessory buffer
- a menu and alert buffer

The Screen Manager

The screen manager monitors the actions of the mouse outside the work area of the active window and reports them as events to the process owning the active (or top) window. The work area of a window is the area excluding the title, information line, scroll bars, close box etc. The screen manager intercepts mouse events and reports high-level events to the application (such as window redraw, menu selected and window scroll). An application can get the raw mouse events itself by calling *wind_update* with a value of `BEG_MCTRL`. Unfortunately it cannot then pass the mouse event on to the Screen Manager if the application decides it really didn't want the mouse event.

The Kernel

The kernel allows a total of six desk accessories and one main application.

The dispatcher portion of the kernel controls the execution of processes to ensure none monopolize the system. This is done by assigning each of the processes – such as the Screen Manager, the primary application, or background processes – to one of two lists.

The two lists are the ready and not-ready. If an application is waiting for an event such as a keystroke, a mouse button press, mouse movement, a message, or time passage, it is assigned to the not-ready list. If a process is presently ready to run it is then assigned to the ready list. The ready list will execute in order.

The process dispatcher runs non-preemptively: it is only executed when an application makes a GEM call. Because of this, processes should make sure some GEM routine is called periodically, even if it serves no other purpose than to run the dispatcher.

Desk accessory buffer

The desk accessory buffer is just the section of memory that contains the desk accessory programs. Desk accessories are executable files ending with the extension "`.ACC`". They are loaded into memory and executed when the GEM desktop program starts at boot time. A desk accessory starts by performing any required initialization, including a *menu_register* call which places the accessory's name in the desk menu, and then calling *event_multi* which waits for an event. At this point GEM desktop takes over and loads the next desk accessory. A desk accessory will be re-started when the user selects its name in the desk menu. This causes an `AC_OPEN` event to be sent to the accessory.

Menu/Alert buffer

The menu/alert buffer is just a section of RAM used to hold the part of the screen bit map covered up by a pull-down menu or alert box so that the screen can be restored when the menu or alert box goes away. The buffer can hold 1/4 of the screen's bit map (which places a maximum size on a menu or alert box).

The area of the screen covered up by a window is not saved, so the screen manager sends redraw events to applications when a window is moved or closed. This is somewhat slower than the buffer method, but doesn't require nearly as much memory.

GEM AES interface

GEM AES is implemented as a series of functions divided into "managers". The functions are defined in the standard C library; but the code that actually performs the operation is in the ROM. The library functions merely translate from the conventional C style function call to the somewhat unusual method employed by the ROM. A C programmer normally need not concern himself with the internal mechanisms, however it is sometimes necessary to know these things.

All data passed to and received from the AES ROM routines are sent through six global arrays. These arrays are defined in the standard C library and are automatically included in any program using the AES functions. The only parameter passed to the ROM is a pointer to a struct of pointers to these six arrays. The arrays are defined as follows:

```
int  control[C_SIZE+1], global[G_SIZE+1];
int  int_in[I_SIZE+1], int_out[O_SIZE+1];
long addr_in[AI_SIZE+1], addr_out[AO_SIZE+1];
```

The constants are defined in "GEMBIND.H".

The **global** array is used as follows:

global[0]	The version of GEM AES, pre-set.
[1]	The largest number of applications the version of AES can support concurrently.
[2]	The application ID, set by AES upon invoking the application.
[3-4]	A LONG value which can be set and used as the application desires.
[5-6]	A LONG address which points to the array of tree addresses initialized by *rsrc_load*
[7-15]	Reserved for use by AES.

A program may open multiple resource files by saving and restoring **global[5]** and **[6]**. This pointer is used by *rsrc_gaddr*.

The other arrays don't have anything useful for the C programmer in them.

16.1 Creating a GEM Application

A GEM application must first call *appl_init*. This function sets up any application specific data structures and returns an application ID **ap_id**. This ID is placed in the **global** array and is used by AES to identify the application. *appl_exit* must be called before the program exits.

The example program developed in this section displays the dialog shown in figure 16.1 when "About test..." is choosen in the Desk menu.

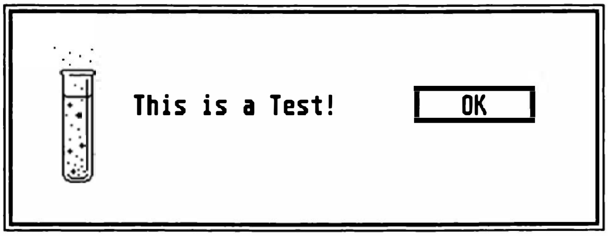


Figure 16.1: Example Dialog

```
int ap_id;          /* Application ID */

main()
{
    ap_id = appl_init();
    . . .
    appl_exit();
}
```

Select the Application's Resource File(s)

The specification of certain graphical/textual objects is kept on disk in a file called a resource file instead of being hard coded within the program. Typically the menu bar, dialog boxes, and icons are stored this way. A resource file will have a ".RSC" extension.

Screen resolutions are 640×400 (monochrome), 640×200 (four color), and 320×200 (sixteen color). It may be necessary to maintain several resource files for icons because of the difference in the screen's aspect ratio; one file for high-res (monochrome) mode and low-res, and one file for medium-res. The aspect ratio for low and high resolution modes are the same.

If knowledge of the screen's resolution is required prior to loading the proper resource file, it may be obtained from the GEMDOS function *Getrez*.

Since the example has an icon, two ".RSC" files will be used. **phys_handle** isn't used in this example, but is typically needed in larger GEM programs.

```
#include <osbind.h>

int gr_hwchar, gr_hhchar;   /* Size of a character cell (in pixels) */
int gr_hwbox, gr_hhbox;     /* Size of box big enough to hold a character */
int resolution;             /* 0=320x200, 1=640x200, 2=640x400 */
```

```
int phys_handle;
    . . .
    phys_handle = graf_handle(&gr_hwchar, &gr_hhchar, &gr_hwbox, &gr_hhbox);
    resolution = Getrez();
    . . .
```

Load the Resource File(s)

To load the resource file, a call is made to *rsrc_load*. This causes the Resource Manager to allocate memory for the resource file, load it, and set up internal pointers.

```
    . . .
    if (resolution == 1)
        rsrc_load("testmed.rsc");  /* Medium rez. */
    else
        rsrc_load("testhigh.rsc"); /* Used for both high and low */
    . . .
```

Obtain Resource Addresses

After *rsrc_load* has been performed, a call to *rsrc_gaddr* is used to return the address of any OBJECT contained in the resource file. Typically symbolic names created by the Resource Construction Program (RCP) are used. These names are defined in a ".H" file created by the RCP (called "test.h" in this example).


```
#include <obdefs.h>
#include "test.h"                /* Header file from RCP */
. . .
    OBJECT *about;               /* Out of paper dialog */
    int x, y, w, h;
    . . .
    rsrc_gaddr(0, TEST, &about);          /* Get address of dialog */
    form_center(about, &x, &y, &w, &h);   /* Center dialog on screen */
    objc_draw(about, 0, 10, x, y, w, h);  /* Draw the dialog */
    form_do(about, 0);                    /* Wait for OK button */
    . . .
```

Capturing an Event

After all initializations are complete, a GEM program goes into its main event loop. This is just a `do ... while` loop that waits for an event, processes it and loops back for the next event. Events are caused by user actions, such as selecting a menu item, pressing a key or moving a window. The event driven method avoids "modes" because the user is always able to do anything (dialog boxes are an exception though).

The function *evnt_multi* will wait for any type of event that can be created. The example only deals with message events so *evnt_mesag* is used instead.

Example

Here is the complete example:

```
/*
 * A simple GEM application
 */
#include <stdio.h>
#include <osbind.h>
#include <obdefs.h>          /* Defines Object Manager symbols */
#include <gemdefs.h>         /* Defines other GEM AES symbols */
#include "test.h"            /* Header file from RCP */

/*
 * Some global variables
 */
int gr_hwchar, gr_hhchar;    /* Size of a character cell (in pixels) */
int gr_hwbox, gr_hhbox;      /* Size of box big enough to hold a character */
int resolution;              /* 0=320x200, 1=640x200, 2=640x400 */
int ap_id;                   /* Application ID */
int phys_handle;
OBJECT *menubar;
int quit;                    /* 1=exit from event loop */

about()
/*
 * Displays the "About test..." dialog box.
 */
{
    OBJECT *about;
    int x,y,w,h;    /* Location and size of dialog box on screen */

    rsrc_gaddr(0, TEST, &about);              /* Get address of dialog */
    form_center(about, &x, &y, &w, &h);       /* Center on screen */

    form_dial(FMD_START, 0,0,0,0, x,y,w,h); /* Reserve screen space */

    objc_draw(about, 0, 10, x,y,w,h);         /* Draw it */
    form_do(about 0);            /* Waits for an exitable item (OK) */

    /* De-hilite the OK button for the next time it is displayed */
    objc_change(about, ABOUTOK, 0, x,y,w,h, NORMAL, 0);

    form_dial(FMD_FINISH, 0,0,0,0, x,y,w,h); /* Release screen space */
```

16

```
}

main()
{
    int message[8];

    ap_id = appl_init();

    phys_handle = graf_handle(&gr_hwchar, &gr_hhchar, &gr_hwbox, &gr_hhbox);
    resolution = Getrez();

    /* Load resource file */
    if (resolution == 1)
        rsrc_load("testmed.rsc");  /* Medium rez. */
    else
        rsrc_load("testhigh.rsc"); /* Used for both high and low */

    /* Display the menu bar */
    rsrc_gaddr(0, MENU, &menubar);
    menu_bar(menubar, 1);
    graf_mouse(ARROW, 1);        /* Change to arrow from bumble bee */

    do {                         /* Enter main event loop */
        evnt_mesag(message);

        switch (message[0]) {
          case MN_SELECTED:
            switch (message[3]) {
              case MNDESK:
                about();
                break;

              case MNFILE:       /* Only "Quit" appears in the File menu so */
                quit = 1;        /*   I don't have to look at the item no. */
                break;
            }

            /* De-hilite the menu title */
            menu_tnormal(menubar, message[3], 1);
            break;
        }
    } while (!quit);

    menu_bar(menubar, 0);        /* Clear menu bar */
    rsrc_free();                 /* Release resource file's memory */
    appl_exit();
}
```

A more complete GEM example may be found in the EXAMPLES folder of the WORK disk.

16

16

16.2 Applications Manager

appl_init	returns application ID and initializes GEM for the application.
appl_read	reads a specific number of bytes from the event managers buffer.
appl_write	writes a specific number of bytes from the event managers buffer.
appl_find	finds another application's ID.
appl_tplay	plays back a series of AES recorded events.
appl_trecord	records a series of user interactions with AES.
appl_exit	exits a session with the application manager.

Introduction

The Applications Manager is a set of routines designed to communicate with the operating system and other applications.

NAME

appl_exit — GEM AES cleanup

SYNOPSIS

```
int appl_exit()
```

DESCRIPTION

appl_exit is used when an AES application is about to shut down. This function cleans up the GEM environment freeing AES related data structures as well as restoring the machine to its state before the start of the application.

NOTE

A call to this function does not terminate the execution of the program.

DIAGNOSTICS

The result of the function is 0 if an error occurs.

NAME

appl_find — find the ID of another application

SYNOPSIS

```
int appl_find(ap_fpname)
    char *ap_fpname;
```

DESCRIPTION

appl_find allows an application to obtain the ID of another application in order to communicate with it. This is done by passing an 8 character string which contains the file name of the application being looked for in the parameter `ap_fpname`. The string must be padded with blanks to make it 8 characters in length. If the application is found, its application ID will be returned as the result of the function.

DIAGNOSTICS

The result of the function is -1 if an error occurs.

SEE ALSO

appl_read, *appl_write*

NAME

appl_init — initialize the application

SYNOPSIS

```
int appl_init()
```

DESCRIPTION

appl_init initializes internal GEM AES arrays. If the application's initialization was successful a positive application ID is returned as the result of the function.

DIAGNOSTICS

The result of the function is −1 if an error occurs.

SEE ALSO

appl_exit

NAME

appl_read — reads a number of bytes from a message pipe.

SYNOPSIS

```
int appl_read(appl_id, length, buff)
    int   appl_id;
    int   length;
    char *buff;
```

DESCRIPTION

appl_read reads a message sent from another active application whose ID is specified by the parameter `appl_id`. The parameter `length` indicates the number of bytes to be read from the message pipe and the pointer `buff` tells the function where the data is to be placed.

DIAGNOSTICS

The result of the function is zero if an error occurs.

SEE ALSO

appl_init, *appl_write*

NAME

appl_tplay — replays a portion of a record of user actions

SYNOPSIS

```
int appl_tplay(ap_tpmem, ap_tpnum, ap_tpscale)
    char *ap_tpmem;
    int   ap_tpnum;
    int   ap_tpscale;
```

DESCRIPTION

appl_tplay replays user events that were recorded through a call to the AES function *appl_trecord*. The parameter **ap_tnum** contains the number of user events that are defined in the buffer pointed to by the parameter **ap_tpmem**. The last parameter **ap_tpscale** is a speed factor which determines the rate at which the user's events will be played back. The values for this parameter range from 1 to 10,000.

50	=	Half speed
100	=	Full speed
200	=	Twice speed

16

DIAGNOSTICS

The result of this function is always 1.

SEE ALSO

appl_trecord

NAME

appl_trecord — records user actions

SYNOPSIS

```
int appl_trecord(ap_trmem, ap_trcount)
    char *ap_trmem;
    int   ap_trcount;
```

DESCRIPTION

appl_trecord records up to **ap_trcount** user actions. These actions can later be replayed by *appl_tplay*. The parameter **ap_trmem** is a pointer to a buffer where the user event messages will be stored. Note that there should be approximately 6 times **ap_trcount** bytes available in the user event buffer. The last parameter **ap_trcount** contains the number of events to record.

Each user action is stored in two parts: two bytes that define the action and four bytes that describe the action. The result of the function is the number of actions actually recorded.

Two byte code for the event:

0x0000	timer event
0x0001	button event
0x0002	mouse event
0x0003	keyboard event

The next four bytes store information dependent upon the event:

timer — the number of milliseconds elapsed

button event — The low word is the button state.

0 = button up
1 = button down

The high word is the number of clicks.

mouse event — low word = mouse x-coordinate
high word = mouse y-coordinate

keyboard event — the high word is the keyboard state, the low word is the character.

SEE ALSO

appl_tplay

16

NAME

appl_write — write a number of bytes to a message pipe

SYNOPSIS

```
int appl_write(appl_id_wid, length, buff)
    int   appl_id;
    int   length;
    char *buff;
```

DESCRIPTION

appl_write sends a message event to another application whose ID is specified by the parameter `appl_id`. The parameter length indicates the number of bytes to be placed in the message pipe, and the pointer `buff` points to the data that is to be placed in the message pipe.

NOTE

This routine is useful for posting message events to the application running. For a complete description of message types refer to page 179. Also, this routine is useful for creating application defined message events.

DIAGNOSTICS

16

The result of the function is zero if an error occurs.

SEE ALSO

appl_read, Event Manager Introduction (pg. 179)

16.3 Event Manager

evnt_keybd	waits for keyboard event
evnt_button	waits for a mouse button event
evnt_mesag	waits for a message event
evnt_timer	waits for a timer event
evnt_multi	waits for any of multiple events
evnt_dclick	sets and obtains the double clicking speed

Introduction

In an attempt to alleviate the need for time consuming polling of inputs, GEM provides routines which allow the operating system to look for many inputs (events), and to activate the application when an input occurs.

Mouse Button Event

The mouse button event occurs when the following equation is true:

$$(\text{current\_state AND mask}) = \text{desired\_state}$$

The LSB is the left-most mouse button. The current state of 01_2 would indicate the left mouse button is pressed. The mask sets the buttons the application is interested in looking at. So, a mask of 10_2 would only look at the second mouse button. The final variable (desired_state) is what is being looked for; a value of 01_2, would look for the left button being pressed.

It is also possible to look for such things as "double clicks" from the mouse. This is done by specifying the number of clicks necessary in an interval.

Mouse Event

A mouse event occurs when the mouse is either inside or outside a rectangle. This could be used to change the mouse form when the mouse enters a specified rectangle on the screen. The application would be inactive as long as the mouse is in the rectangle, and would be re-activated when the mouse leaves the rectangle.

Message Event

GEM allows many applications to run at the same time. The application with the top window has control of the keyboard and the menus which appear in the menu bar. For many reasons the user may wish to manipulate many items which the application does not know about directly. Such as menu selection and user interaction with the window border. The information from these interactions are passed to the application through a message pipe, resulting in a Message Event.

The messages are stored in the message pipe in first-in-first-out order. A Message Event occurs when the application receives the message, and each message is removed as it is read by the application.

There are several pre-defined types of messages. Each message has a maximum length of 8 words (16 bytes), and all of the pre-defined types use the first three elements of `ev_mgpbuff` in the same manner:

`ev_mgpbuff[0]`	message type, a number
`ev_mgpbuff[1]`	the ap_id of the application originating the message.
`ev_mgpbuff[2]`	the message length in excess of the predefined 16 bytes. The portion beyond 16 bytes can be read by the *appl_read* call.

MN_SELECTED

This message notifies the application a user has selected a menu item.

`ev_mgpbuff[0]`	10
`ev_mgpbuff[3]`	the object index of the menu title selected
`ev_mgpbuff[4]`	the object index of the menu item selected

WM_REDRAW

This indicates part of the window work area must be redrawn due to user action. This is the area of the window other than any border, title bar, or information line.

`ev_mgpbuff[0]`	20
`ev_mgpbuff[3]`	the window handle to be redrawn
`ev_mgpbuff[4]`	the screen coordinate x position of the window area to be redrawn
`ev_mgpbuff[5]`	the screen coordinate y position of the window area to be redrawn
`ev_mgpbuff[6]`	screen coordinate width of the area
`ev_mgpbuff[7]`	screen coordinate height of the area

WM_TOPPED

This tells the application it (the application) has requested its or another application's window to be moved to the top and made active.

`ev_mgpbuff[0]`	21
`ev_mgpbuff[3]`	the handle of the window

WM_CLOSED

This indicates the user wishes the application's window closed.

`ev_mgpbuff[0]`	22
`ev_mgpbuff[3]`	the handle of the window

WM_FULLED

This informs the application that the user has clicked the window full box, thus requesting the window be enlarged to its full size. If the window is at its full size, this is interpreted to restore the window to its previous size.

`ev_mgpbuff[0]`	23
`ev_mgpbuff[3]`	the handle of the window

WM_ARROWED

One of the arrows or scroll bars in the application's window border area has been clicked.

`ev_mgpbuff[0]`	24
`ev_mgpbuff[3]`	the handle of the window
`ev_mgpbuff[4]`	one of the following: 0 — page up 1 — page down 2 — row up 3 — row down 4 — page left 5 — page right 6 — column left 7 — column right

Page actions are from the scroll bars, row and column actions are from the arrows.

WM_HSLID

This informs the application of the new position requested for the horizontal slider.

`ev_mgpbuff[0]`	25
`ev_mgpbuff[3]`	the window handle
`ev_mgpbuff[4]`	the requested slider position (0 left most — 1000 right most)

WM_VSLID

This informs the application of the new position requested for the vertical slider.

`ev_mgpbuff[0]`	26
`ev_mgpbuff[3]`	the handle of the application window
`ev_mgpbuff[4]`	the new position (0 top — 1000 bottom)

WM_SIZED

The user has requested a new window size. The new coordinates given by this message include the following applicable title bar, information line and borders.

`ev_mgpbuff[0]`	27
`ev_mgpbuff[3]`	the handle of the window
`ev_mgpbuff[4]`	the requested X-coordinate, which is normally the present one
`ev_mgpbuff[5]`	the requested Y-coordinate, usually the present one
`ev_mgpbuff[6]`	the requested width
`ev_mgpbuff[7]`	the requested height

WM_MOVED

The user has moved a window. The new coordinates include the applicable of the title bar, information line, and borders.

`ev_mgpbuff[0]`	28
`ev_mgpbuff[3]`	the window handle
`ev_mgpbuff[4]`	the requested x coordinate
`ev_mgpbuff[5]`	the requested y coordinate
`ev_mgpbuff[6]`	the requested window width, should stay the same
`ev_mgpbuff[7]`	the requested window height, should stay the same

WM_NEWTOP

This tells the application that its window has been placed on top and thus made active.

`ev_mgpbuff[0]`	29
`ev_mgpbuff[3]`	the handle of the window just placed on top

AC_OPEN

This is sent to a desk accessory when it has been selected from the Desk Menu.

`ev_mgpbuff[0]`	30
`ev_mgpbuff[3]`	`me_rmenuid` — the desk accessory menu item identifier returned by the *menu_register* call.

AC_CLOSE

This is sent to a desk accessory when all of the following are true:

- the current application has just terminated
- the screen is about to be cleared
- window manager structures are about to be reinitialized.

The desk accessory should then zero any window owned by it.

`ev_mgpbuff[0]`	31
`ev_mgpbuff[3]`	`me_raccmenid` — the desk accessory menu item identifier returned by the *menu_register* call.

Timer Event

If the application desires to wait for a time, it can cause a timer event to be generated after a requested number of milliseconds. The intent is to avoid polling the system clock or other cumbersome methods for timing something.

16

Example

The following example is a typical call to the event manager that is used in the sample AES application supplied with Laser C.

```
#include <gemdefs.h>
#include <obdefs.h>

#include "globals.h"

/*
    Handle Application Events.
*/

TaskMaster()
{
    int event;          /* The event code.                  */

    int button = TRUE;  /* desired Button state             */
    int message[8];     /* Event message buffer.            */
    int mousex, mousey; /* The current mouse position.      */
    int mousebutton;    /* The state of the mouse button    */

    int keycode;        /*  The code for the key pressed.   */
    int keymods;        /*  The state of the keyboard modifiers.
                            (shift, ctrl, etc). */
    int clicks;         /*  The number of mouse clicks that occurred in the
                            given time. */

    do {
        event = evnt_multi(
            MU_MESAG | MU_BUTTON | MU_KEYBD,   /* set messages to respond to. */
            1,              /* Time frame for events.                   */
            1,              /* Keyboard Event mask.                     */
            button,         /* desired key state                        */
            0, 0, 0, 0, 0,  /* rectangle one information (ignored)      */
            0, 0, 0, 0, 0,  /* rectangle two information (ignored)      */
            message,        /* The message buffer                       */
            0, 0,           /* Number of Ticks for Timer event.         */
            &mousex,        /* The x-coordinate of the mouse at event.  */
            &mousey,        /* The y-coordinate of the mouse at event.  */
            &mousebutton,   /* The state of the mouse buttons at event. */
            &keymods,       /* The state of the keyboard modifiers.     */
            &keycode,       /* The key code for the key pressed.        */
            &clicks         /* The number of times the event occurred   */
        );

        if (event & MU_MESAG) {
            switch (message[0]) {
                /*
```

16

```
                    Window Support
                */
                case WM_REDRAW:
                case WM_TOPPED:
                case WM_FULLED:
                case WM_ARROWED:
                case WM_HSLID:
                case WM_VSLID:
                case WM_SIZED:
                case WM_MOVED:
                case WM_NEWTOP:
                case WM_CLOSED:
                    do_window(message);
                break;

                /*
                    Menu Support
                */
                case MN_SELECTED:
                    do_menu(message);
                break;

                /*
                    Desk Accessory Support
                */
                case AC_OPEN:
                case AC_CLOSE:
                break;
            }
        }

        if (event & MU_BUTTON)
            button ^= TRUE;

        if (event & MU_KEYBD)
            do_update(message);
    } while(1);
}
```

16

NAME

evnt_button — waits for a mousedown event

SYNOPSIS

```
int evnt_button(ev_bclicks, ev_bmask, ev_bstate, ev_bmx, ev_bmy,
                ev_bbutton, ev_bkstate)
    int  ev_bclicks;
    int  ev_bmask;
    int  ev_bstate;
    int *ev_bmx;
    int *ev_bmy;
    int *ev_bbutton;
    int *ev_bkstate;
```

DESCRIPTION

evnt_button waits until a mouse down event occurs and then returns information about the mouse through the parameters. It is possible to have this routine respond only to certain mouse buttons and to wait until a certain number of clicks have occurred. The result of the function is the number of times that the button achieved the desired state.

16

ev_bclicks the number of times the mouse button needs to be clicked.

ev_bmask is a mask that allows the application to respond to only certain button events.

0x0001	=	Left mouse button
0x0002	=	Right mouse button

ev_bstate The state of the mouse button to wait for (0 is up, 1 is down). The state is indicated with a bit vector as in **ev_bmask**.

ev_bmx the x-coordinate of where the mousedown event occurred.

ev_bmy the y-coordinate of where the mousedown event occurred.

ev_button The state of the mouse buttons upon exit from the routine (using the same bit vector as **ev_bstate**).

`ev_bkstate`	The keyboard's state upon exit from the routine. If a bit is set then that button has been pressed:

0x0001	=	right shift
0x0002	=	left shift
0x0004	=	ctrl key
0x0008	=	alt key

SEE ALSO

evnt_multi

NAME

evnt_dclick — sets or reads the double-click speed

SYNOPSIS

```
int evnt_dclick(ev_dnew, ev_dgetset)
    int ev_dnew;
    int ev_dgetset;
```

DESCRIPTION

evnt_dclick is used to set or read the double click speed for the mouse. If `ev_dgetest` is one, then a new double-click speed is set. The new double click speed is contained in `ev_dnew`. The speeds range from 0 to 4 where 4 is the fastest. If a read of the double click speed is requested the function returns the current double click speed.

`ev_dnew`	this parameter contains the new double click speed.
`ev_dgetset`	determines whether the value in `ev_dnew` is to be used in setting the double click speed. If it is set to zero then the current speed of the double click is returned through the function and the value of `ev_dnew` ignored.

SEE ALSO

evnt_multi

NAME

evnt_keybd — waits for a keyboard event

SYNOPSIS

```
int evnt_keybd()
```

DESCRIPTION

evnt_keybd function waits for a keyboard event (any keyboard input). The result of this function is the keyboard code for the typed character (refer to page 589).

SEE ALSO

evnt_multi, Keyboard Codes

NAME

evnt_mesag — waits for a message

SYNOPSIS

```
evnt_mesag(ev_mgpbuff)
    int ev_mgpbuff[8];
```

DESCRIPTION

evnt_mesag is used to wait for message events from the system. The parameter `ev_mgpbuff` is a pointer to a 8 word (16 byte) buffer in memory where the message will be placed.

NOTE

The standard event messages are described in the event manager introductory section.

DIAGNOSTICS

The result of the function is always 1.

SEE ALSO

evnt_multi, Event Manager Introduction (pg. 179)

NAME

evnt_mouse — waits for a mouse event

SYNOPSIS

```
int evnt_mouse(ev_moflags, ev_mox, ev_moy, ev_mowidth, ev_moheight,
               ev_momx,  ev_momy,  ev_mobutton,  ev_mokstate)
    int  ev_moflags;
    int  ev_mox;
    int  ev_moy;
    int  ev_mowidth;
    int  ev_moheight;
    int *ev_momx;
    int *ev_momy;
    int *ev_mobutton;
    int *ev_mokstate;
```

DESCRIPTION

evnt_mouse waits for the mouse to enter or leave a specified rectangle. The function is passed the size and position of the rectangle in the parameters **ev_mox**, **ev_moy**, **ev_mowidth**, and **ev_moheight**. The function returns the location and button state of the mouse when the event occurred and stores the results at the locations pointed to by their respective integer pointers.

16

ev_moflags If this flag is 1, a mouse event occurs when it exits the rectangle, otherwise the event occurs when the mouse enters the rectangle.

ev_mox the x-position (in pixels) of the defined rectangle.

ev_moy the y-position (in pixels) of the defined rectangle.

ev_mowidth the width of the defined rectangle in pixels.

ev_moheight the height of the rectangle in pixels.

ev_momx the x-coordinate of the mouse when it entered or exited the rectangle.

ev_momy the y-coordinate of the mouse when it entered or exited the rectangle.

`ev_mobutton`	the state of the mouse button when it entered or exited the rectangle. Each bit represents a mouse button 0-15 from lower order to high. If the bit is set then the button has been pressed (e.g. left button has value 0x0001).
`ev_mokstate`	the status of the keyboard special function keys. If the bit is set then the button has been pressed. They are represented as follows:

0x0001	=	right shift
0x0002	=	left shift
0x0004	=	Ctrl
0x0008	=	Alt

SEE ALSO

evnt_multi

16

NAME

evnt_multi — waits for several possible events

SYNOPSIS

```
int evnt_multi(ev_mflags, ev_mbclicks, ev_mbmask, ev_mbstate,
    ev_mm1flags, ev_mm1x, ev_mm1y, ev_mm1width, ev_mm1height,
    ev_mm2flags, ev_mm2x, ev_mm2y, ev_mm2width, ev_mm2height,
    ev_mmgpbuff, ev_mtlocount, ev_mthicount, ev_mmox, ev_mmoy,
    ev_mmobutton, ev_mmokstate, ev_mkreturn, ev_mbreturn)

    int  ev_mmflags;
    int  ev_mbclicks;
    int  ev_mbmask;
    int  ev_mbstate;
    int  ev_mm1flags;
    int  ev_mm1x, ev_mm1y;
    int  ev_mm1height, ev_mm1width;
    int  ev_mm2flags;
    int  ev_mm2x, ev_mm2y;
    int  ev_mm2height, ev_mm2width;
    int  ev_mtlocount;
    int  ev_mthicount;
    int *ev_mmox, *ev_mmoy;
    int *ev_mmobutton;
    int *ev_mmokstate;
    int *ev_mkreturn;
    int *ev_mbreturn;
    int ev_mmgpbuff[8];
```

DESCRIPTION

This simple function will wait for any of 6 possible events. Which events to wait for are indicated by event mask `ev_mflags` by setting the appropriate bit as below:

Bit	Event
0	keyboard event
1	mouse button event
2	mouse event 1
3	mouse event 2
4	message event
5	timer event

16

Any combination is legal which means to wait for any one of the events.

The event which actually occurred is returned (using the same bit representation as above).

`ev_mbclicks`	A mouse event occurs when the keys of interest, defined by `ev_mbmask` are placed in a state defined by `ev_mbstate`, for a count of `ev_mbclicks` in a time generally specified by the front panel.
`ev_mbask`	This sets the mouse button mask for a mouse event. The mask is ANDed with the present state of the mouse keys and then compared to the desired state. The LSB of this mask filters the value of the leftmost mouse button. A value of 0x0001 in this parameter would allow only the left button to be tested.
`ev_mbstate`	This is the state of the mouse buttons of interest which cause a mouse button event. The bits refer to the keys as above, 0 means mouse up, 1 means mouse down.
`ev_mm1flags`	This sets the mouse event for the first rectangle to be generated upon entry or exit from the rectangle. A zero generates it on entry, and a one generates the event on exit.
`ev_mm1x`, `ev_mm1y`	The x and y coordinates of the first mouse event rectangle.
`ev_mm1width`, `ev_mm1height`	The width and height of the first mouse event rectangle.
`ev_mm2flags`, `ev_mm2x`, `ev_mm2y`, `ev_mm2width`, `ev_mm2height`	These are the parameters for the second mouse event rectangle, and have the same meaning as the first — except they act on the second rectangle.
`ev_mmgpbuff`	This is the 8 word message pipe buffer. Refer to page 179 for a further description of the event messages.
`ev_mtlocount`, `ev_mthicount`	The low and high words used to set the timer.

`ev_mmox`, `ev_mmoy`	The x and y coordinates of the mouse when the mouse event occurred.
`ev_mmobutton`	This contains the state of the mouse buttons when the user event occurred. As above, 0x0002 would indicate the mouse button second from the left was depressed.
`ev_mmokstate`	This returns the state of the following keys when the event occurred:

Bit (LSB = bit 0)	**Key**
0	right shift
1	left shift
2	ctrl
3	alt

`ev_mkreturn`	The keyboard code for the key pressed.
`ev_mbreturn`	This is the number of times the mouse key entered the desired state, within the desired time.

SEE ALSO

evnt_keybd, *evnt_button*, *evnt_mouse*, *evnt_mesag*, *evnt_timer*, *evnt_dclick*

NAME

evnt_timer — waits for a specified time.

SYNOPSIS

```
evnt_timer(low_count, high_count)
    int low_count, high_count;
```

DESCRIPTION

evnt_timer delays for a specified number of milliseconds. The number of milliseconds is defined by a long word which is divided into two parts. The parameter `low_count` contains the low sixteen bits of the long word. The parameter `high_count` contains the upper word of the delay count long word.

DIAGNOSTICS

The function result is always 1.

SEE ALSO

envt_multi

16.4 Form Manager

form_do	monitors user interaction with a form
form_dial	allocates and de-allocates space for dialog boxes
form_alert	makes a alert box, saves screen, redraws screen, etc
form_error	makes an error box, saves screen, redraws screen, etc
form_center	centers a dialog box

Introduction

A form is a means of gathering information from the user. The application may use any of the following methods for querying the user:

Radio buttons — for one response only. All but the selected response are deselected.

Check boxes — all boxes checked are selected.

Editable text — for responses that require text reply.

The form must have at least one exit. Usually two are supplied: an **OK** button and a **CANCEL**. The **OK** is traditionally used to record the information obtained from the form, while **CANCEL** is pressed if the response is to be ignored.

Editable Text Fields

The following keys may be used in the editable text fields:

Left and right arrows, down-arrow, delete, backspace, Tab — The Return and Enter keys end editing of the text field. This happens only if one object in the form has been flagged as a DEFAULT object. If there is no DEFAULT object, the Form Manager ignores any Return or Enter.

Escape — Clears the text edit field.

There are three parts to any text edit field. They are the template, the validation string, and the text. The template is used to format text that appears in the text field, the validation string specifies what may be typed into the field, and the text is typed in by the user or may be a default value. These are created in the Resource Construction Program.

If a character is entered that is not valid according to the validation string, it is ignored unless it is the next invalid character in the template. If this occurs the curser moves to the position immediately following the invalid character.

An example of a field follows in which a period is not a valid character:

```
______.___
```

If the string "test.c" were entered,

```
test      .c__
```

would appear. Or, in the case of a date that is entered "1/3/86" into:

```
__/__/__
```

The result is:

```
1_/3_/86
```

Three special forms exist for interaction with the user. They are the dialog box, alert box, and Error box.

Dialog Boxes

16

The dialog box is basically a generic form and thus, is used when the application requires additional information from the user. It usually contains some text and one or more exit buttons. It may fill the screen if desired and contain a large number of buttons, boxes, and text fields. The dialog box appears on top of the screen and may optionally be centered.

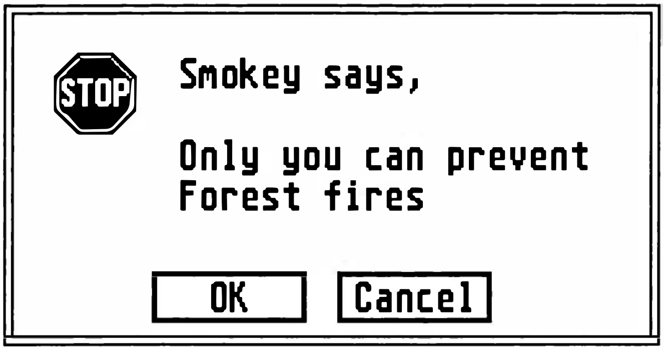
Dialog boxes are resources and are, therefore, created by the Resource Construction Program.

To call a dialog box from an application, the following steps need to be taken:

1. Call *rsrc_gaddr* to get the address of the dialog box object tree.
2. Call *form_dial* to reserve screen space for the dialog box. Call the routine again with `FMD_GROW` set to draw an expanding box.
3. Call *obj_draw* to display the dialog box.
4. Call *form_do* to handle events of the dialog box.
5. Call *form_dial* to free the screen space and to redraw the screen. The routine may be called twice, the first time with `FMD_SHRINK` set to show a shrinking dialog box.

Alert and Error Boxes

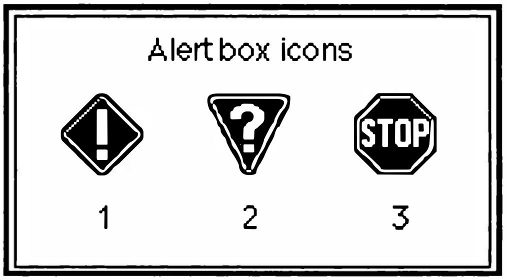
An alert box is used to convey a message to the user for an immediate response. The alert is very easy to handle, you simply call *form_alert* and pass the three required pieces of information. An example follows:




```
[3] [Smokey says,| |Only you can prevent|forest fires][OK|Cancel]
```

The three parts of an alert are:

[⟨icon#⟩] [⟨message text⟩] [⟨exit buttons⟩]



⟨icon#⟩ is a single character that identifies an icon (if any) that appears at the left side of the alert.

0 = no icon
1 = NOTE icon
2 = WAIT icon
3 = STOP icon

⟨message text⟩ is a string consisting of up to 5 lines of 32 characters per line. In the string, the lines are separated by the logical OR symbol "|".

⟨exit buttons⟩ one, two, or three exit buttons; each containing no more that 20 text characters.

In the string, the exit button text is separated by the logical OR symbol.

The area of the screen that is written over by the alert is saved in a buffer and is automatically written back when the alert is exited. The buffer is limited to 25% of the screen size, so this puts a limit on the alert box size.

An error box is just an alert box that receives its text string from the forms manager after a system error occurs.

To display an alert from an application, the following step needs to be performed:

- Call *form_alert*

To display an Error box, do the following:

- Call *form_error*. Pass an operating system error code. A retry or abandon code is returned to the application.

Example

Below is shown a routine which follows the steps in displaying and handling dialogs as described above.


```
#include <gemdefs.h>
#include <obdefs.h>

#include "resource.h"


do_dialog(dialog)
    OBJECT *dialog;
{
    int     x, y, w, h;
    int     itemhit;

    /*
        Center the dialog box.
    */
    form_center(dialog, &x, &y, &w, &h);

    /*
        Reserve screen memory for dialog.
    */
    form_dial(FMD_START, 0, 0, 0, 0, x, y, w, h);
```

```
    /*
        Draw dialog
    */
    objc_draw(dialog, 0, 10, x, y, w, h);

    /*
        Handle Dialog Event.
    */
    itemhit = form_do(dialog, 0);

    /*
        Release reserved screen memory.
    */
    form_dial(FMD_FINISH, 0, 0, 0, 0, x, y, w, h);

    return itemhit;
}
```

NAME

form_alert — is the routine that displays the alert dialog box.

SYNOPSIS

```
int form_alert(fo_adefbttn, fo_astring)
    int   fo_adefbttn;
    char *fo_astring;
```

DESCRIPTION

form_alert displays the alert box, and returns with a number identifying the exit button that was selected by the user. The sequence of steps the routine goes through to display an alert box are as follows:

1. It creates an object tree based upon the alert string that it was given.
2. It saves the screen area that will be taken over by the alert.
3. It calls the *objc_draw* routine to display the alert.
4. It calls the *form_do* routine to let the user respond to the alert.
5. After return from the *form_do* routine the screen is restored, and the exit button that was selected is returned to the application.

16

fo_adefbttn is the form's DEFAULT exit button.

0	=	no DEFAULT exit button
1	=	first exit button
2	=	second exit button
3	=	third exit button

fo_astring the address of the string containing the alert box description. The format of the string is discussed in the Introduction section on Alert Boxes.

SEE ALSO

Form Manager Introduction (pg. 197)

NAME

form_center — centers the dialog box on the screen.

SYNOPSIS

```
form_center(dlog_tree, new_x, new_y, new_w, new_h)
    OBJECT *dlog_tree;
    int    *new_x, *new_y, *new_w, *new_h;
```

DESCRIPTION

form_center takes the OBJECT described by the parameter **dlog_tree** and centers it in relation to the screen boundaries. The OBJECT data structure will be modified to reflect the centering and the new position of the box will be returned in the parameters **new_x, new_y, new_w, new_h**.

dlog_tree	The address of the object tree that describes the dialog.
new_x	the centered x-coordinate of the dialog box.
new_y	the centered y-coordinate of the dialog box.
new_w	the width of the dialog box in pixels.
new_h	the height of the dialog box in pixels.

16

NOTE

Once the dialog box is centered it is not necessary to call the *form_center* function for the dialog again.

DIAGNOSTICS

The result of this function is always 1.

NAME

form_dial — reserves or releases the portion of the screen used for dialog boxes.

SYNOPSIS

```
int form_dial(form_cmd, small_x, small_y, small_w, small_h,
                   big_x, big_y, big_w, big_h)

    int form_cmd;
    int small_x, small_y, small_w, small_h;
    int big_x, big_y, big_w, big_h;
```

DESCRIPTION

form_dial performs the housekeeping functions required for dialog boxes. The four dialog box housekeeping functions are as follows:

`form_cmd` the *form_dial* action being invoked by the current call.

0 (`FMD_START`)	reserves screen space for the dialog box.
1 (`FMD_GROW`)	calls *graf_growbox* to draw an expanding box from small to the large box specified by (`big_`)x, y, `w` and `h`.
2 (`FMD_SHRINK`)	calls *graf_shrinkbox* to draw a shrinking box from the large box to the small box specified by (`small_`)x, y, `w` and `h`.
3 (`FMD_FINISH`)	Releases screen space reserved by `FMD_START`, and causes the application to redraw the screen.

The parameters `small_x`, `small_y`, `small_w`, `small_h` and `big_x`, `big_y`, `big_w`, `big_h` are used with the form commands `FMD_GROW` and `FMD_SHRINK`. The grow and shrink commands call the AES function *graf_growbox* and *graf_shrinkbox* respectively, passing the appropriate set of parameters. The small rectangle for the shrink operation is described by the `small_` parameters. The large rectangle for the grow operation is defined by the `big_` parameters.

DIAGNOSTICS

If an error occurs the result of the function is 0.

16

SEE ALSO

form_do, *graf_growbox*, *graf_shrinkbox*

NAME

form_do — causes the Form Manager to monitor the user's interaction with a form.

SYNOPSIS

```
int form_do(dlog_tree, start_obj)
    OBJECT *dlog_tree;
    int     start_obj;
```

DESCRIPTION

form_do handles the user's interaction with a form (or dialog box). The result of the function is the number of the object that caused the exit from the dialog box.

`dlog_tree`	The address of the form's object tree definition.
`start_obj`	The number of the object (which must be an editable text field) that the application wants active when the form is displayed. The application can pass in a value of 0 if the form does not contain editable text fields.

NAME

form_error — display the error dialog box specified by the DOS Error number parameter.

SYNOPSIS

```
int form_error(error_code)
    int error_code;
```

DESCRIPTION

form_error displays a pre-defined error dialog box specified by the error code number. The result of the function is the number of the exit button. The error dialog is specified by the parameter **error_code**. The pre-defined errors are as follows:

2	=	File not Found Error
3	=	File not Found Error
4	=	Out of Memory Error
5	=	File Exists Error
8	=	Can't Launch application, Out of Memory
10	=	Can't Launch application, Out of Memory
11	=	Can't Launch application, Out of Memory
15	=	Disk Drive does not Exist
16	=	Can't delete Folder
18	=	File Not Found Error

Note that any error dialog that is undefined will default with the message "TOS error #" and the number of the undefined **error_code**.

NOTE

All standard error dialogs have only one exit button. Any **error_code** greater than 63 will return an error for the function and not display an error dialog.

16

16.5 File Selector Manager

fsel_input displays the File Selector dialog box and controls box activities

Introduction

The routine in the File Selector Manager creates a dialog box that displays the current directory name and the list of its files. The directory files are placed in a window on the File Selector box with a scroll bar on the right side of the window. The box also contains a text editable field which contains a file selection (when appropriate). **CANCEL** and **OK** buttons are also part of the box.

Before selecting a file, the user may:

- scroll through files in the directory, or
- change directories.

To change file directories, the user clicks the mouse curser in the DIRECTORY text editable field and types in a new drive identifier, directory path name, and file specification containing a wildcard. For example,

```
B:\GEMSTUFF\*.GEM
```

16

Using the File Selector Manager

The *fsel_input()* routine returns the following information:

- the selected file name
- the current directory and wildcard specification
- which of the exit methods was used (CANCEL or OK)

Example

Below is shown an example of the *fsel_input* function.

```
#include <stdio.h>

#define OK      1
#define CANCEL  0
```

```
main()
{
    char default_path[80];
    char default_name[80];
    int  button;

    appl_init();

    strcpy(default_path, "A:\\*.*");
    strcpy(default_name, "Untitled");

    fsel_input(default_path, default_name, &button);

    if (button == OK)
        printf("You have selected the file <%s>.\n", default_name);
    else
        printf("You have canceled the file selection.\n");

    printf("Press RETURN to end.\n");
    getchar();

    appl_exit();
}
```

NAME

fsel_input — displays a file selector dialog box, and waits for input.

SYNOPSIS

```
int fsel_input(default_path, default_fname, button)
    char *default_path;
    char *default_fname;
    int  *button;
```

DESCRIPTION

fsel_input displays a dialog box which is used to select the name of a file on a disk. The file selector displays the files that are in the directory specified by `default_path`. There is a field on the file selector dialog box which contains a default file name. This field is initialized by the parameter `default_fname`. The results of the user interaction will be placed in the memory pointed to by `default_path` and `default_fname`. The parameter `button` is a pointer to an integer that contains the number of the exit button. The return values of `button` are defined as follows:

0	=	Cancel button
1	=	OK button

16

DIAGNOSTICS

The result of the function is zero if an error occurs.

NOTE

Wildcard characters may be used in the parameter `default_path`. All files ending in ".c" would be displayed by passing the string "`A:*.C`" as the default path.

16

16.6 Graphics Manager

graf_rubberbox	draws an expanding box from a fixed point as the mouse moves
graf_dragbox	moves a box on the screen, keeping the mouse pointer in the same position
graf_movebox	draws a moving box
graf_growbox	draws an expanding box outline
graf_shrinkbox	draws a shrinking outline
graf_watchbox	looks for a mouse-down inside a box
graf_slidebox	keeps a sliding box inside the parent box
graf_handle	returns a VDI handle for the opened screen work-station that AES uses
graf_mouse	changes the mouse form to another predefined or application defined form
graf_mkstate	returns the current mouse location, mouse button state, and keyboard state

Introduction

The Graphics Manager routines are used to control boxes in the GEM environment. A "box" is basically a rectangular outline drawn on the screen. For example, the routine *graf_growbox* is the routine that draws the expanding box when an application is executed by double-clicking an icon. Other Graphics Manager routines perform functions like moving a box shape across the screen, dragging a box on the screen keeping the mouse pointer fixed, and checking to see if a mouse-down event has occurred in a box.

NAME

graf_dragbox — moves a rectangle, keeping the mouse pointer in the same position in the rectangle.

SYNOPSIS

```
int graf_dragbox(start_w, start_h, start_x, start_y,
                 bound_x, bound_y, bound_w, bound_h,
                 finish_x, finish_y)

    int  start_w, start_h, start_x, start_y;
    int  bound_x, bound_y, bound_w, bound_h;
    int  *finish_x, *finish_y;
```

DESCRIPTION

graf_dragbox lets a user drag an outline of a rectangle within an application defined boundary rectangle. When the user presses the mouse button to begin dragging, GEM AES makes a call to VDI to get the mouse's location. As the user drags, this call keeps the mouse pointer in a fixed position relative to the box's upper left corner. The parameters `start_w`, `start_h`, `start_x`, `start_y` define the outline of the rectangle to be drawn. The parameters `bound_x`, `bound_y`, `bound_w`, `bound_h` define a boundary rectangle that will contain the rectangle being drawn. If an error occurs the result of the function is 0. The final (x, y) position, when the mouse button is released, will be stored at the locations pointed to by `finish_x` and `finish_y`, respectively. Note that all parameters are defined in pixels.

16

NOTE

If a call to *graf_dragbox* is made while the mouse button is up the function will return immediately.

SEE ALSO

graf_slidebox

NAME

graf_growbox — draws an expanding box outline.

SYNOPSIS

```
int graf_growbox(small_x, small_y, small_w, small_h,
                 large_x, large_y, large_w, large_h)

    int small_x, small_y, small_w, small_h;
    int large_x, large_y, large_w, large_h;
```

DESCRIPTION

graf_growbox draws a box growing from a smaller rectangle to a larger rectangle. The small rectangle is defined by the parameters `small_x`, `small_y`, `small_w`, `small_h`. The large rectangle is defined by the parameters `large_x`, `large_y`, `large_w`, `large_h`. Note that both rectangles are defined in pixels.

DIAGNOSTICS

The result of the function is zero if an error occurs.

SEE ALSO

graf_shrinkbox

NAME

graf_handle — is a routine that returns a handle to the opened screen workstation that the GEM AES routines use.

SYNOPSIS

```
int graf_handle(char_width, char_height, char_bwidth, char_bheight)
    int *char_width,  *char_height;
    int *char_bwidth, *char_bheight;
```

DESCRIPTION

graf_handle returns a handle to the current active workstation. Information about the system font is returned through the parameters as follows:

`char_width`	the width of a character cell in the system font in pixels.
`char_height`	the height of a character cell in the system font in pixels.
`char_bwidth`	the width of a square box large enough to hold a system font character in pixels.
`char_bheight`	the width of a square box large enough to hold a system font character in pixels.

16

SEE ALSO

vst_height

NAME

graf_mkstate — returns the location of the mouse, the state of the mouse button, and the state of the keyboard.

SYNOPSIS

```
int graf_mkstate(mousex, mousey, mouse_state, keybd_state)
    int *mousex, *mousey, *mouse_state, *keybd_state;
```

DESCRIPTION

graf_mkstate returns information about the mouse state. The mouse (x, y) locations are stored in memory where the parameters `mousex` and `mousey` point. The state of the mouse button and the state of the keyboard are stored at the locations pointed to by the parameters `mouse_state` and `keybd_state`. The integers returned are defined as follows:

`mouse_state` The current mouse button state. If the bit is set then the button is currently down:

0x0001	=	button on left
0x0002	=	second button from the left
0x0004	=	third button from the left, etc.

`keybd_state` The state of the keyboard's modifier keys. If the bit is set then the key is considered down, if it is zero then the key is considered up:

0x0001	=	right-shift
0x0002	=	left-shift
0x0004	=	Ctrl
0x0008	=	Alt

DIAGNOSTICS

The function always returns a 1.

NAME

graf_mouse — lets an application change the mouse form to one of a predefined set or to an application-defined form.

SYNOPSIS

```
int graf_mouse(form_num, form_def)
    int form_num;
    int form_def[37];
```

DESCRIPTION

graf_mouse changes the mouse form to one of a predefined set or to an application defined form. The parameters are defined as follows:

`form_num` a code identifying a predefined mouse form:

0	=	arrow
1	=	hourglass
2	=	bumble bee
3	=	hand with pointing finger
4	=	flat hand, extended fingers
5	=	thin cross hair
6	=	thick cross hair
7	=	outline cross hair
255	=	mouse form stored in `form_def`
256	=	hide mouse form
257	=	show mouse form

`form_def` the address of a 37-word buffer that fits the mouse form definition. See the VDI function *vsc_form* (page 382).

16

NAME

graf_movebox — Draw a moving outlined box

SYNOPSIS

```
int graf_movebox(gr_mwidth, gr_mheight, gr_msourcex,
                 gr_msourcey, gr_mdestx, gr_mdesty)

    int gr_mwidth, gr_mheight;
    int gr_msourcex, gr_msourcey;
    int gr_mdestx, gr_mdesty;
```

DESCRIPTION

graf_movebox is a routine that draws an animated box moving from one position to another without changing size.

`gr_mwidth`	the rectangle's width in pixels.
`gr_mheight`	the rectangle's height in pixels.
`gr_msourcex`	the rectangle's starting x-coordinate.
`gr_msourcey`	the rectangle's starting y-coordinate.
`gr_mdestx`	the rectangle's ending x-coordinate
`gr_mdesty`	the rectangle's ending y-coordinate

DIAGNOSTICS

A positive integer is returned on success, 0 on failure.

NAME

graf_rubberbox — draws a rectangle that expands and contracts from a fixed point as the mouse moves.

SYNOPSIS

```
int graf_rubberbox(gr_rx,  gr_ry, gr_rminwidth,
    gr_rminheight, gr_lastwidth, gr_rlastheight)

    int gr_rx, gr_ry;
    int gr_minwidth, gr_minheight;
    int *gr_rlastwidth, *gr_rlastheight;
```

DESCRIPTION

graf_rubberbox draws the outline of a rectangle that expands and contracts with the movement of the mouse. The position of the rectangle's upper left corner is fixed, but by dragging the lower right corner with the mouse pointer, the user can make the rectangle larger or smaller. When the mouse button is released the width and height of the new rectangle is returned.

16

`gr_rx`	the rectangle's X-coordinate.
`gr_ry`	the rectangle's Y-coordinate.
`gr_rminwidth`	the rectangle's smallest possible width in pixels.
`gr_rminheight`	the rectangle's smallest possible height in pixels.
`gr_rlastwidth`	the resulting width of the rectangle.
`gr_rlastheight`	the resulting height of the rectangle.

DIAGNOSTICS

The result of the function is zero if an error occurs.

NAME

graf_shrinkbox — draws a shrinking rectangle outline.

SYNOPSIS

```
int graf_shrinkbox(start_x, start_y, start_w, start_h,
                   final_x, final_y, final_w, final_h)

    int start_x, start_y, start_w, start_h;
    int final_x, final_y, final_w, final_h;
```

DESCRIPTION

graf_shrinkbox that will draw a shrinking rectangle outline. The large rectangle is defined by the `start_` parameters. The small resulting rectangle is defined by the `final_` parameters. Note that no rectangle will be visible on the screen when this function is finished.

`start_x`	the rectangle's starting x-coordinate.
`start_y`	the rectangle's starting y-coordinate.
`start_w`	the rectangle's starting width in pixels.
`start_h`	the rectangle's starting height in pixels.
`final_x`	the rectangle's ending x-coordinate.
`final_y`	the rectangle's ending y-coordinate.
`final_w`	the rectangle's ending width in pixels.
`final_h`	the rectangle's ending height in pixels.

DIAGNOSTICS

The result of the function is zero if an error occurs.

16

NAME

graf_slidebox — keeps a sliding rectangle inside its parent rectangle.

SYNOPSIS

```
int graf_slidebox(objtree, parent, slider, direction)
    OBJECT *objtree;
    int     parent;
    int     slider;
    int     direction;
```

DESCRIPTION

graf_slidebox tracks a sliding rectangle inside a parent rectangle. An example of the use of this function are the scroll bars commonly seen on windows. The mouse movement causes the sliding rectangle to move, and the parent rectangle defines the sliding rectangle's range of motion. An application calls this routine when the mouse button is depressed and returns control to the application when the user releases the mouse button. Both boxes (slider and parent) defined by the object tree `objtree`.

The return value of the function is a number that indicates the slider position relative to the inside of the parent rectangle. If `direction` is 0, then the routine returns a value from 0 to 1000; left to right. If `direction` is 1, the routine returns a value from 0 to 1000; top to bottom.

16

`parent`	The index of the parent in the object tree.
`slider`	The index of the the slider in the object tree.
`direction`	The direction of the slider's movement. 0 = horizontal 1 = vertical
`objtree`	A pointer to the object tree containing the slider and parent.

NAME

graf_watchbox — "watches" a rectangle to see if the user releases the mouse button inside of a specified rectangle.

SYNOPSIS

```
int graf_watchbox(tree, object, instate, outstate)
    OBJECT *tree;
    int     object;
    int     instate, outstate;
```

DESCRIPTION

graf_watchbox tracks the mouse pointer in and out of a predefined rectangle while the mouse button is depressed, and returns a value based upon where the mouse button is released. The state of the rectangle is changed according to the `instate` and `outstate` parameters.

1 is returned if the mouse button was released inside the rectangle, otherwise 0 is returned.

`tree`	is a pointer to the object tree that contains the defined rectangular area.
`object`	the index of the object in the tree.
`instate`	the rectangle's state when the depressed mouse button goes inside the defined rectangle.

0x0000	=	normal
0x0001	=	selected
0x0002	=	crossed
0x0004	=	checked
0x0008	=	disabled
0x0010	=	outlined
0x0020	=	shadowed

`outstate`	the rectangle's state when the depressed mouse button goes outside the defined rectangle.

16

16.7 Menu Manager

menu_bar	displays or erases the menu bar
menu_icheck	displays or removes checks by menu items
menu_ienable	enables or disables menu items
menu_tnormal	displays the menu title in normal or reverse video
menu_text	changes the text of a menu item
menu_register	registers desk accessories

Introduction

Each GEM application defines its own menu, and the application's menu is displayed when the application is active. The menu's title is selected by moving the mouse onto the text of the title on the menu bar, this causes a drop-down menu to be displayed.

The various selections displayed under a title be enabled or disabled by the application. If disabled, a half-tone or gray title is drawn and the user is prevented from selecting the item. Additionally the application may wish to place a check mark to the left of one or more of the selections.

To display a menu, the application must make two calls. First, it calls *rsrc_load* to load the menu data. Second, it calls *menu_bar* to display the menu bar. The application will then receive a message from the Screen Manager when an item in a drop down menu is selected.

The Screen Manager displays the drop down menu and highlights the menu title when the mouse form touches a menu title. The manager then follows the mouse over the menu. As the mouse passes enabled titles, the manger displays them in reverse video. The user selects an enabled item from the menu by clicking the mouse on one of the enabled items, resulting in in two actions by the manager. First, the manager removes the drop down menu. Then, the manager sends a message to the message pipe of the application.

If the user moves the mouse outside of a drop down menu, the drop down menu remains, but nothing is selected or highlighted. If the user then clicks the mouse, the drop down menu is removed and no messages are sent to the application.

GEM also allows the application to change the text of the menu items. This is useful for different states or modes of the application.

Using the Menu Manager

The programmer creates a menu object tree with the Resource Construction Program which then adds it to a resource file. Then the tree is loaded into memory using the *rsrc_load* call from the application. Finally, *menu_bar* is called to display the titles. Once this is done, the visible menu entries can be accessed by the user.

After the user chooses a menu item, the Screen Manager sends a message to the application, and then control is returned to the application. The application must then read the message in the pipe. Reading the pipe tells the application that the message is about a menu selection, the object index of the menu title chosen, and the object index of the menu item chosen.

Example

The example below shows how to place a menu bar on the screen. The menu bar object tree is created by the Resource Construction Program. The name of the menu bar is MENUBAR is defined by the header file of the resource file. For further information about resources refer to section 11. For further information about handling menu trees refer to section 16.3.

16

```
#include <osbind.h>
#include <gemdefs.h>
#include <obdefs.h>

#include "resource.h"   /* header file created by RCP     */
#include "globals.h"    /* contains definition of menubar */

/*
    init_menu   -  find the address of the menubar and draw it.
*/
init_menu()
{
    /*
        MENUBAR is the name of the menu resource.
        menubar is an (OBJECT *).
    */
    rsrc_gaddr(0, MENUBAR, &menubar);

    /*
        Draw the menu Bar.
    */
    menu_bar(menubar, 1);
}
```

The following example shows how to handle a menu event.

```
#include <gemdefs.h>
#include <obdefs.h>

#include "resource.h"   /* header file created by RCP     */
#include "globals.h"    /* contains definition of menubar */

/*
    do_menu - determines which menu was selected and calls the
        appropriate routine to handle the item selected.
*/
do_menu(message)
    int *message;
{
    int menuid, itemid;

    menuid = message[3];
    itemid = message[4];

    switch(menuid) {
        case DESK:
            handle_desk(itemid);
        break;

        case FILE:
            handle_file(itemid);
        break;

        case EDIT:
            handle_edit(itemid);
        break;
    }

    menu_tnormal(menubar, menuid, 1);
}


/*
    handle_desk - performs the appropriate action for the menu item selected.
*/
handle_desk(itemid)
    int itemid;
{
    switch(itemid) {
        case ABOUT:
            form_alert(1, "[0][ A Sample Application | | rpt. ][ Ok ]");
        break;
    }
}
```

```
/*
    handle_file - performs the appropriate action for the menu item selected.
*/
handle_file(itemid)
    int itemid;
{
    int  button;

    switch(itemid) {
        case FILENEW:
            new_window(SIZER | MOVER | FULLER | CLOSER | NAME);
        break;

        case FILECLOS:
            {
                windowptr thewin = frontwindow();

                if (thewin)
                    dispose_window(thewin);
            }
        break;

        case FILEQUIT:
            button = form_alert(2, "[3][ Are you sure? ][ Yes | No ]");

            if (button == 1)
                shutdown(0);
        break;
    }
}


/*
    handle_edit - performs the appropriate action for the menu item selected.
*/
handle_edit(itemid)
    int itemid;
{
    char string[80];

    switch(itemid) {
        case UNDO:
            sprintf(string, "Edit.  Undo.  itemid == %d.", itemid);
        break;

        case CUT:
            sprintf(string, "Edit.  Cut.  itemid == %d.", itemid);
        break;

        case COPY:
```

16

```
            sprintf(string, "Edit.  Copy.  itemid == %d.", itemid);
        break;

        case PASTE:
            sprintf(string, "Edit.  Paste.  itemid == %d.", itemid);
        break;

        case CLEAR:
            sprintf(string, "Edit.  Clear.  itemid == %d.", itemid);
        break;
    }

    paramdlog(string);
}
```

16

NAME

menu_bar — display or erase the menu bar.

SYNOPSIS

```
int menu_bar(menu_tree, show_menu)
    OBJECT *menu_tree;
    int     show_menu;
```

DESCRIPTION

menu_bar draws or erases a menu object tree on the desktop. The parameter `menu_tree` is a pointer to an OBJECT which describes a menu. The parameter `show_menu` tells the function whether to draw or erase the menu bar. If `show_menu` is 1, the menu bar is drawn, otherwise the menu is erased.

NOTE

The application should always erase the menu bar before exiting with *appl_exit*.

DIAGNOSTICS

The function result will be 0 if an error occurs.

NAME

menu_icheck — displays or erases a check mark next to a menu item

SYNOPSIS

```
int menu_icheck(menu_tree, menu_item, menu_check)
    OBJECT *menu_tree;
    int     menu_item;
    int     menu_check;
```

DESCRIPTION

menu_icheck marks a menu item as checked. The parameter `menu_tree` is an object pointer to the menu definition. The parameter `menu_item` is a integer index into the menu tree which indicates the menu item to mark. The final parameter `menu_check` is an flag which determines the state of the check. If `menu_check` is 1, the menu is marked with a check. If it is zero then the menu is not marked.

NOTE

The value of `menu_item` may be obtained from the resource header file if the menu item has been "named."

DIAGNOSTICS

The result of the function is zero if an error occurs.

SEE ALSO

Resource Construction Program (pg. 81)

NAME

menu_ienable — enable or disable a menu item

SYNOPSIS

```
int menu_ienable(menu_tree, menu_item, menu_enable)
    OBJECT *menu_tree;
    int     menu_item;
    int     menu_enable;
```

DESCRIPTION

menu_ienable enables and disables menu items. When a menu item is disabled it is drawn in half-tone grey and cannot be selected by the user. The parameter `menu_tree` is a pointer to an obect that describes the menu bar. The parameter `menu_item` is an index into the menu bar object tree that indicates the menu to be enabled/disabled. The last parameter `menu_enable` is a flag which determines the operation. If `menu_enable` is 0, the menu will be disabled, otherwise the menu item will be enabled.

NOTE

The value of `menu_item` may be obtained from the resource header file if the menu item has been "named."

DIAGNOSTICS

The result of the function is zero if an error occurs.

NAME

menu_register — places a desk accessories menu on the Desk Menu.

SYNOPSIS

```
int menu_register(appl_id, acc_name)
    int   appl_id;
    char *acc_name;
```

DESCRIPTION

menu_register is used to register the application to AES as a Desk Accessory. The parameter `appl_id` is the application's identifier that is obtained from `appl_init`. The last parameter is a pointer to the name of the Accessory which is placed in the Desk Menu. The result of the function is a menu identifier.

NOTE

The menu identifier is used during the message event `ACC_OPEN` to determine if the Desk Accessory is active.

DIAGNOSTICS

If there are currently more than 6 accessories in the list then the result of the function is −1.

SEE ALSO

appl_init, *evnt_mesag*, *evnt_multi*

NAME

menu_text — changes the text of a menu item.

SYNOPSIS

```
int menu_text(menu_tree, menu_item, menu_text)
    OBJECT *menu_tree;
    int     menu_item;
    char   *menu_text;
```

DESCRIPTION

menu_text changes the text on a specified menu item. The menu is defined by the OBJECT tree pointer `menu_tree`. The menu item that is changed is specified by `menu_item`. The text that replace the existing menu item text is pointed to by `menu_text`.

NOTE

The value of `menu_item` may be obtained from the resource header file if the menu item has been "named."

DIAGNOSTICS

The function result is zero if an error occurs.

NAME

menu_tnormal — displays a menu title in either normal or reversed video.

SYNOPSIS

```
int menu_tnormal(menu_tree, menu_title, menu_normal)
    OBJECT *menu_tree;
    int     menu_title;
    int     menu_normal;
```

DESCRIPTION

menu_tnormal hilites and de-hilites menu titles. The parameter `menu_tree` is an OBJECT pointer to the menu bar tree. The parameter `menu_tree` is an index into the menu tree that specifies the menu title that is affected. The last parameter `menu_normal` specifies the state of the menu title. If it is zero the menu title will be hilited, otherwise the menu title is de-hilited.

NOTE

The value of `menu_title` may be obtained from the resource header file if the menu item has been "named."

DIAGNOSTICS

The result of the function is zero if an error occurs.

16

SEE ALSO

Resource Construction Program (pg. 81)

16

16.8 Object Manager

objc_add	adds an object to an object tree
objc_delete	deletes an object from an object tree
objc_draw	draws an object or object tree
objc_find	determines if the mouse is over an object
objc_offset	computes an object's location relative to the screen
objc_order	changes the order of an object within its tree
objc_edit	lets a user edit text in an object
objc_change	changes an objects state

Introduction

Objects describe visual items, such as icons, characters, and boxes. The Object Manager provides routines to manipulate them. An example of an object (in this case a dialog box) follows:

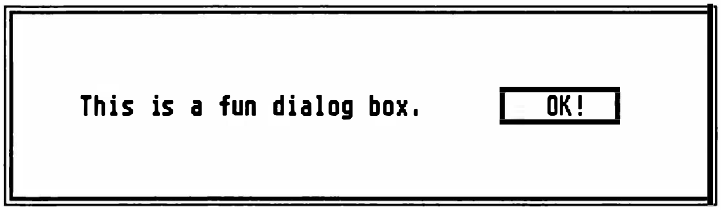


Figure 16.2: Example Dialog

An object tree is an array of `OBJECT`'s (see C typedef in `<obdefs.h>` and shown below). The address of the array is the address of the tree (its type is `OBJECT *`). Each object in the tree can be accessed as an index from the address of the tree. For example the following C code de-hilites the **OK** button in the above dialog box:

```
#include <obdefs.h>

/*
    Note: The following values are supplied by the RCP.
*/
#define TREE      0    /* Get resource type Tree        */
#define NAMEDLOG 10    /* eleventh tree in resource file */
#define OKBUTN    2    /* third object in above tree */
```

```
dehilite_OK()
{
    OBJECT *namedlog;

    /*
        Make OBJECT *namedlog point to dialog tree.
    */
    rsrc_gaddr(TREE, NAMEDLOG, &namedlog);

    /*
        Change the ob_state of the button object
            to not selected.
    */
    namedlog[OKBUTN].ob_state &= ~SELECTED;
}
```

Normally the names NAMEDLOG and OKBUTN would come from a '.H' file created by RCP. The structure of the tree comes from the ob_next, ob_head and ob_last fields of each OBJECT. These fields contain the index numbers of the next sibling, first child and last child of each object in the tree. If there is no such object (such as ob_head for a leaf object) then the value is −1. The ob_next field of the last child of an object points to that object. The first object (index no. 0) is the root of the tree. The ob_next field of the root object is −1. The tree structure for the dialog box is shown below:

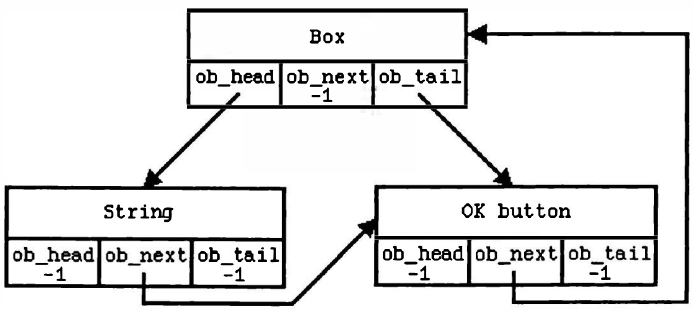


Figure 16.3: Object structure

Object Manager Data Structures

The following structures are used by the Object Manager:

```
OBJECT
TEDINFO
```

```
ICONBLK
BITBLK
APPLBLK
PARMBLK
```

OBJECT

```
typedef struct object {
    int       ob_next;   /* object's next sibling           */
    int       ob_head    /* head of object's children       */
    int       ob_tail    /* tail of object's children       */
    unsigned  ob_type;   /* type of object, i.e. BOX, CHAR ...*/
    unsigned  ob_flags;  /* flags                           */
    unsigned  ob_state;  /*state, i.e. SELECTED, OPEN...    */
    char     *ob_spec;   /* see below                       */
    int       ob_x;      /*upper left corner of object      */
    int       ob_y;      /*upper left corner of object      */
    int       ob_width;  /*width of object                  */
    int       ob_height; /* height of object                */
} OBJECT;
```


The OBJECT structure's values describe the object, its relative position in the tree, and on screen. There is an OBJECT structure for each object in a tree.

ob_next index of the object's next sibling in the object tree array

ob_head index of the object's first child: the head of the list of the object's children in the object tree array.

ob_tail index of the last child in the list of the object's children in the object tree array.

ob_type the object type (see object types paragraph)

ob_flags the object flags (see object flags paragraph)

ob_state the object state (see object state paragraph)

ob_spec Depending on the object's type, this can be a pointer, or other four byte value. All the values of this are described when the object types are described below in Object Types.

For the types G_BOX G_IBOX, and G_BOXCHAR the low word is the object color. The high word is broken into two bytes. For the type G_BOXCHAR, the high byte of this word is a character, for all other types, this is zero.

The low byte of the high word is the thickness of the border of the object, and can have the following values:

0	no thickness
1 – 128	these positive values are the inside thickness, inward from the object's edge
−1 – (−127)	these negative values are outward thickness, from the object edge.

ob_x	For a child, the x coordinate of the object relative to the parent. For the root, relative to the screen.
ob_y	For a child, the y coordinate of the object relative to the parent. For the root, relative to the screen.
ob_width	the width of the object in pixels
ob_height	the height of the object in pixels

16

TEDINFO

The TEDINFO object is an editable field that is used to receive and check keyboard input from the user.

```
typedef struct text_edinfo {
    char    *te_ptext;    /* ptr to text (must be first)       */
    char    *te_ptmplt;   /* ptr to template                   */
    char    *te_pvalid;   /* ptr to validation chrs.           */
    int     te_font;      /* font                              */
    int     te_junk1;     /* junk word 1                       */
    int     te_just;      /* justification, left or right ... */
    int     te_color;     /* color information word            */
    int     te_junk2      /* junk word                         */
```

```
    int     te_thickness; /* border thickness          */
    int     te_txtlen;    /*length of text string      */
    int     te_tmplen;    /* length of template string */
} TEDINFO;
```

This structure allows a user to edit formatted text. The object types G_TEXT, G_BOXTEXT, G_FTEXT, and G_FBOXTEXT use their ob_spec pointers to point to TEDINFO structures. te_ptext a pointer to the actual text. If the first character is "@", then the entire field is blanks, and all following characters are merely place holders; i.e. "@abc" would be four spaces.

te_ptmplt a pointer to a text string template for any further data entry.

te_pvalid a pointer to a text string which validates entered text:

- 9 allow only digits 0–9
- A allow only uppercase letters A–Z and space
- a allow only letters, upper and lower case A–z and space
- N allow 0–9 and uppercase A–Z plus space
- n allow 0–9 and upper and lowercase A–z, plus space
- F allow all valid DOS file name characters, plus ? * :
- P allow all valid DOS file name characters, plus \ : ? *
- p allow all valid DOS file name character, plus \ :
- X allow anything
- x allow anything and do uppercase conversion

te_font an integer identifying the font used to draw the text:

- 3 system font, used in menus, dialogs, etc.
- 5 small font, used in icons

te_junk1 reserved for future use

te_just an integer identifying the type of text justification desired:

- 0 = left justified
- 1 = right justified
- 2 = centered

`te_color`	an integer identifying the color and pattern of box-type objects
`te_junk2`	reserve for future use
`te_thickness`	an integer defining the thickness in pixels of the text box 0 — no thickness 1 – 128 — positive values are the inside thickness inward from the object's edge −1 – (−127) — negative values are the outside thickness outward from the object's edge
`te_txtlen`	an integer which is the length of the text string pointed to by `te_ptext`
`te_tmplen`	an integer containing the length of the string pointed to be te_ptmplt

Underscore characters in the `te_ptmplt` field indicate where characters typed by the user will be displayed. Other characters are for display only and may not be modified by the user. The `te_ptext` string will contain only the characters the user typed in, and will not contain any of the extra characters from the `te_ptmplt` field.

An example. An edit field for entering a file name would use the following field values:

`te_ptmplt`	"`File name: ________.___`"
`te_pvalid`	"`FFFFFFFFFFF`"

On return `te_ptext` will contain only the characters the user typed, for instance if the user typed "`FUN.TXT`" then `te_ptext` will contain:

`te_ptext`	"`FUN TXT`"

Note that the period typed by the user is not legal according to `te_pvalid`, however it was in `te_ptmplt` so the cursor automatically jumped to the next underscore after the period.

ICONBLK

The Object Manager uses this structure to hold the data that defines icons. The object type `G_ICON` points with its `ob_spec` pointer to an `ICONBLK` structure.

```
typedef struct icon_block {
    int     *ib_pmask;
    int     *ib_pdata;
    char    *ib_ptext;
    int     ib_char;
    int     ib_xchar;
    int     ib_ychar;
    int     ib_xicon;
    int     ib_yicon;
    int     ib_wicon;
    int     ib_hicon;
    int     ib_xtext;
    int     ib_ytext;
    int     ib_wtext;
    int     ib_htext;
} ICONBLK;
```

ib_pmask a pointer to an array of integers representing the mask bit-image of the icon

ib_pdata a pointer to an array of integers representing the data bit-image of the icon

ib_ptext a pointer to the icon's text

ib_char an integer containing a character to be drawn in the icon. The character color is defined in the high byte of the integer. The foreground color is defined in the upper nybble and the background color is defined in the lower nybble.

ib_xchar an integer containing the x-coordinate of ib_char

ib_ychar an integer containing the y-coordinate of ib_char

ib_xicon an integer containing the x-coordinate of the icon

ib_yicon an integer containing the y-coordinate of the icon

ib_wicon an integer containing the width of the icon in pixels, and must be a multiple of 16.

16

ib_hicon	an integer containing the height of the icon in pixels
ib_xtext	an integer containing the x-coordinate of the icon's text
ib_ytext	an integer containing the y-coordinate of the icon's text
ib_wtext	an integer containing the width of the icon's text in pixels
ib_htext	an integer containing the height of the icon's text in pixels

BITBLK

The object type G_IMAGE uses the BITBLK structure to draw the bit images like cursor forms or icons.

```
typedef struct bit_block {
    int *bi_pdata;  /* ptr to bit forms data    */
    int  bi_wb;     /* width of forms in bytes  */
    int  bi_hl;     /* height in lines          */
    int  bi_x;      /* source x in bit form     */
    int  bi_y;      /* source y in bit form     */
    int  bi_color;  /* fore-ground color of bit */
} BITBLK;
```

16

bi_pdata	a pointer to an array of ints containing the bit image
bi_wb	an integer containing the width of the bi_pbdata array in bytes. Because the bi_pdata array is made of ints, this value must be an even number.
bi_hl	an integer containing the height of the bit block in scan lines, or pixels
bi_x	an integer containing the source X in bit form, relative to the bi_pdata array
bi_y	an integer containing the source Y in bit form, relative to the bi_pdata array
bi_color	an integer containing the color GEM AES uses when displaying the bit-image.

APPLBLK

This structure is used to locate and call an application-defined routine that will draw and/or change an object. The ob_spec pointer in the object type G_PROGDEF points to an APPLBLK structure.

```
typedef struct appl_blk {
    int     (*ub_code)();
    long    ub_parm;
} APPLBLK;
```

ub_code — a pointer to the routine for drawing and/or changing the object

ub_parm — a long value, optionally provided by the application, passed as a parameter when the Object Manager calls the application's object drawing/changing routine.

PARMBLK

This structure is used to store information relevant to the application's drawing or changing an object.

When it calls the application's object drawing/changing routine(pointed to by ab_code), the Object Manager provides a pointer to a PARMBLK.

```
typedef struct parm_blk {
    OBJECT  *pb_tree;
    int     pb_obj;
    int     pb_prevstate;
    int     pb_currstate;
    int     pb_x, pb_x, pb_w, pb_h;
    int     pb_xc, pb_yc, pb_wc, pb_hc;
    long    pb_parm;
} PARMBLK;
```

pb_tree — a pointer to the object tree that contains the application-defined object

pb_obj — the object index of the application-defined object

16

`pb_prevstate` the old state of an object to be changed

`pb_currstate` the changed or new state of an object

If `pb_prevstate` = `pb_currstate` then the application is drawing the object, not changing it.

`pb_x, pb_y` the x and y-coordinates of a rectangle that define the location of the object on the physical screen

`pb_w, pb_h` the width and height in pixels of a rectangle defining the size of the object on the physical screen

`pb_xc, pb_yc` an integer containing the x and y -coordinates of the current clip rectangle on the physical screen

`pb_wc, pb_hc` an integer containing the width and height in pixels of the current clip rectangle on the physical screen

`pb_param` identical to the `ab_parm` in the `APPLBLK` structure. The Object Manager passes this value to the application when it is time for the application to draw or change the object.

Predefined Values

The Object Manager routines use the following predefined values:

object types
object flags
object states
objects colors

The following sections define these values.

Object Types

```
#define G_BOX       20
#define G_TEXT      21
#define G_BOXTEXT   22
#define G_IMAGE     23
#define G_PROGDEF   24
```

```
#define G_IBOX      25
#define G_BUTTON    26
#define G_BOXCHAR   27
#define G_STRING    28
#define G_FTEXT     29
#define G_FBOXTEXT  30
#define G_ICON      31
#define G_TITLE     32
```

Object types are stored in the **ob_type** section of the OBJECT structure. All object types are graphic or bitmap object types.

G_BOX — A graphic box; **ob_spec** is the object's color and thickness.

G_TEXT — Graphic text; **ob_spec** is as pointer to a TEDINFO structure in which the value of the **te_ptext** points to the displayed text string.

G_BOXTEXT — A graphic box containing graphic text; **ob_spec** is a pointer to a TEDINFO structure in which **te_ptext** points to the actual text string.

G_IMAGE — A graphic bit-image; **ob_spec** is a pointer to a BITBLK structure.

G_PROGDEF — A programmer-defined object; its **ob_spec** is a pointer to an APPLBLK structure.

G_IBOX — An "invisible" graphic box; its **ob_spec** value contains the object's color int and thickness. It has no fill pattern and no internal color. Its border is the only visible part, and the border is only visible if it has thickness.

G_BUTTON — a graphic text object centered in a box; **ob_spec** is a pointer to a null-terminated text string.

G_BOXCHAR — A graphic box containing a single text character; **ob_spec** contains the character, plus the object's color and thickness.

G_STRING — A graphic text object; its **ob_spec** value is a pointer to a null-terminated text string.

G_FTEXT — Formatted graphic text; **ob_spec** is a pointer to a TEDINFO structure in which **te_ptext** points to a text string. The text string is merged with the template pointed to by **te_ptmplt** before it is displayed.

G_FBOXTEXT — A graphic box containing formatted graphic text; **ob_spec** is a pointer to a TEDINFO structure in which **te_ptext** points to a text string. The text string is merged with the template pointed to by the **te_ptmplt** before it is displayed.

G_ICON — An object that describes an icon; **ob_spec** is a pointer to an ICONBLK structure.

G_TITLE — A graphic text string used in menu titles; **ob_spec** is a pointer to a null-terminated text string.

Object Flags

```
#define NONE        0X0000
#define SELECTABLE  0X0001
#define DEFAULT     0X0002
#define EXIT        0X0004
#define EDITABLE    0X0008
#define RBUTTON     0X0010
#define LASTOB      0X0020
#define TOUCHEXIT   0X0040
#define HIDETREE    0X0080
#define INDIRECT    0X0100
```

Object flags are stored as a bit vector in the **ob_flags** portion of the OBJECT structure. Each bit in the **ob_flags** word is significant. Undefined bits should be set to zero.

SELECTABLE — Indicates the user can select the object.

DEFAULT — Indicates the the Form Manager will examine the object if the user enters a carriage return. No more than one object in a form can be flagged DEFAULT. This object is usually an exit button, which lets the user enter a carriage return to exit the form without using the mouse.

EXIT — Indicates that the Form Manager will return control to the caller when the exit condition is satisfied, by the user selecting the object.

EDITABLE — The object is editable by the user in.

RBUTTON — An object called a radio button.

Radio buttons appear in groups of two are more, only one of which may be selected at a given time. When the user selects a button, the currently selected button is automatically de-selected.

All radio buttons in a group must have the same parent.

LASTOB — Indicates that an object is the last object in the object tree.

TOUCHEXIT — Indicates that the Form Manager will return control to the caller after the exit condition is satisfied. The exit condition is satisfied when the user presses the mouse button while the pointer is over ("touching") the object.

HIDETREE — Makes a subtree invisible. It and its children cannot be drawn by *objc_draw* or found by *objc_find()* calls.

INDIRECT — Indicates that the value in **ob_spec** is a pointer to the actual value of **ob_spec**.

16

Object States

```
#define NORMAL      0x0000
#define SELECTED    0x0001
#define CROSSED     0x0002
#define CHECKED     0x0004
#define DISABLED    0x0008
#define OUTLINED    0x0010
#define SHADOWED    0x0020
```

Object states determine how the *objc_draw* routine draws objects. Object states are stored as a bit vector in the **ob_state** portion of the OBJECT structure.

`NORMAL` Indicates that the object is drawn in normal foreground and background colors.

`SELECTED` Indicates that the object is highlighted by reversing the foreground and background colors.

`CROSSED` Indicates that an "X" is drawn in the object. The object must be a box.

`CHECKED` Indicates that the object is drawn with a check mark.

`DISABLED` Indicates that the object is drawn faintly.

`OUTLINED` Indicates that an outline appears around a box object. This state is used for dialog boxes.

`SHADOWED` Indicates that the object (usually a box) is drawn with a drop shadow.

Object Colors

Object colors are stored in the **ob_spec** portion of the OBJECT structure and the **te_color** portion of the TEDINFO structure. An L preceding the name of the color, as in LRED, indicates a light shade of the color.

The color descriptor integer has five portions as indicated below, each portion's bits represented by a letter:

aaaabbbbcdddeeee

The high four bits "**aaaa**" are the border color, with values ranging from 0 to 15. The next four bits, "**bbbb**" are the text color, also with values from 0 to 15. Bit "**c**" indicates whether text is written in transparent mode (**c** = 0) or replace mode (**c** = 1). The next three bits "**ddd**" indicate the object's fill pattern, with values 0 to 7:

0	=	hollow fill
7	=	solid fill
1 – 6	=	other pattern of increasing darkness

The low four bits "**eeee**" are the object's inside color, with values from 0 to 15.

NOTE: A tree is an array of objects, and thus each object is referred to by its index based at the address of the tree. In the Object Manager routine descriptions, references to an object number or ID refer to this index.

NAME

objc_add — adds an object to an object tree.

SYNOPSIS

```
int objc_add(obj_tree, obj_parent, obj_child)
    OBJECT *obj_tree;
    int     obj_parent;
    int     obj_child;
```

DESCRIPTION

objc_add associates two OBJECT trees. The parameter **obj_parent** is an index into the object tree **obj_tree**. This object is considered the parent object. The last parameter **obj_child** is an index into the object tree **obj_tree**. This object is the OBJECT which will be made the child object of **obj_parent**.

In other words the OBJECT that is indexed by **obj_child** will be made the actual child of the OBJECT specified by the parameter **obj_parent**.

DIAGNOSTICS

The function result is zero if an error occurs.

SEE ALSO

Resource Construction Program (pg. 81)

NAME

objc_change — changes an object's **ob_state** value.

SYNOPSIS

```
int objc_change(obj_tree, obj_object, obj_resvd, obj_xclip,
    obj_yclip, obj_wclip, obj_hclip, obj_newstate, obj_redraw)

    OBJECT *obj_tree;
    int     obj_object;
    int     obj_resvd;
    int     obj_xclip;
    int     obj_yclip;
    int     obj_wclip;
    int     obj_hclip;
    int     obj_newstate;
    int     obj_redraw;
```

DESCRIPTION

objc_change changes the **ob_state** field of an OBJECT. The parameter **obj_tree** defines the object tree. The parameter **obj_object** is an index into the object tree **obj_tree**. The **ob_state** field of the object **obj_object** will be changed to the value of the parameter **obj_newstate**. If the **obj_redraw** flag is 1, then the object will be drawn with the new state **obj_newstate** using the clipping rectangle defined by **obj_*clip**.

16

obj_tree	the address of the object tree containing the object
obj_object	the object to be changed
obj_resvd	reserved; the value must be zero
obj_xclip, **obj_yclip**	the x and y-coordinate of the clip rectangle
obj_wclip, **obj_hclip**	the width and height of the clip rectangle in pixels.
obj_newstate	the **ob_state** value of the object
obj_redraw	if 1, then redraw the object, if zero then don't redraw

DIAGNOSTICS

The result of the function is zero if an error occurs.

NAME

objc_delete — removes an object from its parent object.

SYNOPSIS

```
int objc_delete(obj_tree, obj_object)
    OBJECT *obj_tree;
    int     obj_object;
```

DESCRIPTION

objc_delete disassociates an OBJECT from it's parent OBJECT. The object tree is defined by the parameter **obj_tree**. The parameter **obj_object** is an index into the object tree **obj_tree**. The object **obj_object** will be deleted from the tree list **obj_tree**.

DIAGNOSTICS

The result of the function is zero if an error occurs.

16

NAME

objc_draw — draws objects, or object trees

SYNOPSIS

```
int objc_draw(obj_tree, obj_startobj, obj_depth,
              obj_xclip, obj_yclip, obj_wclip, obj_hclip)

    OBJECT *obj_tree;
    int     obj_startobj;
    int     obj_depth;
    int     obj_xclip;
    int     obj_yclip;
    int     obj_wclip;
    int     obj_hclip;
```

DESCRIPTION

objc_draw draws an object tree. The parameter **obj_tree** defines the OBJECT tree being drawn. The parameter **obj_startobj** is an index into the object tree. This index indicates the initial object to be drawn. The parameter **obj_depth** determines how many levels of the tree, from **obj_startobj**, are drawn. When the object are drawn the clipping rectangle **obj_*clip** is used. This means that only the objects defined within the clipping rectangle will be drawn.

16

obj_tree	the address of the object tree containing the object
obj_startob	the starting object on the tree ob_drtree.
obj_depth	how many levels in the object tree to draw, starting from **ob_drstartob**: 0 = starting object only n = n level children of starting object
obj_xclip, **obj_yclip**	the x and y-coordinates of the clip rectangle in pixels
obj_wclip, **obj_hclip**	the width and height of the clip rectangle in pixels

DIAGNOSTICS

The result of the function is zero if an error occurs.

NAME

objc_edit — allows the user to edit the text in an object.

SYNOPSIS

```
int objc_edit(obj_tree, obj_object, obj_char, obj_idx,
              obj_kind, obj_newidx)
    OBJECT *obj_tree;
    int     obj_object;
    int     obj_char;
    int     obj_idx;
    int     obj_kind;
    int    *obj_newidx;
```

DESCRIPTION

objc_edit is used to handle user interaction with a text object tree. The object tree being edited is defined by the parameter `obj_tree`. The actual edit object is defined by the index `obj_object` into the object tree `obj_tree`. This object must be of type `G_TEXT` or `G_BOXTEXT`. The parameter `obj_char` is the character that is to be inserted at position `obj_idx` in the input box string. The type of edit to be performed is controlled by `obj_kind`. Its values are defined as follows:

16

1 = combine the values in `te_ptext` and `te_ptmplt` into a formatted string; turn on text cursor

2 = validate typed characters against `te_pvalid`, update `te_ptext`, and display string

3 = turn off text cursor

After the edit is performed the function returns. The index of the next character in the raw text string is stored at `obj_newidx`.

NOTE

objc_edit does not query the keyboard for the user input. It is strictly a function which performs an edit operation on the editable object and displays the changes specified by the parameters. It is suggested the *form_do* function be used for obtaining user input.

DIAGNOSTICS

The result of the function is zero if an error occurs.

SEE ALSO

Object Introduction (pg. 237, *form_do*)

NAME

objc_find — finds an object under the mouse form.

SYNOPSIS

```
int objc_find(obj_tree, obj_startobj, obj_depth, mousex, mousey)
    OBJECT *obj_tree;
    int     obj_startobj;
    int     obj_depth;
    int     mousex;
    int     mousey;
```

DESCRIPTION

objc_find locates an object which is drawn under a defined point on the screen. The point where the object is searched for is defined by the parameters **mousex** and **mousey**. The object tree that is searched is defined by the parameter **obj_tree**. The object where the search begins is defined by the parameter **obj_startobj**. This number is an index into the object tree **obj_tree**. The number of levels down the object tree that are searched is defined by the parameter **obj_depth**. The result of the function will be the number of the object found under the point. If no object was found the result of the function is −1.

16

NOTE

If **obj_depth** is zero then only the object specified by **obj_startobj** will be searched.

NAME

objc_offset — computes an object's location relative to the screen

SYNOPSIS

```
int objc_offset(obj_tree, obj_object, obj_xoffset, obj_yoffset)
    OBJECT *obj_tree;
    int     obj_object;
    int    *obj_xoffset;
    int    *obj_yoffset;
```

DESCRIPTION

objc_offset returns the position of the specified object on the screen. The object tree is defined by the parameter **obj_tree**. The parameter **obj_object** is an index into the object tree **obj_tree** which defines the object. The coordinates of the object, relative to the upper-left corner of the screen, are stored at the locations pointed to by **obj_xoffset** and **obj_yoffset**.

DIAGNOSTICS

The result of the function is zero if an error occurs.

NAME

objc_order — moves an object to a new position in its parent's list of children.

SYNOPSIS

```
int objc_order(obj_tree, obj_object, obj_newpos)
    OBJECT *obj_tree;
    int     obj_object;
    int     obj_newpos;
```

DESCRIPTION

objc_order moves an object in the object tree to a new position in the object list. The object tree is defined by the parameter **obj_tree**. The object to be moved is defined by the parameter **obj_object**. This number is an index into the object tree **obj_tree**. The parameter **obj_newpos** defines the new position of the object **obj_object**. The new position is defined relative to the bottom of the object list as follows:

0	=	on the bottom
1	=	one from the bottom
2	=	two from the bottom [etc.]
−1	=	on the top

16

DIAGNOSTICS

The result of the function is zero if an error occurs.

16.9 Resource Manager

rsrc_load	loads entire resource file into RAM
rsrc_free	frees the memory allocated during *rsrc_load*
rsrc_gaddr	gets the address of a data structure in memory
rsrc_saddr	stores an index to a data structure
rsrc_obfix	converts an object's (x, y) coordinates from character to pixel coordinates

Introduction

A resource is an application independent interface between the user, or a device, and the application. Its purpose is to allow easy change to the application's interface without changing the application. A common use for this would be localizing an application for a language other than the one for which it was originally written. For example, if an application was originally written for an English literate market and the authors wished to sell it in France, a simple change in the resources using the Resource Construction Program hopefully would be all that is required.

Using the Resource Manager

The *rsrc_load* routine is used to load the resource file into RAM. It also makes any necessary updates to the file. These updates include building the array of tree pointers, storing the tree array address in the application's `global` array, and making the file device specific to the screen's resolution.

Example

The example below illustrates how to load a resource into memory.

```
#include <osbind.h>
#include <gemdefs.h>
#include <obdefs.h>

#include "resource.h"   /* header file created by RCP     */
#include "globals.h"    /* contains definition of menubar */


/*
   init_resources - attempts to load the applications resources into
      memory.  Note that if the file is not found in the current
```

```
        directory the ROM will search the A:\ drive automatically.
*/
init_resources()
{
    if (!rsrc_load("resource.rsc")) {
        form_alert(1, "[0][Cannot find resource.rsc file|Terminating ...][OK]");
        exit(2);
    }
}

/*
    Put up simple dialog to show how resources work.
*/
main()
{
    OBJECT  *dialog;
    int      x, y, w, h;

    /*
        Initiailize the ROMs.
    */
    appl_init();

    /*
        Load resources.
    */
    init_resources();

    /*
        Get address of dialog definition in memory.
    */
    rsrc_gaddr(0, PARMDLOG, &dialog);

    /*
        This next set of functions display a dialog and handle
           the dialogs events.  For a in depth description of
           what is begin done refer to the Form Manager.
    */
    form_center (dialog, &x, &y, &w, &h);
    form_dial   (FMD_START, 0, 0, 0, 0, x, y, w, h);
    objc_draw   (dialog, 0, 10, x, y, w, h);
    form_do     (dialog, 0);
    form_dial   (FMD_FINISH, 0, 0, 0, 0, x, y, w, h);

    /*
        Shut down the application.
    */
    appl_exit();
}
```

16

NAME

rsrc_free — frees the memory allocated during *rsrc_load*.

SYNOPSIS

```
int rsrc_free()
```

DESCRIPTION

rsrc_free release the memory allocated for resourced defined during the *rsrc_load* function.

DIAGNOSTICS

The result of the function is zero if an error occurs.

16

NAME

rsrc_gaddr — gets the address of a data structure in memory.

SYNOPSIS

```
int rsrc_gaddr(re_gtype, re_gindex,  re_gaddr)
    int       re_gtype;
    int       re_gindex;
    struct **re_gaddr;
```

DESCRIPTION

rsrc_gaddr returns the address of a specified OBJECT in the resource list. The type of the object is defined by the parameter **re_gtype**. The parameter **re_gindex** is an index into the object list specifying the object whose address is required. The address of the object is stored at the pointer whose address is defined in **re_gaddr**. The different types of objects whose address may be obtained is as follows:

re_gtype the type of data structure:

0	=	tree
1	=	OBJECT
2	=	TEDINFO
3	=	ICONBLK
4	=	BITBLK
5	=	string
6	=	imagedata
7	=	obspec
8	=	**te_ptext**
9	=	**te_ptmplt**
10	=	**te_pvalid**
11	=	**ib_pmask**
13	=	**ib_pdata**
14	=	**ib_ptext**
15	=	**ad_frstr** — the address of a pointer to a free string
16	=	**ad_frimg** — the address of a pointer to a free image

re_gindex the index of the data structure.

16

re_gaddr	the address of the data structure specified by **re_gtype** and **re_gindex**.

DIAGNOSTICS

The result of the function is zero if an error occurs.

NAME

rsrc_load — loads an entire resource file into memory.

SYNOPSIS

```
int rsrc_load(res_fname)
    char *res_fname;
```

DESCRIPTION

rsrc_load loads a resource file into memory and changes all offset values into specific addresses. The resource file that is loaded is specified by the path name **res_fname**. This routine searches for the file and finds its total size in bytes. Using the DOS allocate call, it allocates the memory space for the resource file. It then opens it and reads the resource into memory, and closes the file. It then makes the required updates to the file:

1. make the file device-specific to the screen's resolution.
2. link up all the `OBJECT` pointers, `TEDINFO` pointers, `ICONBLK` pointers and `BITBLK` pointers.
3. build the array of tree pointers.
4. store the address of the tree array in the application's Global Array.

DIAGNOSTICS

The result of the function is zero if an error occurs.

NAME

rsrc_obfix — Changes a resource object's coordinate system.

SYNOPSIS

```
int rsrc_obfix(res_tree, res_object)
    OBJECT *res_tree;
    int     res_object;
```

DESCRIPTION

rsrc_obfix converts an objects position and size from a character coordinate system to a pixel coordinate system. The object tree is defined by the parameter **res_tree**. The object to be converted is defined by the parameter **res_object** which is an index into the object tree **res_tree**.

DIAGNOSTICS

The result of the function is zero if an error occurs.

NAME

rsrc_saddr — stores an index to a data structure.

SYNOPSIS

```
int rsrc_saddr(res_type, res_index, res_addr)
    int     res_type;
    int     res_index;
    struct *res_addr;
```

DESCRIPTION

rsrc_saddr stores the address **res_addr** into the resource list. The *rsrc_gaddr* function returns the address of a specified type of object at a certain index in the resource list. If it becomes necessary to redefine the object in the resource list the *rsrc_saddr* function is used. The type of the resource to be changed is defined by the parameter **res_type**. The index into the resource type list is defined by the parameter **res_index**. The address to which the resource will now point for the specified object is defined by **res_addr**. The types of resources are:

re_stype the type of data structure.

0	=	tree
1	=	OBJECT
2	=	TEDINFO
3	=	ICONBLK
4	=	BITBLK
5	=	string
6	=	imagedata
7	=	obspec
8	=	**te_ptext**
9	=	**te_ptmplt**
10	=	**te_pvalid**
11	=	**ib_pmask**
13	=	**ib_pdata**
14	=	**ib_ptext**
15	=	**ad_frstr** — the address of a pointer to a free string
16	=	**ad_frimg** — the address of a pointer to a free image

DIAGNOSTICS

The result of the function is zero if an error occurs.

SEE ALSO

rsrc_gaddr

16

16.10 Scrap Manager

scrp_read	reads the scrap directory currently stored in the clipboard
scrp_write	writes the scrap directory to the clipboard

Introduction

The Scrap Manager provides a means for applications to share information.

There are two ways to gather data to be transmitted to another application. First, the data may be extracted from the source file leaving the only version of the data in the clipboard or, second, the data may be copied into the clipboard leaving the source file unaffected by the operation.

The target for a *scrap_write* procedure is the clipboard. It is also the source for a read.

The clipboard only keeps one element at the time. If two *scrap_write* operations are performed, one immediately following the other, the data from the *scrap_write* is overwritten by the data from the second.

The "clipboard" is merely AES keeping track of a directory where a scrap file may be stored. It is up to the application to conform to the standard conventions of creating the scrap file.

The convention states that the file name must be `SCRAP` and the filetype must convey the type of data.

The following types are defined by GEM:

Data Type	**File Extension**
Text strings	`.TXT`
Spreadsheet data	`.DIF`
Metafile	`.GEM`
Bit-images	`.IMG`

NAME

scrp_read — reads the scrap directory currently stored on the clipboard.

SYNOPSIS

```
int scrp_read(scrap_buf)
    char *scrap_buf;
```

DESCRIPTION

scrp_read reads the name of the scrap file from the "clipboard." The path name of this scrap file is stored at `scrap_buf`. The function result is zero if an error occurs.

NOTE

It is up to the application to create the space and read the data in from the scrap file. The "clipboard" is strictly a static piece of memory that may be used to communicate the path of the scrap file between applications.

SEE ALSO

scrp_write

NAME

scrp_write — writes the scrap directory to the clipboard.

SYNOPSIS

```
int scrp_write(scrap_fname)
    char *scrap_fname;
```

DESCRIPTION

scrp_write writes the path name of the scrap directory into an area called the clipboard. **scrap_fname** is the path name of the scrap file.

NOTE

It is up to the application to write the scrap data to the path name that is specified by the *scrp_write* function. The "clipboard" is strictly a static piece of memory used to pass path name information about the scrap file between applications.

16

16.11 Shell Manager

shel_envrn	Get an environment variable value
shel_find	Find an application pathname through DOS search path
shel_read	Get command line variables
shel_write	Launch another application

Introduction

The Shell Manager is a set of functions that may be used by an application to communicate with the outside environment. These functions include the manipulation of the application's command line and environment variables. They can also find and invoke other applications.

NAME

shel_envrn — get address of environment variable table.

SYNOPSIS

```
int shel_envrn(value, name)
    char **value, *name;
```

DESCRIPTION

shel_envrn searches the GEMDOS environment variable list for the name `value`. The form of an environment variable is `name=value`. If the variable name is not in the environment list a NULL pointer is returned.

`value`	a pointer to 4 byte area where the address of the environment variable value is located.
`name`	a character pointer to the name of the environment variable to be searched for. Note that the string should have an '=' sign at the end.

NOTE

16

This function will return a pointer to the byte following the first pattern that was matched with `name`.

DIAGNOSTICS

The result of the function is always 1.

SEE ALSO

getenv

NAME

shel_find — find the full pathname of an application.

SYNOPSIS

```
int shel_find(filename)
    char *filename;
```

DESCRIPTION

shel_find searches for the filename specified by the parameter `filename` using the DOS search path. The full pathname of the file, if found, will be stored at the memory pointed to by the parameter `filename`.

NOTE

The buffer must be large enough to hold the resulting pathname.

DIAGNOSTICS

The result of the function is 0 if an error ocurrs.

SEE ALSO

shel_write

NAME

shel_read — tells an application its name and parameters.

SYNOPSIS

```
int shel_read(programname, commandline)
    char *programname;
    char *commandline;
```

DESCRIPTION

shel_read is used to obtain the application's name and command line parameters when the application was launched.

`programname`	points to a buffer where the program name will be stored.
`commandline`	points to a buffer where the parameters to the program will be stored.

NOTE

The full pathname of the `programname` will be returned if the application does not reside in the current working directory.

DIAGNOSTICS

The result of the function is 0 if an error occurs.

SEE ALSO

shel_write, Progam Parameters (pg. 107)

NAME

shel_write — launch a new application.

SYNOPSIS

```
int shel_write(execcode, graftype, progtype, progname, cmdline)
    int   execcode, graftype, progtype;
    char *progname, *cmdline;
```

DESCRIPTION

shel_write exit GEM or launch new application.

`execcode`	execution code. (0 = Exit GEM, 1 = run new program)
`graftype`	program type. (0 = non graphic, 1 = graphic)
`progtype`	program type. (0 = non GEM, 1 = GEM)
`progname`	character pointer to name of program to launch.
`cmdline`	character pointer to new programs parameters.

DIAGNOSTICS

The result of the function is 0 if an error ocurrs.

SEE ALSO

shel_read, *Pexec*, Shell Introduction (pg. 275)

16

16.12 Window Manager

wind_calc	Calculate the window size
wind_close	Close a window
wind_create	Create a window
wind_delete	Remove a window ID from the window list
wind_find	Find a window under a point
wind_get	Get information about a window
wind_open	Open a created window
wind_set	Set values for display fields
wind_update	Tell GEM that a window has been updated

Introduction

The following picture shows the various components of a window:

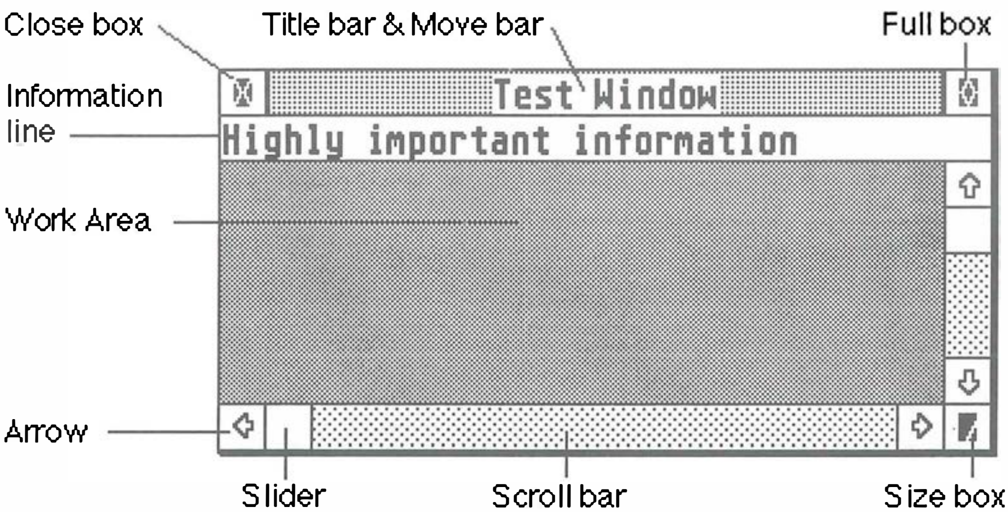


Figure 16.4: Sample Window

The window library supports the creation, deletion and updating of an application's windows. The `<gemdefs.h>` header file should be included in applications using the window library. An application can have up to eight windows simultaneously open. Each window is refered to by an integer known as a window handle.

The only components required for all windows are the title bar and work area. The optional components are called window control areas. The work area is maintained by the application, all other components are handled by the window library routines. The window library communicates information about user actions back to the application through message events. These events occur when a user mouse operation changes the window in some way. For example, by moving the window to a new location or pressing in a scroll bar. The application is notified about the event after the fact; the window library traps the mouse event and generates an appropriate window event for the application. The application can turn off the mouse event trapping and receive all mouse events itself by calling *wind_update* with the value BEG_MCTRL (see the manual page for more information). See the event library for a list of the 'WM_' messages generated by the window library.

Slider Usage

The ratio of the size of the slider to the size of the scroll bar should be the same as the size of the work area to the size of the object being displayed in the window. This allows the user to see at a glance exactly where in (by the position of the slider) and how much of (by the size) the complete object the window is. The size of the slider is specified as a percentage of the scroll bar's size to make this easy to do.

16

Move Bar

The move bar is overlaied on top of the title bar. It just enables mouse tracking and moving of the window when the user presses the mouse over the title bar. There is no visual clue that the move bar exists, the user will discover whether it exists or not when he tries to move the window.

Desktop Window

The menu bar and gray background is called the desktop window (although it doesn't look or work like a normal window). The window library automatically redraws the desktop whenever part of it becomes uncovered. The size of the gray region can be obtained by calling *wind_get* with a value of WF_WORKXYWH for window 0. *wind_get* is used to get information about windows, WF_WORKXYWH asks for the work area's (x, y) location and width and height. Application windows should be initially created to fit inside the desktop work area.

The gray background area can be replaced by an application defined object tree by calling *wind_set* with a value of WF_NEWDESK for window 0. This is the way most applications place icons on the desktop.

Creating a Window

A window is created by a sequence of calls that specify the initial size, the largest size, the location, and the type of window controls the window will have. These values may be changed later with *wind_set*.

The routine that creates the window wants the size of the perimeter of the window. This size is not nearly as important to the application as the size of the work area. The function *wind_calc* is used to calculate the perimeter (or border) size from the work area size and vice versa. The program should call *wind_calc* to determine the border area size for both the initial and largest sizes of the window.

The application calls *wind_create* after determining the size of the border area. *wind_create* reserves one of the eight windows and returns the handle number for it. It does not draw the window.

Next the application calls *wind_open* with the initial size. The window is drawn at this point. A WM_REDRAW event is also generated which will cause the application to draw the interior of the window in the normal course of event processing. The example below is used by the sample application to create a new window.

16

```
#include <gemdefs.h>
#include "globals.h"

/*
    new_window  -  create & draw a new window.

    1)  create the window.
    2)  draw the window with the wind_open()
    3)  create and setup the window record.
*/
windowptr new_window(thekind)
    int         thekind;
{
    int         handle;
    int         xdesk, ydesk, wdesk, hdesk;
    windowptr   thewin;
    static      window_count = 1;

    /*
        Get the desktop coordinates.
    */
    wind_get(0, WF_WORKXYWH, &xdesk, &ydesk, &wdesk, &hdesk);
```

```
/*
    Create the information for the window.
        Max size is the desktop.
*/
handle = wind_create(thekind, xdesk, ydesk, wdesk, hdesk);

/*
    Check for error.
*/
if (handle < 0) {
    paramdlog("Sorry! No more windows available.");
    return NULL;
}

/*
    Allocate space for window record.
*/
thewin                = (windowptr) malloc(sizeof(windowrec));

/*
    Set the title for the window.
*/
sprintf(thewin -> title, " Untitled %d ", window_count++);

wind_set(handle, WF_NAME, thewin -> title, 0, 0);

/*
    A little flim-flammery.
*/
graf_growbox(0, 0, 0, 0, xdesk, ydesk, wdesk/2, hdesk/2);

/*
    Draw the window.
*/
wind_open(handle, xdesk, ydesk, wdesk/2, hdesk/2);

/*
    Initialize window data structure.
*/
thewin -> next        = NULL;
thewin -> handle      = handle;
thewin -> kind        = thekind;
thewin -> fullsize    = FALSE;
thewin -> graf.handle = open_vwork(&thewin -> graf.mfdb);
thewin -> updateproc  = nullproc;

wind_get(handle, WF_WORKXYWH, &xdesk, &ydesk, &wdesk, &hdesk);
setrect(&thewin -> work, xdesk, ydesk, wdesk, hdesk);

wind_get(handle, WF_CURRXYWH, &xdesk, &ydesk, &wdesk, &hdesk);
```

16

```
    setrect(&thewin -> box, xdesk, ydesk, wdesk, hdesk);

    /*
        Insert into windowlist.
    */
    {
        register windowptr  winptr = (windowptr) &firstwindow;

        while(winptr -> next)
            winptr = winptr -> next;

        winptr -> next = thewin;
    }

    make_frontwin(thewin);

    return thewin;
}
```

Closing a Window

A window can be removed from the screen by calling *wind_close* with the window handle. The window is still allocated however and may be redrawn by calling *wind_open* again. If the window is not needed any longer then *wind_delete* should be called to return the window to the window library so it may be used by another application (a desk accessory for instance). The example below is used in the sample application to close and delete a window.

16

```
/*
    dispose_window - Closes the window and disposes the storage for
        the window record.
*/
dispose_window(thewin)
    windowptr   thewin;
{
    int x, y, w, h;
    int handle;

    handle = thewin -> handle;

    wind_close(handle);

    wind_get(handle, WF_CURRXYWH, &x, &y, &w, &h);

    graf_shrinkbox(0, 0, 0, 0, x, y, w, h);

    wind_delete(handle);
```

```
    {
        /*
            Remove window record from window list.
        */
        register windowptr  winptr = (windowptr) &firstwindow;

        while(winptr -> next)
            if (winptr -> next == thewin)
                break;
            else
                winptr = winptr -> next;

        if (winptr -> next)
            winptr -> next = winptr -> next -> next;
        else {
            paramdlog("Internal Error: Window pointer not in list.");
            shutdown(2);
        }

        /*
            Update the front window pointer.
        */
        if (!firstwindow)
            thefrontwin = NULL;
        else
            if (winptr == (windowptr) &firstwindow)
                make_frontwin(winptr -> next);
            else
                make_frontwin(winptr);

        /*
            Close workstation associated with window.
        */
        v_clsvwk(thewin -> graf.handle);

        /*
            Release window data structure storage.
        */
        free(thewin);
    }
}
```

16

Drawing and Updating a Window

The window library maintains a rectangle list of non-intersecting rectangles which together cover all visible parts of each opened window. This list is accessed by calling *wind_get* with WF_FIRSTXYWH to retrieve the first one, and WF_NEXTXYWH to retrieve the rest. The last rectangle is indicated by a width and height of zero.

The application must redraw a window when it receives a WM_REDRAW event. First it should call *wind_update*, passing a "1" to tell the window library not to change any rectangle lists while the window is redrawn.

A window handle and a rectangle to be redrawn is passed in the message buffer. The application should step through the window's rectangle list intersecting each rectangle with the rectangle passed in the message buffer and then redraw the window with clipping set to the resultant rectangle (see the VDI function *vs_clip*). When the entire window has been redrawn, the application should call *wind_update* again, but pass 0 to let the window library change the rectangle lists again. The clipping rectangle should be reset to the desktop work area also.

Any rectangular area of the screen can be invalidated by calling *form_dial* with a value of FMD_FINISH. The rectangle will be redrawn by a series of events for all windows covered by it. The example below is used by the sample application to update all the windows when a redraw event is received.

```
/*
    do_update - Update all of the windows affected by the
                update event.
*/
do_update(message)
    int *message;
{
    int  thewindow;
    rect r1, therect;

    thewindow = message[3];

    setrect(&therect, message[4], message[5], message[6], message[7]);

    wind_get(thewindow, WF_FIRSTXYWH, &r1.x, &r1.y, &r1.w, &r1.h);

    /*
        Cycle through rectangle list.
    */
    while (r1.w && r1.h) {
        if (rc_intersect(&therect, &r1)) {
            /*
                Set clipping so that drawing will only change
                    the changed area.
            */
            setclip(thewindow, &r1);

            /*
                Call the function to redraw the window.
            */
```

```
                update_window(thewindow);
            }

            wind_get(thewindow, WF_NEXTXYWH, &r1.x, &r1.y, &r1.w, &r1.h);
        }

        {
            int x, y, w, h;

            /*
                Restore clip rectangle to desktop rectangle.
            */
            wind_get(0, WF_WORKXYWH, &x, &y, &w, &h);
            setrect(&r1, x, y, x+w, y+h);
            vs_clip(phys_handle, 1, &r1);
        }
    }
```

Examples

The following example functions illustrate common functions that are used in a AES application for handling window events. These functions are used in a complete sample application that is supplied as part of an Laser Development System. The next example shows a method for resizing a window.

16

```
#define WIND_MINW 80
#define WIND_MINH 80

/*
    do_resize - redraws the window at it's new position and updates all
        of the window's position records.
*/
do_resize(message)
    int *message;
{
    int x, y, w, h;
    int handle;

    handle  = message[3];
    x       = message[4];
    y       = message[5];
    w       = message[6];
    h       = message[7];

    /*
        Make sure that the window doesn't become too small.
    */
    if (w < WIND_MINW) w = WIND_MINW;
```

```
    if (h < WIND_MINH) h = WIND_MINH;

    /*
        Redraw the window at it's new size.
    */
    wind_set(handle, WF_CURRXYWH, x, y, w, h);
    wind_get(handle, WF_WORKXYWH, &x, &y, &w, &h);

    {
        /*
            Set the Window record data.
        */
        windowptr   thewin;

        thewin = findwindowptr(handle);

        setrect(&thewin -> work, x, y, w, h);
        setrect(&thewin -> box, x, y, w, h);

        thewin -> fullsize = FALSE;
    }
}
```

This example illustrates the method for making the window become it's full size.

```
/*
    do_fullsize - draws the window at it's fully defined size.  If
        the window is at it's full size then this routines restores
        the window to it's previous size.
*/
do_fullsize(handle)
    int handle;
{
    register windowptr  thewin;

    int x, y, w, h;
    int d;

    thewin = findwindowptr(handle);

    if (thewin -> fullsize) {
        /*
            Back to normal size
        */
        wind_calc(WC_WORK, thewin -> kind,
                  thewin -> box.x, thewin -> box.y,
                  thewin -> box.w, thewin -> box.h,

            &thewin -> work.x, &thewin -> work.y,
```

16

```
            &thewin -> work.w, &thewin -> work.h);

        wind_set(handle, WF_CURRXYWH,
            thewin -> box.x, thewin -> box.y,
            thewin -> box.w, thewin -> box.h);

        thewin -> fullsize = FALSE;
    } else {
        /*
            Draw window at full size;
        */
        wind_get(handle, WF_FULLXYWH, &x, &y, &w, &h);
        wind_set(handle, WF_CURRXYWH, x, y, w, h);
        wind_calc(WC_WORK, thewin -> kind, x, y, w, h,
            &thewin -> work.x, &thewin -> work.y,
            &thewin -> work.w, &thewin -> work.h);

        thewin -> fullsize = TRUE;
    }
}
```

The next example shows a method for decoding and handling window events.

```
/*
    do_window - determines the type of window event and then calls
        the appropriate function to handle the event.
*/
do_window(message)
    int *message;
{
    int handle;

    handle = message[3];

    set_mouse(OFF);
    wind_update(BEG_UPDATE);

    switch (message[0]) {
        case WM_REDRAW:
            do_update(message);
        break;

        case WM_NEWTOP:
        case WM_TOPPED:
            make_frontwin(findwindowptr(handle));
        break;

        case WM_MOVED:
        case WM_SIZED:
            do_resize(message);
```

16

```
        break;

        case WM_FULLED:
            do_fullsize(handle);
        break;

        case WM_CLOSED:
            dispose_window(findwindowptr(handle));
        break;
    }

    wind_update(END_UPDATE);
    set_mouse(ON);
}
```

NAME

wind_calc — calculates the X- and Y-coordinates and the width and height of a window's work area or border area.

SYNOPSIS

```
int wind_calc(wi_ctype,  wi_ckind,
              wi_cinx,   wi_ciny,   wi_cinw,  wi_cinh,
              wi_coutx,  wi_couty,  wi_coutw, wi_couth)

    int  wi_ctype, wi_ckind;
    int  wi_cinx,   wi_ciny,   wi_cinw,   wi_cinh;
    int *wi_coutx, *wi_couty, *wi_coutw, *wi_couth;
```

DESCRIPTION

wind_calc calculates the X- and Y-coordinates and the width and height of a window's border area or work area. The parameter `wi_ctype` indicates the type of calculation that is to be performed. If `wi_ctype` contains the value 0 the function assumes that the input rectangle, `wi_cin*`, describes the size of the work area. The output rectangle will be the dimensions of the total window. If `wi_ctype` contains the value 1 the function assumes that the input rectangle describes the size of the entire window. The output rectangle will be the dimensions of the work area of the window.

16

`wi_ctype` the type of calculation to perform:

0	=	return border area X, Y, width, and height.
1	=	return work area X, Y, width, and height.

`wi_crkind` A bit vector of the window components used for the window in question.

The following bits represent the components:

0x0001 (NAME)	=	title bar with name
0x0002 (CLOSE)	=	close box
0x0004 (FULL)	=	full box
0x0008 (MOVE)	=	move box
0x0010 (INFO)	=	information line
0x0020 (SIZE)	=	size box
0x0040 (UPARROW)	=	up-arrow

`0x0080 (DNARROW)`	=	down-arrow
`0x0100 (VSLIDE)`	=	vertical slider
`0x0200 (LFARROW)`	=	left-arrow
`0x0400 (RTARROW)`	=	right-arrow

This call uses the following bit settings for each component:

0	=	does not have component.
1	=	has the component.

`wi_cinx` the input X-coordinate of the work area (if `wi_ctype` = 0) or border area (if `wi_ctype` = 1).

`wi_ciny` the input X-coordinate of the work area (if `wi_ctype` = 0) or border area (if `wi_ctype` = 1).

`wi_cinw` the input width of the work area (`wi_ctype` = 0) or border area (if `wi_ctype` = 1).

`wi_cinh` the input height of the work area (`wi_ctype` = 0) or border area (if `wi_ctype` = 1).

`wi_coutx` the output X-coordinate of the work area (if `wi_ctype` = 1) or border area (if `wi_ctype` = 1).

`wi_couty` the output Y-coordinate of thework area (if `wi_ctype` = 1) or border area (if `wi_ctype` = 0).

`wi_coutw` the output width of the work area (if `wi_ctype` = 1) or border area (if `wi_ctype` = 0).

`wi_couth` the output height of the work area (if `wi_ctype` = 1) or border area (if `wi_ctype` = 0).

DIAGNOSTICS

The result of the function will be zero if an error occurs.

16

NAME

wind_close — closes an open window.

SYNOPSIS

```
int wind_close(wi_clhandle)
    int wi_clhandle;
```

DESCRIPTION

wind_close closes an open window. The window to be closed is defined by the window handle `wi_clhandle`. Although the window is closed it's data structures remain in memory. The application can re-open the window by calling the *wind_open* function again.

DIAGNOSTICS

The result of the function will be zero if an error occurs.

SEE ALSO

wind_open

NAME

wind_create — allocates the application's full-size window and returns a handle.

SYNOPSIS

```
int wind_create(wi_crkind, wi_crwx, wi_crwy, wi_crww, wi_crwh)
    int wi_crkind;
    int wi_crwx;
    int wi_crwy;
    int wi_crww;
    int wi_crwh;
```

DESCRIPTION

wind_create creates a window definition in memory. A call to this routine allocates the application's full-size window and returns the window's handle (a integer value). The window's full size rectangle is defined by the parameters `wi_crwx`, `wi_crwy`, `wi_crww`, and `wi_crwh`. The result of the function will be the handle of the new created window.

`wi_crkind` A bit vector of the window components to include in the new window.

The following bits represent the components:

0x0001	(NAME)	=	title bar with name
0x0002	(CLOSE)	=	close box
0x0004	(FULL)	=	full box
0x0008	(MOVE)	=	move box
0x0010	(INFO)	=	information line
0x0020	(SIZE)	=	size box
0x0040	(UPARROW)	=	up-arrow
0x0080	(DNARROW)	=	down-arrow
0x0100	(VSLIDE)	=	vertical slider
0x0200	(LFARROW)	=	left-arrow
0x0400	(RTARROW)	=	right-arrow
0x0800	(HSLIDE)	=	horizontal slider

This call uses the following bit settings for each component:

0	=	does not have component.
1	=	has the component.

16

`wi_crwx`	the X-coordinate of the full-size window.
`wi_crwy`	the Y-coordinate of the full-size window.
`wi_crww`	the width (in pixels) of the full-size window.
`wi_crwh`	the height (in pixels) of the full-size window.

NOTE

The window's initial size is determined by the *wind_open* function.

DIAGNOSTICS

If a negative value is returned an error occurred during the creation of the window.

SEE ALSO

wind_open

NAME

wind_delete — de-allocates the application's window and handle.

SYNOPSIS

```
int wind_delete(wind_handle)
    int wind_handle;
```

DESCRIPTION

wind_delete release the memory allocated by the window and removes the window handle from the active window list. The window that is to be deleted is defined by the window handle `wind_handle`.

DIAGNOSTICS

The result of the function is zero if an error occurs.

SEE ALSO

wind_create

NAME

wind_find — find a window under a point on the screen.

SYNOPSIS

```
int wind_find(wi_fmx,  wi_fmy)
    int wi_fmx;
    int wi_fmy;
```

DESCRIPTION

wind_find returns the handle of the frontmost window under the pixel position (`wi_fms`, `wi_fmy`). A value of zero will be returned if only the desktop or background is visible at that location.

`wi_fmx`	the X-coordinate of the position.
`wi_fmy`	the Y-coordinate of the position.

NAME

wind_get — gets the information on the window specified by wi_ghandle.

SYNOPSIS

```
int wind_get(wi_ghandle, wi_gfield,
             wi_gw1, wi_gw2, wi_gw3, wi_gw4)

    int  wi_ghandle;
    int  wi_gfield;
    int *wi_gw1, *wi_gw2, *wi_gw3, *wi_gw4;
```

DESCRIPTION

wind_get returns information about the window with handle **wi_ghandle**. This routine can request information about the windows (x, y) position and size, the active window handle, the slider location and size, the window's current location and size, or the window's previous location and size.

wi_ghandle The handle of the window that the application wants information about.

wi_gfield An integer that tells the routine which information to pass back to the application program. The value of **wi_gfield** determines which of **wi_gw1**, **wi_gw2**, **wi_gw3**, and **wi_gw4** is returned.

4 (WF_WORKXYWH) — request the coordinates of the window work area.

wi_gw1	=	X-coordinate
wi_gw2	=	Y-coordinate
wi_gw3	=	width
wi_gw4	=	height

5 (WF_CURRXYWH) — request the coordinates of the entire current window.

wi_gw1	=	X-coordinate
wi_gw2	=	Y-coordinate
wi_gw3	=	width
we_gw4	=	height

16

6 (WF_PREVXYWH) — request the coordinates of the previous window.

wi_gw1	=	X-coordinate
wi_gw2	=	Y-coordinate
wi_gw3	=	width
wi_gw4	=	height

7 (WF_FULLXYWH) — request the coordinates of the fullsize window.

wi_gw1	=	X-coordinate
wi_gw2	=	Y-coordinate
wi_gw3	=	width
wi_gw4	=	height

8 (WF_HSLIDE) — request the relative position of the horizontal slider. It returns a number between 1 and 1000 which is the relative position of the slider. (1 = leftmost position; 1000 = rightmost position).

wi_gw1	=	Slider position

9 (WF_VSLIDE) — request the relative position of the vertical slider. It returns a number between 1 and 1000, giving the relative position of the vertical slider. (1 = topmost position; 1000 = bottom position).

wi_gw1	=	Slider position

10 (WF_TOP) — request the handle of the active window.

wi_gw1	=	Window Handle

11 (WF_FIRSTXYWH) — request the coordinates of the first rectangle in the window's rectangle list.

wi_gw1	=	X-coordinate
wi_gw2	=	Y-coordinate
wi_gw3	=	width
wi_gw4	=	height

12 (WF_NEXTXYWH) — request the coordinates of the next rectangle in the window's rectangle list.

`wi_gw1`	=	X-coordinate
`wi_gw2`	=	Y-coordinate
`wi_gw3`	=	width
`wi_gw4`	=	height

13 (`WF_RESVD`) — [Reserved].

15 (`WF_HSLSIZE`) — request the size of the horizontal slider.
−1 = default minimum size (a square box).
1 − 1000 = the slider's relative size compared to the horizontal scroll bar
`wi_gw1` = Slider size

16 (`WF_VSLSIZE`) — request the size of the vertical slider.
−1 = default minimum size (a square box).
1 − 1000 = the slider's relative size compared to the vertical scroll bar
`wi_gw1` = Slider size

`wi_gw1`, `wi_gw2`, `wi_gw3`, `wi_gw4` — The return values. The meaning of each is determined by `wi_gfield` above.

16

DIAGNOSTICS

Returns 0 if an error occurs.

NAME

wind_open — opens the created window to a specified size and location.

SYNOPSIS

```
int wind_open(wi_ohandle, wi_owx, wi_owy, wi_oww, wi_owh)
    int wi_ohandle;
    int wi_owx, wi_owy, wi_oww, wi_owh;
```

DESCRIPTION

wind_open draws a window onto the screen. This window's size is defined by `wi_oww` and `wi_owh` at the location (`wi_owx`, `wi_owy`).

`wi_ohandle`	the handle of the window to be opened (returned by *wind_create*).
`wi_owx`	the initial X-coordinate of the window.
`wi_owy`	the initial Y-coordinate of the window.
`wi_oww`	the initial width (in pixels) of the window.
`wi_owh`	the initial height (in pixels) of the window.

16

NOTE

It is necessary to obtain a window handle from the *wind_create* function before opening the window.

DIAGNOSTICS

The result of the function is zero if an error occurs.

SEE ALSO

wind_create

NAME

wind_set — sets new values for the display fields for a window.

SYNOPSIS

```
int wind_set(wi_shandle, wi_sfield, wi_sw1, wi_sw2,
    wi_sw3, wi_sw4)

int wi_shandle;
int wi_sfield;
int wi_sw1, wi_sw2, wi_sw3, wi_sw4;
```

DESCRIPTION

wind_set is used to change window attributes. The parameter `wi_shandle` defines the window whose attributes are to be changed. The attribute to be changed is defined in the parameter `wi_sfield`.

`wi_shandle` The handle of the window whose fields are to be changed.

`wi_sfield` A numerical value identifying the field to change:

1 (`WF_KIND`) — the components of the window (see `wi_crkind`, the *wind_create* routine).
`wi_sw1` = New `wi_crkind`

2 (`WF_NAME`) — a string containing the name of the window.
`wi_sw1` = High word of `char *`
`wi_sw2` = Low word of `char *`

3 (`WF_INFO`) — a string containing the information line.
`wi_sw1` = High word of `char *`
`wi_sw2` = Low word of `char *`

5 (`WF_CURRXYWH`) — defined under *wind_get*.

8 (`WF_HSLIDE`) — defined under *wind_get*.

9 (`WF_VSLIDE`) — defined under *wind_get*.

10 (`WF_TOP`) — defined under *wind_get*.

16

14 (WF_NEWDESK) — the address of a new default desktop for GEM AES to draw. Takes a tree (OBJECT *) and index of subtree to draw.

wi_sw1	=	High word of OBJECT *
wi_sw2	=	Low word of OBJECT *
wi_sw3	=	Subtree index

15 (WF_HSLSIZE) — defined under *wind_get*.

16 (WF_VSLSIZE) — defined under *wind_get*.

wi_sw1, wi_sw2, wi_sw3, wi_sw4 — The value depends on the field named **wi_sfield** above.

DIAGNOSTICS

The result of the function is zero if an error occurs.

EXAMPLE

16

```
/*
    make_frontwin - Force a window to the front.
*/
make_frontwin(thewin)
    windowptr thewin;
{
    wind_set(thewin -> handle, WF_TOP, 0, 0, 0, 0);
    thefrontwin = thewin;
}
```

SEE ALSO

wind_get

NAME

wind_update — notify GEM that a window update is in progress.

SYNOPSIS

```
int wind_update(wind_op)
    int wind_op;
```

DESCRIPTION

wind_update performs several functions that facilitate window updating. The parameter `wind_op` functions that are used during the update of a window's work area. If `wind_op` contains the code `BEGIN_UPDATE` GEM knows that the application is updating the screen and no further drawing will be done by GEM, (e.g. menus, mouse, etc.). When the update is completed the *wind_update* function is called again with the control code `END_UPDATE`.

This routine can also take control of the mouse functions, relieving the screen manager of its control over the mouse, menus, and window control points until it tells GEM AES that it again has control. This is done by passing *wind_update* a control code of `BEG_MCTRL`. When the application is finished with the mouse, control may be returned to GEM by calling *wind_update* with a control code of `END_MCTRL`.

16

NOTE

The udpate control codes are defined in the `gemdefs.h` header file.

DIAGNOSTICS

The result of the function is zero if an error occurs.

16

Chapter 17

VDI

Introduction

GEM VDI (Virtual device interface) handles all drawing and graphic I/O for GEM. VDI attempts to provide a program interface which is device and machine independent. VDI is designed to be able to drive a large variety of graphics hardware devices for both input and output. This is accomplished by using normalized values rather than device dependent values. The problem with normalized values is that they are slow (since everything must be converted to the actual device specific value) and suffer from round off errors which can make a program's output look trashy. For these reasons, most programs by-pass the normalizing mechanism and use device specific values.

17

A VDI drawing environment is called a workstation. It is referenced in a program by a number called a handle which is returned when the workstation is opened. A workstation specifies the output device, any input devices, the currently selected color, pattern, line width and many other attributes (it doesn't include a coordinate origin, however). All drawing is done in the absolute coordinate system of the output device. A program can choose to use normalized coordinates (NDC) or device specific raster coordinates (RC) when it opens a workstation. The GEM desktop program opens a workstation for the screen using raster coordinates. GEM can only have one workstation open for a particular device at a time, and since the desktop program is always run before user applications, there is no way for an application program to open a workstation on the ST. It can however open a virtual workstation that inherits the device specific information from the currently open workstation. Nevertheless, a program on the ST must use raster coordinates (which are probably prefered anyway).

The origin (location $(0,0)$) for raster coordinates is the upper left corner of the screen. Positive coordinates extend to the right and down from the origin. The range of values that appear on the screen is dependent upon the device resolution. Information about a raster (the term used for the memory where the pixels are stored) is passed in a structure called a Memory Form Definition Block:

```
typedef struct fbdstr {
    char *fd_addr;  /* address of raster memory      */
    int fd_w;       /* width of raster in pixels     */
    int fd_h;       /* height of raster in pixels    */
    int fd_wdwidth; /* fd_w / 16                     */
    int fd_stand;   /* 1=normalized, 0=raster coord. */
    int fd_nplanes; /* number of bits / pixel        */
    int fd_r[3];    /* reserved                      */
} MFDB;
```

An `fd_addr` value of -1 indicates the device doesn't have a bitmap. Although some VDI routines use MFDBs, the only way to get one is to create it yourself with the information returned after opening a virtual workstation.

Raster functions that operate on pixels combine a source (S) and destination (D) pixel according to a mode flag. The bitwise logical operation performed for a particular mode flag value is as follows:

Mode	Operation	Mode	Operation
0	set to 0	8	~(S | D)
1	S & D	9	~(S ^ D)
2	S & ~D	10	~D
3	S	11	S | ~D
4	~S & D	12	~S
5	D	13	~S | D
6	S ^ D	14	~(S & D)
7	S | D	15	set to 1

Parameters passed to (and results returned) from VDI routines are placed into five global arrays which *must* be defined in the application program:

```
int contrl[12], intin[128], ptsin[128], intout[128], ptsout[128];
```

These arrays are referenced by the GEM library interface routines, the application doesn't have to worry about them beyond just defining them.

Color numbers used in VDI routines refer to the ST color palette which provides three bits per color gun. The palette will have to be initialized if specific colors are required by an application since the palette is not reset each time a program is run.

VDI angles are integer values ten times the angle desired in degrees (i.e. 0 – 3600). Angles are measured counterclockwise from the positive x-axis.

Workstation Functions

v_opnwk	open workstation	v_clswk	close workstation
v_clrwk	clear workstation	v_updwk	update workstation
v_opnvwk	open virtual workstation	v_clsvwk	close virtual workstation

Output Functions

v_gtext	text output	v_justified	justified text
v_pline	polyline	v_pmarker	polymarker
v_fillarea	filled area	v_contourfill	contour fill
vr_recfl	fill rectangle	v_cellarray	cell array
v_rbox	rounded corner rectangle	v_rfbox	filled rounded corner rectangle
v_bar	bar	v_arc	arc
v_circle	circle	v_pieslice	pie
v_ellarc	elliptical arc	v_ellpie	elliptical pie
v_ellipse	ellipse	vs_clip	set clipping rectangle

17

Attribute Functions

vswr_mode	set writing mode	vs_color	set color palette entry
vsl_type	set polyline line pattern	vsl_udsty	set user-defined line pattern
vsl_width	set polyline line width	vsl_color	set polyline color
vsl_ends	set polyline end styles	vsm_type	set polymarker type
vsm_height	set polymarker height	vsm_color	set polymarker color
vst_height	set character height, absolute mode	vst_point	set character cell height, points mode
vst_rotation	set character baseline vector	vst_font	set text font
vst_color	set graphic text color	vst_effects	set graphic text special effects
vst_load_font	load extended fonts	vst_unload_font	unload extended fonts
vst_alignment	set graphic text alignment	vsf_interior	set fill interior style
vsf_style	set fill pattern	vsf_color	set fill color
vsf_perimeter	set fill perimeter visibility	vsf_udpat	set user defined fill pattern

Raster Functions

vro_cpyfm	copy raster, opaque	vrt_cpyfm	copy raster, transparent
vr_trnfm	transform form	v_get_pixel	get pixel

Input Functions

vsc_form	set mouse form	v_show_c	show cursor
v_hide_c	hide cursor	vq_mouse	sample mouse button state
vex_timv	exchange timer interrupt vector	vex_butv	exchange button change vector
vex_motv	exchange mouse movement vector	vex_curv	exchange cursor change vector
vq_key_s	sample keyboard state information		

Inquire Functions

vq_extnd	extended inquire function	vq_color	inquire color representation
vql_attributes	inquire polyline attributes	vqm_attributes	inquire polymarker attributes
vqf_attributes	inquire fill area attributes	vqt_attributes	inquire graphic text attributes
vqt_extent	inquire text extent	vqt_width	inquire character cell width
vqt_name	inquire face name and index	vq_cellarray	inquire cell array
vqt_fontinfo	inquire current face information		

Escapes

vq_chcells	inquire addressable character cells	v_exit_cur	exit alpha mode
v_enter_cur	enter alpha mode	v_curup	cursor up
v_curdown	cursor down	v_curright	cursor right
v_curleft	cursor left	v_curhome	cursor home
v_eeos	erase to end of screen	v_eeol	erase to end of line
vs_curaddress	direct cursor address	v_curtext	output cursor addressable text
v_rvon	reverse video on	v_rvoff	reverse video off

17.1 VDI Examples

Included with most of the function descriptions is a small example that illustrates the usage of the particular example. However, due to space constraints in the documentation it was necessary to omit the repetetive portion of the example which initialized the operating system. For each of the examples in the VDI it is necessary to add the example source to the file `vmain.c`. This file contains the code that initializes GEM, VDI, and calls the example function.

```
/*
    main() - This is the main function for the example VDI functions.
*/
#include <gemdefs.h>

/*
    Define VDI Global Variables
*/
```

```
int contrl[12];
int intin[256],  ptsin[256];
int intout[256], ptsout[256];

main()
{
    MFDB    theMFDB;     /*  Screen definition structure  */
    int     handle;      /*  Virtual Workstation Handle   */

    /*
        Start up ROM.
    */
    appl_init();
    handle = open_workstation(&theMFDB);

    /*
        Call example function here.
    */
    sample_function();

    /*
        Wait for a Carriage Return.
    */
    wait(handle);

    /*
        Close the virtual workstation, and shutdown application.
    */
    v_clsvwk(handle);
    appl_exit();
}
```


The file "**GRAFSTUF.C**" contains functions that some of the examples depend on. If an example calls a function defined in this source file then compile "**GRAFSTUF.C**" and link the example file with "**GRAFSTUF.O**".

NAME

v_arc — Arc; draw an arc

SYNOPSIS

```
int v_arc(handle, x, y, radius, start_angle, end_angle)
    int handle;
    int x, y;
    int radius;
    int start_angle, end_angle;
```

DESCRIPTION

This function draws a hollow arc centered on the point (x, y). The beginning and ending angles are in **start_angle** and **end_angle**. The angles are expressed in tenths of degrees (0 – 3600), clockwise, with the positive x-axis as 0. The radius is in **radius**, which is expressed in pixels. The arc is drawn with the current line attributes.

EXAMPLE

```
/*
    draw_arcs - show how to use the v_arc() vdi function.
        The handle that is passed as a parameter is the
        vdi workstation handle.  For further information
        refer to the vdi function v_opnvwk().
*/
draw_arcs(handle)
    int handle;
{
    int px          = 60;
    int py          = 70;
    int start_angle = 0;
    int end_angle   = 3600 - 900;
    int radius;

    for (radius = 10; radius < 40; radius += 10) {
        /*
            draw an arc.
        */
        v_arc(handle, px, py, radius, start_angle, end_angle);

        /*
            Make the arc larger while moving the start angle.
        */
        end_angle   += 300;
        start_angle += 300;
    }
}
```

17

SEE ALSO

vswr_mode, *vsl_color*, *vsl_type*, *vsl_width*, *vsl_ends*

NAME

v_bar — Draw a filled bar.

SYNOPSIS

```
int v_bar(handle, rect)
    int handle;
    int rect[4];
```

DESCRIPTION

The bar is drawn by placing the lower lefthand and the upper righthand corners into the array **rect**. The lower lefthand corner and the upper righthand corner are defined in the array as $[x_1, y_1, x_2, y_2]$ respectively. The bar is drawn with the current fill area attributes.

EXAMPLE

```
/*
    draw_bars  -  An example of how to use the v_bar() function to
        draw solid rectangles.  The parameter handle is the vdi
        workstation handle that is returned from the function
        v_opnvwk().
*/
draw_bars(handle)
    int handle;
{
    int rect[4];
    int px = 200;
    int py = 100;
    int y  = 90;
    int x;

    for (x=0; x < 100; x += 25, px += 25, y -= 10) {
        rect_set(rect, px, py, px+20, py-y);

        v_bar(handle, rect);
    }
}
```

SEE ALSO

vswr_mode, *vsf_interior*, *vsf_style*, *vsf_color*, *vsf_perimeter*

17

NAME

v_circle — Draw a Circle.

SYNOPSIS

```
int v_circle(handle, x, y, radius)
    int handle;
    int x, y;
    int radius;
```

DESCRIPTION

This function draws a solid circle with center (x, y) and with a radius in pixels defined by the parameter **radius**. The current fill area attributes will be used to fill the circle when it is drawn.

EXAMPLE

```
/*
    draw_circles  -  An example of how to use the v_circle()
        function to draw circles.  In this case a circle
        within a circle, ...  The parameter handle is the
        vdi workstation handle that is returned from the
        function v_opnvwk().
*/
draw_circles(handle)
    int handle;
{
    int radius;
    int px = 150;
    int py = 70;

    for (radius = 10; radius < 40; radius += 10)
        v_circle(handle, px, py, radius);
}
```

SEE ALSO

vswr_mode, *vsf_interior*, *vsf_style*, *vsf_color*, *vsf_perimeter*

NAME

v_clrwk — clear the workstation.

SYNOPSIS

```
int v_clrwk(handle)
    int handle;
```

DESCRIPTION

This function sets the defined workstation to its initial state. If the workstation is defined as the screen it is cleared to the background color (index 0). If the device is a printer a form feed is given to the device, and it's buffer is cleared. If the device is a plotter with manual paper load, the operator is prompted to load a new sheet. Finally, if the device is a metafile the opcode is flushed to the output file.

NAME

v_clsvwk — close virtual workstation

SYNOPSIS

```
int v_clsvwk(handle)
    int handle;
```

DESCRIPTION

This function closes a virtual workstation, preventing furthur output through the handle. All virtual workstations opened by a program should be closed before the program exits.

SEE ALSO

v_clswk, *v_opnvwk*

17

NAME

v_clswk — close the workstation defined by handle.

SYNOPSIS

```
int v_clswk(handle)
    int handle;
```

DESCRIPTION

v_clswk closes the workstation device and prevents any further output to be received by the device. If the device was a printer, then an update results unless one occurred previously. For screens, the graphics device is released, and the alpha device is selected. For metafiles, the buffer is flushed and the metafile closed.

NOTE

You should close virtual work stations before closing the workstation.

SEE ALSO

v_clsvwk, *v_opnwk*

NAME

v_contourfill — Contour Fill; flood or seed fill, fill an area to the edge or a color.

SYNOPSIS

```
int v_contourfill(handle, x, y, color)
    int handle;
    int x,y;
    int color;
```

DESCRIPTION

This fills an area to either the edge of the display surface, or a specified color. Also called flood or seed fill, the algorithm starts coloring at a seed (x, y) and colors the area until it reaches the color specified by the parameter `color`. This parameter is an index into the workstation's color table. If `color` is negative, the function will fill the area until any color other than the color at the seed. The area is filled using the current fill area attributes other than fill perimeter.

EXAMPLE

```
do_fill(handle)
    int handle;
{
    int x     = 300;
    int y     = 70;
    int color = 1;

    /*
        Draw an empty circle
    */
    v_arc(handle, x, y, 70, 0, 3600);

    /*
        Fill the circle
    */
    v_contourfill(handle, x, y, color);
}
```

SEE ALSO

vswr_mode, *vsf_interior*, *vsf_style*, *vsf_color*, *vq_extnd*

17

NAME

v_curdown, v_curhome, v_curleft, v_curright, v_curup — Cursor movement operations.

SYNOPSIS

```
int v_curdown(handle)
    int handle;

int v_curhome(handle)
    int handle;

int v_curleft(handle)
    int handle;

int v_curright(handle)
    int handle;

int v_curup(handle)
    int handle
```

DESCRIPTION

17

v_curdown moves the cursor down one row; unless the cursor is on the bottom, in which case, the cursor stays put.

v_curhome moves the cursor to the home position, generally the upper left corner cell.

v_curleft moves the cursor left one column, but not past the left margin.

v_curright moves the cursor right one column. It will not move the cursor past the right margin.

v_curup moves the cursor up one row; unless the cursor is at the top, in which case the cursor is not moved.

EXAMPLE

An example of the VDI cursor movement routines is shown on the examples disk `vcursor.c`.

SEE ALSO

v_enter_cur

NAME

v_curtext — Output Cursor Addressable Alpha Text

SYNOPSIS

```
int v_curtext(handle, string)
    int   handle;
    char *string;
```

DESCRIPTION

This prints `string` starting at the current cursor location. The current alpha text attributes are used for the text attributes (reverse or standard video).

EXAMPLE

```
print_hello(handle)
    int handle;
{
    char *string = "Hello world ...";

    v_curtext(handle, string);
}
```

SEE ALSO

v_enter_cur, *v_cursor movement*, *v_rvon*, *v_rvoff*

NAME

v_eeol — Erase to End of Alpha Text Line

SYNOPSIS

```
int v_eeol(handle)
    int handle;
```

DESCRIPTION

This erases the text from the present cursor location to the end of the line. The cursor location remains the same.

SEE ALSO

v_enter_cur, *v_cursor movement*

NAME

v_eeos — Erase to End of Alpha Screen

SYNOPSIS

```
int v_eeos(handle)
    int handle;
```

DESCRIPTION

The *v_eeos* function erases the text from the current cursor position to end of screen. The cursor location is not changed.

EXAMPLE

```
clear_screen(handle)
    int handle;
{
    int row = 1;
    int col = 1;

    /*
        Place the cursor at the top left part of screen.
    */
    vs_curaddress(row, col);

    /*
        Clear to the end of the screen.
    */
    v_eeos(handle);
}
```

SEE ALSO

v_enter_cur, *v_cursor movement*

NAME

v_ellarc — Elliptical Arc

SYNOPSIS

```
int v_ellarc(handle, x, y, xradius, yradius, start_angle,
                end_angle)
    int handle;
    int x,y;
    int xradius, yradius;
    int start_angle, end_angle;
```

DESCRIPTION

v_ellarc draws a hollow elliptical arc with the center at (x, y) and the beginning and ending angles in `start_angle` and `end_angle`. The x and y radius, defined in pixels, are in `xradius` and `yradius`. The arc is drawn using the current line attributes.

EXAMPLE

```
/*
    draw_ellarc  -  An example of how to use the v_ellarc()
        function to draw elliptical hollow arcs.

    Note: circular drawing functions use tenth's of degress
        for angles.
*/
draw_ellarc(handle)
    int handle;
{
    int x           = 50;
    int y           = 130;
    int xradius     = 10;
    int yradius     = 30;
    int start_angle = 0;
    int end_angle   = 3600;

    /*
        Draw the elliptical arc.
    */
    v_ellarc(handle, x, y, xradius, yradius, start_angle, end_angle);
}
```

SEE ALSO

vswr_mode, *vsl_type*, *vsl_width*, *vsl_color*, *vsl_ends*

17

NAME

v_ellipse — draw an ellipse.

SYNOPSIS

```
int v_ellipse(handle, x, y, xradius, yradius)
    int handle;
    int x,y;
    int xradius, yradius;
```

DESCRIPTION

This routine draws a filled ellipse with the center at (x, y). The x-radius and the y-radius are defined by **xradius** and **yradius** in pixels. This function uses the current fill area attributes.

EXAMPLE

```
/*
    draw_ellipse - An example of how to use the function v_ellipse().
        This example will draw a solid ellipse at the point x, y.
*/
draw_ellipse(handle)
    int handle;
{
    int x       = 180;
    int y       = 130;
    int xradius = 40;
    int yradius = 10;

    v_ellipse(handle, x, y, xradius, yradius);
}
```

SEE ALSO

vswr_mode, *vsf_interior*, *vsf_style*, *vsf_color*, *vsf_perimeter*

NAME

v_ellpie — draw an elliptical Pie Slice

SYNOPSIS

```
int v_ellpie(handle, x, y, xradius, yradius, start_angle,
                end_angle)
    int handle;
    int x,y;
    int xradius, yradius;
    int start_angle, end_angle;
```

DESCRIPTION

v_ellpie draws a filled elliptical pie slice with its center at (x, y) and the beginning and ending angles in **start_angle** and **end_angle**. The x and y radius, defined in pixels, are in **xradius** and **yradius**. This function uses the current fill area attributes.

EXAMPLE

```
/*
    draw_ellpie  -  An example of how to use the function v_ellpie() to
        draw and elliptical pie slice.  This function will use the
        current fill attributes when drawing the slice of pie.
*/
draw_ellpie(handle)
    int handle;
{
    int x           = 260;
    int y           = 130;
    int xradius     = 30;
    int yradius     = 10;
    int start_angle = 0;
    int end_angle   = 1200;

    v_ellpie(handle, x, y, xradius, yradius, start_angle, end_angle);
}
```

17

NOTE

All angles are expressed in tenths of degrees.

SEE ALSO

vswr_mode, *vsf_interior*, *vsf_style*, *vsf_color*, *vsf_perimeter*

NAME

v_enter_cur — Enter Alpha Mode

SYNOPSIS

```
int v_enter_cur(handle)
    int handle;
```

DESCRIPTION

This switches from graphics mode to alpha mode, or cursor addressing mode (text). The function will clear the screen and leave the cursor in the upper left character cell.

SEE ALSO

v_cursor movement, *v_exit_cur*

NAME

vex_butv — Exchange Button Change Vector.

SYNOPSIS

```
int vex_butv(handle, user_code_ptr, save_area_ptr)
    int     handle;
    void  (*user_code_ptr)();
    long   *save_area_ptr;
```

DESCRIPTION

vex_butv will change the interrupt vector for the mouse button handler to point to a user defined interrupt handler. This allows the user to write a routine which will be executed each time a mouse button changes state. The parameter `user_code_ptr` is the address of the function to be executed during the interrupt. The parameter `save_area_ptr` points to a 4 byte area where the address of the old interrupt handler will be stored.

The new interrupt routine will be executed from a JSR instruction with the interrupts disabled and should exit by an RTS instruction. The state of the mouse keys will be passed in the lower 16 bits of the 68000's `D0` register. The least significant bit of the word will contain the state of the leftmost mouse button with a 1 indicating that the button has been depressed.

17

The user routine receives control after the buttons are decoded, but prior to the driver, so any changes made to the `D0` register before exiting will affect the driver's knowledge of the button states.

NOTE

Preserve the states of any registers that are used during the interrupt, and do not enable the interrupts.

EXAMPLE

```
int     leftbutton, rightbutton;
long    oldmouse;

set_mymouse(handle)
    int handle;
{
    extern mymouse();

    vex_butv(handle, mymouse, &oldmouse);
}

restore_mouse(handle)
```

```
    int handle;
{
    long dummy;

    vex_butv(handle, oldmouse, &dummy);
}

mymouse()
{
    unsigned buttonstate;

    /*
        Save registers used by compiler and
        move button state into local var.
    */
    asm {
            movem.l A0-A1/D1-D2, -(A7)
            move    D0, buttonstate(A6)
    }

    /*
        Handle the button event
    */
    leftbutton = buttonstate & 1;
    rightbutton = buttonstate & 2;

    /*
        Restore the registers used and put the
        new button state into D0.
    */
    asm {
            movem.l (A7)+, A0-A1/D1-D2
            move    buttonstate(A6), D0
    }
}
```

SEE ALSO

Available Registers (pg. 24)

NAME

vex_curv — Exchange Cursor Change Vector.

SYNOPSIS

```
int vex_curv(handle, user_code_ptr, save_area_ptr)
    int     handle;
    void  (*user_code_ptr)();
    long   *save_area_ptr;
```

DESCRIPTION

vex_curv will change the vector for the mouse cursor drawing routine to a user defined drawing routine. This allows the user to write a function which will be executed each time the mouse cursor is drawn. The parameter `user_code_ptr` is the address of the function that will be executed. The parameter `save_area_ptr` points to a 4 byte area where the address of the old drawing routine will be stored.

The new drawing routine will be receive control from a JSR instruction with the interrupts disabled and should exit by an RTS instruction. The x location of the new cursor is passed in lower 16 bits of the 68000's `D0` register, and the new y is passed in the lower 16 bits of the `D1` register.

NOTE

17

Preserve the states of any registers that are used during the interrupt, and do not enable the interrupts.

EXAMPLE

```
long old_mouse_draw;

set_mouse_draw(handle)
    int handle;
{
    extern my_mouse_draw();

    vex_curv(handle, my_mouse_draw, &old_mouse_draw);
}

restore_mouse_draw(handle)
    int handle;
{
    long dummy;

    vex_curv(handle, old_mouse_draw, &dummy);
}
```

```
my_mouse_draw()
{
    unsigned    mousex, mousey;

    /*
        Save registers used by compiler and move mouse position
        into local variables.
    */
    asm {
            movem.l A0-A1/D0-D2, -(A7)
            move    D0, mousex(A6)
            move    D1, mousey(A6)
    }

    /*
        Draw the mouse cursor.
    */
    a_fillrect(mousex, mousey, mousex+16, mousey+16);

    /*
        Restore the registers used.
    */
    asm {
            movem.l (A7)+, A0-A1/D0-D2
    }
}
```

17

NAME

v_exit_cur — Exit Alpha Mode

SYNOPSIS

```
int v_exit_cur(handle)
    int handle;
```

DESCRIPTION

This function is used to exit cursor addressing mode, and to enter graphics mode.

SEE ALSO

v_enter_cur, v_cursor movement

NAME

vex_motv — Exchange mouse movement vector.

SYNOPSIS

```
int vex_motv(handle, user_code_ptr, save_area_ptr)
    int    handle;
    void  (*user_code_ptr)();
    long   *save_area_ptr;
```

DESCRIPTION

vex_motv will change the interrupt vector for the mouse handler to point to a user defined interrupt handler. This allows the user to write a routine which will be executed each time the mouse is moved. The parameter `user_code_ptr` is the address of the function to be executed during the interrupt. The parameter `save_area_ptr` points to a 4 byte area where the address of the old interrupt routine will be stored.

The new interrupt routine will be executed from a JSR instruction with the interrupts disabled and should exit by an RTS instruction. The x location of the mouse is passed in lower 16 bits of the 68000's `D0` register, and the new y is passed in the lower 16 bits of the `D1` register.

The user routine receives control after the new (x, y) position is computed, but prior to the driver receiving the information. This means that any changes that are made to the `D0` or `D1` registers will affect the driver's knowledge of the mouse's position.

17

NOTE

Preserve the states of any registers that are used during the interrupt, and do not enable the interrupts.

EXAMPLE

```
long old_mousexy;

set_mousexy(handle)
    int handle;
{
    extern mousexy();

    vex_motv(handle, mousexy, &old_mousexy);
}

restore_mousexy(handle)
    int handle;
```

```
{
    long dummy;

    vex_motv(handle, old_mousexy, &dummy);
}

mousexy()
{
    /*
        Save registers used in interrupt function and set up local
        variables to use in function.
    */
    asm {
            movem.l A0-A1/D0-D2, -(A7)
            move    D0, mousex
            move    D1, mousey
    }

    /*
        Work with the new (x,y) position of the mouse.
    */
    if (mousex > 300)
        mousex = 300;
    if (mousey > 150)
        mousey = 150;

    /*
        Restore registers changed during interrupt and reset D0 & D1
        to contain the modified mouse (x, y) coordinates.
    */
    asm {
            movem.l (A7)+, A0-A1/D0-D2
            move    mousex, D0
            move    mousey, D1
    }
}
```

17

NAME

vex_timv — Exchange Timer Interrupt Vector.

SYNOPSIS

```
int vex_timv(handle, user_code_ptr, save_area_ptr, mils_per_tick)
    int   handle;
    void  (*user_code_ptr)();
    long   *save_area_ptr;
    int    *mils_per_tick;
```

DESCRIPTION

vex_timv will change the interrupt vector for the timer interrupt handler to point to a user defined interrupt handler. This allows the user to write a routine which will be executed each time the timer clock ticks. The parameter `user_code_ptr` is the address of the function to be executed during the interrupt. The parameter `save_area_ptr` points to a 4 byte area where the address of the old interrupt routine will be stored. The last parameter `mils_per_tick` is a pointer to a 2 byte area where the number of milliseconds per tick will be stored.

The new interrupt routine will be executed from a JSR instruction with the interrupts disabled and should exit by an RTS instruction.

NOTE

Preserve the states of any registers that are used during the interrupt, and do not enable the interrupts.

17

EXAMPLE

```
long tickcount, old_timer;

set_timer(handle)
    int handle;
{
    extern mytimer();
    int    mils_per_tick;

    vex_timv(handle, mytimer, &old_timer, &mils_per_tick);
}

restore_timer(handle)
    int handle;
{
    long dummy;

    vex_timv(handle, old_timer, &dummy, &dummy);
```

```
}

mytimer()
{
    /*
        Preserve register states
    */
    asm {
            movem.l A0-A1/D0-D2, -(A7)
    }

    /*
        Handle the tick event.
    */
    tickcount++;

    /*
        Restore register states.
    */
    asm {
            movem.l (A7)+, A0-A1/D0-D2
    }
}
```

NAME

v_fillarea — fill a complex polygon.

SYNOPSIS

```
int v_fillarea(handle, count, points)
    int handle;
    int count;
    int points[][2];
```

DESCRIPTION

v_fillarea fills a complex polygon defined in the parameter `points`.

`points` contains a series of points which define the lines in the polygon.

`count` contains the number of points in the polygon array. The lines are drawn begining at `points[0]` and continuing through `points[count-1]`. The current fill area attributes are used when drawing the polygon.

EXAMPLE

```
do_fillarea(handle)
{
    int cx      = 100;
    int cy      = 100;
    int count   = 5;
    int points[5][2];

    /*
        Create a diamond.
    */
    pt_set(points[0], cx     , cy - 50);
    pt_set(points[1], cx + 50, cy);
    pt_set(points[2], cx     , cy + 50);
    pt_set(points[3], cx - 50, cy);
    pt_set(points[4], cx     , cy - 50);

    /*
        Now fill the diamond.
    */
    v_fillarea(handle, count, points);
}
```

SEE ALSO

vsf_perimeter, *vsf_interior*, *vsf_color*, *vswr_mode*, *vsf_style*

NAME

v_get_pixel — Get Pixel

SYNOPSIS

```
int v_get_pixel(handle, x, y, state, color)
    int handle
    int x, y;
    int *state;
    int *color;
```

DESCRIPTION

v_get_pixel returns the color and state of the pixel at then point (x, y).

The parameters x and y represent the point where the pixel to be checked is present. The variable `state` is a pointer to a two byte location where the state of the pixel is to be stored. A one will be stored if the the pixel is set and a zero will be stored if the pixel is not set. The last parameter `color` is a pointer to a two byte area where the color index of the pixel is stored.

EXAMPLE

```
#define ON 1

check_pixel(handle)
    int handle;
{
    int x = 100;
    int y = 100;
    int state;
    int color;

    v_get_pixel(handle, x, y, &state, &color);

    printf("The Pixel at (%d, %d) is ", x, y);

    if (state == ON)
        puts("on");
    else
        puts("off");
}
```

SEE ALSO

v_opnwk, *vq_extnd*

NAME

v_gtext — text; write text to the display

SYNOPSIS

```
int v_gtext(handle, x, y, text)
    int   handle;
    int   x;
    int   y;
    char *text;
```

DESCRIPTION

v_gtext writes the string, defined by the character pointer `text`, to the display device. The string is written at the reference position: (x, y). The relationship between the reference position and the actual location of the text on the display is determined by the Set-Graphic-Text-Alignment function, *vst_alignment*. By default the alignment of the string is the left baseline position.

If a character is not defined by the character set, an undefined character symbol is displayed.

EXAMPLE

```
drawtext(handle)
    int handle;
{
    int x = 100;
    int y = 100;
    char *text = "Hello, World ...";

    v_gtext(handle, x, y, text);
}
```

SEE ALSO

vst_alignment, *vst_height*, *vst_rotation*, *vst_font*, *vst_color*, *vst_effects*

17

NAME

v_hide_c — Hide Cursor

SYNOPSIS

```
int v_hide_c(handle)
    int handle;
```

DESCRIPTION

v_hide_c makes the mouse cursor invisible. The cursor visibility may be "nested" to any depth. Every call to *v_hide_c* must be balanced with a call to *v_show_c*. The cursor may be shown at any time with a call to *v_show_c* with the reset parameter set to zero.

The parameter `handle` is the virtual device handle obtained from the *v_opnvwk* call.

SEE ALSO

v_show_c

NAME

v_justified — Justify Graphics Text; write justified text to the device.

SYNOPSIS

```
int v_justified(handle, x, y, string, length, word_space,
                char_space)
    int handle;
    int x, y;
    int length;
    int word_space;
    int char_space;
    char *string;
```

DESCRIPTION

The *v_justified* outputs left and right justified text to the device, starting at the alignment point (x, y). Extra spacing may be inserted or deleted between words and/or characters so that the string is the expected length. The inter-word spacing modification is determined by the value of `word_space`. If it is set to TRUE, then the inter-word spacing modification is used. If the value of `char_space` is set to TRUE, then the inter-character spacing is used.

The desired output length of the string, in x-coordinate units, is the value of `length`. The string is in `string`.

This function uses the current text attributes.

17

EXAMPLE

```
justtext(handle)
    int handle;
{
    int x = 100;
    int y = 150;
    char *text = "Hello, World";

    v_justified(handle, x, y, text, 150, 0, 1);
}
```

SEE ALSO

vst_height, *vst_rotation*, *vst_font*, *vst_color*, *vst_effects*

NAME

v_opnvwk — open virtual workstation

SYNOPSIS

```
v_opnvwk(work_in, handleptr, work_out)
    int  work_in[11];
    int *handleptr;
    int  work_out[57];
```

DESCRIPTION

This function creates a virtual workstation from an existing physical workstation for a device. A workstation is a drawing environment; it defines all attributes used by VDI functions. Only one physical workstation is allowed per device. The screen's workstation is opened by GEM Desktop, so virtual workstations must be used by all applications running under GEM Desktop.

The parameters `work_in` and `work_out` are described in the function description of *v_opnwk*. The difference between the call to *v_opnwk* and *v_opnvwk* is that the function *v_opnvwk* requires the parameter `handleptr` to point to a handle of an open physical workstation.

EXAMPLE


```
/*
    open_workstation  -  Open a VDI virtual workstation.

    Note:
      information about the workstation is returned in the
      parameter 'form'. appl_init() must be called previously.
*/
int open_workstation(form)
    register MFDB   *form;
{
    register int x;
    int      work_in[11];
    int      work_out[57];
    int      handle;
    int      dummy;
    int      GDOS = 0;

    /*
        Does GDOS exist?
    */
    asm {
            move.w  #-2, D0

            trap    #2
```

```
        cmp.w   #-2, D0
        beq     gdos_not_installed
        move.w  #1, GDOS(A6)
gdos_not_installed:
}

/*
    Initialize workstation variables.
*/
if (GDOS)
    work_in[0] = Getrez() + 2;
else
    work_in[0] = 1;

for(x=1; x<10; x++)
    work_in[x] = 1;

/*
    Set for Raster Coordinate System.
*/
work_in[10] = 2;

/*
    Open Virtual Workstation
*/
handle = graf_handle(&dummy, &dummy, &dummy, &dummy);
v_opnvwk(work_in, &handle, work_out);

/*
    Check for error.
*/
if (!handle) {
    Cconws("\033E Error: Cannot open Virtual Device");
    Bconin(2);
    exit(1);
}

/*
    Set up the Memory Form Definition Block (MFDB).  This
    structure is defined in <gemdefs.h>.
*/

/*
    The Base address of the drawing screen.
*/
form -> fd_addr = Logbase();

/*
    The width of the screen in pixels.
```

```
    */
    form -> fd_w    = work_out[0] + 1;

    /*
        The height of the screen in pixels.
    */
    form -> fd_h    = work_out[1] + 1;

    /*
        The number of words in the width of the screen.
    */
    form -> fd_wdwidth  = form -> fd_w / 16;

    /*
        Working in a raster coordinate system.
    */
    form -> fd_stand       = 0;

    /*
        The number of drawing planes.
    */
    switch(work_out[13]) {
        case 16: form -> fd_nplanes = 4; break;
        case 08: form -> fd_nplanes = 3; break;
        case 04: form -> fd_nplanes = 2; break;
        default: form -> fd_nplanes = 1; break;
    }

    /*
        Return the workstation handle.
    */
    return handle;
}
```


NOTE

Not all input devices associated with the virtual workstation will work.

SEE ALSO

v_opnwk

NAME

v_opnwk — initialize a workstation.

SYNOPSIS

```
v_opnwk(work_in, handle, work_out)
    int  work_in[11];
    int *handle;
    int  work_out[57];
```

DESCRIPTION

This function prepares a workstation for use. It initializes the workstation to the parameters in work_in, and places information about the workstation in handle and work_out. The display of the workstation is cleared and set to graphics mode.

A failure to open or initialize the device returns a zero as the device handle.

work_in[0] Device id number. The drivers loaded are determined by the file "assign.sys"

1	Screen	11	Plotter
21	Printer	31	Metafile
41	Camera	51	Tablet

[1] Linetype

1	solid	5	short dashes
2	long dashes	6	dash, dot, dot
3	dots	7	user defined
4	dashes plus dots	> 7	device dependent

[2] Polyline color index. See page 349.

[3] Marker type

1	dot	5	diagonal cross
2	plus sign	6	diamond
3	asterisk	7	device dependent
4	square		

[4] Polymarker color index. See page 349.

[5] Text face; refer to *vqt_font_info* description

[6] Text color index. See page 349.

[7]	Fill interior style 0 = holow 1 = solid 2 = patterned 3 = cross-hatched 4 = user defined.	
[8]	Fill style index; refer to *vsf_interior*	
[9]	Fill color index. See page 349.	
[10]	NDC to RC transformation flag 0 = Map the full NDC space to the full RC space 1 = Reserved 2 = Use the RC system	

The following data is returned by *v_opnwk*:

work_out [0]	Addressable width of device in rasters or steps. A value of 512 means one could address from 0 – 512.
[1]	Height of device in rasters or steps.
[2]	Device Coordinate units flag; tells if the image can be precisely scaled as on a printer, or only close as on a a film recorder. 0 = precise scaling. 1 = no precise scaling.
[3]	Micron width of one addressable unit for the device.
[4]	Micron height of one addressable unit of the device.
[5]	Number of character heights, or zero if the device has continuous scaling.
[6]	Number of line types
[7]	Number of line widths, or zero if the device has continuous scaling.
[8]	Number of marker types
[9]	Number of marker sizes, or zero if the device has continuous scaling.
[10]	Number of type faces supported by the device, not the highest numbered face.
[11]	Number of patterns
[12]	Number of hatch styles
[13]	Number of available, predefined colors the device can display at one time.

17

[14] Number of Generalized Drawing Primitives (GDP).

[15 - 24] The GDPs supported by the device. If the device supports less than 10, the list will be terminated be a −1. The 10 GEM VDI GDPs will be represented by the following numbers:

1	=	Bar
2	=	Arc
3	=	Pie slice
4	=	Circle
5	=	Ellipse
6	=	Elliptical arc
7	=	Elliptical pie
8	=	Rounded rectangle
9	=	Filled rounded rectangle
10	=	Justified graphics text

[25 - 34] a list of attributes available for each GDP above:

0	=	Polyline
1	=	Polymarker
2	=	Text
3	=	Fill area
4	=	None

[35] Color; 1 if capable, 0 if not

[36] Text rotation; 1 if capable, 0 if not

[37] Fill area; 1 if capable, 0 if not

[38] Cell array operations; 1 if capable, 0 if not

[39] Number of available colors. Zero indicates the device has more than 32767, while 2 indicates black and white.

[40] Locator devices: 1 – keyboard only, 2 – keyboard and something else.

[41] Valuator device: 1 – keyboard only, 2 – another available

[42] Keypads: 1 – function keys on keyboard, 2 – another available

[43] String devices: 1 – keyboard

[44] Workstation type:

0	=	output only
1	=	input only
2	=	input/output
3	=	reserved
4	=	metafile output

17

The following dimensions are all in the current coordinate system.

[45]	Minimum character width
[46]	Minimum character height, excluding extends.
[47]	Maximum character width
[48]	Maximum character height, excluding extends
[49]	Minimum line width (x-axis). Line widths of 1 device unit may not display.
[50]	0
[51]	Maximum line width
[52]	0
[53]	Minimum marker width
[54]	Minimum marker height
[55]	Maximum marker width
[56]	Maximum marker height

The default values for certain VDI attributes are listed in the following table.

Defaults

Attribute	Default Value
Character Height	Nominal character height
Character baseline rotation	0 degrees
Text alignment	Left baseline
Text Style	Normal intensity
Line width	Nominal line width
Marker height	Nominal marker height
Polyline end styles	Squared
Writing mode	Replace
Input mode	Request for all input classes
Fill area perimeter visibility	Visible
User-defined line style	Solid
User-defined fill pattern	Solid
Cursor	Hidden
Clipping	Disabled

17

The default assignment of colors to color indices is shown in the table on the next page:

Default Color Index Values

0	White	8	White
1	Black	9	Black
2	Red	10	Light Red
3	Green	11	Light Green
4	Blue	12	Light Blue
5	Cyan	13	Light Cyan
6	Yellow	14	Light Yellow
7	Magenta	15	Light Magenta

Color numbers 16 and greater are device dependent.

SEE ALSO

vq_extnd

NAME

v_pieslice — Pie slice; draw a pie slice

SYNOPSIS

```
int v_pieslice(handle, x, y, radius, start_angle, end_angle)
    int handle;
    int x, y;
    int radius;
    int start_angle, end_angle;
```

DESCRIPTION

v_pieslice draws a filled pie slice with its center at (x, y). The beginning and ending angles are defined in tenths of degrees in the parameters `start_angle` and `end_angle`. The radius is set by the parameter `radius` and is defined in pixels. This function uses the current fill area attributes when filling the pie slice.

EXAMPLE

```
/*
    draw_pieslice  -  This is an example of how to use the vdi
        function v_pieslice().
*/
draw_pieslice(handle)
    int handle;
{
    int x           = 320;
    int y           = 130;
    int radius      = 30;
    int start_angle = 0;
    int end_angle   = 1200;

    v_pieslice(handle, x, y, radius, start_angle, end_angle);
}
```


SEE ALSO

vswr_mode, *vsf_interior*, *vsf_style*, *vsf_color*, *vsf_perimeter*

NAME

v_pline — polyline; connects n vertices

SYNOPSIS

```
int v_pline(handle, count, points)
    int handle;
    int count;
    int points[][2];
```

DESCRIPTION

v_pline draws a complex polygon defined in the parameter **points**.

The array **points** contains a series of points which define the lines in the polygon. The parameter **count** contains the number of points in the polygon array. The lines drawn begin at the point **points[0]** and connect the points in the array until **points[count]**. All points are represented in pixels. The current line attributes are used to draw the polygon.

NOTE

The line must have at least two coordinate pairs, though they may be coincident.

EXAMPLE

```
do_polyline(handle)
{
    int cx      = 400;
    int cy      = 70;
    int count   = 5;
    int points[5][2];

    /*
        Create a diamond.
    */
    pt_set(points[0], cx     , cy - 50);
    pt_set(points[1], cx + 50, cy);
    pt_set(points[2], cx     , cy + 50);
    pt_set(points[3], cx - 50, cy);
    pt_set(points[4], cx     , cy - 50);

    /*
        Now draw the diamond.
    */
    v_pline(handle, count, points);
}
```

SEE ALSO

vsl_type, *vsl_width*, *vsl_color*, *vsl_ends*, *vswr_mode*

NAME

v_pmarker — polymarker; draws count number of markers in pxyarray

SYNOPSIS

```
int v_pmarker(handle, count, points)
    int handle;
    int count;
    int points[][2];
```

DESCRIPTION

v_pmarker draws a hollow complex polygon defined in the parameter points. At each of the points the function will draw a mark to highlight the point using the current marker attributes.

The array points contains a series of points which define the lines in the polygon. The parameter count contains the number of points in the polygon array. The lines drawn begin at the point points[0] and connect the points in the array until points[count]. All points are represented in pixels. The current polymarker attributes are used when drawing the polygon.

EXAMPLE


```
/*
   do_polymarker  -  This function draws a diamond with a marker at
      each of the points of the diamond.
*/
do_polymarker(handle)
{
   int cx      = 400;
   int cy      = 130;
   int count   = 5;
   int points[5][2];

   /*
      Create a diamond.
   */
   pt_set(points[0], cx     , cy - 50);
   pt_set(points[1], cx + 50, cy);
   pt_set(points[2], cx     , cy + 50);
   pt_set(points[3], cx - 50, cy);
   pt_set(points[4], cx     , cy - 50);

   /*
      Now draw the diamond.
   */
   vsm_type(handle, 3);        /* Asterisks */
   vsm_height(handle, 30);     /* Make that BIG asterisks */
```

```
        v_pmarker(handle, count, points);
    }
```

SEE ALSO

vsm_type, *vsm_height*, *vsm_color*, *vswr_mode*

NAME

vq_chcells — Inquire Addressable Character Cells

SYNOPSIS

```
int vq_chcells(handle, rows, columns)
    int  handle;
    int *rows;
    int *columns;
```

DESCRIPTION

vq_chcells returns the maximum number of rows and columns that are used by the text mode screen. The number of rows is stored at the location pointed to by `rows`. The number of columns is stored at the location pointed to by `columns`. If such addressing is not possible, the returned values will be −1.

EXAMPLE

```
show_dimensions(handle)
    int handle;
{
    int rows;
    int cols;

    vq_chcells(handle, &rows, &cols);

    printf("The screen has %d addressable rows.\n", rows);
    printf("The screen has %d addressable columns.\n", cols);
}
```


SEE ALSO

v_enter_cur, *v_cursor movement*

NAME

vq_color — Inquire Color Representation

SYNOPSIS

```
int vq_color(handle, color, set_flag, rgb)
    int handle;
    int color;
    int set_flag;
    int rgb[3];
```

DESCRIPTION

vq_color returns the red, green and blue settings for the color index specified by `color` (see page 349 for the default values). The values are returned in the `rgb` array as integer values from 0 to 1000.

The ST only allows 8 levels per color which are mapped into the 0 – 1000 range. These numbers are called the *actual* levels and are returned when `set_flag` is 1. The values used to set the color index on the last *vs_color* call are returned when `set_flag` is 0.

EXAMPLE

```
#define SET  1
#define ACTUAL   1

#define RED   0
#define GREEN 1
#define BLUE  2

fade_to_black(handle)
int handle;
{
    int rgb[3];
    int color;

    /*
        For each color
    */
    for (color=0; color<16; color++) {
        vq_color(handle, color, ACTUAL, rgb);

        /*
            Fade each color gun value
        */
        while(rgb[RED] | rgb[GREEN] | rgb[BLUE]) {
            if (rgb[RED])   rgb[RED]--;
            if (rgb[GREEN]) rgb[GREEN]--;
```

17

```
                if (rgb[BLUE])  rgb[BLUE]--;

                vs_color(handle, color, rgb);
            }
        }
    }
```

NOTE

If the index color is out of range a random value will be returned.

SEE ALSO

vs_color, *v_opnwk*

17

NAME

vq_curaddress — returns the current position of the text cursor.

SYNOPSIS

```
int vq_curaddress(handle, row, column)
    int handle;
    int *row, *column;
```

DESCRIPTION

vq_curaddress returns the current row and column position of the text cursor. The row position is stored at the location pointed to by the parameter `row`. The column position is stored at the location pointed to by `column`.

EXAMPLE

```
show_cursor_position(handle)
    int handle;
{
    int row, col;

    vq_curaddress(handle, &row, &col);

    printf("The cursor is at (%d, %d).\n", col, row);
}
```

SEE ALSO

v_enter_cur, *v_cursor movement*

NAME

vq_extnd — Extended Inquire Function

SYNOPSIS

```
int vq_extnd(handle, owflag, work_out)
    int handle;
    int owflag;
    int work_out[57];
```

DESCRIPTION

This function allows access to information not returned in the open workstation call, *v_opnwk*. If `owflag` is 1 the extended inquire values are returned, if it is a 0, the open workstation values are returned.

`work_out[0]` Screen Type:

- 1 = Separate alpha, graphic controllers; separate video screens.
- 2 = Separate alpha, graphic controllers; common video screen.
- 3 = Common alpha, graphic controller; separate image memory.
- 4 = Common alpha, graphic controller; common image memory.

17

`[1]` Number of available background colors. This may not equal the number returned from *v_opnwk*.

`[2]` Number of graphic special effects. See *vst_effects*.

`[3]` If 1 then scaling possible, if 0 then not possible.

`[4]` Number of planes.

`[5]` If 0 then look-up table supported, if 1 it is not supported.

`[6]` Number of 16 by 16 pixel raster operations per second.

`[7]` Contour fill capability.

`[8]` Character rotation:

- 0 = None
- 1 = in 90 degree increments (only)
- 2 = arbitrary angles

`[9]` Number of writing modes available.

`[10]` Input modes available:

- 0 = none
- 1 = request only
- 2 = sample and request

[11]	0 – No text alignment; 1 – available
[12]	0 – device cannot ink; 1 – device can
[13]	Rubberbanding: 0 = none 1 = rubberband lines 2 = rubberband lines and rectangles
[14]	Maximum number of vertices for polylines, polymarkers, or filled areas; or −1 if there is no limit.
[15]	Maximum `intin` size, −1 if no maximum
[16]	Number of available mouse keys
[17]	0 – no styles for wide lines; 1 – there are
[18]	Writing modes for wide lines
[19-56]	Reserved, all 0's

SEE ALSO

v_opnwk

NAME

vqf_attributes — Inquire Fill Area Attributes.

SYNOPSIS

```
int vqf_attributes(handle, attrib)
    int handle;
    int attrib[5];
```

DESCRIPTION

vqf_attributes returns the current fill area attributes. The current settings of the fill area attributes are returned in `attrib`:

`attrib[0]`	fill interior style
`[1]`	fill area color index
`[2]`	fill area style index
`[3]`	writing style
`[4]`	fill perimeter status

SEE ALSO

vswr_mode, *vsf_interior*, *vsf_style*, *vsf_color*, *vsf_perimeter*

NAME

vq_key_s — Sample Keyboard State Information

SYNOPSIS

```
int vq_key_s(handle, status)
    int handle;
    int *status;
```

DESCRIPTION

The *vq_key_s* function returns the status of the keyboard modifier keys. The parameter **status** is a pointer to a two-byte area of memory where the status of the keyboard modifiers will be stored.

The low byte of the status word contains the status of the four keys with a 1 indicating the key that has been depressed. The bits representations are as follows:

Bit	Key
0	Left Shift key
1	Right Shift key
2	Control key
3	Alternate key

17

EXAMPLE

```
#define LSHIFT  0x001
#define RSHIFT  0x002
#define CTRL    0x004
#define ALT     0x008

check_key_status(handle)
    int handle;
{
    int status;

    vq_key_s(handle, &status);

    if (status & RSHIFT)
        puts("The Left Shift key is down.   ");

    if (status & LSHIFT)
        puts("The Right Shift key is down.  ");

    if (status & CTRL)
        puts("The Control key is down.      ");
```

```
    if (status & ALT)
        puts("The Alternate key is down.    ");
}
```

NAME

vql_attributes — Inquire Polyline Attributes

SYNOPSIS

```
int vql_attributes(handle, attrib)
    int handle;
    int attrib[6];
```

DESCRIPTION

vql_attributes returns the current line drawing attributes. These attributes are returned in the array `attrib` as follows:

`attrib[0]`	Polyline type
`[1]`	Polyline color index
`[2]`	Current writing mode
`[3]`	Start point style
`[4]`	End Point style
`[5]`	Line width

The start and end point styles may be one of the following:

0	squared
1	arrow
2	rounded

SEE ALSO

vsl_type, *vsl_width*, *vsl_color*, *vsl_ends*, *vswr_mode*

17

NAME

vqm_attributes — Inquire Polymarker Attributes

SYNOPSIS

```
int vqm_attributes(handle, attrib)
    int handle;
    int attrib[5];
```

DESCRIPTION

vqm_attributes returns the current attributes for the line marker. The attributes are returned in the array **attrib** as follows:

`attrib[0]`	type
`[1]`	color index
`[2]`	writing mode
`[3]`	width
`[4]`	height

SEE ALSO

vsm_type, *vsm_height*, *vsm_color*, *vswr_mode*

NAME

vq_mouse — Sample Mouse Button State

SYNOPSIS

```
int vq_mouse(handle, status, x, y)
    int  handle;
    int *status;
    int *x, *y;
```

DESCRIPTION

vq_mouse returns the current state of the mouse as well as it's current position. The parameter `status` points to a location where the status of the mouse buttons can be stored. The bits in the status word represent the state of the mouse buttons. The LSB, least significant bit, represents the leftmost button, and a 1 indicates the button has been depressed. The mouse's current x position will be stored at the location pointed to by the parameter x. The mouse's current y position will be stored at the location pointed to by the parameter y.

EXAMPLE

```
#define LBUTTON 0x1
#define RBUTTON 0x2

int check_mouse(handle)
    int handle;
{
    int status, x, y;

    vq_mouse(handle, &status, &x, &y);

    printf("The mouse is at (%d, %d) and \n", x, y);

    if (status & LBUTTON)
        printf("the left button is down.\n");

    if (status & RBUTTON)
        printf("the right button is down.\n");

    if (!status)
        printf("no buttons are down.\n");

    return status;
}
```

17

NAME

vqt_attributes — Inquire Graphic Text Attributes

SYNOPSIS

```
int vqt_attributes(handle, attrib)
    int handle;
    int attrib[10];
```

DESCRIPTION

vqt_attributes returns the current text attributes. The attributes are returned in the array `attrib` as follows:

`attrib[0]`	graphic text face
`[1]`	graphic text color
`[2]`	angle of text baseline rotation
`[3]`	horizontal alignment
`[4]`	vertical alignment
`[5]`	writing mode
`[6]`	character width
`[7]`	character height
`[8]`	character cell width
`[9]`	character cell height

17

SEE ALSO

vst_height, *vst_font*, *vst_color*, *vst_alignment*, *vswr_mode*, *vst_rotation*

NAME

vqt_extent — Inquire Text Extent

SYNOPSIS

```
int vqt_extent(handle, string, extent)
    int    handle;
    int    extent[8];
    char *string;
```

DESCRIPTION

vqt_extent returns an array of points which defines a rectangle that surrounds the text specified in the parameter `string`. The box corners returned in `extent` are in a coordinate system where the lower left corner (point number 1) is on the X-axis, and the last point is on the Y-axis. The points are enumerated as follows:

Figure 17.1: Text Box Extent

They are returned in extent in the following order: $[x_1, y_1, x_2, y_2, x_3, y_3, x_4, y_4]$.

If the text baseline were to be rotated, the coordinate system would still have the Y-axis vertical and the X-axis horizontal, but the box would be rotated and the values of the points would reflect this.

The size of the box is affected by all current text attributes.

SEE ALSO

vst_height, *vst_rotation*, *vst_font*, *vst_color*, *vst_effects*, *vst_alignment*

17

NAME

vqt_fontinfo — Inquire Current Face Information

SYNOPSIS

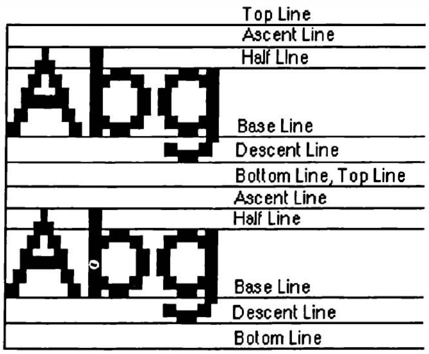
```
int vqt_fontinfo(handle, firstchar, lastchar, distances,
                  maxwidth, effects)
    int handle;
    int *firstchar, *lastchar;
    int distances[5];
    int *maxwidth;
    int effects[3];
```

DESCRIPTION

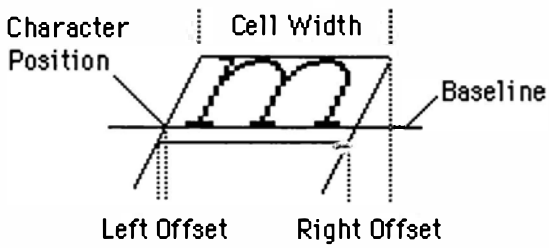
vqt_fontinfo obtains sizing information on the current face at the current height, taking account of the current special effects.

`firstchar` `lastchar`	The ASCII equivalent of the first and last characters in the current font.
`distances[0]`	the bottom line distance, relative to the baseline.
`[1]`	the descent line distance, relative to the baseline.
`[2]`	half height distance, relative to the baseline.
`[3]`	ascent distance relative to the baseline.
`[4]`	top distance relative to the baseline.

17



`maxwidth`	the maximum cell width, not including the left and right offsets.



`effects[0]`	the increase in character width due to special effects such as italics.
`[1]`	left offset.
`[2]`	right offset.

SEE ALSO

vqt_width, *vst_font*, *vst_effects*

17

NAME

vqt_name — Inquire Face Name and Index

SYNOPSIS

```
int vqt_name(handle, face_num, font_name)
    int   handle;
    int   face_num;
    char *font_name;
```

DESCRIPTION

vqt_name will derive the name and font ID from a type face number. The parameter `face_num` is the face number of the font whose name is required. The number of faces available may be obtained through *v_extnd* with `owflag` set to 0 by looking at output parameter `work_out[10]`. The last parameter `font_name` is a pointer to an array of 32 characters where the name of the font will be stored. The first sixteen characters are the font name, and the next sixteen are the style. The function result is the ID number of the current typeface.

EXAMPLE

17

```
show_font_info(handle, face_num)
    int handle;
    int face_num;
{
    struct {
        char font_name[16];
        char font_style[16];
        int  font_id;
    } fontinfo;

    fontinfo.font_id = vqt_name(handle, face_num, &fontinfo);

    printf("Font name  == <%s>\n", fontinfo.font_name);
    printf("Font style == <%s>\n", fontinfo.font_style);
    printf("Font ID    == <%d>\n", fontinfo.font_id);
}
```

SEE ALSO

vst_font, *v_extnd*

NAME

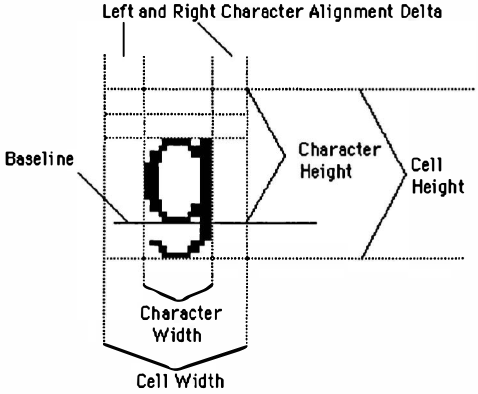
vqt_width — Inquire Character Cell Width

SYNOPSIS

```
int vqt_width(handle, character, cell_width, left_delta,
              right_delta)
    int  handle;
    char character;
    int *cell_width;
    int *left_delta, *right_delta;
```

DESCRIPTION

vqt_width returns the width of a graphics character. The character to be measured is passed in the parameter `character`. The total width of the graphical representation of the `characater` is stored at a location pointed to by the parameter `cell_width`. The unused space on the left and right side of the character will be stored at the locations pointed to by the parameter's `left_delta` and `right_delta`, respectively. The special effects and rotation of the character do not affect the size.



EXAMPLE

```
show_char_info(handle, thechar)
    int  handle;
    char thechar;
{
    int cell_width, left, right;

    vqt_width(handle, thechar, &cell_width, &left, &right);
```

```
        printf("The character [%c] is %d pixels wide.\n",
                          thechar,  cell_width - (left + right));
    }
```

DIAGNOSTICS

The function will return a -1 if the character cannot be measured.

SEE ALSO

vst_height, *vst_font*

17

NAME

v_rbox — Rounded Rectangle

SYNOPSIS

```
int v_rbox(handle, rect)
    int handle;
    int rect[4];
```

DESCRIPTION

v_rbox draws a hollow rectangle with rounded edges. `rect` is an array of integers which define the corners of the box to be drawn. The first two elements define the lower left corner and the last two elements define the upper right corner of the box. This function uses the current line attributes.

EXAMPLE

```
/*
    draw_bars  -  An example of how to use the v_rbox() function to
        draw hollow rounded edge boxes.
*/
draw_bars(handle)
    int handle;
{
    int rect[4];
    int px = 200, py = 100;
    int x       , y  = 90;

    for (x=0; x < 100; x += 25, px += 25, y -= 10) {
        rect_set(rect, px, py, px+20, py-y);
        v_rbox(handle, rect);
    }
}
```

17

SEE ALSO

vswr_mode, *vsl_type*, *vsl_width*, *vsl_color*, *vsl_ends*

NAME

v_rfbox — Rounded and Filled Rectangle

SYNOPSIS

```
int v_rfbox(handle, rect)
    int handle;
    int rect[4];
```

DESCRIPTION

v_rfbox draws a filled rectangle with rounded edges. `rect` is an array of integers which define the corners of the box to be drawn. The first two elements define the lower left corner and the last two elements define the upper right corner of the box. This function uses the current fill area attributes.

EXAMPLE

```
/*
    draw_filled_bars  -  An example of how to use the v_rfbox()
        function to draw filled rounded edge boxes.
*/
draw_filled_bars(handle)
    int handle;
{
    int rect[4];
    int px = 200, py = 100;
    int x      , y  = 90;

    for (x=0; x < 100; x += 25, px += 25, y -= 10) {
        rect_set(rect, px, py, px+20, py-y);
        v_rfbox(handle, rect);
    }
}
```

SEE ALSO

vswr_mode, *vsf_interior*, *vsf_style*, *vsf_color*, *vsf_perimeter*

NAME

vro_cpyfm — Copy Raster, Opaque

SYNOPSIS

```
#include <gemdefs.h>

int vro_cpyfm(handle, write_mode, pxyarray, source_MFDB,
              destin_MFDB)
    int   handle;
    int   write_mode;
    int   pxyarray[8];
    MFDB *source_MFDB:
    MFDB *destin_MFDB;
```

DESCRIPTION

vro_cpyfm copies a source rectangle (defined by `source_MFDB`) to a destination rectangle (defined by `destin_MFDB`) using the logical transfer operation passed in `write_mode` (see the VDI introduction, pg. 307, for the writing modes available for raster functions).

The two rectangles are each specified by two diagonally opposite corners. The corners (xs_1, ys_1), (xs_2, ys_2) [source corners] and (xd_1, yd_1), (xd_2, yd_2) [destination] are passed in pxyaray in the order: $[xs_1, ys_1, xs_2, ys_2, xd_1, yd_1, xd_2, yd_2]$.

17

If the "from" and "to" rectangles overlap, a copy is made of the original information before any modification of the overlap area is performed.

No rotation results from this function. The data will be scaled if the rectangles are of different sizes and `work_out[3]` as obtained from *vq_extend* is a 1.

The source and destination must be in device-specific form (see *vr_trnfm*).

The pointers `source_MFDB` and `destin_MFDB` point to the source Memory Form Definition Block and destination Memory Form Definition Block, respectively. Refer to the VDI Introduction, pg. 307 for more information.

EXAMPLE

```
#include <gemdefs.h>
#include <osbind.h>
#include <obdefs.h>

/*
   Declare VDI globals
*/
int contrl[12];
```

```
int intin[256],  ptsin[256];
int intout[256], ptsout[256];

main()
{
    int     handle;
    int     pxyarray[8];
    MFDB    source, destin;

    /*
        Initialize AES & VDI.
    */
    appl_init();
    handle = open_workstation(&source);
    destin = source;

    v_gtext(handle, 80, 60, "Moving the Menu Bar...");

    /*
        Set the source and destination rectangles.
    */
    rect_set(&pxyarray[0], 0, 0, source.fd_w, 40);
    rect_set(&pxyarray[4], 0, 100, source.fd_w, 140);

    /*
        Do the Copy.
    */
    vro_cpyfm(handle, S_ONLY, pxyarray, &source,  &destin);

    v_gtext(handle, 80, 80, "Press RETURN to end.");
    Bconin(2);

    appl_exit();
}
```


SEE ALSO

VDI introduction (pg. 307), *vq_extnd*, *vr_trnfm*

NAME

vr_recfl — Fill Rectangle

SYNOPSIS

```
int vr_recfl(handle, rect)
    int handle;
    int rect[4];
```

DESCRIPTION

vr_recfl draws a filled rectangle without a perimeter. The rectangle is defined by the parameter **rect**. The first two elements define the lower left corner and the last two elements define the upper right corner of the box. This function uses the current fill area attributes, but since it does not draw a perimeter it does not use the fill perimeter setting.

EXAMPLE

```
/*
    draw_recfl  -  An example of how to use the vr_recfl() function to
        draw filled rectangles.
*/
draw_recfl(handle)
    int handle;
{
    int rect[4];
    int px = 300, py = 100;
    int x      , y  = 90;

    for (x=0; x < 100; x += 25, px += 25, y -= 10) {
        rect_set(rect, px, py, px+20, py-y);
        vr_recfl(handle, rect);
    }
}
```

SEE ALSO

vsf_interior, *vsf_style*, *vsf_color*, *vswr_mode*

17

NAME

vrt_cpyfm — Copy Raster, Transparent

SYNOPSIS

```
#include <gemdefs.h>

int vrt_cpyfm(handle, write_mode, pxyarray, source_MFDB,
              destin_MFDB, color)
    int   handle;
    int   write_mode;
    int   pxyarray[8];
    MFDB *source_MFDB;
    MFDB *destin_MFDB;
    int   color[2];
```

DESCRIPTION

vrt_cpyfm copies a source rectangle (defined by `source_MFDB`) to a destination rectangle (defined by `destin_MFDB`). The function is similar to *vro_cpyfm*, except that it copies a single color raster to a color raster.

The two rectangles are each specified by two diagonally opposite corners. The corners (xs_1, ys_1), (xs_2, ys_2) [source corners] and (xd_1, yd_1), (xd_2, yd_2) [destination] are passed in pxyaray in the order: $[xs_1, ys_1, xs_2, ys_2, xd_1, yd_1, xd_2, yd_2]$.

If the "from" and "to" rectangles overlap, a copy is made of the original information before any modification of the overlap area is performed.

No rotation results from this function. The data will be scaled if the rectangles are of different sizes and `work_out[3]` as obtained from *vq_extend* is a 1.

The source and destination must be in device-specific form (see *vr_trnfm*).

The pointers `source_MFDB` and `destin_MFDB` point to the source Memory Form Definition Block and destination Memory Form Definition Block, respectively. Refer to the VDI introduction, pg. 307, for more information.

The writing mode in `write_mode` can be:

Replace Mode (1) — All source pixels are transfered; source pixels with a 1 will have `color[0]` in the destination, all those with a 0 will have `color[1]`.

Transparent Mode (2) — Only the source pixels with a value of 1 will write over the destination pixels. `color[0]` contains the color index to use for writing.

XOR (3) — The source monochrome raster area is logically XOR'd (Exclusive OR'd) with each destination plane. The values in `color` are ignored.

Reverse Transparent Mode (4) — This is the reverse of mode 2, only the destination pixels with associated source pixels of 0 are affected. Those with a value of 0 are mapped to `color[1]`.

SEE ALSO

vro_cpyfm, *vswr_mode*, *vq_extnd*, VDI Introduction (pg. 307)

NAME

vr_trnfm — Transform Form

SYNOPSIS

```
#include <gemdefs.h>

int vr_trnfm(handle, source_MFDB, destin_MFDB)
    int   handle;
    MFDB *source_MFDB;
    MFDB *destin_MFDB;
```

DESCRIPTION

vr_trnfm transforms the raster image from device-specific raster coordinates to standard normalized coordinates and vice-versa.

The number of planes transformed is determined by the source MFDB, the address of which is passed in **source_MFDB**. The format flag (**fd_stand**) from the source is toggled and placed in the destination MFDB, who's address is passed in **destin_MFDB**.

The user must ensure all the other parameters in the destination MFDB are correct.

SEE ALSO

17

VDI Introduction (pg. 307)

NAME

v_rvoff, v_rvon — Video switches.

SYNOPSIS

```
int v_rvoff(handle)
    int handle;

int v_rvon(handle)
    int handle;
```

DESCRIPTION

These functions set flags which determine where the alpha text will be displayed in normal or reverse video. The parameter `handle` is a handle to the device's virtual workstation.

v_rvon turns reverse video on for text

v_rvoff turns reverse video off for text

SEE ALSO

v_enter_cur, v_cursor movement

NAME

vsc_form — Set Mouse Form; Change the cursor pattern

SYNOPSIS

```
int vsc_form(handle, pcur_form)
    int handle;
    int pcur_form[37];
```

DESCRIPTION

vsc_form allows the user to set his own form for the mouse cursor. The array `pcur_form` contains the cursor definition information in the following positions:

`pcur_form[0,1]` the x and y locations, relative to the upper left corner of the cursor, of the "center" of the cursor. The location of the pixel in the center is defined as the location of the cursor.

`[2]` For future use, must be 1.

`[3]` This is the color index the 1's in the cursor background will have.

`[4]` This the color index the 1's in the cursor foreground will have.

17

`[5-20]` The (16 × 16) array of background bits. The MSB (most significant bit) of the first word (index 5) is the upper left corner, and the LSB of the last word (index 20) is the lower right.

`[21-36]` The foreground data, organized as above.

EXAMPLE

```
/*
    set_mouse - redefine the current mouse cursor using vsc_form().

    Note: stuffbits() is a routine that converts ascii 0's & 1's to
          real binary and stores the result at the location pointed to
          by the first parameter.  It is defined in grafstuf.c
*/
set_mouse(handle)
    int handle;
{
    unsigned form[37];
    int      x;
```

```
    /*
        Define the mouses ''Hot Spot''
    */
    pt_set(&form[0], 5, 2);

    /*
        Setup array information
    */
    form[2] = 1;    /*  reserved by Atari   */
    form[3] = 2;    /*  Background color    */
    form[4] = 3;    /*  Foreground color    */

    /*
        Define Background mouse form
    */
    stuffbits(&form[5], "0000000000000000");
    stuffbits(&form[6], "0000011111100000");
    stuffbits(&form[7], "0000001111100000");
    stuffbits(&form[8], "0000000001100000");
    stuffbits(&form[9], "0000000000000000");

    for (x=10; x<21; x++)
        stuffbits(&form[x], "0000000000000000");

    /*
        Define Foreground mouse form
    */
    stuffbits(&form[21], "0011111111110000");
    stuffbits(&form[22], "0011100000010000");
    stuffbits(&form[23], "0011110000010000");
    stuffbits(&form[24], "0011111110010000");
    stuffbits(&form[25], "0011111111110000");

    for (x=26; x<37; x++)
        stuffbits(&form[x], "0000000000000000");

    vsc_form(handle, form);
}
```

SEE ALSO

v_show_c, *graf_mouse*

NAME

vs_clip — set clipping rectangle; set or reset clipping of all primitives

SYNOPSIS

```
int vs_clip(handle, clip_flag, rect)
    int handle;
    int clip_flag;
    int rect[4];
```

DESCRIPTION

vs_clip defines a rectangular area within which all future drawing will be restricted until clipping is disabled or redefined. Clipping is enabled if `clip_flag` is 1 and disabled if it is 0. Open workstation (*v_opnwk*) initially disables clipping.

The rectangle is defined by 2 diagonally opposed (x, y) coordinates in `rect` in the following order: $[x_1, y_1, x_2, y_2]$.

NAME

vs_color — Set Color Representation; define colors

SYNOPSIS

```
int vs_color(handle, color, rgb)
    int handle;
    int color;
    int rgb[3];
```

DESCRIPTION

vs_color sets the intensity values of the color electron guns for the color index specified. The intensities of the three colors have a range of 0 – 1000. Any intensity above 1000 is mapped to 1000, and any less than 0 is mapped to 0. If the device is monochrome, each of the colors are mapped to a percentage of white. The first parameter **color** is an index into the color table defined by the *v_opnvwk* call. The second parameter is the **rgb** defines the values of each color gun where **rgb[0]** is red, **rgb_in[1]** is green, and **rgb_in[2]** is blue.

NOTE

No action takes place if the device does not have a color look-up table, or the color index (**index**) is out of range. See page 349 for the default color assignments.

SEE ALSO

v_opnwk, *vq_extnd*, *vq_color*

NAME

vs_curaddress — Direct Alpha Cursor Address

SYNOPSIS

```
int vs_curaddress(handle, row, column)
    int handle;
    int row;
    int column;
```

DESCRIPTION

vs_curaddress moves the text cursor to (`row,column`). The cursor will not move beyond the maximum displayable range of the screen if over range coordinates are passed, instead it will move to the maximum value of each.

EXAMPLE

```
gotoxy(handle, x, y)
    int handle, x, y;
{
    vs_curaddress(handle, y, x);
}
```

SEE ALSO

vq_curaddress, *v_enter_cur*, v_cursor movement

NAME

vsf_color — Set Fill Color Index

SYNOPSIS

```
int vsf_color(handle, color)
    int handle;
    int color;
```

DESCRIPTION

vsf_color will set the color index used for future polygon fill operations. If the color requested in `color` is out of range, color 1 will be selected. The colors 0, and 1 are always present, others may be available (see *vq_extnd*).

SEE ALSO

v_opnwk, *vq_extnd*, *vs_color*, *vsf_interior*, *vsf_style*

NAME

vsf_interior — Set Fill Interior Style

SYNOPSIS

```
int vsf_interior(handle, style)
    int handle;
    int style;
```

DESCRIPTION

vsf_interior will set the fill style used in future interior fill operations. The function result is the value of the style selected. The value selected will be 0, or hollow if the requested style does not exist. See *vsf_style* for a complete list of the pattern and hatch styles.

Available fill styles

0	=	Hollow – fills with background color (index 0)
1	=	Solid – fills with current fill color
2	=	Pattern
3	=	Hatch
4	=	User-defined style

SEE ALSO

vsf_color, *vs_color*, *vsf_udpat*, *vsf_style*

17

NAME

vsf_perimeter — Set Fill Perimeter Visibility

SYNOPSIS

```
int vsf_perimeter(handle, visible)
    int handle;
    int visible;
```

DESCRIPTION

vsf_perimeter sets a flag which determines whether or not the perimeter of a polygon when drawn should be visible. If `visible` is 0, the perimeter of a filled area is not visible. If `visible` is any value other than 0 the perimeter is visible. If the perimeter is set visible, it is drawn as a solid line in the current fill color. The default perimeter set by *v_opnwk* is visible.

NAME

vsf_style — Set Fill Style Index

SYNOPSIS

```
int vsf_style(handle, style_index)
    int handle;
    int style_index;
```

DESCRIPTION

vsf_style sets a fill style which is based on the fill interior style, set with *vsf_interior*. This fill style has effect only if the fill interior style is set to Pattern or Hatch. The desired index is passed in `style_index`, and the one chosen will be the return value of the function. If the requested style does not exist or is invalid then the function will default to style 1.

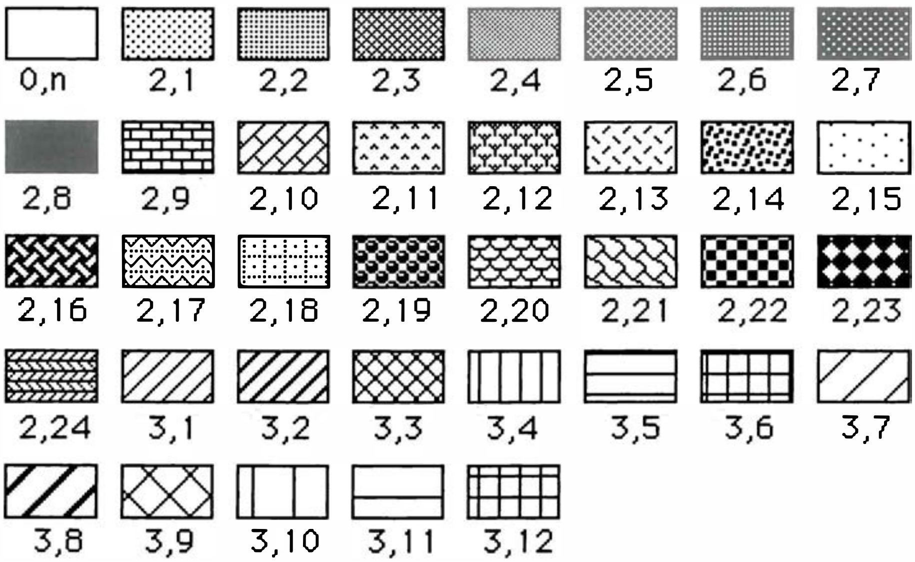


Figure 17.2: Available Fill Styles

The available fill style indices start at 1 and continue to a device-dependent maximum. The chart above shows the resulting patterns for some combinations of the fill interior style and the fill style, set by this function. In the paired numbers on the chart, the left number is the fill interior style, and the right is the fill style (set with this function).

The two colors displayed will be the fill color set by *vsf_color* and the darkest color on the device (index 1).

SEE ALSO

vsf_interior, *vsf_color*

NAME

vsf_udpat — Set User-defined Fill Pattern

SYNOPSIS

```
int vsf_udpat(handle, fill_pattern, planes)
    int handle;
    int fill_pattern[];
    int planes;
```

DESCRIPTION

vsf_udpat allows the user to define a customized fill pattern and make it the user-defined fill pattern, (see *vsf_interior*. The pattern is a 16 × 16 bit array. The defined pattern may consist of more than one plane. The number of planes is defined by the parameter `planes`. For single plane patterns, a 1 maps to the present foreground color, and 0 to the background. The foreground color is set by *vsf_color*. The bit pattern is stored in `fill_patttern` as follows:

1010101010101010	word 0, `fill_pattern`$[16 \times i]$ for plane i
0101010101010101	word 1, `fill_pattern`$[16 \times i + 1]$ for plane i
⋮	
1010101010101010	word 15, `fill_pattern`$[16 \times i + 15]$ for plane i

17

NOTE

The interior fill style must be set to 4 (user defined) using *vsf_interior*.

EXAMPLE

```
set_fill_pattern(handle)
    int handle;
{
    unsigned x;
    unsigned fill_pattern[16];

    /*
        Create checker board fill pattern
    */
    for (x=0; x<16; x+= 2)
        stuffbits(&fill_pattern[x], "0101010101010101");

    for (x=1; x<16; x += 2)
        stuffbits(&fill_pattern[x], "1010101010101010");

    vsf_udpat(handle, fill_pattern, 1);
}
```

SEE ALSO

vsf_interior, *vsf_style*, *vsf_color*

NAME

v_show_c — Show Cursor

SYNOPSIS

```
int v_show_c(handle, reset)
    int handle;
    int reset;
```

DESCRIPTION

v_show_c will cause the cursor to be displayed based upon its level of visibility. **reset** indicates if the level of visibility should be reset. Calling this funntion with **reset** = 0 will make the cursor appear and reset the visibility level to 0. If **reset** is non-zero, the cursor will be displayed based upon its level of visibility.

The cursor visibility may be "nested" to any depth. Every call to *v_hide_c* must be balanced with a call to *v_show_c*. If the cursor needs to be shown at any time, a call to *v_show_c* with the **reset** parameter set to zero will cause the cursor to become visible despite its "nesting" level. This also causes the nesting level to be set to zero.

SEE ALSO

v_hide_c

NAME

vsl_color — Set Polyline Color

SYNOPSIS

```
int vsl_color(handle, color)
    int handle
    int color;
```

DESCRIPTION

vsl_color sets the color index with which polylines are drawn. An out of range `color` results in a color of 1 being set. The color index actually used will be returned as the value of the function.

SEE ALSO

v_opnwk, *vq_extnd*, *vql_attributes*, *vsl_ends*

NAME

vsl_ends — Set Polyline End Styles

SYNOPSIS

```
int vsl_ends(handle, start_style, end_style)
    int handle;
    int start_style, end_style;
```

DESCRIPTION

vsl_ends defines how the ends of a line appear. The first parameter **start_style** defines the style of the beginning of the line. The second parameter **end_style** defines the style of the end of the line. The types of styles available are:

0	squared
1	arrow
2	rounded

The rounded style extends past the end point of the line by one-half the line width, or the radius of the half-circle. The others end at the end point.

NOTE

If an out of bounds style is asked for, then style 0, the default, is used.

SEE ALSO

vql_attributes

NAME

vsl_type — Set Polyline Type

SYNOPSIS

```
int vsl_type(handle, style)
    int handle;
    int style;
```

DESCRIPTION

vsl_type sets the line style to `style`. Although the total number of styles available is device dependent, at least six will always be available. If the style requested is out of range, style 1 will be used.

In the chart below, a bit value of 1 represents a pixel on and 0 off. The MSB (most significant bit) is displayed first. The user defined style (7) defaults to all on, it is set with *vsl_udsty*.

1 solid	1111111111111111
2 long dash	1111111111110000
3 dot	1110000011100000
4 dash-dot	1111111000111000
5 dash	1111111100000000
6 dash-dot-dot	1111000110011000
7 User-defined	16 bits defined by *vsl_udsty*
$8 - n$	Device dependent

If a non-default line width is used, the device may use the solid pattern, and may change the writing mode.

SEE ALSO

vsl_udsty, *v_opnwk*, *vq_extnd*, *vql_attributes*

NAME

vsl_udsty — Set User-defined Line Style Pattern; define your polyline

SYNOPSIS

```
int vsl_udsty(handle, pattern)
    int handle;
    int pattern;
```

DESCRIPTION

The argument **pattern** contains a sixteen bit pattern which is used to define which pixels of the user-defined line style (style 7) are on. The default for style 7 is a solid line, or all 1's. The pattern is displayed MSB (most significant bit) first, with 0's indicating off and 1's on.

EXAMPLE

```
set_line_pattern(handle)
    int handle;
{
    int pattern;

    /*
        Define the line pattern.  (16 bits wide)
    */
    stuffbits(&pattern, "0101010101010101");

    /*
        Set the pattern to the user defined line drawing pattern.
    */
    vsl_udsty(handle, pattern);
}
```

17

SEE ALSO

vsl_type

NAME

vsl_width — Set Polyline Line Width

SYNOPSIS

```
int vsl_width(handle, width)
    int handle;
    int width;
```

DESCRIPTION

vsl_width defines the width that all lines will be drawn at. If the width of the line requested does not exist then the next smaller available line width will be used. The function's return value will be the set width of a line. The number of line widths available may be obtained through *v_extnd* with `owflag` set to 0 by looking at output parameter `work_out[7]`.

NOTE

Wide lines may be rendered with a solid pattern.

SEE ALSO

v_opnwk, *vq_extnd*

NAME

vsm_color — Set Polymarker Color

SYNOPSIS

```
int vsm_color(handle, color)
    int handle;
    int color;
```

DESCRIPTION

vsm_color sets the output color index for polymarkers, and returns it as the value of the function. The colors 0 and 1 will always be present. If the value requested is out of range, the color selected will default to 1 (black).

SEE ALSO

v_opnwk, *vq_extnd*

NAME

vsm_height — Set Polymarker Height

SYNOPSIS

```
int vsm_height(handle, height)
    int handle;
    int height;
```

DESCRIPTION

vsm_height changes the height of a marker. The value in **height** sets the height of the polymarker's output in y-axis units. If the height of the marker requested does not exist then the next smaller available marker height will be used. The function's return value will be the actual height the marker was set at. The number of line widths available may be obtained through *v_extnd* with **owflag** set to 0 by looking at output parameter **work_out[7]**.

SEE ALSO

vq_extnd, *vqm_attributes*

NAME

vsm_type — Set Poly Marker Type

SYNOPSIS

```
int vsm_type(handle, symbol)
    int handle;
    int symbol;
```

DESCRIPTION

vsm_type sets the polymarker type. Although the number of types is device dependent, a minimum of six will always be available.

1	· – dot
2	+ – plus
3	* – asterisk
4	□ – square
5	× – diagonal cross
6	◇ – diamond
> 6	device dependent

The function returns the value of the marker used. If the value requested is out of range the function will default to marker type 3.

17

NOTE

The smallest displayable dot is type 1, and it cannot be scaled.

SEE ALSO

v_opnwk, *vq_extnd*, *vqm_attributes*, *vsm_height*, *vsm_color*

NAME

vsm_valuator — input Valuator, Sample Mode

SYNOPSIS

```
int vsm_valuator(handle, val_in, val_out, term, status)
    int  handle;
    int  val_in;
    int *val_out;
    int *term;
    int *status;
```

DESCRIPTION

This function returns the new valuator value in `val_out` if a key was pressed. The result of the function is nothing happened. The parameter `status` contains a 0 if nothing happened, a 1 if the valuator has changed, or a 2 if a key was pressed. The parameter `term` contains the keypress if one occurred.

NOTE

As this function is not required, it may not be available on all devices.

SEE ALSO

vrq_valuator

NAME

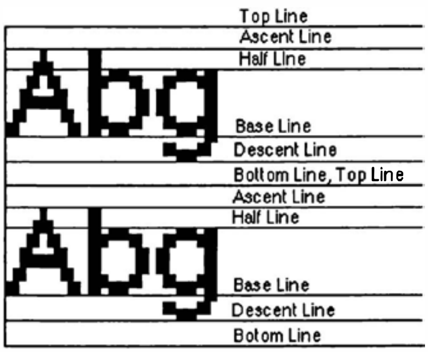
vst_alignment — Set Graphic Text Alignment

SYNOPSIS

```
int vst_alignment(handle, horiz, vertical, new_horiz, new_vertical)
    int  handle;
    int  horiz;
    int  vertical;
    int *new_horiz;
    int *new_vertical;
```

DESCRIPTION

vst_alignment sets the horizontal and vertical alignment of graphic text. The parameters **horiz** and **vertical** are the requested values that the horizontal and vertical alignment be set at. The actual values that are set are stored at the locations pointed to by **new_horiz** and **new_vertical**.



There are 6 valid values for the vertical alignment:

0	=	Baseline, the default
1	=	Halfline
2	=	Ascent line
3	=	Bottom
4	=	Descent
5	=	Top

There are three valid values for horizontal alignment:

0	=	Left justified, the default
1	=	Center justified
2	=	Right justified

NOTE

The default alignment of the text vertically is on the base line. The default alignment of the text horizontally is against the left edge.

SEE ALSO

v_gtext

NAME

vst_color — Set Graphic Text Color

SYNOPSIS

```
int vst_color(handle, color)
    int handle;
    int color;
```

DESCRIPTION

vst_color sets the color index for all future graphic text output. The parameter `color` is an index into the color table defined by the *v_opnvwk* function. If the value passed in `color` is out of range then the color index is set to 1 (black). All devices support at least 2 colors 0 (white) and 1 (black). The result of the function is the color that was actually set.

SEE ALSO

v_opnwk, *vq_extnd*, *vs_color*, *v_gtext*

NAME

vst_effects — Set Graphic Text Special Effects

SYNOPSIS

```
int vst_effects(handle, effects)
    int handle;
    int effects;
```

DESCRIPTION

vst_effects controls the setting of special effects for graphic text. The special text effects are controlled by the bits set in the argument `effects`. The bits 0 – 5 have represent the following effects:

0	Thickened (bold)
1	Light Intensity
2	Skewed (italicized)
3	Underlined
4	Outline
5	Shadow

If the bit is set then that particular type style is in effect. Any combination of these styles may be used. Example:

0000 0000 0000 1001 indicates bold underlined text.

17

The result of the function is an integer which contains the bits that are actually set. If an effect is not supported that bit will be set to 0. If the function is not available then the result will be 0.

EXAMPLE

```
#define BOLD        0x001
#define PLAIN       0x002
#define ITALICS     0x004
#define UNDERLINE   0x008
#define OUTLINE     0x010
#define SHADOW      0x020

laserproc(grafhandle)
    int grafhandle;
{
    int  dummy, cw, ch;
    char *text = "Laser C";

    /*
```

```
        Set text size 25 pts &
        set slant mode       &
        write mode = transparent.
    */
    vst_height(grafhandle, 25, &dummy, &dummy, &cw, &ch);
    vst_effects(grafhandle, OUTLINE | ITALICS | UNDERLINE);
    vswr_mode(grafhandle, 2);

    v_gtext(grafhandle, 40, 80, text);
}
```

SEE ALSO

v_gtext

NAME

vst_font — Set Text Face

SYNOPSIS

```
int vst_font(handle, font)
    int handle;
    int font;
```

DESCRIPTION

vst_font changes the type face for all future graphics text output. The new font is defined in the parameter `font`. The result of the function is the font number of the type face that was set. There are several pre-defined font numbers as follows:

1	=	System face
2	=	Swiss 721
3	=	Swiss 721 Thin
4	=	Swiss 721 Thin Italic
5	=	Swiss 721 Light
6	=	Swiss 721 Light Italic
7	=	Swiss 721 Italic
8	=	Swiss 721 Bold
9	=	Swiss 721 Bold Italic
10	=	Swiss 721 Heavy
11	=	Swiss 721 Heavy Italic
12	=	Swiss 721 Black
13	=	Swiss 721 Black Italic
14	=	Dutch 801 Roman
15	=	Dutch 801 Italic
16	=	Dutch 801 Bold
17	=	Dutch 801 Bold Italic

Only type face number 1 is built-in. Any others, if available, will need to be loaded by *vst_load_fonts*.

NOTE

The number of type faces available may be obtained through *v_extnd* with `owflag` set to 0 by looking at output parameter `work_out[10]`.

SEE ALSO

vq_extnd, *vqt_name*, *vst_load_fonts*

17

NAME

vst_height — Set Character Height, Absolute Mode

SYNOPSIS

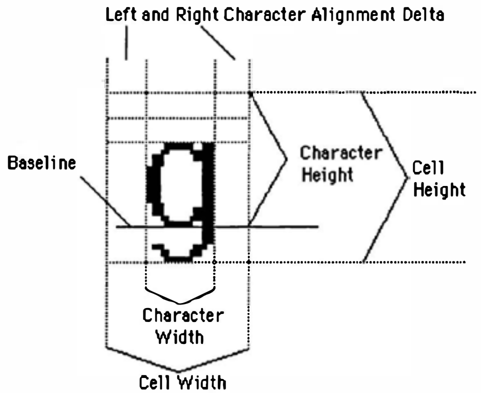
```
int vst_height(handle, height, char_width, char_height,
    cell_width, cell_height)

    int handle;
    int height;
    int *char_width;
    int *char_height;
    int *cell_width;
    int *cell_height;
```

DESCRIPTION

vst_height sets the character height (not the cell height) in units of pixels. The requested height is defined in the parameter **height**. If the height of the character cell is not available then the next smaller height will be used. The character and cell widths and heights are stored at the locations pointed to by their associated variables. If the face (or font) has proportional spacing, the width returned is that of the widest character and cell.

17



The function's return value will be the set height of the character cell.

SEE ALSO

vst_point, *v_gtext*, *graf_handle*

NAME

vst_load_fonts — Load Fonts

SYNOPSIS

```
int vst_load_fonts(handle, select)
    int handle;
    int select;
```

DESCRIPTION

vst_load_fonts loads the fonts for a driver into RAM. The number of fonts loaded is returned as the function result. Zero is returned if the fonts for the driver are already in RAM. The `select` parameter is reserved for future use and should be set to 0.

This function need not be called if the default fonts for a driver are sufficient.

NOTE

This function should only be used with the GDOS driver installed. Any use of this function outside of that environment will have disasterous results.

SEE ALSO

vst_unload_fonts

NAME

vst_point — Set Cell Height, Points Mode

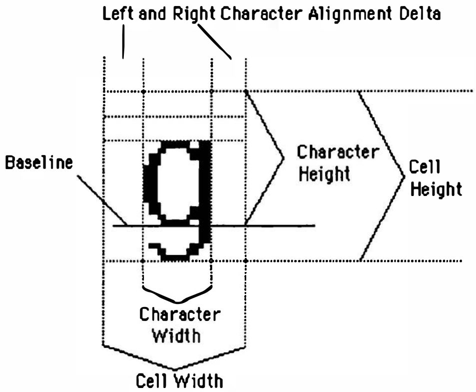
SYNOPSIS

```
int vst_point(handle, points, char_width, char_height,
    cell_width, cell_height)

    int handle;
    int points;
    int *char_width, *char_height;
    int *cell_width, *cell_height;
```

DESCRIPTION

vst_point sets the character height based upon a system of points. After this call characters drawn will not be based upon graphic pixels, but upon printer points where (one **point** = **1/72 inch**). The height is the distance from the base line of one text line to base line of the next and is defined by the parameter **points**. If the height of the character cell is not available then the next smaller height will be used. The character and cell widths and heights are stored at the locations pointed to by their associated variables. If the face (or font) has proportional spacing, the width returned is that of the widest character and cell.



The function's return value will be the set height of the character cell.

SEE ALSO

vst_height, *v_gtext*, *graf_handle*

NAME

vst_rotation — Set Character Baseline Vector

SYNOPSIS

```
int vst_rotation(handle, angle)
    int handle;
    int angle;
```

DESCRIPTION

vst_rotation sets the angle at which all future text will be drawn. This sets the character baseline vector for the best fit of the angle requested in **angle**. The angle is specified in tenths of degrees (0 – 3600), with 0 being the positive x-axis. The angle actually set will be returned as the function result.

NOTE

This function may not be available for every workstation device.

SEE ALSO

vq_extnd

NAME

vst_unload_fonts — Unload Fonts

SYNOPSIS

```
int vst_unload_fonts(handle, select)
    int handle;
    int select;
```

DESCRIPTION

The *vst_unload_fonts* function unloads (disassociates from the driver and possibly removes from RAM) the fonts for a driver loaded by the *vst_load_fonts* function. The fonts will only be removed from RAM if all workstations which have loaded them have been closed or have called *vst_unload_fonts*.

Only fonts loaded with the *vst_load_fonts* function are unloaded; the default fonts will continue to be available.

The `select` parameter is reserved for future use and should be set to 0.

NOTE

This function should only be used with the GDOS driver installed. Any use of this function outside of that environment will have disasterous results.

SEE ALSO

vst_load_fonts

NAME

vswr_mode — Set Writing Mode; define how output affects existing information on the display.

SYNOPSIS

```
int vswr_mode(handle, mode)
    int handle;
    int mode;
```

DESCRIPTION

vswr_mode changes the way graphics is written to the display. The requested mode is defined in the parameter `mode`. If the mode number is out of range then this function will default to mode 1. The writing mode is used when drawing lines, markers, filled areas and text. The four modes available are listed below. The source pixels come from the line style, fill pattern, or black parts of graphic text.

The writing mode in `write_mode` can be:

Replace Mode (1) — All source pixels are transfered; source pixels with a 1 will write the foreground color in the destination, all those with a 0 will write color index 1.

Transparent Mode (2) — Only the source pixels with a value of 1 will write over the destination pixels. The foreground color is used.

XOR (3) — The foreground color is XOR'd (exclusive OR'd) with a destination pixel when the source pixel value is 1. Otherwise the background color is XOR'd with the destination pixel.

Reverse Transparent Mode (4) — This is the reverse of mode 2, only the destination pixels with associated source pixels of 0 are written. Color index 1 is used.

The mode actually set will be returned as the function result.

NAME

v_updwk — update workstation; execute all pending workstation commands, start printers

SYNOPSIS

```
int v_updwk(handle)
    int handle;
```

DESCRIPTION

v_updwk causes the workstation defined by `handle` to be updated, and any pending graphics commands to be executed in the order of their occurrence in the command queue. If the workstation is a printer or plotter, this will cause the device driver to begin output to the device. If a picture is drawn to a printing device, no form feed will be issued. If the device is the screen, there is no effect. If the workstation is defined as a metafile, GEM VDI outputs the opcode.

SEE ALSO

v_clrwk

Chapter 18

BIOS, GEMDOS, XBIOS Routines

Introduction

Digital Research Corp.'s GEMDOS operating system is the programmer's interface to the Atari ST hardware. GEMDOS was designed to be portable, in that it's hardware dependent functions are isolated in a section called the BIOS (Basic Input Output System). A computer manufacturer may port GEMDOS by providing the BIOS routines for his particular hardware. Additional hardware functionality not required by GEMDOS is included in the XBIOS (eXtended BIOS). GEMDOS, BIOS, and XBIOS routines are called through the Motorola 68000's `TRAP` instruction. The header file "**OSBIND.H**" contains C preprocessor macros for the various calls, and must be included by a GEM application.

18.1 BIOS Interface

The BIOS interface routines provide the basis for higher level GEMDOS input/output functionality. Basic input/output includes:

Screen Output

Keyboard Input

Printer Output

RS-232 Input/Output

Disk Input/Output

"**OSBIND.H**" contains macros which convert the name of the function to a call to the function *bios* with an appropriate opcode. The opcode is then passed to the ROM via a 68000 instruction `TRAP #13`. All BIOS functions are accessed through this trap.

18.2 XBIOS Interface

The XBIOS interfaces special hardware features of the Atari ST, including:

68901 MFP (Multi-Function Peripheral) Timer Chip

YM-2149 Sound Generator Chip

6850 ACIA (Asynchronous Communications Interface Adapter)

MIDI Port Input/Output

"**OSBIND.H**" contains macros which convert the name of the function to a call to the function *xbios* with an appropriate opcode. The opcode is then passed to the ROM via a 68000 instruction `TRAP #14`. All XBIOS functions are accessed through this trap.

18

18.3 GEMDOS Interface

GEMDOS routines include high level file input/output, disk directory management, and memory allocation. The "**OSBIND.H**" header file contains macros which convert each GEMDOS function call into a call to the function *gemdos* with an appropriate opcode. The *gemdos* function then calls the ROM via a 68000 instruction `TRAP #1`.

18.4 GEM Run-time Structure

When executed, a GEM application is loaded into the section of RAM known as the TPA (Transient Program Area). The base page, a data structure containing run-time information, marks the base of the TPA. The TPA is contiguous and extends from the base page to the top of usable RAM. In the TPA are the program's code, globals, stack, and heap. The heap is the memory pool from which memory is dynamically allocated. The format of the base page is:

Offset	Name	Description
0x00	`p_lowtpa`	Base address of TPA
0x04	`p_hitpa`	Address of byte just past end of TPA
0x08	`p_tbase`	Address of text segment (code of program)
0x0C	`p_tlen`	Length of text segment
0x10	`p_dbase`	Address of data segment (strings)
0x14	`p_dlen`	Length of data segment
0x18	`p_bbase`	Address of BSS segment (globals)
0x1C	`p_blen`	Length of BSS segment
0x2C	`p_env`	Address of environment string
0x80	`p_cmdlin`	Address of command line image

The extern variable `_base` points to the base page of the currently executing program. Figure 18.1 shows how the memory in the TPA is partitioned.

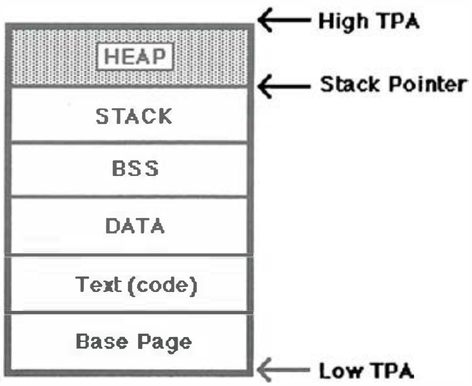


Figure 18.1: Transient Program Area

When a program is loaded into memory it is stored in the heap of its parent program (the program from which it is executed). The parent waits until its child program terminates before continuing. As a program terminates, its memory is returned to the heap from which it was allocated. Program termination is extremely fast since the parent program remains in memory.

A program's stack is initially 8K bytes. This size may be changed in Laser C by declaring the initialized global variable `_stksize`. For example:

```
/*  Make this program's stack 4K bytes */
long _stksize = 4096L;
```

Note that the error codes for the GEMDOS functions are described in the DOS Error Codes, pg. 587.

NAME

Cauxin, Cauxout, Cauxis, Cauxos — Auxiliary port read/write/status

SYNOPSIS

```
#include <osbind.h>

int Cauxin()

Cauxout(chr)
    int chr;

int Cauxis()

int Cauxos()
```

DESCRIPTION

These function handle I/O through the serial ports. These routines are defined as macros in `<osbind.h>`

Cauxin returns the next character from the RS232 port.

Cauxout writes `chr` to the RS232 port.

Cauxis returns non-zero if a character is available at the RS232 port.

Cauxos returns non-zero if the RS232 port is ready to send a character.

EXAMPLE

```
#include <osbind.h>

#define ESC 27

main()
{
    char c;

    while (c != ESC) {
        /*
            Display characters that come across the serial port.
        */
        if (Cauxis())
            Cconout((int)Cauxin()&127);
```

```
            /*
                Check for keyboard data
            */
            if (Cconis()) {
                /*
                    Get keyboard data
                */
                c = Cconin();

                /*
                    Wait for OK to send char to RS-232
                */
                while(!Cauxos())
                    ;

                /*
                    Send character to serial port.
                */
                Cauxout(c);
            }
        }
    }
```

18

NAME

Bconstat, Bconin, Bconout, Bcostat — Character input, output, status

SYNOPSIS

```
#include <osbind.h>

int Bconstat(dev)
    int dev;

long Bconin(dev)
    int dev;

Bconout(dev, c)
    int dev, c;

long Bcostat(dev)
    int dev;
```

DESCRIPTION

`dev` is one of the following:

0	=	`PRT:` (parallel printer port)
1	=	`AUX:` (auxiliary RS-232 port)
2	=	`CON:` (console/keyboard)
3	=	MIDI port
4	=	Keyboard port (KBD)

18

Legal operations for each device:

Operation	`PRT:`	`AUX:`	`CON:`	MIDI	KBD
Bconstat	no	yes	yes	yes	no
Bconin	yes	yes	yes	yes	no
Bconout	yes	yes	yes	yes	yes
Bcostat	yes	yes	yes	yes	yes

Bconstat checks the status of a specified `device` and determines if any data is available for input. *Bconstat* returns −1 if characters are available for input, 0 if no characters are available.

Bconin waits until a character is available on the device specified by `dev`. The result of the function is a 32-bit long which contains the character typed and a keycode. The character is returned in the low word of the long. If bit 3 of the system global `conterm` is set, then the high word will contain the value of the system variable `kbshift` at the time of the keystroke.

Bconout writes the character `c` to the device specified in `dev`. *Bconout* will wait until the character has been written before returning.

Bcostat checks the status of a specified `device` and determines if the device is available for output. It returns −1 if the device is ready for output, and 0 if it is not.

These functions are defined as macros in `<osbind.h>`.

EXAMPLE

```
#include <osbind.h>

#define ESC     27
#define AUX     1
#define CONSOLE 2

/*
    The dumb terminal Handler using Bcon's
*/
main()
{
    char c;

    while (c != ESC) {
        /*
            Display characters that come across the serial port.
        */
        if (Bconstat(AUX))
            Bconout(CONSOLE, (int)Bconin(AUX)&127);

        /*
            Check for keyboard data
        */
        if (Bconstat(CONSOLE)) {
            /*
                Get keyboard data
            */
            c = Bconin(CONSOLE);

            /*
                Wait for OK to send char to RS-232
                Note: Bcostat() not Bconstat().
            */
            while(!Bcostat(AUX))
```

18

```
                    ;

                /*
                    Send character to serial port.
                */
                Bconout(AUX, c);
            }
        }
    }
```

SEE ALSO

Printer I/O, Console I/O

NAME

Cconin, Cconout, Cconws, Cconrs, Cconis, Cconos, Crawio, Crawcin, Cnecin — Console input/output/status.

SYNOPSIS

```
#include <osbind.h>

int Cconin()

Cconout(chr)
int chr;

Cconws(str)
char *str;

Cconrs(buf)
char *buf;

int Cconis()

int Cconos()

int Crawio(wrd)
int wrd;

int Crawcin()

int Cnecin()
```


DESCRIPTION

Cconin returns and echoes (to the console) the next character from the console.

Cconout writes `chr` onto the console.

Cconws writes the null terminated string `str` to the console.

Cconrs reads an edited string from the console. `buf[0]` is the size of the buffer beginning with `buf[2]`. `buf[1]` contains the number of characters read on exit with the characters starting at `buf[2]`. The returned string is also null terminated.

Cconis returns non-zero if a character is available at the console.

Cconos returns non-zero if the console is ready to receive a character.

Crawio writes `wrd` to the console if `wrd` isn't `0xFF`. If it is then a character is read from the console and returned.

Crawcin returns the next character from the console without echoing. All control characters are returned.

Cnecin returns the next character from the console without echoing, but the control characters: `^S` (stop output), `^Q` (continue output), `^C` (terminate program) are trapped and acted upon.

These routines are defined as macros in `<osbind.h>`

SEE ALSO

Printer I/O, Character I/O

NAME

Cursconf — Configure the VT52 emulator cursor

SYNOPSIS

```
#include <osbind.h>

int Cursconf(function, operand)
int function, operand;
```

DESCRIPTION

The VT52 emulator cursor is configured according the the value in `function`:

`function`	**Operation performed**
0	Hide cursor
1	Show cursor
2	Set blinking cursor
3	Set non-blinking cursor
4	Set blink time according to value in operand.
5	Return cursor blink time.

The cursor blink rate is based on the vertical blanking interrupt (which occurs at a 70hz rate on the B/W monitor, a 60hz rate on the color monitor and a 50hz rate for PAL). The time for the cursor to turn off and back on again is two times `operand` divided by the screen frequency.

Cursconf is defined as a macro in `<osbind.h>`

NAME

Dcreate — Create a subdirectory

SYNOPSIS

```
#include <osbind.h>

int Dcreate(path)
    char *path;
```

DESCRIPTION

Dcreate creates a new subdirectory on a disk with the path name specified by the parameter path.

Dcreate is defined as a macro in `<osbind.h>`

DIAGNOSTICS

A non-zero error code is returned if an error occurred.

EXAMPLE

```
#include <osbind.h>

main()
{
    mkdir("C:\MEGAMAX\");
}

mkdir(path)
    char *path;
{
    if (Dcreate(path)) {
        printf("Error in creating path <%s>\n", path);
    } else
        printf("Success in creating path <%s>\n", path);
}
```

18

SEE ALSO

Ddelete, DOS error codes (pg. 587)

NAME

Ddelete — Delete a subdirectory

SYNOPSIS

```
#include <osbind.h>

int Ddelete(path)
    char *path;
```

DESCRIPTION

Ddelete deletes the directory specified by the parameter **path**.

Ddelete is defined as a macro in **<osbind.h>**

DIAGNOSTICS

A non-zero error code is returned if an error occurs.

EXAMPLE

```
#include <osbind.h>

main()
{
    rmdir("A:\JUNKDIR\");
}

rmdir(path)
    char *path;
{
    if (Ddelete(path))
        rmdir("Error in deleting path <%s>\n", path);
    else
        rmdir("Success in deleting path <%s>\n", path);
}
```

SEE ALSO

Dcreate, DOS Error Codes (pg. 587)

NAME

Dfree — Get information about disk allocation

SYNOPSIS

```
#include <osbind.h>

Dfree(buf, drv)
    disk_info *buf;
    int        drv;
```

DESCRIPTION

The *Dfree* function returns allocation information about drive `drv` where a 0 means the default drive, 1 means drive `A:`, 2 means drive `B:`, etc. The parameter buf points to the following structure which is filled in by the call:

```
    typedef struct _disk_info {
        long b_free;    /* no. of free clusters on drive */
        long b_total;   /* total no. of clusters on drive */
        long b_secsiz;  /* no. of bytes in a sector */
        long b_clsiz;   /* no. of sectors in a cluster */
    } disk_info;
```

Dfree is defined as a macro in `<osbind.h>`

18

EXAMPLE

```
#include <osbind.h>

show_disk_info(drive)
    int drive;
{
    disk_info myinfo;

    Dfree(&myinfo, drive);

    if (!drive)
        printf("The default disk has:\n");
    else
        printf("The disk %c:\ has:\n", 'A' + (drive-1));

    printf("%ld free clusters\n", myinfo.b_free);
    printf("%ld total clusters\n", myinfo.b_total);
    printf("%ld bytes per sector\n", myinfo.b_secsiz);
    printf("%ld sectors per cluster\n", myinfo.b_clsiz);
```

```
    printf("%ld free bytes in disk\n",
        myinfo.b_free * myinfo.b_clsiz * myinfo.b_secsiz);
}
```

NAME

Dosound — Set sound process "program counter"

SYNOPSIS

```
#include <osbind.h>

Dosound(ptr)
    char *ptr;
```

DESCRIPTION

The *Dosound* function starts the sound generator. The sound process "program counter" is set to **ptr**. The parameter **ptr** points at a series of "instructions" with the following meanings:

0x00 – 0x0F	Put the next byte into a sound register. 0x00 puts the byte in register 0, 0x01 in register 1 etc.
0x80	Put the next byte into the temporary register.
0x81	`for (register no. next byte = temp. reg.t; temp. reg. != next+2 byte; temp reg += next+1 byte) wait until next update; /* next+1 byte is signed */`
0x82 – 0xFF	Set update time. If next byte is zero then the sound is terminated. Otherwise the update rate is set to the next byte divided by 50 hertz.

18

Dosound is defined as a macro in <osbind.h>

NOTE

The sound chip registers are defined in detail in *giaccess*.

EXAMPLE

```
#include <osbind.h>

/*
    Sound definition
*/
unsigned char crash[] = {
      0x06, 0x1f,   /* Noise Period              */
      0x07, 0x2f,   /* Mixer                     */
      0x09, 0x10,   /* Channel B volume          */
      0x0c, 0x20,   /* Duration Course tune      */
      0x0d, 0x00,   /* Envelope Shape            */
```

```
        0x81, 0x12, 0x02, 0xf8,  /* Sustain time for tone  */
        0xff, 0x00    /* End Tone                    */
};

do_crash()
{
    Dosound(crash);
}
```

SEE ALSO

Giaccess

18

NAME

Drvmap — Return bit vector of on-line drives

SYNOPSIS

```
#include <osbind.h>

long Drvmap()
```

DESCRIPTION

Drvmap returns a bit map of available drives. Each bit in the returned long represents the availability of a drive. A value of 1 means the drive is available, 0 means it isn't (e.g. a 1 in bit 0 means drive 0 is available).

Drvmap is implemented as a macro in `<osbind.h>`.

NOTE

Mountable drives must set the `_drvbits` system global properly.

EXAMPLE

```
#include <osbind.h>

show_drives()
{
    unsigned long drives;
    int           drive;

    drives = Drvmap();

    for (drive= 'A'; drive < 'P'; drive++, drives >>= 1)
        if (drives & 0x001)
            printf("Drive %c:\ is available\n", drive);
}
```


SEE ALSO

Dsetdrv

NAME

Dsetdrv, Dgetdrv — Set/get the default disk drive

SYNOPSIS

```
#include <osbind.h>

long Dsetdrv(drv)
    int drv;

int Dgetdrv()
```

DESCRIPTION

These functions are used in setting and discovering which drive is the default disk drive. The default disk drive is the drive that is initially searched when looking for a file.

Dsetdrv sets the default drive (the drive to use if a drive is not specified in a path) to `drv` where a value of 0 means drive A, 1 means B, etc. The return value of the function is a long containing a bit map of the available drives, where bit 0 is 1 if drive A is on-line, bit 1 is 1 if drive B is on-line, etc.

Dgetdrv returns the number of the current default drive (see above description).

These routines are defined as macros in `<osbind.h>`

NAME

Dsetpath, Dgetpath — Set/get current working directory

SYNOPSIS

```
#include <osbind.h>

int Dsetpath(path)
    char *path;

int Dgetpath(pathbuf, drive)
    char *pathbuf;
    int drive;
```

DESCRIPTION

These functions are used in setting and discovering the default path name. The default path name is prepended to file names which contain no path (directory) specification.

Dsetpath sets the default directory to **path**.

Dgetpath stores the name of the default directory for drive **drive** in the character array pointed to by **pathbuf**. A drive value of 0 means the default drive, a value 1 means drive `A:`, 2 means `B:` etc.

pathbuf must point to a buffer space of at least 64 bytes.

18

These functions are defined as macros in `<osbind.h>`

DIAGNOSTICS

A negative error code is returned if an error occurs.

SEE ALSO

DOS Error Codes, (pg. 587)

NAME

Fattrib — Get/set file attributes

SYNOPSIS

```
#include <osbind.h>

int Fattrib(path, mode, attr)
    char *path;
    int mode, attr;
```

DESCRIPTION

Fattrib gets and sets information about a file's attributes. The parameter `path` is a path name to the file whose attributes are to be investigated. The parameter `mode` is used to determine if the function attributes are to be returned or set. If the value of `mode` is 0, the attributes of the file will be returned. If the value of `mode` is 1, then the attributes from the `attr` parameter will be used to set the file's attributes. The file attribute bits and meanings are:

Bit	Meaning
0	Read only
1	Hidden from directory search
2	System file (implies hidden from directory search)
3	File is Volume label
4	File is really a subdirectory
5	The file has been written to and closed

18

Fattrib is defined as a macro in `<osbind.h>`

EXAMPLE

```
#include <osbind.h>

#define READ 0

#define READONLY    0x01
#define HIDDEN      0x02
#define SYSTEM      0x04
#define VOLUME      0x08
#define DIRECTORY   0x10
#define WRITCLOSED  0x20

show_attributes(pathname)
    char *pathname;
{
```

```
    int mode = READ;
    int attr;

    attr = Fattrib(pathname, mode, attr);

    printf("<%s> is ", pathname);
    if (attr & READONLY)  printf("read only, ");
    if (attr & HIDDEN)    printf("hidden, ");
    if (attr & SYSTEM)    printf("a system file, ");
    if (attr & VOLUME)    printf("a volume label, ");
    if (attr & DIRECTORY) printf("a directory, ");
    printf("and a file.\n");
}
```

SEE ALSO

Fsfirst, *Fsnext*

NAME

Fclose — Close an open file

SYNOPSIS

```
#include <osbind.h>

Fclose(fd)
    int fd;
```

DESCRIPTION

Fclose closes the file specified by the file descriptor `fd`. This will cause any data in the file buffers to be flushed from memory and written to the file before the file is closed.

Fclose is defined as a macro in `<osbind.h>`

DIAGNOSTICS

A negative error number is returned upon failure.

SEE ALSO

Fopen, DOS Error Codes (pg. 587)

NAME

Fcreate — Create a file

SYNOPSIS

```
#include <osbind.h>

int Fcreate(name, attr)
    char *name;
    int   attr;
```

DESCRIPTION

Fcreate creates files on disks. A file is created and opened with the pathname `name`. Bits in `attr` contain extra information about the file for the directory:

Bit	Meaning
0	File is read only
1	File is hidden from directory search commands
2	File is a system file (also hidden from directory search)
3	`name` contains a volume label in first 11 bytes.

A positive file descriptor number is returned upon successful creation.

Fcreate is defined as a macro in `<osbind.h>`

DIAGNOSTICS

A negative error number is returned if an error occurs.

EXAMPLE

An example of *Fcreate* is shown in *Fopen*.

SEE ALSO

Fopen, DOS Error Codes (pg. 587)

NAME

Fdelete — Delete a file

SYNOPSIS

```
#include <osbind.h>

int Fdelete(path)
    char *path;
```

DESCRIPTION

Fdelete deletes files from disks. The parameter `path` is the path name of the file that is to be deleted.

Fdelete is defined as a macro in `<osbind.h>`

DIAGNOSTICS

A negative error number is returned if an error occurs.

EXAMPLE

An example of *Fdelete* is shown in *Fopen*.

SEE ALSO

Fopen, DOS Error Codes (pg. 587)

18

NAME

Fdatime — Get/set file "last modified" time and date stamp

SYNOPSIS

```
#include <osbind.h>

int Fdatime(buf, fd, set)
    long *buf;
    int   fd, set;
```

DESCRIPTION

Fdatime returns the date and time of a file. The parameter `buff` points to a long integer with the time in the low word and the date in the high word. The format is as described in the time functions. The next parameter `fd` is the file descriptor of the file to set or get the time stamp for. If `set` is 1 then the file's time stamp is set with the long at `*buff`, otherwise the time stamp is read into the long at `*buff`.

This routine is defined as a macro in `<osbind.h>`

DIAGNOSTICS

The function result is negative if an error occurs.

EXAMPLE


```
#include <stdio.h>
#include <osbind.h>

show_file_date_and_time(fname)
    char *fname;
{
    int  fd;
    int  datime[2];
    int  err;

    if ((fd = Fopen(fname, 0)) < 0)
        fatal("Error in opening file.");

    if ((err = Fdatime(datime, fd, 0)) < 0)
        fatal("Error reading date and time.");

    Fclose(fd);

    if (err > )
        showtime(datime[0], datime[1]);
}
```

```
/*
    showtime - display the date and time.
*/
showtime(mytime, mydate)
    time mytime;
    date mydate;
{
    printf("\t\t date \n Day: %d \t Month: %d \t Year: %d\n",
            mydate.part.day, mydate.part.month, mydate.part.year + 80
    );

    printf("\t\t time \n Hour: %d \t Minute: %d \t Seconds: %d\n",
            mytime.part.hours, mytime.part.minutes, mytime.part.seconds * 2
    );
}


/*
    fatal - works like printf() except that it waits for
            a <CR> and then dies.
*/
fatal(args)
    char *args;
{
    _fprintf(stderr, &args);

    puts("Press RETURN to exit...");
    getchar();
    exit(1);
}
```


SEE ALSO

Protobt, DOS Error Codes (pg. 587)

NAME

Fdup — Duplicate file handle

SYNOPSIS

```
#include <osbind.h>

int Fdup(stdfd)
    int stdfd;
```

DESCRIPTION

Fdup duplicates the standard file descriptor defined in `stdfd`. The function returns a file descriptor which is a normal file descriptor except that it refers to the same file as the standard file descriptor. The first six, 0 – 5, file descriptors are considered "standard" file descriptors. The rest are considered non-standard. The standard file descriptors are:

0	=	Console input (stdin)
1	=	Console output (stdout)
2	=	Serial interface (AUX:)
3	=	Printer interface (PRT:)
4	=	Not used by GEMDOS.
5	=	Not used by GEMDOS.

Fdup is defined as a macro in `<osbind.h>`

DIAGNOSTICS

A negative error number is returned upon failure.

SEE ALSO

Fforce, DOS Error Codes (pg. 587)

NAME

Fforce — Force standard file descriptor to use same file as a non-standard one

SYNOPSIS

```
#include <osbind.h>

int Fforce(stdfd, nstdfd)
    int stdfd, nstdfd;
```

DESCRIPTION

Fforce forces the standard file descriptor **stdfd** to use the same file or device as the non-standard file descriptor **nstdfd**. This permits standard input and output to be redirected to a file. The first six, 0 – 5, file descriptors are considered "standard" file descriptors, while the rest are considered non-standard. The standard file descriptors are:

0	=	Console input (stdin)
1	=	Console output (stdout)
2	=	Serial interface (AUX:)
3	=	Printer interface (PRT:)
4	=	Not used by GEMDOS.
5	=	Not used by GEMDOS.

A typical non-standard file descriptor is returned by the function *Fopen*.

Fforce is defined as a macro in `<osbind.h>`

DIAGNOSTICS

A negative error number is returned upon failure.

SEE ALSO

Fdup, *Fopen*, DOS Error Codes (pg. 587)

NAME

Fgetdta, Fsetdta — Get/set DTA (disk transfer address)

SYNOPSIS

```
#include <osbind.h>

long Fgetdta()

Fsetdta(ptr)
    char *ptr;
```

DESCRIPTION

These functions get and set the DTA, which is used in getting directory information.

Fgetdta returns a pointer to the current DTA (disk transfer address). The DTA is a 44 byte buffer used when getting directory information.

Fsetdta sets the DTA to `ptr`.

Both routines are defined as macros in `<osbind.h>`

EXAMPLE

An example of this function is in Fsfirst().

SEE ALSO

Fsfirst, *Fsnext*

NAME

Floprd, Flopwr, Flopfmt, Flopver — floppy disk operations

SYNOPSIS

```
#include <osbind.h>

int Floprd(buf, filler, devno, sectno, trackno, sideno, count)
    int *buf;
    long filler;
    int  devno, sectno, trackno, sideno, count;

int Flopwr(buf, filler, devno, sectno, trackno, sideno, count)

int Flopver(buf, filler, devno, sectno, trackno, sideno, count)

int Flopfmt(buf, filler, devno, spt, trackno, sideno, interlv,
            magic, virgin)
    char *buf;
    long  filler;
    int   devno, spt, trackno, sideno, interlv;
    long  magic;
    int   virgin;
```

DESCRIPTION

18

These functions are interface routines for the low-level disk operations. Great care should be taken when using these functions.

`buf` — points to a word aligned array of bytes for reading or writing. It must be large enough to hold `count` sectors for read and write, or an entire track when formatting.

`filler` — an unused long value.

`devno` — the floppy drive number (0 or 1).

`sectno` — the first sector to read or write from/to (usually 1 – 9).

`trackno` — the track to read or write from/to, or the track to format (usually 0 – 79).

`sideno`	the side number (0 or 1).
`count`	the number of sectors to read or write (must be less than or equal to the number of sectors in a track.)

The following variables are used only when formatting:

`spt`	the the number of sectors per track (usually 9).
`interlv`	the sector interleaving factor (the number of physical sectors between two logical sectors.) This number must be relatively prime with `spt`.
`magic`	must be the long value 0x87654321.
`virgin`	an `int` sized value with which to fill the newly created sectors.

Floprd reads `count` sectors into `buf`.

Flopwr writes `count` sectors from `buf`. Writing to side 0, track 0, sector 1 will cause *Mediach* and *Rwabs* to enter the "might have changed" state.

18

Flopfmt returns a 0 terminated `int` sized list of bad sectors in `buf`. `virgin` should be set to 0xE5E5 if another value isn't required. The high four bits cannot be 0xF. Formatting causes *Mediach* and *rwabs* to enter the "definitely changed" state.

Flopver verifies `count` sectors by non-destructive reading. `buf` must point to at least 1024 bytes. Bad sectors are returned in `buf` as in *Flopfmt* above.

These routines are defined as macros in `<osbind.h>`

DIAGNOSTICS

Each routine returns a non-zero error number if an error occurs.

EXAMPLE

```
#include <osbind.h>
#include <stdio.h>
```

```
/*
    Program to format a floppy disk
*/

#define DRIVE       0
#define SECTORS     9
#define BUFSIZ      SECTORS*1024

char buf[BUFSIZ];

main()
{
    setbuf(stdout, NULL);

    printf("Floppy disk format program v1.0\n\n");

    format();

    create_boot_blocks();

    verify();

    printf("\nFormat Complete.");
    wait();
}


format()
{
    /*
        Variables for formating floppy.
    */
    int     interleave        = 1;
    long    filler            = NULL;
    int     devno             = DRIVE;
    int     sectors_pertrack  = SECTORS;
    int     trackno;
    int     sideno            = 0;
    long    magic             = 0x87654321L;
    int     virgin            = 0xe5e5;


    printf("Place disk to be formatted in drive A:.");
    wait();

    puts("Formatting track:");
    for (trackno = 0; trackno < 80; trackno++) {
        printf("[%02d] ", trackno);
```

```
        if (!((trackno+1) % 10))
            printf("\n");

        if (Flopfmt(buf, filler, devno, sectors_pertrack, trackno, sideno,
                    interleave, magic, virgin))
            printf("\nError on track %02d\n", trackno);
    }
}


verify()
{
    /*
        Variables required for disk verify.
    */
    long    filler      = NULL;
    int     devno       = DRIVE;
    int     sectno      = 1;
    int     trackno;
    int     sideno      = 0;

    puts("Verifying track:");
    for (trackno = 0; trackno < 80; trackno++) {
        printf("[%02d] ", trackno);

        if (!((trackno+1) % 10))
            printf("\n");

        if (Flopver(buf, filler, devno, sectno, trackno, sideno, SECTORS))
            printf("\nError on track %02d\n", trackno);
    }
}


create_boot_blocks()
{
    /*
        Variables required for disk write.
    */
    long    filler      = NULL;
    int     devno       = DRIVE;
    int     sectno      = 1;
    int     trackno;
    int     sideno      = 0;
    int     i;

    /*
        Variables for Building Boot Blocks.
    */
    long    serialno         = 0x01000000L;
```

18

```
    int     disktype        = 2;
    int     execflag        = 0;


    printf("\nCreating Boot Blocks.\n\n");

    /*
        Zero out buffer.
    */
    for (i=0; i<BUFSIZ; i++)
        buf[i] = 0;

    /*
        Write out zero'd buffer to track zero
    */
    for(trackno = 0; trackno < 1; trackno++)
        Flopwr(buf, filler, devno, sectno, trackno, sideno, SECTORS);

    /*
        Build prototype boot blocks.
    */
    Protobt(buf, serialno, disktype, execflag);

    /*
        Write boot blocks to disk.
    */
    trackno = 0;
    Flopwr(buf, filler, devno, sectno, trackno, sideno, 1);
}


/*
    A routine to keep the number of printf() & getchar()'s to a minimum.
*/
wait()
{
    printf("\nPress RETURN to continue.\n");
    getchar();
}
```

18

SEE ALSO

DOS Error Codes (pg. 587)

NAME

Fopen — Open a file

SYNOPSIS

```
#include <osbind.h>

int Fopen(name, mode)
    char *name;
    int   mode;
```

DESCRIPTION

Fopen opens the file defined in `name`. The mode that the file will be opened in is defined in `mode`. The parameter `mode` has the following values:

mode	Meaning
0	Read only mode
1	Write only mode
2	Read/write mode

A positive file descriptor number is returned upon successfully opening the file.

Fopen is defined as a macro in `<osbind.h>`

NOTE

18

If the file does not exist *Fopen* will ***not*** create the file.

DIAGNOSTICS

A negative error number is returned on failure.

EXAMPLE

```
#include <stdio.h>
#include <osbind.h>

#define CREATE  '1'
#define RENAME  '2'
#define DELETE  '3'
#define READ    '4'
#define WRITE   '5'
#define QUIT    '6'

extern char *gname();

main()
{
```

```
char fname1[80];
char fname2[80];
int  done = 0;
int  err;
int  fd;

while(!done) {
    switch(gmenu()) {
        case CREATE:
            fd = Fcreate(gname(fname1, "File to create:"), 0);

            if (fd < 0)
                puts("Error occurred during create.");

            printf("File: '%s' created.\n\n", fname1);
            Fclose(fd);
        break;

        case RENAME:
            gname(fname1, "New filename:");
            gname(fname2, "Old filename:");
            err = Frename(0, fname2, fname1);

            if (err < 0)
                puts("Error occurred during rename.");

            printf("File: '%s' renamed to '%s'.\n\n", fname2, fname1);
        break;

        case DELETE:
            err = Fdelete(gname(fname1, "File to Delete:"));

            if (err < 0)
                puts("Error occurred during delete.");

            printf("File: '%s' deleted.\n\n", fname1);
        break;

        case READ:
            readfrom(gname(fname1, "File to read:"));

            puts("\n");
        break;

        case WRITE:
            writeto(gname(fname1, "File to write:"));

            puts("\n");
        break;
```

18

```
            case QUIT:
                done = 1;
            break;

            default:
                puts("\nError: Unknown function.\n");
            break;
        }
    }
}


int gmenu()
{
    int c;

    puts("1) Create a file.");
    puts("2) Rename a file.");
    puts("3) Delete a file.");
    puts("4) Read  text from file.");
    puts("5) Write text to file.");
    puts("");
    puts("6) Quit.");
    puts("");
    printf("Enter number: ");  fflush(stdout);

    return Bconin(2);
}


char *gname(fname, literal)
    char *fname, *literal;
{
    printf("\n%s ", literal);  fflush(stdout);
    scanf("%s", fname);

    return fname;
}


/*
    readfrom - reads the data from the file ''fname'' and displays
        that file's data on the screen.
*/
readfrom(fname)
    char *fname;
{
    int  fd;
    char c;
```

```
    fd = Fopen(fname, 0);

    if (fd < 0)
        printf("Error: Couldn't open file '%s'.\n", fname);
    else {
        while(Fread(fd, 1L, &c))
            printf("%c", c);

        Fclose(fd);
    }
}


/*
    writefrom - writes the data from the character pointer
        ``testtext'' to the file defined by ``fname.''
*/
writeto(fname)
    char *fname;
{
    int   fd;
    int   c;
    int   err;
    char *testtext = "This is a test line1\r\nThis is a test line2\r\n";
    long  l;

    fd = Fopen(fname, 1);

    if (fd < 0)
        printf("Error: Couldn't open write file '%s'.\n", fname);

    else {
        err = Fwrite(fd, (long) strlen(testtext), testtext);

        if (!err)
            printf("Error writing Data..., error = %d\n", err);

        Fclose(fd);
    }
}
```

SEE ALSO

DOS Error Codes, (pg. 587)

18

NAME

Fread, Fwrite — File binary I/O

SYNOPSIS

```
#include <osbind.h>

long Fread(fd, count, buf)
    int   fd;
    long  count;
    char *buf;

long Fwrite(fd, count, buf)
    int   fd;
    long  count;
    char *buf;
```

DESCRIPTION

These functions are used to read and write data to/from disks. The number of bytes actually read or written is returned.

Fread reads `count` bytes from the open file with file descriptor `fd` into the array of bytes pointed to by `buf`.

Fwrite writes `count` bytes to the open file with file descriptor `fd` from the array of bytes pointed to by `buf`.

18

Both functions are defined as macros in `<osbind.h>`

DIAGNOSTICS

A zero is returned if an error occurs.

EXAMPLE

Examples of Fread() and Fwrite() are in Fopen().

SEE ALSO

Fopen, DOS Error Codes (pg. 587)

NAME

Frename — Rename a file

SYNOPSIS

```
#include <osbind.h>

int Frename(zero, old, new)
    int   zero;
    char *old, *new;
```

DESCRIPTION

Frename takes the name of an existing file and renames it. The first parameter is zero whose value must be zero. The `old` file name is a pointer to the name of the file to change. The `new` file name is a pointer to the name to change the `old` file name to. Note that the new file name must not exist. This function can also be used to move a file between subdirectories on the same drive.

Frename is defined as a macro in `<osbind.h>`

DIAGNOSTICS

A negative error number is returned upon failure.

EXAMPLE

An example of Frename() is shown in Fopen().

SEE ALSO

Fopen, DOS Error Codes (pg. 587)

NAME

Fseek — Reposition file pointer

SYNOPSIS

```
#include <osbind.h>

long Fseek(offset, fd, mode)
    long offset;
    int  fd;
    int  mode;
```

DESCRIPTION

The file pointer (the location in the file where the next read or write will occur) is set for file descriptor `fd`. The location set is `offset` bytes from the location defined by `mode` as follows:

mode	Start location for offset
0	From beginning of file
1	From current position
2	From end of file

The location of the file pointer from the beginning of the file is returned upon successful operation.

Fseek is defined as a macro in `<osbind.h>`

DIAGNOSTICS

A negative error number is returned upon failure.

EXAMPLE

```
#include <osbind.h>

/*
    Seek modes
*/
#define BEG 0   /* seek from beginning of file. */
#define CUR 1   /* seek from current file mark. */
#define END 2   /* seek from end of file.       */

long recsize;  /*  the size of a record in the data file.  */

/*
    Read a record from a file.
*/
```

```
DBread(fd, buf, recnum)
    int fd, recnum;
    char *buf;
{
    long recpos;

    /*
        Calculate the record position
    */
    recpos = recnum * recsize;

    /*
        Seek to mark.
    */
    Fseek(recpos, fd, BEG);

    /*
        Read in the record.
    */
    Fread(fd, recsize, buf);
}
```

SEE ALSO

DOS Error Codes (pg. 587)

NAME

Fsfirst, Fsnext — Search a directory

SYNOPSIS

```
#include <osbind.h>

int Fsfirst(path, attr)
    char *path;
    int   attr;

int Fsnext()
```

DESCRIPTION

These functions are used to read a disk's directory information. The *Fsfirst* begins a directory search. Any further directory entries may be obtained through calls to the *Fsnext* function.

`path` A pathname which may contain the wildcard characters '*' and '?' in the file name part (but not in the drive or directory part).

`attr` A word containing the file attribute settings that the search will be limited to. The bit meanings of `attr` are:

Bit	Meaning
0	Readonly
1	Hidden from directory search
2	System file (implies hidden from directory search)
3	File is Volume label
4	File is really a subdirectory
5	The file has been written to and closed

If `attr` is 0, then no volume labels, subdirectories, hidden or system files will be matched. If the hidden, system or subdirectory bits are set then those file types are included in the search along with normal files. If the volume label bit is set then only volume labels will be searched.

18

Information on matched file names is returned in the DTA (disk transfer address) as follows:

Offset	Size	Meaning
00	21 bytes	Reserved for OS
21	1 byte	File attribute
22	2 bytes	File time stamp (int)
24	2 bytes	File date stamp (int)
26	4 bytes	File size (long)
30	14 bytes	File name and extension

Fsnext finds the next file in the search, and places the information about the file in the DTA. The end of the search is indicated by a negative error result from either function.

Both functions are defined as macros in `<osbind.h>`

EXAMPLE

```
#include <osbind.h>
#include <stdio.h>

/*
    DTA : Disk Transfer Address.  A buffer where directory information
          is stored.
*/


main()
{
    ls("*.*");

    printf("Press return..." );
    getchar();
}


/*
    ls - list a disk directory using the path specification 'pathspec'.
*/
ls(pathspec)
    char *pathspec;
{
    long    olddta;
    int     err;
    struct {
        char    reserved[21];
        char    fattr;
        int     ftime;
        int     fdate;
        long    fsize;
        char    fname[14];
    } newdta;
```

```
    olddta = Fgetdta();
    Fsetdta(&newdta);

    printf(" File                Size   Date      Time      Attributes\n");

    err = Fsfirst(pathspec, 0x003f);    /* find all files   */
    while(!err) {
        printf("%-14.14s  ", newdta.fname );
        printf("%8ld  ", newdta.fsize );
        ls_date( newdta.ftime, newdta.fdate );
        printf("  0x%02x\n", newdta.fattr);

        err = Fsnext(); /* find next file   */
    }

    Fsetdta(olddta);
}

/*
    Print time and date in human-readable form
*/
ls_date( date, time )
    int                 date, time;
{
    int                 mth = (date>>5)     & 0xf;
    int                 day = (date)        & 0x1f;
    int                 yr  = ((date>>9)    & 0x7f) + 80;
    int                 hrs = (time>>11)    & 0x1f;
    int                 min = (time>>5)     & 0x3f;
    int                 sec = ((time)       & 0x1f) * 2;

    if ( hrs == 0 )
        hrs = 12;
    printf( " %02d-%02d-%02d", mth, day, yr );
    printf( "  %02d:%02d:%02d", hrs, min, sec );
}
```

18

SEE ALSO

Fattrib, Fgetdta, Gettime

NAME

Getbpb — Get BIOS parameter block

SYNOPSIS

```
#include <osbind.h>

bpb *Getbpb(dev)
    int dev;
```

DESCRIPTION

Getbpb returns a pointer to a BIOS parameter block. The block contains information pertaining to the disk that is contained in the drive specified by the parameter `dev`. The parameter `dev` is the device number where 0 indicates drive A: and 1 indicates drive B:. The result of the function is a pointer to the BIOS parameter block.

Getbpb is implemented as a macro in `<osbind.h>`.

DIAGNOSTICS

If no block exists then a null pointer (0L) is returned.

EXAMPLE

```
#include <osbind.h>
#include <obdefs.h>

show_bios_info(device)
    int device;
{
    bpb *mybpb;

    mybpb = Getbpb(device);

    printf("Sectors size   == %d (bytes)\n",   mybpb -> sector_size_bytes);
    printf("Cluster size   == %d (sectors)\n", mybpb -> cl_sectors);
    printf("Cluster size   == %d (bytes)\n",   mybpb -> cl_bytes);
    printf("Directory size == %d (sectors)\n", mybpb -> dir_length_sectors);
    printf("FAT size       == %d (sectors)\n", mybpb -> FAT_size_sectors);

    printf("Second FAT Sector    == %d\n", mybpb -> FAT_sector);
    printf("Start Data Clusters  == %d\n", mybpb -> data_sector);
    printf("Total Data Clusters  == %d\n", mybpb -> total_data_clusters);
    printf("Miscellaneous Flags  == %x\n", mybpb -> flags);
}
```

SEE ALSO

Rwabs

18

NAME

Getmpb — Get Memory Parameter Block

SYNOPSIS

```
#include <osbind.h>

Getmpb(mpb)
    mpb *mpb;
```

DESCRIPTION

Getmpb stores a copy of the initial memory parameter block at the location pointed to by the parameter mpb. The parameter mpb will contain a information about the internal memory structure of the machine as follows:

```
typedef struct _md {
    struct _md *m_link;   /* next memory block       */
    long        m_start;  /* start address of block  */
    long        m_length; /* No. of bytes in block   */
    long        m_own;    /* Memory block's owner ID */
} md;

typedef struct _mpb {
    md *mp_mfl;       /* memory free list      */
    md *mp_mal;       /* memory allocated list */
    md *mp_rover;     /* roving pointer        */
} mpb;
```

Getmpb is defined as a macro in <osbind.h>.

NOTE

The memory parameter block lists are in protected memory. Since this is the case any accesses to the memory data structure will have to be done in supervisor mode.

EXAMPLE

```
#include <osbind.h>

show_freemem()
{
    mpb   my_mpb;
    md   *free,   *used;
    long  freemem = 0;
```

```
    long  usedmem = 0;

    /*
        Do supervisor mode.
    */
    long        save_ssp = Super(0L);

    /*
        Get the memory parameter block
    */
    Getmpb(&my_mpb);

    /*
        Let's count free memory chunks
    */
    for (free = my_mpb.mp_mfl; free; free = free -> m_link)
        freemem += free -> m_length;

    /*
        How much have we used?
    */
    for (used = my_mpb.mp_mal; used; used = used -> m_link)
        usedmem += used -> m_length;

    /*
        Restore to user mode.
    */
    Super(save_ssp);

    /*
        Print the compiled statistics.
    */
    printf("Free memory: %ld\n", freemem);
    printf("Used memory: %ld\n", usedmem);
}
```

NAME

Gettime, Settime — Get/Set time of day clock

SYNOPSIS

```
#include <osbind.h>

long Gettime()

Settime(thedatetime)
    datetime thedatetime;
```

DESCRIPTION

These functions are designed to manipulate and read the system date and time. The parameter `thedatetime` to *Settime* is defined as follows:

Bits	Meaning
0 – 4	Seconds times 2 (range 0 – 30)
5 – 10	Minutes (range 0 – 60)
11 – 15	Hours (range 0 – 24)
16 – 20	Day in month (range 1 – 31)
21 – 24	Month (range 1 – 12)
25 – 31	Year since 1980 (range 0 – 119)

18

Gettime returns the current date and time in the above format as the function result.

Settime sets the date and time with the value in `datetime`.

Both functions are defined as macros in `<osbind.h>`

EXAMPLE

```
#include <stdio.h>
#include <osbind.h>

/*
    This structure is a bit field that represents the different components of
    the date and time words.  A union structure was used so that a long
    could be used for the assignment from the gettime() function and the
    bit-field structure could be used to easily decode the long word.

    Note:  This data structure was designed to work with Megamax C.  Not all
           compilers allocate bit-fields in the same manner.  rpt
```

```
    Note : To set the time just assign the ''part'' fields of the
        structure and then pass Settime() the  real datetime. Ex:

        mytime.part.day  = 10;
        mytime.part.year = 7;

        Settime(mytime.realtime);
*/
typedef union {
    struct {
        unsigned day     : 5;
        unsigned month   : 4;
        unsigned year    : 7;
        unsigned seconds : 5;
        unsigned minutes : 6;
        unsigned hours   : 5;
    } part;
    long realtime;
} time;


/*
    Example of how to get information  from the Gettime Xbios functions.
*/

main()
{
    time mytime;

    /*
        Get the date and time with the long word of the time data structure.
    */
    mytime.realtime = Gettime();

    /*
        Send it off to be printed.
    */
    showtime(mytime);
    puts( "Press return" );
    getchar();
}


/*
    Print the date and time based on the time data structure.
*/
showtime(mytime)
    time mytime;
{
    /*
```

```
        Print the date.

        Note:  The years are represented from 1980.
    */
    printf("\t\t date \n Day: %d \t Month: %d \t Year: %d\n",
            mytime.part.day,
            mytime.part.month,
            mytime.part.year + 80
    );

    /*
        Print the time.

        Note:  The seconds are represented in multiples of 2.
    */
    printf("\t\t time \n Hour: %d \t Minute: %d \t Seconds: %d\n",
            mytime.part.hours,
            mytime.part.minutes,
            mytime.part.seconds * 2
    );
}
```

NAME

Giaccess, Offgibit, Ongibit — Modify register on the sound chip

SYNOPSIS

```
#include <osbind.h>

char Giaccess(data, regno)
    char data;
    int  regno;

Offgibit(bitno)
    int bitno;

Ongibit(bitno)
    int bitno;
```

DESCRIPTION

These functions are a high level interface used to modify the sound chip sound registers.

Giaccess reads or writes a sound chip register. Logically OR the value 0x80 with `regno` to write data. The *Giaccess* function will return the register value for a read operation.

Offgibit clears bit number `bitno` in the PORT A register.

Ongibit sets bit number `bitno` in the PORT A register.

The sound chip contains 16 8-bit registers (labeled 0 – F). Registers E and F are not used for sound but to control the floppy disk drives. These registers are called ports A and B respectively. *Offgibit* and *Ongibit* modify selected bits of port A. The other registers are modified with *Giaccess*.

The Port A bits are defined as follows:

0	=	Disk side select (for double sided drives)
1	=	Drive A select
2	=	Drive B select
3	=	RS-232 RTS (Request to Send) line
4	=	RS2322 DTR (Data Terminal Ready) line
5	=	Centronics data strobe
6	=	General purpose output on video connector
7	=	Unused

NOTE

The sound chip used in the Atari is Yamaha's YM-2149 programmable sound synthesis chip. This chip was initially designed to be used by arcade games before it found it's home in the Atari ST. Some of the special features of the YM-2149 are:

- Three independent programmable tone generators (called channels A, B and C)
- Programmable noise generator
- Software controlled analog output
- Programmable mixer for tone and noise
- Programmable envelopes (ADSR)
- Two bi-directional 8-bit data ports

All of the sound capabilities of the YM-2149 are controlled through sixteen 8-bit registers. These registers are defined as follows:

Reg. 0
Reg. 1

18

The period for the channel A freqency generator. This is a twelve bit number with the low eight bits in register 0 and the high four bits in the low four bits of register 1.
The period determines the frequecy of the tone generator by use of the following formula:

frequency (Hz) = 62500 / (12-bit register value);

The register values may be calculated accordingly:

register 0 = (62500 / frequency (Hz)) & `0x00ff`;
register 1 = ((62500 / frequency (Hz)) >> 8) & `0x000f`;

Reg. 2

This register is the same as register 0 except that it affects the frequency of the channel B tone generator.

Reg. 3

This register is the same as register 1 except that it affects the pitch of the channel B tone generator.

Reg. 4	This register is the same as register 0 except that it affects the frequency of the channel C tone generator.
Reg. 5	This register is the same as register 1 except that it affects the pitch of the channel C tone generator.
Reg. 6	This register controls the pitch of the noise generator. Only the low order 5 bits are used. Note that smaller values cause the noise to be generated at a higher pitch.
Reg. 7	This register is the mixer for all of the tone generators and the noise generator. The bits for this register are defined as follows:

Bit	Description
0	Channel A tone generator on/off
1	Channel B tone generator on/off
2	Channel C tone generator on/off
3	Channel A noise generator on/off
4	Channel B noise generator on/off
5	Channel C noise generator on/off
6	Port A I/O select input/output
7	Port B I/O select input/output

Note that in the bit settings above a value of zero indicates that the channel is on. Conversely, if the value of the bit is 1 then the channel is off.

Reg. 8	This register controls the amplitude or volume of the Channel A tone generator. The lower 4 bits, bits 0 – 3, contain the volume of the channel. If bit 4 is set then bits 0 – 3 are ignored and the tone's loudness will decay (e.g. go from loud to soft or from soft to loud). The waveform of this decay is determined by registers B, C, and D.
Reg. 9	This register is the same as register 8 except that it controls the volume of the Channel B tone generator.
Reg. A	This register is the same as register 8 except that it controls the volume of the Channel C tone generator.

Reg. B This register contains the low-byte of the sustain counter.
Reg. C This register contains the high-byte of the sustain counter.

Reg. D This register determines the waveform of the envelope generator. The lower 4 bits, bits 0 – 3, are used to select the waveform, and have the following representations:

Bit	Description
0	Hold : If this bit is set then the tone and the end of the initial decay will be held. (see Continue)
1	Alternate : If this bit is set then the Attack will alternate directions while being repeated. (see Continue)
2	Attack : A value of zero in this field will cause the tone to go from loud to soft (decay), whereas a value of one in this fields will cause the tone to go from soft to loud (attack).
3	Continue : If this bit is set then the Attack will repeat itself until stopped by another sound. Note that the Alternate and Hold bits are only valid when this bit is set.

The possible envelopes are:

00xx	=	Decay and hold.
01xx	=	Attack, sharp decay, and hold.
1000	=	Decay, sharp attack, decay (reverse saw-tooth wave).
1001	=	Decay and hold.
1010	=	Decay, attack (Triangle wave).
1011	=	Decay, sharp attack, and hold.
1100	=	Attack, sharp decay, Attack (saw-tooth wave).
1101	=	Attack and hold.
1110	=	Attack, Decay (Triangle wave).
1111	=	Attack, sharp decay, and hold.

Reg. E This register controls port A of the sound chip (not to be confused with channel A). These ports are not used for sound

18

	generation on the Atari ST. They are used to control the floppy disk drive select signals. Note that the state of this port (input/output) is determined by register 7.
Reg. F	This register performs the same function as register E, except that Port B of the sound chip is affected.

These routines are defined as macros in <osbind.h>.

EXAMPLE

```
#include <osbind.h>

#define WRITE 0x80

/*
    Define register set for each tone.
*/
char tone1[] = { 0x1b, 0x01, 0x1c, 0x01, 0x1d, 0x01, 0x00,
                 0x38, 0x10, 0x10, 0x10, 0x00, 0x30, 0x03
};

char tone2[] = { 0xa7, 0x00, 0xab, 0x00, 0xa9, 0x00, 0x00,
                 0x38, 0x10, 0x10, 0x10, 0x00, 0x30, 0x03
};

char tone3[] = { 0xd3, 0x00, 0xd4, 0x00, 0xd5, 0x00, 0x00,
                 0x38, 0x10, 0x10, 0x10, 0x00, 0x30, 0x03
};

char tone4[] = { 0xa8, 0x01, 0xa9, 0x01, 0xaa, 0x01, 0x00,
                 0x38, 0x10, 0x10, 0x10, 0x00, 0x30, 0x03
};

char *song[] = { tone1, tone2, tone3, tone4, tone1 };

main()
{
    int x;
    int reg7;       /*  bits 7 and 6 are used by the OS */

    puts("Phone home?");

    /*
        Save bits 7 and 6 of register 7
    */
    reg7 = Giaccess(0, 7);
```

18

```
    /*
        Play each tone.
    */
    for(x=0; x<(sizeof(song) / sizeof(char *)); x++) {
        /*
            Play tone.
        */
        do_tone(song[x], reg7 & 0xc0);

        /*
            Wait for tone to finish.
        */
        wait60(30);
    }

    Bconin(2);
}

/*
    do_tone -  setup the sound chips registers.
        As the registers change the tone is produced.
*/
do_tone(thetone, mask)
    char *thetone;
    int mask;
{
    int x;

    for(x=0; x<0x0e; x++)
        if ( x == 7 )
            Giaccess((unsigned)thetone[x]|mask, x|WRITE);
        else
            Giaccess((unsigned)thetone[x], x|WRITE);
}

/*
    wait60 - wait for delay 1/60th seconds
*/
wait60(delay)
    int delay;
{
    while(delay--)
        Vsync();
}
```

18

SEE ALSO

Dosound

NAME

Ikbdws — Write a string to the intelligent keyboard processor

SYNOPSIS

```
#include <osbind.h>

Ikbdws(cnt, ptr)
    int   cnt;
    char *ptr;
```

DESCRIPTION

The *Ikbdws* functions writes a string of characters to the intelligent keyboard processor. The parameter `ptr` points to an array of characters whcih are commands for the keyboard processor. The last parameter `cnt` is the number of characters to write minus one.

Ikbdws is defined as a macro in `<osbind.h>`

NOTE

There is a set of commands that the keyboard processor understands pertaining to the handling of the mouse and keyboard. These are defined as follows:

0x07 Return the result of the mouse buttons when pressed. This command is only valid in absolute mode. The following byte defines how the keyboard controller will react to mouse events as follows:

Bit	Meaning
0	return position when the button is pressed.
1	return position when the button is released.
2	affect mouse position through keyboard.
3 – 7	zero.

0x08 Return the mouse position in relative mode. This will return the mouse position in terms of the distance from the last position of the mouse. A mouse packet is generated when the threshold value of the mouse is exceeded. The mouse packet returned in this mode is as follows:

Header byte whose values range from `0xf8` to `0xfb` (The low order bits represent the state of the mouse buttons)

1 byte msb x position.
1 byte lsb x position.
1 byte msb y position.
1 byte lsb y position.

0x09 — Return the mouse position is absolute mode. This will return the mouse position in terms of an absolute coordinate system. This command must be followed by two bytes. The first byte indicates the maximum X value of the mouse and the second byte indicates the maximum Y value.

0x0a — Sets the keyboard controller to treat the mouse movement like the movement described by the cursor keys. This command must be followed by two bytes. The first byte indicates the stepping for the X coordinate counter, and the second indicates the stepping for the Y coordinate counter when the mouse keyboard equivalent is struck.

0x0b — Sets the threshold value for the mouse. This function determines how responsive the mouse is in terms of how far the mouse must move before a mouse packet is sent to the mouse handler. The command must be follwed by two bytes. The first byte defines the threshold for movement in the X direction. The second byte defines the threshold for movement in the Y direction. This command can only be used in relative mode (command 0x08).

0x0c — Set the mouse scale. This function determines how responsive the mouse is in terms of how far the mouse must move before the coordinate is changed. The command must be followed by two bytes. The first bytes defines the X scale. The second bytes defines the Y scale. This command an only be used in absolute mode (command 0x09).

0x0d — Get the mouse's absolute position. This function will cause a keyboard packet to be returned via the keyboard packet handler. The following bytes will be returned:

1 byte header = 0xf7
1 byte mouse button status.

Bit	Meaning
0	right button pressed since the last read.
1	right button not pressed since last read.
2	left button pressed since the last read.
3	left button not pressed since last read.

1 byte msb X coordinate
1 byte lsb X coordinate
1 byte msb Y coordinate
1 byte lsb Y coordinate

`0x0e` Set the mouse position. This command requires five bytes. The first bytes is a 0 byte. The next two bytes define the mouse's X position. The last two bytes define the mouse's Y position.

`0x0f` Set mouse Y-axis origin at bottom.

`0x10` Set mouse Y-axis origin at top.

`0x11` Resume the sending of data packets. (see command `0x13`)

`0x12` Turn off mouse handling. If the mouse mode is changed the keyboard controller will resume mouse handling.

`0x13` Pause the sending of data packets and buffer any keyboard or mouse commands.

`0x14` Force the keyboard controller to return a data packet for each movement of the joystick. The data packet returned has the following format:

1 byte Header (`0xfe` = joystick 0, `0xff` = joystick 1)
1 byte status:

Bits 0 – 3: position of joystick.
Bit 7: status of fire button.

`0x15` stop the keyboard controller from automatically returning joystick data packets.

18

`0x16`	read the joystick. This command causes the keyboard controller to send a data packet to the joystick packet handler. The format of the packet is the same as with command `0x14`.
`0x17`	joystick timeout. This command sets the interval in 1/100'ths of a second between each joystick packet that is sent. Once this command is invoked the following packet is sent at the end of every interval:

1 byte time since last message in 1/100'ths of a second.
1 byte (bit 0 fire button joystick 1, bit 1 fire button joystick 2)
1 byte
  bits 0 – 3: position of joystick 1
  bits 4 – 7: position of joystick 0.

`0x1a`	turn off joystick handling.
`0x1b`	set clock time. This command is followed by 6 bytes which are defined in BCD (Binary Coded Decimal, every four bits is a decimal digit) format. These bytes are defined as follows:

18

1 byte year
1 byte month
1 byte day
1 byte hour
1 byte minute
1 byte seconds

`0x1c`	read block time. This command will cause the keyboard controller to send a data packet to the clock packet handler. The header byte for this packet is `0xfc`. The header byte will be followed by BCD values in the format described in command `0x1b`.
`0x80`	reset keyboard controller without affecting the internal clock. This command must be followed by a single byte `0x01`.

NOTE

The packet handling routines may be defined by the *Kbdvbase* function. Also, for a complete discussion of the keyboard commands refer to the HD-6301 technical reference manual.

SEE ALSO

Kbdvbase

NAME

Initmous — Initialize mouse packet handler

SYNOPSIS

```
#include <osbind.h>

Initmous(type, paramp, vec)
    int           type;
    mouse_data   *paramp;
    int          (*vec)();
```

DESCRIPTION

Initmous sets up the mouse's initial state and the mouse's interrupt handler. The parameters are defined as follows:

`type` indicates the operation to be performed:

0	=	disable mouse
1	=	enable mouse and set to relative mode
2	=	enable mouse and set to absolute mode
3	=	unused
4	=	enable mouse and set to keycode mode

18

`param` points to a param struct:

```
struct _mouse_data {
    char topmode;
    char buttons;
    char xparam;
    char yparam;
    int  xmax, ymax;          /* absolute only */
    int  xinital, yinital;   /* absolute only */
}
```

`topmode` values:

0	y position of 0 at bottom of screen
1	y position of 0 at top of screen

buttons is a parameter to the keyboard's "set mouse buttons" command.
txparam and yparam have the following meanings depending on the mode:

mode	meaning of xparam and yparam
relative	x and y interrupt threshold values
absolute	x and y scale factors
keycode	x and y delta factors

The absolute mode requires the additional x and y maximum and x and y initial values.

vec — points the the mouse interrupt handler (see *kbdvbase*).

Initmous is defined as a macro in <osbind.h>

SEE ALSO

kbdvbase

18

NAME

Iorec — Get serial device input buffer descriptor

SYNOPSIS

```
#include <osbind.h>

iorec *Iorec(device)
    int device;
```

DESCRIPTION

Iorec returns a pointer to a device input buffer descriptor record. The result returned is a pointer to the input buffer record for the device specified in the parameter device. The parameter device will be defined as one of the following:

Device Number	Device Name
0	RS232
1	Keyboard
2	MIDI

The structure the returned pointer points to is defined as follows:

```
typedef struct _iorec {
    char *ibuf;    /* pointer to queue       */
    int ibufsiz;   /* size of queue in bytes */
    int ibufhd;    /* head index of queue    */
    int ibuftl;    /* tail index of queue    */
    int ibuflow;   /* low water mark         */
    int ibufhigh;  /* high water mark        */
} iorec;
```

ibuf is a pointer to the I/O buffer. ibuftl points at the last character to enter the queue. ibufhd points just before the next character to be removed from the queue. The queue is empty if ibufhd equals ibuftl.

The ST will request the sender to stop transmitting when the number of characters in the queue equals ibufhigh. It will request the sender to resume when the number drops to ibuflow. Output flow control for RS-232 operates in a similar manner.

Iorec is defined as a macro in <osbind.h>

NOTE

An output buffer descriptor (just like the input descriptor) immediately follows the input descriptor in memory if the device is RS-232.

EXAMPLE

```
#include <osbind.h>

show_iorec(device)
    int    device;
{
    iorec *therec;

    therec = (iorec *)Iorec(device);

    printf("\nThe device %d is:\n", device);
    printf("buffer      == %lx\n", therec -> ibuf);
    printf("buffer size == %d\n",  therec -> ibufsiz);
    printf("head index  == %d\n",  therec -> ibufhd);
    printf("tail index  == %d\n",  therec -> ibuftl);
    printf("low mark    == %d\n",  therec -> ibuflow);
    printf("high mark   == %d\n",  therec -> ibufhigh);
}
```

NAME

Kbrate — Get/set keyboard repeat rate

SYNOPSIS

```
#include <osbind.h>

int Kbrate(initial, repeat)
    unsigned char initial, repeat;
```

DESCRIPTION

Kbrate establishes the time before a key repeat is performed and how much of a delay between each repeat. The parameter `initial` establishes the delay before the auto-repeat begins. The last parameter `repeat` determines the delay between each repeated key. If a value of -1 is passed for either of the parameters then that value will not be changed. Note that all times are measured in ticks, each tick being about 20 microseconds. The previous setting of `initial` and `repeat` is returned as an integer with `initial` in the high byte and `repeat` in the low byte.

Kbrate is defined as a macro in `<osbind.h>`

EXAMPLE

```
#include <osbind.h>

/*
    set_keyrepeat() - set the keyboard repeat rate at 1 second before
        repeat and repeat 4 keys per second.
*/
set_keyrepeat()
{
    int initial = 50;   /* delay for a second.  1000 per second / 20 ticks */
    int repeat  = 12;   /* repeat.  50 ticks per sec / 4 reps per sec
                             == 12 ticks per rep */

    Kbrate(50, 12);
}
```

18

NAME

Kbdvbase — Get list of various system vectors

SYNOPSIS

```
#include <osbind.h>

kbdvecs *Kbdvbase()
```

DESCRIPTION

Kbdvbase returns a pointer to a list of system vectors. The pointer to the following structure is returned:

```
typedef struct {
    int (*midivec)();    /* MIDI-input */
    int (*vkbderr)();    /* keyboard error */
    int (*vmiderr)();    /* MIDI error */
    int (*statvec)();    /* ikbd status packet */
    int (*mousevec)();   /* mouse packet */
    int (*clockvec)();   /* clock packet */
    int (*joyvec)();     /* joystick packet */
    int (*midisys)();    /* system MIDI vector */
    int (*ikbdsys)();    /* system IKBD vector */
} kbdvecs;
```

18

`midivec` points to a routine in the BIOS that returns the character read from the MIDI port in the low byte of `D0`.

`vkbderr` and `vmiderr` are called whenever an overrun condition is detected on the keyboard or MIDI 6850s.

`statvec`, `mousevec`, `clockvec` and `joyvec` point to ikbd (intelligent keyboard) packet handlers for the status, mouse, real-time clock and joystick. A pointer to the packet is passed to the routine in `A0` and on the stack. The handler returns with an `rts` instruction (as opposed to `rte`) and must not spend more than 1 millisecond in the routine.

`midisys` and `ikbdsys` are called when characters are ready on the appropriate 6850. They dispatch to the other vectors. The initial `midisys` just calls `midivec`. The initial `ikbdsys` figures out what kind of thing has happened and calls one of `statvec`, `mousevec`, `clockvec` or `joyvec`.

Kbdvbase is defined as a macro in `<osbind.h>`

EXAMPLE

```
#include <osbind.h>

extern packet_handler();
extern status();
extern state();

kbdvecs *thevecs;
int     *thestate  = (int *) state;
long     savevec;


main()
{
    int x, y;

    appl_init();

    /*
        Get pointer to vector table and replace the mouse
            packet handler.
    */
    thevecs = (kbdvecs *)Kbdvbase();
    savevec             = (long) thevecs -> mousevec;
    thevecs -> mousevec = packet_handler;

    /*
        Loop Until done.
    */
    while (x < 100) {
        if (*thestate) {
            char *p = (char *)status;
            printf("status == %02x %02x %02x %02x %02x %02x\n",
                *(p+0)&0xff, *(p+1)&0xff, *(p+2)&0xff,
                *(p+3)&0xff, *(p+4)&0xff, *(p+5)&0xff);

            x++;

            *thestate = 0;
        } else
            if (y++ > 100) {
                puts("Waiting...");
                y = 0;
            }
    }

    /*
        Restore the old mouse packet handler.
    */
    thevecs -> mousevec = (int(*)()) savevec;
```

18

```
    appl_exit();
}


/*
    Packet_handler  -  move packet information to a more easily accessible
        place.
*/
asm {
status:
        dc.b    0, 0, 0, 0, 0, 0
state:
        dc.w    0

packet_handler:
        lea     status(PC), A1  /*  get the address of our status work space */

        move.b  (A0)+, (A1)+    /*  move it over fast.                        */
        move.b  (A0)+, (A1)+
        move.b  (A0)+, (A1)+
        move.b  (A0)+, (A1)+
        move.b  (A0)+, (A1)+
        move.b  (A0)+, (A1)+

        lea     state(PC), A0
        addq    #1, (A0)
        rts                     /* kiss it goodbye.                           */
}
```

18

NAME

Kbshift — Gets or sets the keyboard shift bits

SYNOPSIS

```
#include <osbind.h>

long Kbshift(mode)
    int mode;
```

DESCRIPTION

Kbshift returns information about the keyboard's special keys. The parameter `mode` controls the setting or getting of the keyboard shift bits. If `mode` is negative then the current settings are returned. If `mode` is non-negative then the value of `mode` is used to set the shift bits. The shift bit assignments are as follows:

Bit	Key
0	Right shift key
1	Left shift key
2	Control key
3	ALT key
4	Caps-lock key
5	Right mouse button (CLR/HOME)
6	Left mouse button (INSERT)
7	Reserved

18

Kbshift is implemented as a macro in `<osbind.h>`.

NAME

Keytbl, Bioskeys — Sets keyboard translation tables

SYNOPSIS

```
#include <osbind.h>

long Keytbl(unshift, shift, capslock)
    char unshift[128], shift[128], capslock[128];

Bioskeys()
```

DESCRIPTION

Keytbl sets the keyboard translation tables. `unshift`, `shift` and `capslock` point to keycode-to-ASCII translation tables for unshifted, shifted and capslock down while pressed keys respectively. The pointers are stored into the following structure for which a pointer to is returned (as a long):

```
struct keytab {
    char *unshift;
    char *shift;
    char *capslock;
};
```

Bioskeys restores the initial boot up values of the translation tables.

Both routines are defined as a macro in `<osbind.h>`

18

NAME

Malloc, Mfree, Mshrink — Memory allocator

SYNOPSIS

```
#include <osbind.h>

long Malloc(amount)
    long amount;

int Mfree(addr)
    char *addr;

int Mshrink(zero, mem, size)
    int   zero;
    char *mem;
    long  size;
```

DESCRIPTION

These functions are used to manipulate memory dynamically.

Malloc allocates `amount` bytes of memory from the current program's heap and returns a pointer to the beginning of the block. The block is word aligned. If `amount` is −1L then the amount of free space in the heap is returned. A NULL value is returned if the requested number of bytes is not available or an error occurs

Mfree releases the block pointed to by `addr` and returns the space to the program's heap. The block must have been allocated by *Malloc*. A non-zero value is returned if an error occurs.

Mshrink changes the size of the heap. The parameter `zero` must have a value of zero. `mem` points to the base of the TPA. `size` is the number of bytes to retain in the TPA for the program (basically the size of the base page + code + data + bss + stack). This function is used by the system library before *main* is called. A non-zero value is returned if an error occurs.

These functions are defined as macros in `<osbind.h>`

EXAMPLE

```
#include <osbind.h>
```

```
show_freemem()
{
    printf("There are %ld bytes free in the heap.\n", Malloc(-1L));
}
```

NAME

Mediach — Return current "media-changed" value for a device

SYNOPSIS

```
#include <osbind.h>

long Mediach(dev)
    int dev;
```

DESCRIPTION

Mediach is used internally by BIOS before reading and writing to ensure the disk has not been replaced by another disk. The parameter **dev** is the device to return the "media-changed" value for.

The return value is one of:

0	Media definitely has not changed
1	Media might have changed
2	Media definitely has changed

Mediach is implemented as a macro in **<osbind.h>**.

NAME

Mfpint, Jdisint, Jenabint — Set, disable and enable interrupts on the MFP

SYNOPSIS

```
#include <osbind.h>

Mfpint(interno, vector)
    int interno;
    int (*vector)();

Jdisint(interno)
    int interno;

Jenabint(interno)
    int interno;
```

DESCRIPTION

Mfpint sets the 68901 MFP chip to interrupt to a user defined routine. The interrupt number is specified in the parameter `interno`. The user defined interrupt handler is defined by the parameter `vector`.

Jdisint disables the interrupt specified by the parameter interno.

Jenabint enables the interrupt specified by the parameter interno.

18

The 68901 MFP (Multi-Function Peripheral) supports up to 16 interrupt functions. The address of each function is stored in the 68000's vector numbers 64 – 79 (4 bytes each starting at address 0x100). Interrupt functions on the 68000 must return with the RTE instruction instead of the usual RTS. See the Morotola 68000 reference manual for more information. Additionally, the bit in the 68901's "Interrupt Status Register" (ISR) corresponding to the interrupt number must be cleared before returning.

There are two 8-bit ISRs labeled ISRA and ISRB. ISRA has a bit for functions 8-15 while ISRB has a bit for functions 0-7.

Interrupt numbers are assigned as follows:

interno	ISR bit	Used for
0	ISRB 0	Parallel port (initially disabled)
1	1	RS232 Carrier Detect (initially disabled)
2	2	RS232 Clear-To-Send (initially disabled)
3	3	Unused, disabled
4	4	Unused, disabled (Timer D)
5	5	200hz system clock (Timer C)
6	6	Keyboard/MIDI (6850)
7	7	Polled FDC/HDC (initially disabled)
8	ISRA 0	HSync (initially disabled) (Timer B)
9	1	RS232 transmit error
10	2	RS232 transmit buffer empty
11	3	RS232 receive error
12	4	RS232 receive buffer empty
13	5	Unused, disabled (Timer A)
14	6	RS232 Ring detect (initially disabled)
15	7	Polled monitor type (initially disabled)

ISRA is stored at memory location `0xfffa0f`. ISRB is at `0xfffa11`.

See *Xbtimer* for information about programming the four timers (A,B,C and D).

These routines are defined as macros in `<osbind.h>`

NOTE

The interrupt priority levels are from 0 to 15, lowest to highest priority.

EXAMPLE

An example of a 68901 interrupt routine is provided in *Xbtimer*.

SEE ALSO

Xbtimer(), Rsconf()

NAME

Midiws — Write a string to the MIDI port

SYNOPSIS

```
#include <osbind.h>

Midiws(cnt, ptr)
    int   cnt;
    char *ptr;
```

DESCRIPTION

Midiws writes characters out accross the MIDI out port of the ST. The parameter `cnt` is the number of characters to write minus 1. The parameter `ptr` is a pointer to the data that is to be written out.

NOTE

MIDI is an acronym which stands for "Musical Instrument Device Interface." This interface is a standard for most of the electronic synthesizers on the market today.

Below is listed some of the commands that are defined in the MIDI standard. This is not a complete list, nor can it be due to the fact that each synthesizer has a set of commands which shows of it's own individual talents as well as the manufacturers thoughts. These channel messages are defined as three 8-bit bytes where the individual bits are represented as follows:

18

Flag	Description
c	Channel. There are sixteen total channels available (0 – 15).
k	Key pressed. From piano: 21 (low D) – 108 (high C).

Middle C	=	60
Sharps	=	note number + 1
Flats	=	note number - 1
Octave jumps	=	note number + 12

v	Velocity. Determines how loud a note is to be played.
	`(soft) 0_1___________64____________127 (loud)`
	`ppp  pp   mp     mf    f  ff fff`

p	Program number (0 – 127).
b	Pitch bender range (0 – 127). 64 = center (i.e. no bend)
x	Don't care

MIDI command	**Message Description in bits**
note OFF	1000 cccc + 0kkk kkkk + 0100 0000
note ON	1001 cccc + 0kkk kkkk + 0vvv vvvv
OSC modulation	1011 cccc + 0000 0001 + 0vvv vvvv
VCF modulation	1011 cccc + 0000 0010 + 0vvv vvvv
Damper pedal OFF	1011 cccc + 0100 0000 + 0000 0000
Damper pedal ON	1011 cccc + 0100 0000 + 0111 1111
Portamento OFF	1011 cccc + 0100 0001 + 0000 0000
Portamento ON	1011 cccc + 0100 0001 + 0111 1111
Program change	1100 cccc + 0ppp pppp + xxxx xxxx
Channel Pressure	1101 cccc + 0vvv vvvv + xxxx xxxx
Pitch bender change	1110 cccc + 0000 0000 + 0bbb bbbb

Note that if a note is specified that is outside the range of the synthesizer, then the note is transposed to the nearest octave.

Midiws is defined as a macro in `<osbind.h>`

EXAMPLE

An example of *Midiws* is in the file `midi.c` on the Examples disk supplied with the Laser package.

18

NAME

Pexec — Load another program

SYNOPSIS

```
#include <osbind.h>

long Pexec(mode, path, commandline, environment)
    int   mode;
    char *path, *commandline, *environment;
```

DESCRIPTION

Pexec is used to launch an application from another application. There are several modes that may be specified by the **mode** parameter. These modes are defined as follows:

Mode	Function	Description
0	load and go	Set up the parameters as described in the description section.
3	just load	Exactly like mode = 0, however, the address of the base page is returned and the application is not executed.
4	just go	pathname = address of the base page.
5		create a base page and allocate free memory.

path the file containing the program to load.

commandline the command line image to be placed in the base page. The command line may include I/O redirection.

environment the environment string to be placed in the base page. If environment is 0L then the parent program's environment string is used.

Pexec is defined as a macro in **<osbind.h>**

NOTE

The **commandline** parameter is actually a Pascal style string (i.e. length byte with character data following).

DIAGNOSTICS

If the load fails, then a negative error number is returned.

EXAMPLE

```
#include <osbind.h>
#include <stdio.h>

#define LOADNGO 0
/*
    This program demonstrates using the Pexec() function to launch
        itself.  Note that in order to work it's executable name
        must be ''pexec.prg''.
*/
main(argc, argv)
    int   argc;
    char *argv[];
{
    if (argc < 2) {
        printf("This is the first time through the Pexec Test Program\n");
        launch("pexec.prg", "Hello world ...");
    } else {
        printf("This is the second time through the Pexec Test Program\n");
        exit(0);
    }

    puts("End of program.");
}

launch(command, commandline)
    char *command;
    char *commandline;
{
    char work[128];    /* the max size of a command line is 128 chars. */

    /*
        Convert to Pascal Style string.
    */
    work[0] = strlen(commandline);
    strcpy(&work[1], commandline);

    Pexec(LOADNGO, command, work, "");
}
```

SEE ALSO

DOS Error Codes (pg. 587)

18

NAME

Cprnout, Cprnos — Printer port write and status.

SYNOPSIS

```
#include <osbind.h>

int Cprnout(chr)
    int chr;

int Cprnos()
```

DESCRIPTION

The Printer I/O functions are designed to facilitate output to a printing device.

Cprnout writes the character `chr` to the printer port. If the character was successfully sent to the printer then a value of −1 is returned as the function's result. If the printer is offline or inactive for more than 30 seconds then a value of zero will be returned.

Cprnos returns non-zero if the printer port is ready to receive a character.

These routines are defined as macros in `<osbind.h>`

EXAMPLE

```
print_text(thetext, thecount)
    char *thetext;
    int   thecount;
{
    if (!Cprnos())
        fatal("Printer is offline.\n");
    else
        while(thecount--)
            if (!(Cprnout(*thetext++)))
                fatal("Error during print.\n");
}
```


SEE ALSO

Character I/O, Console I/O

NAME

Protobt — Construct a prototype boot sector

SYNOPSIS

```
#include <osbind.h>

Protobt(buf, serialno, disktype, execflag)
    char buf[512];
    long serialno;
    int  disktype, execflag;
```

DESCRIPTION

Protobt creates a prototype boot sector at the memory pointed to by the parameter **buf** which may be written to the disk. The rest of the parameters are defined as follows:

serialno A serial number to be stamped into the boot sector. **buf** may already point at an existing boot sector, if it does and **serialno** is −1 then the previous serial number will be used. If **serialno** is greater than 0x01000000 then a random serial number will be used.

disktype The disk type. If it is −1 and **buf** points at an existing boot sector then the **disktype** information is left unchanged. Other values for **disktype** are:

0	single sided 180K, 40 tracks
1	double sided 360K, 40 tracks
2	singed sided 360K, 80 tracks
3	double sided 720K, 80 tracks

execflag The executable status of the boot sector. If **execflag** is −1 and **buf** points at an existing boot sector, then the sector is left unchanged with respect to executable status. If **execflag** is 1, the boot sector is made executable. If it is 0 then the boot sector is made non-executable.

Protobt is defined as a macro in **<osbind.h>**

EXAMPLE

Refer to Floppy() for an example of *Protobt*

18

SEE ALSO

Floppy

NAME

Pterm0, Pterm, Ptermres — Terminate current process

SYNOPSIS

```
#include <osbind.h>

Pterm0()

Pterm(code)
    int code;

Ptermres(keep, ret)
    long keep;
    int ret;
```

DESCRIPTION

These *Pterm* functions terminate the current program and return control to the calling program. Each of these functions has a slightly different behavior as follows:

Pterm0 terminates the current process with an exit status of 0.

Pterm terminates the current process with an exit status of `code`.

18

Ptermres terminates the current process with an exit status of `ret`, but leaves it in memory. The `keep` parameter is the number of bytes to leave in the process descriptor.

These routines are defined as macros in `<osbind.h>`

NAME

Puntaes — Throw away GEM AES freeing up its space

SYNOPSIS

```
#include <osbind.h>

Puntaes()
```

DESCRIPTION

Puntaes will cause the system to reboot, but won't load the AES routines or GEM desktop. *Puntaes* will just return if it has already been called.

Puntaes is defined as a macro in `<osbind.h>`

NOTE

This won't work with the system in ROM.

NAME

Random — Generate a 24-bit pseudo-random number

SYNOPSIS

```
#include <osbind.h>

long Random()
```

DESCRIPTION

Random generates a 24-bit pseudo-random number which is returned as the function's result as a long. A linear congruential algorithm is used:

$$S = (S \times C) + K$$

K is 1 and C is 3141592621. The initial value for S is taken from the frame-counter global (_frclock). S >> 8 is returned.

Random is defined as a macro in <osbind.h>

NAME

Rsconf — Configure the RS232 port

SYNOPSIS

```
#include <osbind.h>

Rsconf(speed, flowctl, ucr, rsr, tsr, scr)
    int speed, flowctl, ucr, rsr, tsr, scr;
```

DESCRIPTION

Rsconf sets communication parameters for the serial port.

`speed` Sets the baud rate for the RS232 port as follows:

`speed`	**Baud rate**
0	19,200
1	9600
2	4800
3	3600
4	2400
5	2000
6	1800
7	1200
8	600
9	300
10	200
11	150
12	134
13	110
14	75
15	50

`flowctl` Sets the flow control as follows:

`flowctl`	**Type of flow control**
0	No flow control (default value)
1	XON/XOFF
2	RTS/CTS
3	Both XON/XOFF and RTS/CTS

ucr, rsr, tsr, scr — Set the corresponding 68901 registers. A −1 for one of these parameters will not set the register (so you don't have to set them all). Only the **ucr** register is useful:

Bit	Meaning
0	Not used
1	Parity. 1=even parity, 0=odd parity
2	Parity enable. 1=enabled.
3,4	Start/Stop bits:

Bit 4	Bit 3	Start	Stop	Format
0	0	0	0	Sync.
0	1	1	1	Async.
1	0	1	1.5	Async.
1	1	1	2	Async.

Bit	Meaning
5,6	Word length:

Bit 6	Bit 5	Word length
0	0	8 bits
0	1	7 bits
1	0	6 bits
1	1	5 bits

Bit	Meaning
7	Clock mode. 1 = 1/16 rate (use this one), 0 = full speed.

18

Rsconf is defined as a macro in `<osbind.h>`. The 68901 MFP Timer D is used to control the baud rate. See *Xbtimer* for information on how to program the timer for other baud rates.

EXAMPLE

```
#define BAUD19200   0
#define XON_XOFF    1

/*
    Initialize RS-232 port to 19.2 kbaud using XON/XOFF
        flow control.
*/
initRS232()
{
    Rsconf(BAUD19200, XON_XOFF, -1, -1, -1, -1);
}
```

SEE ALSO

Mfpint(), Xbtimer()

NAME

Rwabs — Read/write blocks on a device.

SYNOPSIS

```
#include <osbind.h>

long Rwabs(rwflag, buf, count, recno, dev)
    int   rwflag;
    char *buf;
    int   count, recno, dev;
```

DESCRIPTION

Rwabs allows the user to read and write to the disk using absolute block references. The parameter `rwflag` is contains the operation to be performed which is defined as follows:

0	Read
1	Write
2	Read, but doesn't affect "media-change"
3	Write, but doesn't affect "media-change"

If an error occurs during the requested operation then a negative value will be returned. A return value of 0L indicates successful operation. The rest of the parameters are defined as follows:

18

`buf`	points to a buffer to be read or written. Note that if the buffer begins on an odd address the performance of the operation will decrease.
`count`	the number of blocks to transfer.
`recno`	the logical sector to start transferring at.
`dev`	the device number:

0	=	Floppy drive `A:`
1	=	Floppy drive `B:`
> 1	=	Hard disks, networks or other devices.

Rwabs is implemented as a macro in `<osbind.h>`.

SEE ALSO

mediach

NAME

Scrdmp — Dump B/W screen to printer

SYNOPSIS

```
#include <osbind.h>

Scrdmp()
```

DESCRIPTION

Scrdmp sends the current screen data to the printer. Note that this only works with the black and white monitor.

Scrdmp is defined as a macro in `<osbind.h>`

EXAMPLE

```
#include <osbind.h>

print_screen()
{
    if (Getrez() != 2)
        printf("Cannot print screen in this screen resolution!");
    else
        Scrdmp();
}
```

18

NAME

Physbase, Logbase, Setscreen, Getrez — Screen functions

SYNOPSIS

```
#include <osbind.h>

long Physbase()

long Logbase()

int Getrez()

Setscreen(log_loc, phys_loc, rez)
    char *log_loc, *phys_loc;
    int   rez;
```

DESCRIPTION

These functions are designed to facilitate the manipulation of the graphics screens. The functions are defined as follows:

Physbase returns the screen's physical location in memory at the next vertical blanking interrupt.

Logbase returns the screen's logical location immediately. Note that the logical screen base is where all drawing is done. This may contrast with the physical screen base where the video hardware looks for the data to display on the monitor.

Getrez returns the current screen resolution (0=low, 1=medium, 2=high).

Setscreen sets the logical base, physical base and resolution for the screen. A negative value for a parameter is ignored so it is possible to set only one or two of the values. The logical base is set immediately. The physical base will not be changed until the next vertical blanking interrupt. Changing the screen resolution causes the screen to be cleared and the VT52 emulator to be reset. The address of the screen must be on a page (256 byte) boundary.

These routines are defined as macros in `<osbind.h>`

NOTE

Even when the screen resolution is changed certain parts of GEM are not aware of the change.

EXAMPLE

```
#include <stdio.h>
#include <osbind.h>

/*
    Example showing the use of Vsync(), Physbase(), Setscreen().
*/
main()
{
    register char *vis_screen, *back_screen, *temp;
    int rez;
    int count = 10;

    /*
        Allocate memory for second screen.
    */
    back_screen = malloc(32768 + 256);

    /*
        The screen address must be on a 256 byte boundary.
    */
    if ((long) back_screen & 0xff)
        back_screen = back_screen + (0x100 - (long)back_screen & 0xff);

    /*
        Get visible screen address.
    */
    vis_screen  = (char *)Physbase();

    /*
        Get the current resolution
    */
    rez = Getrez();

    /*
        Reset VT52 cursor to top-left of screen
    */
    printf( "\033Y%c%c", 0 + ' ', 0 + ' ' );
    fflush( stdout );

    puts("Example showing the use of Vsync(), Physbase(), Setscreen()");
    puts("Press return to start the pageflip...");
    getchar();

    /*
        Wait until the vertical blank interrupt and then swap the screens.
```

```
		The physical screen address is the screen image that is displayed.
		The logical screen address is the back screen where everything is
		drawn.
	*/

	while ( count-- ) {
		Setscreen(back_screen, vis_screen, -1);

		/*
			do your thing.
		*/
		draw(back_screen,count);

		/*
			swap screens.
		*/
		temp        = vis_screen;
		vis_screen  = back_screen;
		back_screen = temp;
	}

	Setscreen(vis_screen, vis_screen, -1);
	puts("Press return...");
	getchar();
}


draw(back,count)
	char *back;
{
	int i   = 60;

	while(i--)
		Vsync();
	/*
		Draw and undraw the animation here in the background screen.
	*/
	if ( count & 1 )
		puts( "This is the first screen" );
	else
		puts( "This is the second screen" );
}
```

18

NAME

Setcolor — Set an entry in the hardware color palette

SYNOPSIS

```
#include <osbind.h>

int Setcolor(colornum, color)
    int colornum, color;
```

DESCRIPTION

Setcolor allows the user to change a color in the color palette. The color palette entry `colornum` is set to `color`. If the `color` is negative no change is made to the color. The result of the function is the previous value of the `colornum` before the call. The color is defined by intensity values for each of the different colors in the monitors rgb gun.

Bits	Description
0 – 2	The intensity of blue.
4 – 6	The intensity of green.
8 – 10	The intensity of red.

Setcolor is defined as a macro in `<osbind.h>`

EXAMPLE

```
#include <osbind.h>

#define RED     8
#define GREEN   4
#define BLUE    0

display_palette_values()
{
    int colornum;
    int color;

    for(colornum = 0; colornum < 16; colornum++) {
        color = Setcolor(colornum, -1);
        printf("The color %d contains %d red, %d green, %d blue\n",
                colornum,
                (color >> RED)   & 0xf,
                (color >> GREEN) & 0xf,
                (color >> BLUE)  & 0xf);
    }
}
```

18

NAME

Setexc — Set exception vector for 68000

SYNOPSIS

```
#include <osbind.h>

long Setexc(vecnum, vec)
    int vecnum;
    int (*vec)();
```

DESCRIPTION

Setexc changes one of the 68000's exception vectors. The parameter vecnum defines the number of the vector that is to be changed. The parameter vec is the address of the new vector routine. The function result will be the previous value of the vector if it was changed. If the vector was not changed then the function will return a value of −1.

Setexc is implemented as a macro in <osbind.h>.

NOTE

The 68000 reserves vectors 0x00 through 0xFF. Vectors 0x100 through 0x1FF are reserved for GEM DOS. The following are currently implemented:

0x100	System timer interrupt
0x101	Critical error handler
0x102	Process termination vector
0x103 – 0x107	Unused but reserved

18

The vectors above 0x200 are reserved for OEM use.

EXAMPLE

```
extern death();

set_error_handler()
{
    Setexc(0x101, death);
}

death()
{
    puts("Oops!");
    Pterm(1);
}
```

NAME

Setpalette — Set the contents of the hardware color palette

SYNOPSIS

```
#include <osbind.h>

Setpalette(newpalette)
    int newpalette[16];
```

DESCRIPTION

Setpalette sets the color palette to user defined values. The 16 color video lookup table is loaded with the values from **newpalette**. The assignment will occur at the next vertical blanking interrupt.

Setpalette is defined as a macro in **<osbind.h>**

NOTE

The colors in the palette are described in the color word in terms of the intensities of each of the rgb guns.

Bits	Description
0 – 2	The intensity of blue.
4 – 6	The intensity of green.
8 – 10	The intensity of red.

NAME

Setprt — Set/get printer configuration word

SYNOPSIS

```
#include <osbind.h>

int Setprt(config)
    int config;
```

DESCRIPTION

Setprt allows the setting and querying of the printer's current configuration. If the value of the parameter `config` is negative then the configuration word is returned as the function's result. If `config` is a positive value then the printer will be set to the value of config and the previous configuration word is returned as the function's result. The bits in `config` represent information about the printer as follows:

Bit	Meaning if 0	Meaning if 1
0	Dot matrix	Daisy wheel
1	Color device	Monochrome device
2	Atari printer	Epson style printer
3	Draft mode	Final mode
4	Parallel port	RS232 port
5	Continuous feed	Single sheet
6 – 14	Unused	
15	Must be zero	

Setprt is defined as a macro in `<osbind.h>`

18

NAME

Super — Change 68000 privilege status

SYNOPSIS

```
#include <osbind.h>

long Super(stack)
    long stack;
```

DESCRIPTION

Super allows the user to change or query the 68000's supervisor mode. If the parameter `stack` is −1, then a 0 is returned if the processor is in user mode, and a 1 if it is in supervisor mode.

If the processor is in user mode and `stack` is greater than zero, then *Super* switches to the supervisor state and sets the SSP (supervisor stack pointer) to `stack`.

If the processor is in user mode and `stack` is zero, then *Super* switches to the supervisor state and sets the SSP to the current USP (user stack pointer).

If the processor is in the supervisor state, then *Super* switches to the user state and uses `stack` as the new SSP.

Super returns the old SSP value for the above three conditions.

Super is defined as a macro in `<osbind.h>`

EXAMPLE

```
#include <osbind.h>

/*
    tickcount() - returns the number of 200 hertz ticks that have
        occurred since system power-up.  Since this information lies in
        protected memory it is necessary to move into the 68000 supervisor
        mode.
*/
long tickcount()
{
    /*
        Put in supervisor mode.  Save user stack in User_stack.  Get
        Hz200 tickcount from the System global.
    */
    long User_stack = Super(0L);
    long ticks      = *(long *)0x4ba;

    /*
```

```
        Restore the processor to user mode.
    */
    Super(User_stack);

    /*
        Return the tickcount
    */
    return ticks;
}
```

SEE ALSO

Supexec

NAME

Supexec — Execute a function in 68000 supervisor mode.

SYNOPSIS

```
#include <osbind.h>

long Supexec(func)
    void (*func)();
```

DESCRIPTION

Supexec calls the function pointed to by `func` with the 68000 supervisor mode set. The function should not expect any parameters, and should not return a result.

Supexec is defined as a macro in `<osbind.h>`

EXAMPLE

```
#include <osbind.h>

extern tickcount();

long thetickcount;

/*
    fetch_tickcount() - calls the tickcount function which needs to
        run in supervisor mode.
*/
fetch_tickcount()
{
    Supexec(tickcount);
}

/*
    tickcount() - stores the current 200 hz tick count to the global
        variable thetickcount.
*/
tickcount()
{
    thetickcount = *(long *)0x4ba;
}
```

SEE ALSO

Super

18

NAME

Sversion — Returns current version number of GEM

SYNOPSIS

```
#include <osbind.h>

int Sversion()
```

DESCRIPTION

Sversion returns the current version number of GEM. The low byte contains the major version number and the high byte contains the minor version number.

Sversion is defined as a macro in `<osbind.h>`

NAME

Tickcal — Return system timer calibration value to nearest millisecond.

SYNOPSIS

```
#include <osbind.h>

long Tickcal()
```

DESCRIPTION

Tickcal tells the user how long a system tick is in milliseconds. The function's result is the system timer calibration value rounded to the nearest millisecond.

NOTE

This is not very useful since the number of elapsed milliseconds is passed on the stack when a system timer exception occurs.

Tickcal is implemented as a macro in `<osbind.h>`.

NAME

Tgetdate, Tsetdate, Tgettime, Tsettime — Get/set date and time

SYNOPSIS

```
#include <osbind.h>

int Tgetdate()

Tsetdate(date)
    dateinfo date;

int Tgettime()

Tsettime(time)
    timeinfo time;
```

DESCRIPTION

The time functions are used to manipulate the system's date and time. The functions are defined as follows:

Tgetdate returns the current date in an int using the following format:

bits 0 – 4	Day in month (1 – 31)
bits 5 – 8	Month in year (1 – 12)
bits 9 – 15	Year since 1980 (0 – 119)

Tsetdate sets the date to date using the above format.

Tgettime returns the current time in an integer using the following format:

bits 0 – 4	Current second divided by 2 (0 – 30)
bits 5 – 10	Current minute (0 – 59)
bits 11 – 15	Current hour (0 – 23)

Tsettime sets the time to `time` using the above format.

These functions are defined as macros in `<osbind.h>`

EXAMPLE

```
#include <osbind.h>

/*
    Example of how to get information from the Tgettime() functions.

    Note : To set the time just assign the ''part'' fields of the
        structure and then pass Settime() the  real datetime. Ex:

                time.part.hours = 5;
                time.part.minutes = 34;
                time.part.seconds = 15 / 2;

                Tsettime(time.realtime);
*/
show_date_and_time()
{
    timeinfo mytime;
    dateinfo mydate;

    mytime.realtime = Tgettime();
    mydate.realdate = Tgetdate();

    printf("\t\t date \n Day: %d \t Month: %d \t Year: %d\n",
            mydate.part.day, mydate.part.month, mydate.part.year + 80
    );

    printf("\t\t time \n Hour: %d \t Minute: %d \t Seconds: %d\n",
            mytime.part.hours, mytime.part.minutes, mytime.part.seconds * 2
    );

    printf("\nPress RETURN to exit...\n");
    Bconin(2);
}
```

NAME

Vsync — Wait until the next vertical blanking interrupt

SYNOPSIS

```
#include <osbind.h>

Vsync()
```

DESCRIPTION

Vsync helps in syncing with the video's vertical retrace. This function will not return until the next vertical retrace occurs.

Vsync is defined as a macro in `<osbind.h>`

EXAMPLE

```
/*
    delay() - this function is used to delay a specified number of
        seconds.  Note that this depends on the vertical retrace
        occurring every 1/60th of a second.
*/
delay(secs)
    int secs;
{
    int tsecs;

    while(secs--)
        for(tsecs=0; tsecs<60; tsecs++)
            Vsync();
}
```

NAME

Xbtimer — Set timer on 68901 MFP (Multi-Function Peripheral)

SYNOPSIS

```
#include <osbind.h>

Xbtimer(timer, control, data, vec)
    int timer, control, data;
    int (*vec)();
```

DESCRIPTION

`timer`	The 68901 timer to set (0=A, 1=B, 2=C, 3=D).
`control`	The 8-bit value for the timer control register.
`data`	The 8-bit value for the timer data register. This is used as the inital value for the counter.
`vec`	The address of a new interrupt vector.

The timers are used as follows:

Timer	Usage
A	Reserved for applications
B	Reserved for graphics (HSYNC signal)
C	200hz system timer
D	RS232 baud-rate control. The interrupt vector for this timer may be used for any purpose.

18

Bit 4 of the control register for timers A and B is used to reset the TA0 and TB0 outlines of the 68901 respectively. These lines are not used by the ST.

Timers A and B can be in one of three modes:

Delay	Timer continuously counts down to 0 and interrupts
Pulse width	Used to measure external signals
Event count	Used to count external events

Bits 0 – 3 have the following meaning:

Bit 3	Bit 2	Bit 1	Bit 0	**Meaning**
0	0	0	0	Stop timer
0	0	0	1	Delay mode, divide by 4 prescale
0	0	1	0	Delay mode, divide by 10 prescale
0	0	1	1	Delay mode, divide by 16 prescale
0	1	0	0	Delay mode, divide by 50 prescale
0	1	0	1	Delay mode, divide by 64 prescale
0	1	1	0	Delay mode, divide by 100 prescale
0	1	1	1	Delay mode, divide by 200 prescale
1	0	0	0	Event Count mode
1	0	0	1	Pulse width mode, divide by 4 prescale
1	0	1	0	Pulse width mode, divide by 10 prescale
1	0	1	1	Pulse width mode, divide by 16 prescale
1	1	0	0	Pulse width mode, divide by 50 prescale
1	1	0	1	Pulse width mode, divide by 64 prescale
1	1	1	0	Pulse width mode, divide by 100 prescale
1	1	1	1	Pulse width mode, divide by 200 prescale

Timer registers C and D use the same control register. Bits 0 – 2 are used for register D and bits 4 – 6 for register C. The meaning of these bits is as follows:

Bit 2,6	Bit 1,5	Bit 0,4	**Meaning**
0	0	0	Stop timer
0	0	1	Delay mode, divide by 4 prescale
0	1	0	Delay mode, divide by 10 prescale
0	1	1	Delay mode, divide by 16 prescale
1	0	0	Delay mode, divide by 50 prescale
1	0	1	Delay mode, divide by 64 prescale
1	1	0	Delay mode, divide by 100 prescale
1	1	1	Delay mode, divide by 200 prescale

Xbtimer is defined as a macro in `<osbind.h>`

18

NOTE

The timers on the 68901 MFP are controlled by a 2.4576 Mhz crystal. Using timer 'A' in the delay mode with a pre-scale set at 200 (i.e. by setting the timer 'A' control register to 7) creates a 12288 Hz counter (2457600 Hz / 200 pre-scale). Using a count of 256 (i.e. by loading the timer 'A' data register with 0) you get an interrupt frequency of 48 Hz (12288 Hz / 256 ticks).

See the description of *Mfpint* for details of writing interrupt functions for the 68901.

EXAMPLE

```
#include <osbind.h>
#include <stdio.h>

/*
    xbtimer.c

    Sample code that demonstrates the use of TIMER A on the 68901 MFP
    Also demonstrates how to handle the process terminate interrupt.

        The program begins by installing the address of the function 'terminate'
    into the exception vector 0x102 (Process terminate exception), saving its
    old pointer. It then starts up Timer A on the 68901 MFP, configured to
    interrupt the function 'dispatcher' at 48Hz. The main loop continuously
    displays the value of a counter, that the function ''ticker'' increments.
    If CTRL-C is struck, the 'terminate' function is called to handle the
    termination of the program. It stops timer, then restores the original
    process terminate vector and returns (to the default system Interrupt
    Service Routine).

        The timers on the 68901 MFP are controlled by a 2.4576 Mhz crystal.
    Using timer 'A' in the delay mode with a prescale set at 200 (ie. by
    setting the timer 'A' control register to 7) gives you a 12288 Hz counter
    (2457600/200). Using a count of 256 (ie. by loading the timer 'A' data
    register with 0) you get an interrupt frequency of 48 Hz (12288/256).  For
    other values for the control & data registers see the 68901 manual available
    for free by calling Motorola.

        The timer interrupt is handled by the function 'dispatcher'. This
    function calls a routine to increment the long counter, then clears Bit 5
    of the ISRA (In Service Register A), and then returns from the exception
    by doing an RTE.

Note:   In the ST the 68901 always operates in the Software End of Interrupt
    Mode (ie. bit 3 of the Vector Register VR is always set).  In this mode the
    ISR bit of the ISRA, the ISR bit correpsonding to the Timer A interrupt is
    bit 5, is automatically set when an interrupt occurs and the processor
    requests the interrupt vector.  As long as the bit is set, that interrupt
    and any other interrupts of lower priority cannot occur. Once the bit is
    cleared the same interrupt or any lower priority interrupts can once
    again occur. This is why it is important to clear bit 5 of the ISRA before
    performing the RTE. The address of the ISRA register is 0xfffA0F
*/

#define MyApp   0  /*  my application          */
#define Control 7  /*  divide by 200 prescale  */
#define Data    0  /*  Countdown from          */
                   /* 1 byte, 1, 2, ... 254, 255, 0 = 256 */
#define Off     0
```

```
#define MAXITER 100 /* Iterations                  */
#define MAXTICKS 10 /* Maximum timer count-down events  */

extern  dispatcher();   /* labels in in-line assembly must be declared. */
extern  set_timer();
extern  unset_timer();


long ticks = MAXTICKS;  /* local tick counter.                      */
long oldvector;         /* storage for old terminate vector.        */


/*
    This routine is called by the interrupt handler to increment the
    local tick counter.
*/
ticker()
{
    ticks++;
}


main()
{
    int count = 0;

    puts("Sample code demonstrating the use of TIMER A on the 68901 MFP");
    printf("Iterations                : %d\n", MAXITER);
    printf("Timer events per iteration: %d\n\n", MAXTICKS);

    set_timer();        /* turn on timer */
    /*
        Set the terminate vector so that the user can't leave without
        turning of the timer first.
    */
    set_terminate();

    /*
        Keep on ticking...
    */
    while(count < MAXITER) {
        if (ticks == MAXTICKS) {
            printf("count == %d\r", count++);
            fflush(stdout);
            ticks = 0L;
            }
        }

    unset_terminate();
    unset_timer();      /* turn off timer */
```

```
    puts("\nPress return...");
    getchar();
}


/*
    My terminate application function.
*/
terminate()
{
    /*
        Clear 68901 timer interrupt
    */
    unset_timer();

    /*
        Restore the old process terminate vector
    */
    Setexc(0x0102, oldvector);
}

/*
    Get the old terminate application vector and setup
    the local terminate function.
*/
set_terminate()
{
    long user_stack = Super(0L);

    oldvector = Setexc(0x0102, -1L);
    Setexc(0x0102, terminate);

    Super(user_stack);
}


unset_terminate()
{
    /*
        Restore the old process terminate vector
    */
    Supexec(Setexc(0x0102, oldvector));
}


/*
    This is the interrupt dispatcher routine.
*/
asm {
```

```
dispatcher:
            jsr     ticker      /* our function                           */

            bclr.b  #5,0xfffa0f /* Tell MFP the interrupt has been serviced */
            rte                 /* return from exception                  */
}


/*
    This function is called by the main() function to set up the
    application terminate function and the 68901 function timer.
*/
set_timer()
{
    register    char    *globals;


    /*
        Tell the timer chip to call the dispatcher routine for the interrupt.
    */
    Xbtimer(MyApp, Control, Data, dispatcher);
}


/*
    Turn off the timer and reset the terminate vector.
*/
unset_timer()
{
    /*
        Turn off the application timer.
    */
    Xbtimer(MyApp, Off, Off, NULL);
}
```

SEE ALSO

Mfpinit, Rsconf

18

Chapter 19

Line-A Graphics Kernal

Introduction

The Atari ST's ROM contains some low-level graphic drawing routines, called the line-A routines, which are named after their calling mechanism (the 68000 line-A emulation). The line-A routines provide a hardware independent interface for all graphic operations. The Atari ST's VDI (Virtual Device Interface) calls line-A routines to perform its actual drawing. However, due to the overhead associated with the VDI calling mechanism, drawing operations can be performed more quickly by calling line-A routines directly, rather than calling VDI routines which in turn call line-A routines.

19.1 Line-A Graphics Routines

In this section an explanation of the graphics sub-system of the Atari is discussed. It is suggested that the programmer have a solid understanding of GEM and VDI before delving into this section.

As mentioned previously, the low-level graphics routines make use of a special type of instruction on the 68000 called line-A emulation. The 68000 processor has no instructions whose upper four bits are 0xa. These unimplemented instructions have been defined by Atari to call graphic drawing routines.

The following line-A opcodes are defined on the ST:

0xA000	=	Initialize the graphics system
0xA001	=	Plot a point
0xA002	=	Get a value for a point
0xA003	=	Draw a line
0xA004	=	Draw a horizontal line
0xA005	=	Fill a rectangle
0xA006	=	Fill a polygon
0xA007	=	Bit Block Transfer
0xA008	=	Text Block Transfer
0xA009	=	Show Mouse Cursor
0xA00A	=	Hide Mouse Cursor
0xA00B	=	Change Mouse Form
0xA00C	=	Draw Sprite
0xA00D	=	Undraw Sprite
0xA00E	=	Copy Memory Form Definition Block (MFDB)
0xA00F	=	Flood Fill

19.2 Graphics Modes

When the Atari ST is initially turned on, a 32000 byte block of RAM is defined near the top of memory as the screen RAM. Screen RAM is the memory which is scanned by special video hardware to produce the screen display. Although this block of memory is contiguous, it is logically arranged into rows of bytes, each representing a scan line (row) on the display. The dots, or pixels (picture elements), on the screen reflect bit patterns in these rows of bytes.

19.2.1 High-resolution Mode

The high-resolution mode displays 640 pixels on each of 400 scan lines. Each pixel displayed on the screen represents a single bit, either on or off in a row, with each row using 80 bytes. If a bit is zero (0), that pixel is displayed as white, if it is one (1), it is displayed as black. The top-left pixel on the screen is the upper bit of the first byte of screen RAM.

19.2.2 Medium-resolution Mode

The medium-resolution mode displays 640 pixels on each of 200 scan lines. This mode divides screen RAM into two equally sized planes, displaying only one-half the number of scan lines of high resolution. The planes are actually alternate words (16 bit) in memory, such that even words comprise the top plane and odd words comprise the bottom plane. The video hardware overlays the two planes and uses the binary number formed by corresponding bits of the top and bottom planes as an index into a color table, which determines the actual pixel color displayed. Note that two planes allow indices to range from 0 – 3, yielding four possible colors.

19.2.3 Low-resolution Mode

The low-resolution mode displays 320 pixels on each of 200 scan lines. This mode divides screen RAM into four equally sized planes, displaying one-half the number of scan lines and one-half the number of pixels per scan line of high-resolution. As with medium-resolution mode, the planes are alternate words (16 bit) in memory, such that every fourth word from the base of screen RAM lies in a plane. The video hardware overlays the four planes and uses the binary number formed by corresponding bits of these planes as an index into a color table, which determines the actual pixel color displayed. Note that four planes allow indices to range from 0 – 15, yeilding sixteen possible colors.

19.3 Line-A Port

The line-A routines operate from a set of variables contained in the `lineaport` Data Structure. The table is initialized through the call to *a_init*. The port data structure is defined as follows:

```
typedef struct {
    /*
        Drawing Environment
    */
    int  vplanes;           /* Number of video planes */
    int  vwrap;             /* Number of bytes per video scan */
    int *contrl;            /* pointer to VDI contrl array */
    int *intin;             /* pointer to VDI intin array */
    int *ptsin;             /* pointer to VDI ptsin array */
    int *intout;            /* pointer to VDI intout array */
    int *ptsout;            /* pointer to VDI ptsout array */
    int  plane0;            /* color bit mask for plane 0 */
    int  plane1;            /* color bit mask for plane 1 */
    int  plane2;            /* color bit mask for plane 2 */
    int  plane3;            /* color bit mask for plane 3 */
    int  minusone;          /* -1 used in XOR mode */
    int  linemask;          /* VDI line style */
    int  writemode;         /* VDI write mode */
    int  x1, y1, x2, y2;    /* drawing rectangle */
    int *patptr;            /* pointer to current VDI fill patter */
```

19

```
    int  patmask;           /* size of fill pattern mask */
    int  planefill;         /* number of planes to fill (0 = 1 plane) */
    int  clipflag;          /* clipping flag (0 = no clipping) */
    int  xminclip, yminclip;    /* clipping rectangle */
    int  xmaxclip, ymaxclip;

    /*
        Font Information
    */
    textblock thetext;      /* Text Drawing Block                  */

    /*
        Miscellaneous Drawing Variables
    */
    int  copymode;          /* copy mode for raster operations */
    int (*seedabort)();     /* pointer to seed fill abort routine */
} lineaport;
```

19.4 Line-A Data Structures

The data structure below is used with the *a_bitblit* function for describing the bit block to move.

```
typedef struct {
    int   x;
    int   y;
    int  *base;
    int   offset;
    int   width;
    int   plane_offset;
} bitblock;

typedef struct {
    int   width;        /* width of bit block   */
    int   height;       /* height of bit block  */
    int   planecount;   /* number of planes     */
    int   ForeColor;
    int   BackColor;
    char  table[4];

    /*
        Bit blocks to Blit
    */
    bitblock source;
    bitblock destin;

    /*
        Pattern Information
```

```
    */
    int  *patbuf;
    int   pat_offset;
    int   pat_width;
    int   pat_plane_offset;
    int   pat_mask;

    /*
        Temp Work space
    */
    int   work[12];
} blitblock;
```

The data structure below is used with the *a_drawsprite* function for describing the image of the sprite. Note that this data structure is also used to define the mouse form in the function *a_transformmouse*.

```
typedef struct {
    int x;              /* x offset of hot spot */
    int y;              /* y offset of hot spot */
    int format;         /* 0 = Copy, 1 = XOR    */
    int forecolor;      /* background color     */
    int backcolor;      /* foreground color     */
    int image[32];      /* bit-image of sprite  */
} sprite;

/*
    Save area for area behind Sprite.  Needs to be
        4 * sizeof(Sprite) so that all four color
        planes can be saved.
*/
typedef sprite spriteBack[4];
```

The data structure below is used with the *a_textblit* function for describing the block of text that is moved.

```
typedef struct  {
    int       xdda;        /* drawing work variable             */
    int       ddainc;      /* drawing work variable             */
    int       scaledir;    /* drawing work variable             */
    int       mono;        /* monospaced font flag              */
    int       fontx;       /* character (x, y) in font def      */
    int       fonty;
    int       scrnx;       /* charcter (x, y) on screen         */
    int       scrny;
    int       charheight;  /* width of character                */
    int       charwidth;   /* height of character               */
    char     *fontdata;    /* pointer to font bit-image data    */
    int       fontwidth;   /* width of font form                */
```

19

```
    int       fontstyle;    /* font style                          */
    int       litemask;     /* mask for dehilited text             */
    int       skewmask;     /* mask for italics text               */
    int       boldmask;     /* mask for bold text                  */
    int       fsuper;       /* offset for superscript text         */
    int       fsub;         /* offset for subscript text           */
    int       scaleflag;    /* 0 = no scaling                      */
    int       textdir;      /* text orientation flag               */
    int       forecolor;    /* foreground text color               */
    int      *textefx;      /* pointer to start of text, special   */
                            /* effects buffer                      */
    int       scalebuf;     /* offset for scale buffer in textefx  */
    int       backcolor     /* background text color               */
} textblock;
```

The next data structure is a definition of the Atari Font header. This header gives the Atari drawing routines information about a font.

```
typedef struct _FontForm {
    int  fontid;         /* Font Identifier                */
    int  fontsize;       /* Font Size in points            */
    char fontname[32];   /* Font name                      */

    int  lowascii;       /* lowest displayable ASCII char  */
    int  highascii;      /* highest displayable ASCII char */

    /*
        Character drawing offsets (see vst_alignment())
    */
    int  top;            /* offset from baseline to top    */
    int  ascent;         /* offset from baseline to ascent */
    int  half;           /* offset from baseline to half   */
    int  descent;        /* offset from baseline to descent */
    int  bottom;         /* offset from baseline to bottom */

    int  largechar;      /* widest character in font       */
    int  largeboxchar;   /* widest character cell in font  */

    int  kern;           /* kerning offset                 */
    int  rightoffset;    /* right offset for italics       */

    /*
        Text Effects masks
    */
    int  boldmask;
    int  underlinemask;
    int  litemask;
    int  skewmask;

    struct {
```

19

```
        unsigned system    : 1;  /* is it a system font?   */
        unsigned horiz     : 1;  /* horiz offset table?    */
        unsigned swapbytes : 1;  /* integers are reversed? */
        unsigned monospace : 1;  /* is font monospace?     */
    } flags;

    int *horztable;      /* pointer to horizontal offset table */
    int *chartable;      /* pointer to character offset table  */
    int *fonttable;      /* pointer to font bit-image data */

    int formwidth;
    int formheight;

    struct _FontForm *nextfont;  /* pointer to next font def */
} fontform;
```

Example

```
#include <linea.h>
#include <osbind.h>

#define CONSOLE 2

#define WHITE   0
#define RED     1
#define GREEN   2
#define BLACK   3

int top    = 15;
int left   = 10;
int bottom = 195;
int right  = 630;

int x, y;
int color;

main()
{
    lineaport *myport;

    myport = a_init();

    myport -> plane0 = BLACK;
    myport -> plane1 = BLACK;

    a_hidemouse();

    drawbox();
```

19

```
    bounce();

    a_showmouse();
}

bounce()
{
    int mx, my, sx ,sy;
    sprite     thesprite;
    spriteback theback;

    mx = left + 10;
    my = top  + 10;
    sx = 1;
    sy = 1;

    makesprite(&thesprite);

    while(!(Bconstat(CONSOLE))) {
        mx += sx;
        my += sy;

        if ((mx < left) | (mx > right-16))
            sx *= -1;

        if ((my < top+2) | (my > bottom-16))
            sy *= -1;

        a_putpixel(mx, my, RED);
        a_drawsprite(mx, my, &thesprite, theback);
        Vsync();
        a_undrawsprite(theback);
    }
    Bconin(CONSOLE);
}

drawbox()
{
    a_line(left,  top,    right, top);
    a_line(right, top,    right, bottom);
    a_line(right, bottom, left,  bottom);
    a_line(left,  bottom, left,  top);
}

makesprite(thesprite)
    sprite *thesprite;
{
    int x;

    thesprite -> x        = 0;
```

```
    thesprite -> y         = 0;
    thesprite -> format    = 0;
    thesprite -> forecolor = WHITE;
    thesprite -> backcolor = BLACK;

    for (x=0; x<32; x+=2) {
        stuffbits(&thesprite -> image[x],   "1010101010101010");
        stuffbits(&thesprite -> image[x+1], "0101010101010101");
    }
}
```

NAME

a_bitblit — move a rectangular block of bits.

SYNOPSIS

```
a_bitblit(theblock)
    blitblock *theblock;
```

DESCRIPTION

a_bitblit copies blocks of screen bits from a source rectangle to a destination rectangle. The description of the block to move is communicated by the parameter **theblock**. The block is a pointer to the data structure **blitblock** which is defined in section 19.4.

SEE ALSO

vro_cpyfm

NAME

a_copyraster — Copy Raster Form

SYNOPSIS

```
#include <gemdefs.h>

a_copyraster(source, destin)
    MFDB      *source, *destin;
```

DESCRIPTION

a_copyraster performs a raster block move. The MFDB data structure is defined in the VDI Introduction (pg. 307).

NOTE

The blocks copied must be defined on word boundaries and it's width must be defined in words.

SEE ALSO

vro_cpyfm

NAME

a_drawsprite — Draw a grahics entity on the screen

SYNOPSIS

```
a_drawsprite(x, y, thesprite, thebackgnd)
    int         x, y;
    sprite     *thesprite;
    spriteback *thebackgnd;
```

DESCRIPTION

a_drawsprite copies the background of the screen into the area of memory defined by the parameter **thebackgnd**. The graphics entity **thesprite** is then drawn on the screen at the location (x, y).

NOTE

The **sprite** data structure is described in section 19.4.

SEE ALSO

a_undrawsprite

NAME

a_fillpoly — Draw a filled polygon

SYNOPSIS

```
a_fillpoly(vert, points, numpts)
    int  vert;
    int *points;
    int  numpts;
```

DESCRIPTION

a_fillpoly fills a scan line specified by the parameter **vert**. The scan line is bounded by the polygon defined by the parameters **points** and **numpts**. The horizontal line is drawn with the current fill attributes. The following fields of the **lineaport** data structure are used:

x1, **x2**, **y1**

plane0, **plane1**, **plane2**, **plane3**

writemode

patptr

patmask

planefill

clipflag

xminclip, **yminclip**, **xmaxclip**, **ymaxclip**

19

NOTE

The starting point of the polygon must also be defined as the ending point.

EXAMPLE

```
#include <osbind.h>
#include <linea.h>

int pts[4][2] = {
    320, 050,
    120, 150,
    520, 150,
    320, 050
};

main()
{
    lineaport *theport;
```

```
    int fillpat[4], y;

    stuffbits(&fillpat[0], "1100110011001100");
    stuffbits(&fillpat[1], "0110011001100110");
    stuffbits(&fillpat[2], "0011001100110011");
    stuffbits(&fillpat[3], "1001100110011001");

    theport = a_init();

    theport -> plane0 = 1;
    theport -> plane1 = 0;
    theport -> plane2 = 0;
    theport -> plane3 = 0;

    theport -> writemode = 2;
    theport -> patptr    = fillpat;
    theport -> planefill = 0;
    theport -> clipflag  = 0;

    for(y=50; y<150; y++)
        a_fillpoly(y, pts, 3);

    Cconin();
}
```

SEE ALSO

v_fillarea

NAME

a_fillrect — Fill a rectangle

SYNOPSIS

```
a_fillrect(x1, y1, x2, y2)
    int x1, y1;
    int x2, y2;
```

DESCRIPTION

a_fillrect fills the rectangle defined by $(x1, y1)$ and $(x2, y2)$ with the current fill area attributes. The following fields of the **lineaport** data structure are used:

> **x1**, **y1**, **x2**, **y2**
>
> **plane0**, **plane1**, **plane2**, **plane3**
>
> **writemode**
>
> **patptr**
>
> **patmask**
>
> **planefill**
>
> **clipflag**
>
> **xminclip**, **yminclip**, **xmaxclip**, **ymaxclip**

NOTE

The rectangle is filled from the top left corner to the bottom right hand corner.

SEE ALSO

vr_recfl

NAME

a_getpixel — get pixel value

SYNOPSIS

```
int a_getpixel(x, y)
    int x, y;
```

DESCRIPTION

a_getpixel returns the color value of the pixel at the position (x, y).

NOTE

The coordinates of the point are placed in the **ptsin[]** array.

SEE ALSO

v_get_pixel

NAME

a_hidemouse — hide the mouse cursor.

SYNOPSIS

```
a_hidemouse()
```

DESCRIPTION

The *a_hidemouse* function hides the mouse cursor. Note that the mouse cursor hide level is nested.

SEE ALSO

v_hide_c, *v_show_c*, *graf_mouse*

NAME

a_hline — draw a horizontal line.

SYNOPSIS

```
a_hline(x1, x2, y)
    int x1, x2, y;
```

DESCRIPTION

a_hline draws a horizontal line from the pixel position **x1** to **x2**. The line is drawn on the scan line defined by the parameter **y**. The line drawing function uses the following fields of the **lineaport** data structure:

x1, **y1**, **x2**, **y2**

plane0, **plane1**, **plane2**, **plane3**

linemask

writemode

minusone (used for XOR mode only)

NOTE

The line is always drawn from left to right and the line style mask is also applied from left to right. The line style mask is word aligned pattern for the horizontal lines, (i.e. any bit of the mask may be used at the left-most endpoint.)

SEE ALSO

v_pline

NAME

a_init — Initialize the Line-A drawing routines

SYNOPSIS

```
#include <linea.h>

lineaport *a_init()
```

DESCRIPTION

The *a_init* function initializes the drawing variables that are used by the Line-A drawing routines. The result of the function is a pointer to the `lineaport` data structure.

SEE ALSO

v_opnvwk

NAME

a_line — Draw a line

SYNOPSIS

```
a_line(x1, y1, x2, y2)
    int x1, y1;
    int x2, y2;
```

DESCRIPTION

a_line draws a line that connects the two points $(x1, y1)$ and $(x2, y2)$. The line drawing function uses the following fields of the `lineaport` data structure:

`x1, y1, x2, y2`

`plane0, plane1, plane2, plane3`

`linemask`

`writemode`

`minusone` (used for XOR mode only)

NOTE

The line is always drawn from left to right. The line style mask is also applied from left to right. The line style mask is a word aligned pattern.

SEE ALSO

a_hline, *v_pline*

NAME

a_putpixel — Plot a pixel point

SYNOPSIS

```
int a_putpixel(x, y, color)
    int x, y;
    int color;
```

DESCRIPTION

a_putpixel plots a pixel at the screen location (x, y). The point is set to the color index defined by the parameter **color** The return value of the function will be the value of the pixel at the point.

NOTE

The coordinates for the point are placed in the **ptsin[]** array. The result of the function is stored in **intin[0]**.

SEE ALSO

v_pline

19

NAME

a_showmouse — show the mouse cursor.

SYNOPSIS

```
a_showmouse()
```

DESCRIPTION

a_showmouse displays the mouse cursor.

NOTE

The level of display for the mouse is nested. This means that the number of calls to *a_showmouse()* should be balanced with the number of *a_hidemouse()* calls.

SEE ALSO

graf_mouse, *v_hide_c*, *v_show_c*

NAME

a_textblit — Copies a charcter using special effects.

SYNOPSIS

```
a_textblit(charblock)
    textblock *charblock;
```

DESCRIPTION

a_textblit performs a copy block operation of a character to the screen. The graphics text character copied is defined by the parameter **charblock**. The following fields of the **lineaport** data structure are used:

writemode

textfg, textbg

fontdata, fontwidth, fontstyle

srcx, srcy

destx, desty

charheight, charwidth

skewmask, boldmask

upoffset, downoffset

scaleflag, scale

xdda

txtdirect

mono

textefx, scalebuf

SEE ALSO

v_gtext

19

NAME

a_transformmouse — change the mouse form

SYNOPSIS

```
a_transformmouse(theform)
    sprite *theform;
```

DESCRIPTION

The *a_transformmouse* function changes the current form of the mouse cursor. The parameter `theform` is a pointer to a mouse form data structure described in section 19.4.

EXAMPLE

```
#include "linea.h"
#include <osbind.h>

#define CONSOLE 2

int top    = 15;
int left   = 10;
int bottom = 195;
int right  = 630;

int x,y;
int color;

main()
{
    lineaport *myport;
    mouse      themouse;

    myport = a_init();

    a_hidemouse();

    drawbox();

    makemouse(&themouse);

    a_transformmouse(&themouse);

    a_showmouse();

    while(!(Bconstat(CONSOLE)))
        ;
}
```

```
drawbox()
{
    a_line(left,  top,    right, top);
    a_line(right, top,    right, bottom);
    a_line(right, bottom, left,  bottom);
    a_line(left,  bottom, left,  top);
}

makemouse(thesprite)
    sprite *thesprite;
{
    int x;

    thesprite -> x         = 1;
    thesprite -> y         = 1;
    thesprite -> format    = 1;
    thesprite -> forecolor = 2;
    thesprite -> backcolor = 3;

    stuffbits(&thesprite -> image[0],   "0000000000000000");
    for (x=1; x<15; x++)
        stuffbits(&thesprite -> image[x],   "0111111111111110");
    stuffbits(&thesprite -> image[15],   "0000000000000000");

    stuffbits(&thesprite -> image[16],   "1111111111111111");
    for (x=17; x<32; x++)
        stuffbits(&thesprite -> image[x],   "1000000000000001");
    stuffbits(&thesprite -> image[32],   "1111111111111111");
}
```

SEE ALSO

graf_mouse

19

NAME

a_undrawsprite — restores screen behind sprite

SYNOPSIS

```
a_undrawsprite(thebackgnd)
    spriteback *thebackgnd;
```

DESCRIPTION

a_undrawsprite restores the screen to the contents pointed to by the parameter `thebackgnd`. The buffer `thebackgnd` is filled by the *a_drawsprite* function.

SEE ALSO

a_drawsprite

Chapter 20

Utility Routines

Introduction

Though not a part of the Atari ST ROM, the routines described in this chapter can be useful when writting a GEM application. Note that structure passing may be used for some routines which require point or rectangle coordinates. For example:

```
GRECT  rect1, rect2;

rect_equal(rect1.g_x, rect1.g_y, rect1.g_w, rect1.g_h,
           rect2.q_x, rect2.g_y, rect2.q_w, rect2.g_h);
```

...is equivalent (due to structure passing) to:

```
GRECT  rect1, rect2;

rect_equal(rect1, rect2);
```


Coordinate Functions

pt_set	set a point	rect_equal	are rects equal
pt_inrect	is point in rect	rect_offset	offset a rect
pt_sub	subtract two points	rect_set	set a rect
pt_2rect	two points to rect	rect_empty	is a rect empty
pt_equal	are points equal	rect_inset	inset a rect
pt_add	add two points	rect_union	union two rects
rect_intersect	intersect two rectangles	rect_set	set a rect

Object Tree Functions

change_item	change and redraw objects	clear_tree	clear and redraw objects
change_aux	change and redraw objects with given clipping		

Miscelaneous Functions

min	return min of two integers	stuffbits	value from binary string
max	return max of two integers	stuffhex	value from hex string

NAME

change_aux, change_item, clear_tree — set or clear object trees

SYNOPSIS

```
change_aux(tree, item, mask, value, cx, cy, cw, ch)
    OBJECT *tree;
    int item, mask, value, cx, cy, cw, ch;

change_item(tree, item, mask, value)
    OBJECT *tree;
    int item, mask, value;

clear_tree(tree, cx, cy, cw, ch)
    OBJECT *tree;
    int cx, cy, cw, ch;
```

DESCRIPTION

These routines recursively traverse object trees, beginning with `item`, and change the `ob_state` field of each `OBJECT` visited. The new value assigned to each `ob_state` field is given in `value`. The `mask` parameter specifies which bits of the source are to be preserved (set bits in the mask correspond to preserved bits). The objects are redrawn to reflect the new `ob_state`. The `ob_state` of an object determines how an object is displayed (i.e. NORMAL, SELECTED, CROSSED, etc.). See the Object Manager, section 16.8, for more information on object states.

change_aux change objects state with specified clipping. The `cx`, `cy`, `cw`, `ch` parameters determine the clipping rectangle used when the objects are redrawn.

change_item change objects state with clipping restricted to the size of the item being redrawn.

clear_tree set all object states to `NORMAL`. The `cx`, `cy`, `cw`, `ch` parameters determine the clipping rectangle used when the objects are redrawn.

NAME

max, min — return the maximum or minimum of two integers

SYNOPSIS

```
int max(a, b)
    int a, b;

int min(a, b)
    int a, b;
```

DESCRIPTION

Returns the maximum or minimum of two integers

NAME

pt_2rect — convert two points into a rectangle.

SYNOPSIS

```
pt_2rect(x1, y1, x2, y2, rect)
    int x1, y1, x2, y2;
    GRECT *rect;
```

DESCRIPTION

The two points are converted into rectangle coordinates. Point `(x1, y1)` is the top left of the rectangle and `(x2, y2)` is the bottom right. The result is placed in `rect`.

Alternate method:

```
pt_2rect(point1, point2, rect)
    GPOINT point1, point2;
    GRECT *rect;
```

NAME

pt_add — add two points

SYNOPSIS

```
pt_add(x, y, point)
    int x, y;
    GPOINT *point;
```

DESCRIPTION

Corresponding coordinates of (x, y) are added to and assigned **point**.

Alternate method:

```
pt_add(point1, point2)
    GPOINT point1;
    GPOINT *point2;
```

20

NAME

pt_equal — are two points equal

SYNOPSIS

```
int pt_equal(x1, y1, x2, y2)
    int x1, y1, x2, y2;
```

DESCRIPTION

Returns non-zero if corresponding point coordinates are both equal, else returns zero. Points are given by `(x1, y1)` and `(x2, y2)`.

Alternate method:

```
int pt_equal(point1, point2)
    GPOINT point1, point2;
```

NAME

pt_inrect — is a point in a rectangle

SYNOPSIS

```
int pt_inrect(px, py, rx, ry, rw, rh)
    int px, py, rx, ry, rw, rh;
```

DESCRIPTION

Returns non-zero if the point given by `px` and `py` lies inside the rectangle given by `rx`, `ry`, `rw`, `rh`. returns zero.

Alternate method:

```
int pt_inrect(point, rect)
    GPOINT point;
    GRECT rect;
```

NAME

pt_set — set a point

SYNOPSIS

```
int pt_set(point, x, y)
    GPOINT *point;
    int x, y;
```

DESCRIPTION

The coordinates (`x, y`) are copied into the `point`.

Alternate method:

```
int pt_set(dest_pt, src_pt)
    GPOINT *dest_pt;
    GPOINT src_pt;
```

NAME

pt_sub — subtract two points

SYNOPSIS

```
pt_sub(x, y, point)
    int x, y;
    GPOINT *rect;
```

DESCRIPTION

Corresponding coordinates of (x, y) are subtracted from and assigned point.

Alternate method:

```
pt_sub(point1, point2)
    GPOINT point1;
    GPOINT *point2;
```

NAME

rect_empty — is a rectangle empty

SYNOPSIS

```
int rect_empty(x, y, w, h)
    int x, y, w, h
```

DESCRIPTION

Returns non-zero if either the width or the height is less than or equal to zero.

Alternate method:

```
int rect_empty(rect)
    GRECT rect;
```

NAME

rect_equal — are two rectangles equal

SYNOPSIS

```
int rect_equal(x1, y1, w1, h1, x2, y2, w2, h2)
    int x1, y1, w1, h1, x2, y2, w2, h2;
```

DESCRIPTION

Returns non-zero if corresponding coordinates are all equal, else returns zero.

Alternate method:

```
int rect_equal(rect1, rect2)
    GRECT rect1, rect2;
```

NAME

rect_inset — change the size of a rectangle

SYNOPSIS

```
int rect_inset(rect, delta_x, delta_y)
    GRECT *rect;
    int delta_x, delta_y;
```

DESCRIPTION

The rectangle given in `rect` is made smaller or larger by `delta_x` and `delta_y`. Positive delta values make the rectangle smaller and negative values make the rectangle larger.

NAME

rect_intersect — produce the intersection of two rectangles

SYNOPSIS

```
rect_intersect(x1, y1, w1, h1, x2, y2, w2, h2, rect)
    int x1, y1, w1, h1, x2, y2, w2, h2;
    GRECT *rect;
```

DESCRIPTION

The intersection of the two rectangles given by x1, y1, w1, h1, and x2, y2, w2, h2 is placed in the rectangle given by `rect`. The function returns non-zero if the two rectangles actually intersect, else it returns zero.

Alternate method:

```
rect_intersect(rect1, rect2, rect3)
    GRECT rect1, rect2;
    GRECT *rect3;
```

NAME

rect_offset — offset the x and y of a rectangle

SYNOPSIS

```
rect_offset(rect, delta_x, delta_y)
    GRECT *rect;
    int delta_x, delta_y;
```

DESCRIPTION

The x and y coordinates of the rectangle given in `rect` are incremented by `delta_x` and `delta_y`, respectively.

NAME

rect_set — set a rectangle

SYNOPSIS

```
rect_set(rect, x, y, w, h)
    GRECT *rect;
    int x, y, w, h;
```

DESCRIPTION

The fields of the rectangle given in `rect` are set to x, y, w, h.

Alternate method:

```
rect_set(dest_rect, src_rect)
    GRECT *dest_rect;
    GRECT src_rect;
```

NAME

rect_union — produce the union of two rectangles

SYNOPSIS

```
rect_union(x1, y1, w1, h1, x2, y2, w2, h2, rect)
    int x1, y1, w1, h1, x2, y2, w2, h2;
    GRECT *rect;
```

DESCRIPTION

The union of the two rectangles given by **x1**, **y1**, **w1**, **h1**, and **x2**, **y2**, **w2**, **h2** is placed in the rectangle given by **rect**.

Alternate method:

```
rect_union(rect1, rect2, rect3)
    GRECT rect1, rect2;
    GRECT *rect3;
```

NAME

stuffbits — fill a data structure from a string of binary digits

SYNOPSIS

```
stuffbits(ptr, bits)
    char *ptr, *bits;
```

DESCRIPTION

The character string in `bits` is a string of ones and zeros. The string is translated into bits which are copied into the destination `ptr`. Each bit in the string is stuffed into the destination starting from the highest bit of the destination. Any character of the string which is not a zero or one is ignored, leaving the corresponding bit of the destination unaffected.

EXAMPLE

```
char des = 0x01;

/*
    des will be 0x55 after this...
*/
stuffbits(&des, "0101010 ");
```

20

NAME

stuffhex — fill a data structure from a string of hex digits

SYNOPSIS

```
stuffhex(ptr, hex)
    char *ptr, *hex;
```

DESCRIPTION

The character string in hex is a string of hex digits (0 – 9, A – F). The string is translated into bits which are copied into the destination ptr. Each hex digit in the string is stuffed into the destination starting from the highest four bits of the destination. Any character of the string which is not a hex digit is ignored, leaving the corresponding four bits of the destination unaffected.

EXAMPLE

```
long des = 0x000f0000;

/*
    des will be 0xffffffff after this...
*/
stuffhex(&des, "fff ffff");
```

Appendix A

File Formats

A.1 Laser Object File Format

A.out is the name of the format of object files produced by the C compiler. This object file format is same as that used by UNIX systems. The file has five sections: a header, the program TEXT, the program DATA, relocation information, a symbol table, and a string table (in that order). The TEXT segment contains the actual machine code for the program, while the DATA segment contains initialized variables. A segment for uninitialized variables, called the BSS segment, is set up at by the loader when the program is run.

Formats using the C structure definitions are:

```
/*  Header prepended to each object file.
*/
typedef struct {
    long       a_magic;    /* magic number 0x0107         */
    long       a_text;     /* size of text segment        */
    long       a_data;     /* size of initialized data    */
    long       a_bss;      /* size of uninitialized data  */
    long       a_syms;     /* size of symbol table        */
    long       a_entry;    /* entry point                 */
    long       a_trsize;   /* size of text relocation     */
    long       a_drsize;   /* size of data relocation     */
    } exec;

/*  Format of a relocation datum.
```

A

```
*/
typedef struct {
    long        r_address;  /* address which is relocated  */
    long        r_info;     /* r_symbolnum, r_pcrel,       */
                            /* r_length, r_extern.         */
    } relocation_info;


/*  Macros to access the r_info field
*/
#define  r_symbolnum(x) ((x>>8) & 0xffffffL)
#define  r_pcrel(x)     ((x>>7) & 0x1L)
#define  r_length(x)    ((x>>5) & 0x3L)
#define  r_extern(x)    ((x>>4) & 0x1L)
```

If r_extern is zero, then r_symbolnum is actually the N_TYPE (see below) for the relocation rather than an index into the symbol table.

```
/*  Format of a symbol table entry.
*/
typedef struct {
    char        *n_name;    /* string table index          */
    char        n_type;     /* type flag, i.e. N_TEXT etc  */
    char        n_other;    /* unused                      */
    char        n_desc;     /* currently not used          */
    long        n_value;    /* value of this symbol        */
    } nlist;


/*  Simple values for n_type.
*/
#define N_UNDF          0x0     /* undefined                   */
#define N_ABS           0x2     /* absolute                    */
#define N_TEXT          0x4     /* text                        */
#define N_DATA          0x6     /* data                        */
#define N_BSS           0x8     /* bss                         */
#define N_FN            0x1f    /* file name symbol            */

#define N_EXT           01      /* external bit, or'ed in      */
#define N_TYPE          0x1e    /* mask for all the type bits */
```

A

A.2 DRI Object File Format

In addition to Laser C's *a.out* format, Laser utility programs (the linker, archiver, disassembler, and symbol namer) support DRI's CP/M-68K object file format. These files are composed of up to four sections: A header, the TEXT and DATA segments, an optional symbol table, and optional relocation information.

The header, the first component in the file, specifies the size and starting address of the other components in the application which are listed below.

```
/*  CP/M-68K header
*/
typedef struct {
    int         c_magic;    /* magic number (0x601A)       */
    long        c_text;     /* size of text segment        */
    long        c_data;     /* size of initialized data    */
    long        c_bss;      /* size of uninitialized data  */
    long        c_syms;     /* size of symbol table        */
    long        c_entry;    /* entry point                 */
    long        c_res;      /* reserved, always zero       */
    int         c_reloc;    /* size of data relocation     */
    } header;

/*  Symbol table entry
*/
typedef struct {
    char        name[8];    /* Symbol name                 */
    int         type;       /* Type (i.e. DEFINED|TEXT_REL)*/
    long        value;      /* Symbol value                */
    } symbol;

/*  CP/M-68K values for symbol types
*/
#define DEFINED        0x8000 /* The symbol is defined       */
#define EQUATED        0x4000 /* The symbol is an equate     */
#define GLOBAL         0x2000 /* The symbol is global        */
#define EQU_REG        0x1000 /* The symbol is a register    */
#define EXTERNAL       0x0800 /* The reference is external   */
#define DAT_REL        0x0400 /* Data segment reference      */
#define TEX_REL        0x0200 /* Text segment reference      */
#define BSS_REL        0x0100 /* Bss segment reference       */
```

A

The above values may be OR'd together to indicate symbol type.

One word (16-bit) of relocation information exists for each word of TEXT and DATA. The type of relocation is indicated in bits 0-2 of the word. If the relocation is an external reference, the remaining bits (15-3) form an index into the symbol table, thus indicating the name of the external reference.

```
/*  CP/M-68K relocation word values (bits 0-2)
*/
#define NO_RELOC      0      /* No relocation necessary    */
#define DATA_BASED    1      /* Relocate from Data segment */
#define TEXT_BASED    2      /* Relocate from Text segment */
#define BSS_BASED     3      /* Relocate from Bss segment  */
#define UNDEF_SYMBOL  4      /* Symbolic reference         */
#define LONG_REF      5      /* Next relocation is long    */
#define PC_RELATIVE   6      /* Is a PC relative reference */
#define INSTRUCTION   7      /* Is an instruction          */
```

A.3 GEMDOS Application File Format

The file format output by the linker (GEMDOS) is identical to the DRI object file format excepting the relocation information. The GEMDOS loader will only relocate 32-bit references. GEMDOS relocation information consists of a long (32-bit) word, indicating the offset into the program of the first long word to be relocated, followed by a series of relocation bytes (8-bit). These bytes indicate the distance from the last offset relocated to the current offset to be relocated. If a relocation byte is equal to 254, the last offset is incremented, but no relocation is done. A relocation byte of zero means end-of-relocation-information.

A

Appendix B

System Globals

The addresses of the globals in this list of BIOS variables is guaranteed not to change with future releases of the Atari ST, so programs can rely on their locations.

etv_timer (long) 0x400 The System Timer interrupt vector (logical vector 0x100).

etv_critic (long) 0x404 Critical error handler vector (logical vector 0x101).

etv_term (long) 0x408 Process-terminate vector (logical vector 0x102).

etv_xtra (longs) 0x40c Space for logical vectors 0x103 through 0x107.

memvalid (long) 0x420 The magic number 0x752019F3, which (combined with memval2) validates memcntlr and indicates a successful coldstart.

memcntlr (char) 0x424 Memory controller configuration nibble (the low nibble). Some common values are:

Memory size	Value
128K	0
512K	4
256K (2 banks)	0
1MB (2 banks)	5

resvalid (long) 0x426 If resvalid contains the magic number 0x31415926 on system RESET, the system will jump through **resvector**.

resvector (long) 0x42a System RESET trap vector. Called only if resvalid has the correct magic number in it. The vector is called early during system initialization before any hardware registers are configured.

phystop (long) 0x42e Physical end of RAM. Contains a pointer to the first unusable byte (i.e. 0x80000 on a 512K machine).

_membot (long) 0x432 Bottom of available memory. The *Getmpb* BIOS function uses this value as the start of the TPA.

_memtop (long) 0x436 Top of available memory. The *Getmpb* BIOS function uses this value as the end of the TPA.

memval2 (long) 0x43a Contains the magic number 0x237698AA which (combined with memvalid) validates memcntlr and indicates a successful coldstart.

flock (int) 0x43e Locks usage of the DMA chip. A nonzero value ensures that the operating system does not alter the DMA chip registers during vertical retrace. This variable must be nonzero for the DMA bus to be used.

seekrate (int) 0x440 Default floppy disk seek rate. Bits zero and one have the following meaning:

Bits 0,1	Seek rate
00	6ms
01	12ms
10	2ms
11	3ms (default)

_timr_ms (int) 0x442 System timer calibration (in ms). Should be set to 20 since the system timer interrupt vector is called at 50hz. This variable is returned by the BIOS function *Tickcal*, and is passed on the stack to the timer interrupt vector.

B

_fverify (int) 0x444 Floppy disk verify flag. A nonzero value means all write operations to floppies are read-verified (default value). A zero value indicates no verification.

_bootdev (int) 0x446 Boot device number. An environment string is constructed from this variable by the BIOS before GEM desktop is loaded.

palmode (int) 0x448 A nonzero value indicates the PAL (50hz video) mode is in use. A zero value means the NTSC (60hz video) mode is being used.

defshiftmd (char) 0x44a Contains the resolution for the color monitor the system will use if it must change from monochrome mode to color mode.

sshiftmd (int) 0x44c Contains the current value for the shiftmd hardware register:

0	$320 \times 200 \times 4$ (low resolution color)
1	$640 \times 200 \times 2$ (medium resolution color)
2	$640 \times 400 \times 1$ (high resolution B/W)

_v_bas_ad (long) 0x44e Address of screen memory (32K, any resolution). Must be on a 512 byte boundary.

vblsem (int) 0x452 A semaphore used to ensure mutual exclusion in the vertical-blank interrupt handler. Should be 1 to allow vertical-blank processing.

nvbls (int) 0x454 Number of pointers that **_vblqueue** points to. Set to 8 on system RESET.

_vblqueue (long) 0x456 Pointer to a vector of pointers to vertical-retrace handlers to be executed at each vertical retrace interrupt.

colorptr (long) 0x45a Address of an array of 16 integers to be loaded into the hardware color palette during the next vertical retrace. The palette is not loaded if the value is 0L. A 0L is stored in **colorptr** after the palette is loaded.

screenpt (long) 0x45e New screen memory address which will be stored into **v_bas_ad** during the next vertical retrace. If screenpt contains 0L then the screen base will not be changed.

_vbclock (long) 0x462 Count of vertical-blank interrupts that have occurred since last RESET.

_frclock (long) 0x466 Number of vertical retrace interrupts that were processed (i.e. not blocked by **vblsem**)

hdv_init (long) 0x46a Address of hard disk initialization routine. 0L if unused.

swv_vec (long) 0x46e Address of routine to be executed when the monitor is physically changed from monochrome to color or vise-vera. Initially set to system RESET vector.

B

hdv_bpb (long) 0x472 Address of the routine that returns a hard disk's BIOS parameter block (BPB). Parameters and return value are the same as *Getbpb*. Contains 0L if unused.

hdv_rw (long) 0x476 Address of routine to read or write on hard disk. Works like the *Rwabs* BIOS function. Contains 0L if unused.

hdv_boot (long) 0x47a Address of routine to boot from hard disk. Contains 0L if unused.

hdv_mediach (long) 0x47e Address of routine that returns the hard disk's media change mode. Works like the *Mediach* BIOS function. Contains 0L if unused.

_cmdload (int) 0x482 A non-zero value means to attempt to execute the program `COMMAND.PRG` on the boot device. This value can be set by a boot sector so that an application can be loaded instead of GEM desktop.

conterm (char) 0x484 Contains the attribute bits for the console system:

Bit	Function
0	1 = enable bell when `^G` is written to `CON:`
1	1 = enable auto key-repeat
2	1 = enable audible key-click
3	1 = Return the current value of kbshift in bits 24 – 31 when a *Bconin* is called.

themd (long) 0x48e Points at the GEM DOS TPA limits. Filled in by the BIOS with a *Getmbp* call. The structure has the following format:

```
struct MD
  {
    struct MD   *m_link;  /* ->next MD must be 0L */
    long        m_start;  /* start of TPA */
    long        m_length; /* size of TPA in bytes */
    struct PD   *m_own;   /* ->MD's owner (0L) */
  };
```

The structure may not be changed after GEM DOS has been initialized.

savptr (long) 0x4a2 Pointer to register save area for BIOS functions.

_nflops (int) 0x4a6 Number of floppy disks actually attached to the system (0, 1, or 2).

B

`sav_context` `(long)` `0x4ae` Pointer to saved processor context when a catastrophic error occurs (like odd address trap or divide by zero).

`_bufl` `(2 longs)` `0x4b4` Two BCB (buffer control block) pointers. The first is to the sector BCB and the second to the FAT (file allocation table) and directory sectors BCB. A BCB has the following format:

```
struct BCB
  {
    struct BCB *b_link;/* next BCB */
    int     b_bufdrv;  /* drive#, or -1 */
    int     b_buftyp;  /* buffer type */
    int     b_bufrec;  /* record# cached here */
    int     b_dirty;   /* dirty flag */
    DMD     *b_dm;     /* ->Drive Media Descriptor */
    char    *b_bufr;   /* ->buffer itself */
  };
```

`_hz_200` `(long)` `0x4ba` Count of 200hz timer ticks. Divided by four to generate the 50hz system timer.

`the_env` `(char[4])` `0x4be` The default environment string (four NULL characters).

`_drvbits` `(long)` `0x4c4` Value returned by *Drvmap* BIOS function.

`_dskbufp` `(long)` `0x4c6` Address of a 1024 byte disk buffer in the systems global area. This buffer should not be used by interrupt handlers.

`_prt_cnt` `(int)` `0x4ee` Count of number of times the `ALT-HELP` key combination has been pressed. Initially -1, a value of 0 causes the screen dump routine being printing the screen. A non-zero value causes the dump routine to abort the print and reset this value to -1.

`_sysbase` `(long)` `0x4f2` Points to the base of TOS (in ROM or RAM).

`_shell_p` `(long)` `0x4f6` Address of some shell-specific data.

`end_os` `(long)` `0x4fa` Address of byte immediately after the last byte used by TOS. This is also the start of the TPA.

`exec_os` `(long)` `0x4fe` Address of the shell program. The shell is executed by the BIOS after system initialization if complete. This normally points at the first byte of the AES code.

B

B

Appendix C

DOS Error Codes

These error numbers are returned by some of the BIOS and GEMDOS routines. The error code is always in the low 16 bits of the return value so mask long values with `0xffff` before checking the error.

Code	Error	Description
0	OK	No error.
−1	ERROR	General error.
−2	DRIVE_NOT_READY	Device was not ready, was not attached, or has been busy for too long.
−3	UNKNOWN_CMD	Device didn't understand the command.
−4	CRC_ERROR	Soft read error.
−5	BAD_REQUEST	Device couldn't handle the command, although it understood it. Check command parameters.
−6	SEEK_ERROR	Drive couldn't perform the seek.
−7	UNKNOWN_MEDIA	Attempt to read un-formatted or foreign media. Usually caused by a trashed or zeroed boot block.
−8	SECTOR_NOT_FOUND	The requested sector could not be found.
−9	NO_PAPER	The printer is out of paper.
−10	WRITE_FAULT	A write operation failed.
−11	READ_FAULT	A read operation failed.
−12	GENERAL_MISHAP	Reserved for future errors.
−13	WRITE_PROTECT	Attempt to write onto write-protected or read-only media.

C

−14	MEDIA_CHANGE	The media has changed since the last write. The operation did not take place.
−15	UNKNOWN_DEVICE	The operation specified a device that the BIOS couldn't recognize.
−16	BAD_SECTORS	A format operation detected bad sectors.
−17	INSERT_DISK	Request to ask user to insert a disk.

GEMDOS error codes

−32	EINVFN	Invalid function number.
−33	EFILNF	File not found.
−34	EPTHNF	Path not found.
−35	ENHNDL	No file descriptors left (too many files are open).
−36	EACCDN	Access denied.
−37	EIHNDL	Invalid file descriptor.
−39	ENSMEM	Insufficient memory.
−40	EIMBA	Invalid memory block address.
−46	EDRIVE	Invalid drive specified.
−49	ENMFIL	No more files.
−64	ERANGE	Range error.
−65	EINTRN	Internal error.
−66	EPLFMT	Invalid program load format.
−67	EGSBF	Setblock failure due to growth restrictions.

C

Appendix D

Key Codes

The first two numbers are the high and low bytes returned by *evnt_keybd()* or *evnt_multi()* for each key on the keyboard.

03	00	Control	2	(NULL)
1E	01	Control	A	
30	02	Control	B	
2E	03	Control	C	
20	04	Control	D	
12	05	Control	E	
21	06	Control	F	
22	07	Control	G	
23	08	Control	H	
17	09	Control	I	
24	0A	Control	J	
25	0B	Control	K	
26	0C	Control	L	
32	0D	Control	M	
31	0E	Control	N	
18	0F	Control	O	
19	10	Control	P	
10	11	Control	Q	
13	12	Control	R	
1F	13	Control	S	
14	14	Control	T	
16	15	Control	U	
2F	16	Control	V	
11	17	Control	W	
2D	18	Control	X	
15	19	Control	Y	
2C	1A	Control	Z	
1A	1B	Control	[	
2B	1C	Control	\	
1B	1D	Control	]	
07	1E	Control	6	
0C	1F	Control	-	
39	20	Space		
02	21	!		
28	22	"		
04	23	#		
05	24	$		
06	25	%		
08	26	&		
28	27	'		
0A	28	(		
0B	29	)		
09	2A	*		
0D	2B	+		
33	2C	,		
0C	2D	-		
34	2E	.		
35	2F	/		
0B	30	0		
02	31	1		
03	32	2		
04	33	3		
05	34	4		
06	35	5		

07	36	6	
08	37	7	
09	38	8	
0A	39	9	
27	3A	:	
27	3B	;	
33	3C	<	
0D	3D	=	
34	3E	>	
35	3F	?	
03	40	@	
1E	41	A	
30	42	B	
2E	43	C	
20	44	D	
12	45	E	
21	46	F	
22	47	G	
23	48	H	
17	49	I	
24	4A	J	
25	4B	K	
26	4C	L	
32	4D	M	
31	4E	N	
18	4F	O	
19	50	P	
10	51	Q	
13	52	R	
1F	53	S	
14	54	T	
16	55	U	
2F	56	V	
11	57	W	
2D	58	X	
15	59	Y	
2C	5A	Z	
1A	5B	[	
2B	5C	\	
1B	5D	]	
07	5E	^	
0C	5F	_	Underscore
29	60	`	
1E	61	a	
30	62	b	
2E	63	c	
20	64	d	
12	65	e	
21	66	f	
22	67	g	
23	68	h	
17	69	i	
24	6A	j	
25	6B	k	
26	6C	l	
32	6D	m	
31	6E	n	
18	6F	o	
19	70	p	
10	71	q	
13	72	r	
1F	73	s	
14	74	t	
16	75	u	
2F	76	v	
11	77	w	
2D	78	x	
15	79	y	
2C	7A	z	
1A	7B	{	
2B	7C	|	
1B	7D	}	
29	7E	~	
53	7F	Rubout	(DEL)
81	00	Alt 0	
78	00	Alt 1	
79	00	Alt 2	
7A	00	Alt 3	
7B	00	Alt 4	
7C	00	Alt 5	
7D	00	Alt 6	
7E	00	Alt 7	
7F	00	Alt 8	
80	00	Alt 9	
1E	00	Alt A	
30	00	Alt B	

2E	00	Alt	C
20	00	Alt	D
12	00	Alt	E
21	00	Alt	F
22	00	Alt	G
23	00	Alt	H
17	00	Alt	I
24	00	Alt	J
25	00	Alt	K
26	00	Alt	L
32	00	Alt	M
31	00	Alt	N
18	00	Alt	O
19	00	Alt	P
10	00	Alt	Q
13	00	Alt	R
1F	00	Alt	S
14	00	Alt	T
16	00	Alt	U
2F	00	Alt	V
11	00	Alt	W
2D	00	Alt	X
15	00	Alt	Y
2C	00	Alt	Z
3B	00	F1	
3C	00	F2	
3D	00	F3	
3E	00	F4	
3F	00	F5	
40	00	F6	
41	00	F7	
42	00	F8	
43	00	F9	
44	00	F10	
54	00	F11	
55	00	F12	
56	00	F13	
57	00	F14	
58	00	F15	
59	00	F16	
5A	00	F17	
5B	00	F18	
5C	00	F19	

5D	00	F20
5E	00	F21
5F	00	F22
60	00	F23
61	00	F24 (Help)
62	00	F25 (Undo)
63	00	F26
64	00	F27
65	00	F28
66	00	F29
67	00	F30
68	00	F31
69	00	F32
6A	00	F33
6B	00	F34
6C	00	F35
6D	00	F36
6E	00	F37
6F	00	F38
70	00	F39
71	00	F40
73	00	Control Left Arrow
4D	00	Right Arrow
4D	36	Shift Right Arrow
74	00	Control Right Arrow
50	00	Down Arrow
50	32	Shift Down Arrow
48	00	Up Arrow
48	38	Shift Up Arrow
51	00	Page Down
51	33	Shift Page Down
76	00	Control Page Down
49	00	Page Up
49	39	Shift Page Up
84	00	Control Page Up
77	00	Control Home
47	00	Home
47	37	Shift Home
52	00	Insert
52	30	Shift Insert
53	00	Delete
53	2E	Shift Delete
72	00	Control Print Screen

37	2A	Print Screen
01	1B	Escape
0E	08	Backspace
82	00	Alt -
83	00	Alt =
1C	0D	Carriage Return
1C	0A	Control Carriage Return
4C	35	Shift Numeric Pad 5
4A	2B	Numeric Pad -
4E	2B	Numeric Pad +
0F	09	Tab
0F	00	Backtab
4B	00	Left Arrow
4B	34	Shift Left Arrow
4F	00	End
4F	31	Shift End
75	00	Control End
72	0D	Enter

Appendix E

Header Files

ctype.h

```
/*
    Character type tables

    NOTE
    If changes/additions are made, please ensure that the argument
    to the macro is referenced ONLY ONCE in the macro.
*/
#ifndef DL_CTYPE
#define DL_CTYPE

#define ctUCASE 0x01
#define ctLCASE 0x02
#define ctDIGIT 0x04
#define ctSPACE 0x08
#define ctPUNCT 0x10
#define ctCNTRL 0x20
#define ctHEXDG 0x40

extern char _ct_[];

#define isalpha(c)  (_ct_[(c) + 1] & (ctUCASE | ctLCASE))
#define isupper(c)  (_ct_[(c) + 1] & ctUCASE)
#define islower(c)  (_ct_[(c) + 1] & ctLCASE)
#define isdigit(c)  (_ct_[(c) + 1] & ctDIGIT)
#define isxdigit(c) (_ct_[(c) + 1] & (ctDIGIT | ctHEXDG))
#define isspace(c)  (_ct_[(c) + 1] & ctSPACE)
#define ispunct(c)  (_ct_[(c) + 1] & ctPUNCT)
#define isalnum(c)  (_ct_[(c) + 1] & (ctUCASE | ctLCASE | ctDIGIT))
#define isprint(c)  (_ct_[(c) + 1] & (ctPUNCT | ctUCASE | ctLCASE | ctDIGIT))
#define iscntrl(c)  (_ct_[(c) + 1] & ctCNTRL)
#define isascii(c)  ((unsigned) c <= 0x7F)
```

E

```
#define _toupper(c) ((c) - 'a' + 'A')
#define _tolower(c) ((c) - 'A' + 'a')
#define toascii(c)  ((c) & 0x7F)

#endif /* DL_CTYPE */
```

define.h

```
/****************************************************************************/
/*  DEFINE.H  Typical miscellaneous C definitions.                          */
/*      Copyright 1985 Atari Corp.                                          */
/****************************************************************************/
#ifndef DL_DEFINE
#define DL_DEFINE

#define NIL 0           /* Nil Pointer */

#define NO 0            /* "FALSE" */
#define YES 1           /* "TRUE" */

#define TRUE 1
#define FALSE 0

#define EOS '\0'        /* End of String marker */
#define EOF (-1)        /* End of File marker */
#define NEWLINE '\n'        /* Carriage Return */

#define FAILURE (-1)        /* Function failure return val */
#define SUCCESS (0)     /* Function success return val */
#define FOREVER for(;;)     /* Infinite loop declaration   */

#endif
```

fcntl.h

```
#ifndef O_RDONLY

#define O_RDONLY 0
#define O_WRONLY 1
#define O_RDWR 2
#define O_CREAT 4
#define O_APPEND 8
#define O_TRUNC 16
#define O_BINARY 8192  /* low 12 bits not used to comform with UNIX */

#endif
```

E

filefmt.h

```
/*  Laser C object file format definitions
```

```
*/

#define  LMAGIC     0x0107  /*  Laser C magic number              */

/*  Header prepended to each Laser object file.
*/
typedef struct {
    long                a_magic;    /* magic number                    */
    long                a_text;     /* size of text segment            */
    long                a_data;     /* size of initialized data        */
    long                a_bss;      /* size of uninitialized data      */
    long                a_syms;     /* size of symbol table            */
    long                a_entry;    /* entry point                     */
    long                a_trsize;   /* size of text relocation         */
    long                a_drsize;   /* size of data relocation         */
    }
    exec;

/*  Format of a relocation datum.
*/
typedef struct {
    long                r_address;      /* address which is relocated      */
    unsigned long       r_info;         /* r_symbolnum, r_pcrel, r_length, */
                                        /* r_extern.                       */
    }
    reloc_info;

/*  NOTE:  If r_extern is zero, then r_address is actually and N_TYPE,
        and no symbol entry is present for the relocation.
*/

/*  Fields for r_info (above)
*/
#define  r_symbolnum(x) ((x>>8) & 0xffffffL)
#define  r_pcrel(x)     ((x>>7) & 0x1L)
#define  r_length(x)    ((x>>5) & 0x3L)
#define  r_extern(x)    ((x>>4) & 0x1L)

/*  Symbol table entry
*/
typedef struct {
    char                    *n_name;    /* index into string table         */
    char                    n_type;     /* type flag, i.e. N_TEXT etc      */
    char                    n_other;    /* unused                          */
    char                    n_desc;     /* currently not used              */
    long                    n_value;    /* value of this sym               */
    }
    nlist;

/*  Values for n_type (above)
```

E

```
*/
#define  N_UNDF     0x0     /*  undefined                          */
#define  N_ABS      0x2     /*  absolute                           */
#define  N_TEXT     0x4     /*  text                               */
#define  N_DATA     0x6     /*  data                               */
#define  N_BSS      0x8     /*  bss                                */

#define  N_EXT      0x01    /*  external bit, or'ed in             */
#define  N_TYPE     0x1e    /*  mask for all the type bits         */

/*  Following the relocation information is a long word (32-bit)
    which tells the length of the string table which follows.
    The length includes the four bytes of the long word (it
    includes own size).  Strings are zero (0) terminated.
*/

/*  GEMDOS executable file format
*/

/*  CP/M-68K header
*/
typedef struct {
    int         c_magic;    /* magic number (0x601A)       */
    long        c_text;     /* size of text segment        */
    long        c_data;     /* size of initialized data    */
    long        c_bss;      /* size of uninitialized data  */
    long        c_syms;     /* size of symbol table        */
    long        c_entry;    /* entry point                 */
    long        c_res;      /* reserved, always zero       */
    int         c_reloc;    /* size of data relocation     */
    } header;

/*  Symbol table entry
*/
typedef struct {
    char        name[8];    /* Symbol name                 */
    int         type;       /* Type (i.e. DEFINED|TEXT_REL)*/
    long        value;      /* Symbol value                */
    } symbol;

/*  CP/M-68K values for symbol types
*/
#define DEFINED       0x8000 /* The symbol is defined     */
#define EQUATED       0x4000 /* The symbol is an equate   */
#define GLOBAL        0x2000 /* The symbol is global      */
#define EQU_REG       0x1000 /* The symbol is a register  */
#define EXTERNAL      0x0800 /* The reference is external */
#define DAT_REL       0x0400 /* Data segment reference    */
#define TEX_REL       0x0200 /* Text segment reference    */
#define BSS_REL       0x0100 /* Bss segment reference     */
```

```
/* The above values may be OR'd together to indicate
    symbol type.

    One word (16-bit) of relocation information exists for each
    word of TEXT and DATA.  The type of relocation is indicated
    in bits 0-2 of the word.  If the relocation is an external
    reference, the remaining bits (15-3) form an index into the
    symbol table, thus indicating the name of the external
    reference.
*/

/*  CP/M-68K relocation word values (bits 0-2)
*/
#define NO_RELOC      0      /* No relocation necessary    */
#define DATA_BASED    1      /* Relocate from Data segment */
#define TEXT_BASED    2      /* Relocate from Text segment */
#define BSS_BASED     3      /* Relocate from Bss segment  */
#define UNDEF_SYMBOL  4      /* Symbolic reference         */
#define LONG_REF      5      /* Next relocation is long    */
#define PC_RELATIVE   6      /* Is a PC relative reference */
#define INSTRUCTION   7      /* Is an instruction          */

/*  The file format output by the linker (GEMDOS) is identical to the
    DRI object file format excepting the relocation information.  The
    GEMDOS loader will only relocate 32-bit references.  GEMDOS
    relocation information consists of a long (32-bit) word, indicating
    the offset into the program of the first long word to be relocated,
    followed by a series of relocation bytes (8-bit).  These bytes
    indicate the distance from the last offset relocated to the current
    offset to be relocated.  If a relocation byte is equal to 254, the
    last offset is incremented, but no relocation is done.  A
    relocation byte of zero means end-of-relocation-information.
*/
```

gembind.h

```
/*************************************************************************/
/*  GEMBIND.H Do-It-Yourself GEM binding kit.          */
/*      Copyright 1985 Atari Corp.            */
/*                              */
/*  WARNING: This file is not supported!              */
/*       We reccomend you use the supplied binding libraries    */
/*************************************************************************/
#ifndef DL_GEMBIND
#define DL_GEMBIND

/*
   Global arrays references.  rpt  8-21-87
*/
```

E

```
extern crystal();   /*  Used by crys_if to do the actual AES trap call. */
extern ctrl_cnts(); /* actually a table of numbers */

extern int control[], global[];
extern int_in[], int_out[];
extern long addr_in[], addr_out[];

extern int gl_apid; /* application ID */

extern struct __c {
    int *cb_pcontrol;
    int *cb_pglobal;
    int *cb_pintin;
    int *cb_pintout;
    long *cb_padrin;
    long *cb_padrout;
} _c, *_ad_c;


                /* Application Manager          */
#define APPL_INIT 10
#define APPL_READ 11
#define APPL_WRITE 12
#define APPL_FIND 13
#define APPL_TPLAY 14
#define APPL_TRECORD 15
#define APPL_EXIT 19
                /* Event Manager            */
#define EVNT_KEYBD 20
#define EVNT_BUTTON 21
#define EVNT_MOUSE 22
#define EVNT_MESAG 23
#define EVNT_TIMER 24
#define EVNT_MULTI 25
#define EVNT_DCLICK 26
                /* Menu Manager             */
#define MENU_BAR 30
#define MENU_ICHECK 31
#define MENU_IENABLE 32
#define MENU_TNORMAL 33
#define MENU_TEXT 34
#define MENU_REGISTER 35
                /* Object Manager           */
#define OBJC_ADD 40
#define OBJC_DELETE 41
#define OBJC_DRAW 42
#define OBJC_FIND 43
#define OBJC_OFFSET 44
#define OBJC_ORDER 45
```

E

```
#define OBJC_EDIT 46
#define OBJC_CHANGE 47
                /* Form Manager             */
#define FORM_DO 50
#define FORM_DIAL 51
#define FORM_ALERT 52
#define FORM_ERROR 53
#define FORM_CENTER 54
#define FORM_KEYBD 55
#define FORM_BUTTON 56
                /* Graphics Manager         */
#define GRAF_RUBBOX 70
#define GRAF_DRAGBOX 71
#define GRAF_MBOX 72
#define GRAF_GROWBOX 73
#define GRAF_SHRINKBOX 74
#define GRAF_WATCHBOX 75
#define GRAF_SLIDEBOX 76
#define GRAF_HANDLE 77
#define GRAF_MOUSE 78
#define GRAF_MKSTATE 79
                /* Scrap Manager            */
#define SCRP_READ 80
#define SCRP_WRITE 81
                /* File Selector Manager        */
#define FSEL_INPUT 90
                /* Window Manager           */
#define WIND_CREATE 100
#define WIND_OPEN 101
#define WIND_CLOSE 102
#define WIND_DELETE 103
#define WIND_GET 104
#define WIND_SET 105
#define WIND_FIND 106
#define WIND_UPDATE 107
#define WIND_CALC 108
                /* Resource Manager         */
#define RSRC_LOAD 110
#define RSRC_FREE 111
#define RSRC_GADDR 112
#define RSRC_SADDR 113
#define RSRC_OBFIX 114
                /* Shell Manager            */
#define SHEL_READ 120
#define SHEL_WRITE 121
#define SHEL_GET 122
#define SHEL_PUT 123
#define SHEL_FIND 124
#define SHEL_ENVRN 125
                    /* max sizes for arrays     */
```

E

```
#define C_SIZE 4
#define G_SIZE 15
#define I_SIZE 16
#define O_SIZE 7
#define AI_SIZE 2
#define AO_SIZE 1
                    /* Crystal funtion op code  */
#define OP_CODE control[0]
#define IN_LEN control[1]
#define OUT_LEN control[2]
#define AIN_LEN control[3]

#define RET_CODE int_out[0]
                    /* application lib parameters   */
#define AP_VERSION global[0]
#define AP_COUNT global[1]
#define AP_ID global[2]
#define AP_LOPRIVATE global[3]
#define AP_HIPRIVATE global[4]
#define AP_LOPNAME global[5]        /* long ptr. to tree base in rsc*/
#define AP_HIPNAME global[6]
#define AP_LO1RESV global[7]        /* long address of memory alloc.*/
#define AP_HI1RESV global[8]
#define AP_LO2RESV global[9]        /* length of memory allocated   */
#define AP_HI2RESV global[10]       /* colors available on screen   */
#define AP_LO3RESV global[11]
#define AP_HI3RESV global[12]
#define AP_LO4RESV global[13]
#define AP_HI4RESV global[14]

#define AP_GLSIZE int_out[1]

#define AP_RWID int_in[0]
#define AP_LENGTH int_in[1]
#define AP_PBUFF addr_in[0]

#define AP_PNAME addr_in[0]

#define AP_TBUFFER addr_in[0]
#define AP_TLENGTH int_in[0]
#define AP_TSCALE int_in[1]

#define SCR_MGR 0x0001          /* pid of the screen manager*/

#define AP_MSG 0
#define MN_SELECTED 10

#define WM_REDRAW 20
#define WM_TOPPED 21
#define WM_CLOSED 22
```

E

```
#define WM_FULLED 23
#define WM_ARROWED 24
#define WM_HSLID 25
#define WM_VSLID 26
#define WM_SIZED 27
#define WM_MOVED 28
#define WM_NEWTOP 29

#define AC_OPEN 40
#define AC_CLOSE 41

#define CT_UPDATE 50
#define CT_MOVE 51
#define CT_NEWTOP 52
                        /* event lib parameters */
#define IN_FLAGS int_in[0]

#define B_CLICKS int_in[0]
#define B_MASK int_in[1]
#define B_STATE int_in[2]

#define MO_FLAGS int_in[0]
#define MO_X int_in[1]
#define MO_Y int_in[2]
#define MO_WIDTH int_in[3]
#define MO_HEIGHT int_in[4]

#define ME_PBUFF addr_in[0]

#define T_LOCOUNT int_in[0]
#define T_HICOUNT int_in[1]

#define MU_FLAGS int_in[0]
#define EV_MX int_out[1]
#define EV_MY int_out[2]
#define EV_MB int_out[3]
#define EV_KS int_out[4]
#define EV_KRET int_out[5]
#define EV_BRET int_out[6]


#define MB_CLICKS int_in[1]
#define MB_MASK int_in[2]
#define MB_STATE int_in[3]

#define MMO1_FLAGS int_in[4]
#define MMO1_X int_in[5]
#define MMO1_Y int_in[6]
#define MMO1_WIDTH int_in[7]
#define MMO1_HEIGHT int_in[8]
```

E

```
#define MMO2_FLAGS int_in[9]
#define MMO2_X int_in[10]
#define MMO2_Y int_in[11]
#define MMO2_WIDTH int_in[12]
#define MMO2_HEIGHT int_in[13]

#define MME_PBUFF addr_in[0]

#define MT_LOCOUNT int_in[14]
#define MT_HICOUNT int_in[15]
                        /* mu_flags     */
#define MU_KEYBD 0x0001
#define MU_BUTTON 0x0002
#define MU_M1 0x0004
#define MU_M2 0x0008
#define MU_MESAG 0x0010
#define MU_TIMER 0x0020

#define EV_DCRATE int_in[0]
#define EV_DCSETIT int_in[1]
                        /* menu library parameters */

#define MM_ITREE    addr_in[0]      /* ienable,icheck,tnorm */

#define MM_PSTR     addr_in[0]

#define MM_PTEXT    addr_in[1]

#define SHOW_IT     int_in[0]       /* bar          */

#define ITEM_NUM    int_in[0]       /* icheck, ienable  */
#define MM_PID      int_in[0]       /* register     */
#define CHECK_IT    int_in[1]       /* icheck       */
#define ENABLE_IT   int_in[1]       /* ienable      */

#define TITLE_NUM   int_in[0]       /* tnorm        */
#define NORMAL_IT   int_in[1]       /* tnormal      */

                    /* form library parameters  */
#define FM_FORM addr_in[0]
#define FM_START int_in[0]

#define FM_TYPE int_in[0]

#define FM_ERRNUM int_in[0]

#define FM_DEFBUT int_in[0]
#define FM_ASTRING addr_in[0]
```

E

```
#define FN_IX int_in[1]
#define FN_IY int_in[2]
#define FN_IW int_in[3]
#define FN_IH int_in[4]
#define FN_X int_in[5]
#define FN_Y int_in[6]
#define FN_W int_in[7]
#define FN_H int_in[8]

#define FN_XC int_out[1]
#define FN_YC int_out[2]
#define FN_WC int_out[3]
#define FN_HC int_out[4]

#define FND_START 0
#define FND_GROW 1
#define FND_SHRINK 2
#define FND_FINISH 3
                    /* object library parameters    */

#define OB_TREE addr_in[0]      /* all ob procedures        */

#define OB_DELOB int_in[0]      /* ob_delete            */

#define OB_DRAWOB int_in[0]     /* ob_draw, ob_change       */
#define OB_DEPTH int_in[1]
#define OB_XCLIP int_in[2]
#define OB_YCLIP int_in[3]
#define OB_WCLIP int_in[4]
#define OB_HCLIP int_in[5]

#define OB_STARTOB int_in[0]        /* ob_find          */
#define OB_MX int_in[2]
#define OB_MY int_in[3]

#define OB_PARENT int_in[0]     /* ob_add           */
#define OB_CHILD int_in[1]
#define OB_OBJ int_in[0]        /* ob_offset, ob_order      */
#define OB_XOFF int_out[1]
#define OB_YOFF int_out[2]
#define OB_NEWPOS int_in[1]     /* ob_order         */

                    /* ob_edit          */
#define OB_CHAR int_in[1]
#define OB_IDX int_in[2]
#define OB_KIND int_in[3]
#define OB_ODX int_out[1]

#define OB_NEWSTATE int_in[6]       /* ob_change            */
#define OB_REDRAW int_in[7]
```

E

```
                    /* graphics library parameters  */
#define GR_I1 int_in[0]
#define GR_I2 int_in[1]
#define GR_I3 int_in[2]
#define GR_I4 int_in[3]
#define GR_I5 int_in[4]
#define GR_I6 int_in[5]
#define GR_I7 int_in[6]
#define GR_I8 int_in[7]

#define GR_O1 int_out[1]
#define GR_O2 int_out[2]

#define GR_TREE addr_in[0]
#define GR_PARENT int_in[0]
#define GR_OBJ int_in[1]
#define GR_INSTATE int_in[2]
#define GR_OUTSTATE int_in[3]

#define GR_ISVERT int_in[2]

#define M_OFF 256
#define M_ON 257

#define GR_MNUMBER int_in[0]
#define GR_MADDR addr_in[0]

#define GR_WCHAR int_out[1]
#define GR_HCHAR int_out[2]
#define GR_WBOX int_out[3]
#define GR_HBOX int_out[4]

#define GR_MX int_out[1]
#define GR_MY int_out[2]
#define GR_MSTATE int_out[3]
#define GR_KSTATE int_out[4]
                    /* scrap library parameters */
#define SC_PATH addr_in[0]
                    /* file selector library parms  */

#define FS_IPATH addr_in[0]
#define FS_ISEL addr_in[1]

#define FS_BUTTON int_out[1]
                    /* window library parameters    */
#define XFULL 0
#define YFULL gl_hbox
#define WFULL gl_width
#define HFULL (gl_height - gl_hbox)
```

E

```
#define NAME 0x0001
#define CLOSER 0x0002
#define FULLER 0x0004
#define MOVER 0x0008
#define INFO 0x0010
#define SIZER 0x0020
#define UPARROW 0x0040
#define DNARROW 0x0080
#define VSLIDE 0x0100
#define LFARROW 0x0200
#define RTARROW 0x0400
#define HSLIDE 0x0800

#define WF_KIND 1
#define WF_NAME 2
#define WF_INFO 3
#define WF_WXYWH 4
#define WF_CXYWH 5
#define WF_PXYWH 6
#define WF_FXYWH 7
#define WF_HSLIDE 8
#define WF_VSLIDE 9
#define WF_TOP 10
#define WF_FIRSTXYWH 11
#define WF_NEXTXYWH 12
#define WF_IGNORE 13
#define WF_NEWDESK 14
#define WF_HSLSIZ 15
#define WF_VSLSIZ 16
                                        /* arrow message    */
#define WA_UPPAGE 0
#define WA_DNPAGE 1
#define WA_UPLINE 2
#define WA_DNLINE 3
#define WA_LFPAGE 4
#define WA_RTPAGE 5
#define WA_LFLINE 6
#define WA_RTLINE 7
                                        /* wm_create        */
#define WM_KIND int_in[0]
                                        /* wm_open, close, del  */
#define WM_HANDLE int_in[0]
                                        /* wm_open, wm_create   */
#define WM_WX int_in[1]
#define WM_WY int_in[2]
#define WM_WW int_in[3]
#define WM_WH int_in[4]
                                        /* wm_find      */
#define WM_MX int_in[0]
#define WM_MY int_in[1]
```

E

```
                        /* wm_calc      */
#define WC_BORDER 0
#define WC_WORK 1
#define WM_WCTYPE int_in[0]
#define WM_WCKIND int_in[1]
#define WM_WCIX int_in[2]
#define WM_WCIY int_in[3]
#define WM_WCIW int_in[4]
#define WM_WCIH int_in[5]
#define WM_WCOX int_out[1]
#define WM_WCOY int_out[2]
#define WM_WCOW int_out[3]
#define WM_WCOH int_out[4]
                        /* wm_update        */
#define WM_BEGUP int_in[0]


#define WM_WFIELD int_in[1]

#define WM_IPRIVATE int_in[2]

#define WM_IKIND int_in[2]
                        /* for name and info    */
#define WM_IOTITLE addr_in[0]

#define WM_IX int_in[2]
#define WM_IY int_in[3]
#define WM_IW int_in[4]
#define WM_IH int_in[5]

#define WM_OX int_out[1]
#define WM_OY int_out[2]
#define WM_OW int_out[3]
#define WM_OH int_out[4]

#define WM_ISLIDE int_in[2]

#define WM_IRECTNUM int_in[6]
                    /* resource library parameters  */

#define RS_PFNAME addr_in[0]        /* rs_init,             */
#define RS_TYPE int_in[0]
#define RS_INDEX int_in[1]
#define RS_INADDR addr_in[0]
#define RS_OUTADDR addr_out[0]

#define RS_TREE addr_in[0]
#define RS_OBJ int_in[0]

#define R_TREE 0
```

E

```
#define R_OBJECT 1
#define R_TEDINFO 2
#define R_ICONBLK 3
#define R_BITBLK 4
#define R_STRING 5
#define R_IMAGEDATA 6
#define R_OBSPEC 7
#define R_TEPTEXT 8     /* sub ptrs in TEDINFO  */
#define R_TEPTMPLT 9
#define R_TEPVALID 10
#define R_IBPMASK 11        /* sub ptrs in ICONBLK  */
#define R_IBPDATA 12
#define R_IBPTEXT 13
#define R_BIPDATA 14        /* sub ptrs in BITBLK   */
#define R_FRSTR 15      /* gets addr of ptr to free strings */
#define R_FRING 16      /* gets addr of ptr to free images  */


                /* shell library parameters */
#define SH_DOEX int_in[0]
#define SH_ISGR int_in[1]
#define SH_ISCR int_in[2]
#define SH_PCMD addr_in[0]
#define SH_PTAIL addr_in[1]

#define SH_PDATA addr_in[0]
#define SH_PBUFFER addr_in[0]

#define SH_LEN int_in[0]

#define SH_PATH addr_in[0]
#define SH_SRCH addr_in[1]

#endif DL_GEMBIND
```

gemdefs.h

```
/*************************************************************************/
/*  GEMDEFS.H Common GEM definitions and miscellaneous structures.      */
/*      Copyright 1985 Atari Corp.                                       */
/*************************************************************************/
#ifndef DL_GEMDEFS
#define DL_GEMDEFS

/*  EVENT Manager Definitions                           */
                        /* multi flags      */
#define MU_KEYBD  0x0001
#define MU_BUTTON 0x0002
#define MU_M1     0x0004
#define MU_M2     0x0008
#define MU_MESAG  0x0010
```

E

```
#define MU_TIMER  0x0020
                        /* keyboard states  */
#define K_RSHIFT 0x0001
#define K_LSHIFT 0x0002
#define K_CTRL   0x0004
#define K_ALT    0x0008
                        /* message values   */
#define MN_SELECTED 10
#define WM_REDRAW   20
#define WM_TOPPED   21
#define WM_CLOSED   22
#define WM_FULLED   23
#define WM_ARROWED  24
#define WM_HSLID    25
#define WM_VSLID    26
#define WM_SIZED    27
#define WM_MOVED    28
#define WM_NEWTOP   29
#define AC_OPEN     40
#define AC_CLOSE    41

/*  FORM Manager Definitions                    */
                        /* Form flags       */
#define FMD_START  0
#define FMD_GROW   1
#define FMD_SHRINK 2
#define FMD_FINISH 3

/*  RESOURCE Manager Definitions                    */
                        /* data structure types */
#define R_TREE      0
#define R_OBJECT    1
#define R_TEDINFO   2
#define R_ICONBLK   3
#define R_BITBLK    4
#define R_STRING    5       /* gets pointer to free strings */
#define R_IMAGEDATA 6       /* gets pointer to free images  */
#define R_OBSPEC    7
#define R_TEPTEXT   8       /* sub ptrs in TEDINFO  */
#define R_TEPTMPLT  9
#define R_TEPVALID 10
#define R_IBPMASK  11       /* sub ptrs in ICONBLK  */
#define R_IBPDATA  12
#define R_IBPTEXT  13
#define R_BIPDATA  14       /* sub ptrs in BITBLK   */
#define R_FRSTR    15       /* gets addr of ptr to free strings */
#define R_FRIMG    16       /* gets addr of ptr to free images  */

                        /* used in RSCREATE.C   */
typedef struct rshdr
```

E

```
{
    int     rsh_vrsn;
    int     rsh_object;
    int     rsh_tedinfo;
    int     rsh_iconblk;    /* list of ICONBLKS     */
    int     rsh_bitblk;
    int     rsh_frstr;
    int     rsh_string;
    int     rsh_imdata; /* image data           */
    int     rsh_frimg;
    int     rsh_trindex;
    int     rsh_nobs;   /* counts of various structs    */
    int     rsh_ntree;
    int     rsh_nted;
    int     rsh_nib;
    int     rsh_nbb;
    int     rsh_nstring;
    int     rsh_nimages;
    int     rsh_rssize; /* total bytes in resource  */
} RSHDR;
#define F_ATTR  0           /* file attr for dos_create */

/*  WINDOW Manager Definitions.                 */
                            /* Window Attributes    */
#define NAME    0x0001
#define CLOSER  0x0002
#define FULLER  0x0004
#define MOVER   0x0008
#define INFO    0x0010
#define SIZER   0x0020
#define UPARROW 0x0040
#define DNARROW 0x0080
#define VSLIDE  0x0100
#define LFARROW 0x0200
#define RTARROW 0x0400
#define HSLIDE  0x0800
                            /* wind_create flags    */
#define WC_BORDER 0
#define WC_WORK   1
                            /* wind_get flags   */
#define WF_KIND     1
#define WF_NAME     2
#define WF_INFO     3
#define WF_WORKXYWH 4
#define WF_CURRXYWH 5
#define WF_PREVXYWH 6
#define WF_FULLXYWH 7
#define WF_HSLIDE   8
#define WF_VSLIDE   9
#define WF_TOP          10
```

```
#define WF_FIRSTXYWH    11
#define WF_NEXTXYWH     12
#define WF_RESVD    13
#define WF_NEWDESK  14
#define WF_HSLSIZE      15
#define WF_VSLSIZE      16
#define WF_SCREEN       17
                        /* update flags     */
#define END_UPDATE 0
#define BEG_UPDATE 1
#define END_MCTRL  2
#define BEG_MCTRL  3

/*  GRAPHICS Manager Definitions                       */
                        /* Mouse Forms      */
#define ARROW       0
#define TEXT_CRSR   1
#define HOURGLASS   2
#define POINT_HAND  3
#define FLAT_HAND   4
#define THIN_CROSS  5
#define THICK_CROSS 6
#define OUTLN_CROSS 7
#define USER_DEF   255
#define M_OFF      256
#define M_ON       257

/*  MISCELLANEOUS Structures                       */
                    /* Memory Form Definition Block */
typedef struct fdbstr
{
    long    fd_addr;
    int     fd_w;
    int     fd_h;
    int     fd_wdwidth;
    int     fd_stand;
    int     fd_nplanes;
    int     fd_r1;
    int     fd_r2;
    int     fd_r3;
} MFDB;
                    /* Mouse Form Definition Block */
typedef struct mfstr
{
    int mf_xhot;
    int mf_yhot;
    int mf_nplanes;
    int mf_fg;
    int mf_bg;
    int mf_mask[16];
```

E

```
    int mf_data[16];
} MFORM;

#endif DL_GEMDEFS
```

linea.h

```
#ifndef Lna_INIT

#define Lna_INIT        0xa000  /* Initialize Line-A data structures    */
#define Lna_PUTPIXEL    0xa001  /* Put Pixel onto graphics screen       */
#define Lna_GETPIXEL    0xa002  /* Get Pixel value on graphics screen   */
#define Lna_LINE        0xa003  /* Draw a Line                          */
#define Lna_HLINE       0xa004  /* Draw a Horizontal Line               */
#define Lna_FILLRECT    0xa005  /* Draw a filled box (rectangle)        */
#define Lna_FILLPOLY    0xa006  /* Draw a line polygon and fill it      */
#define Lna_BITBLIT     0xa007  /* Bit Block Transfer                   */
#define Lna_TEXTBLIT    0xa008  /* Text Block Transfer                  */
#define Lna_SHOWMOUSE   0xa009  /* Show Mouse Cursor                    */
#define Lna_HIDEMOUSE   0xa00a  /* Hide Mouse Cursor                    */
#define Lna_NEWMOUSE    0xa00b  /* Change Mouse form                    */
#define Lna_UNSPRITE    0xa00c  /* Undraw sprite                        */
#define Lna_DRAWSPRITE  0xa00d  /* Draw sprite                          */
#define Lna_COPYRASTER  0xa00e  /* Copy Raster Form                     */
#define Lna_SEEDFILL    0xa00f  /* Do Seed fill on polygon              */


/*
    Miscellaneous Data Structure
*/
typedef struct {
    int x, y;
} point;


typedef struct {
    int top;
    int left;
    int bottom;
    int right;
} rect;


/*
    Font Header Data Structure
*/
typedef struct _fontform {
    int  fontid;        /* Font Identifier              */
    int  fontsize;      /* Font Size in points          */
    char fontname[32];  /* Font name                    */
```

E

```
    int  lowascii;       /* lowest displayable ASCII char   */
    int  highascii;      /* highest displayable ASCII char  */

    /*
        Character drawing offsets (see vst_alignment())
    */
    int  top;            /* offset from baseline to top     */
    int  ascent;         /* offset from baseline to ascent  */
    int  half;           /* offset from baseline to half    */
    int  descent;        /* offset from baseline to descent */
    int  bottom;         /* offset from baseline to bottom  */

    int  largechar;      /* widest character in font        */
    int  largeboxchar;   /* widest character cell in font   */

    int  kern;           /* kerning offset                  */
    int  rightoffset;    /* right offset for italics        */

    /*
        Text Effects masks
    */
    int  boldmask;
    int  underlinemask;
    int  litemask;
    int  skewmask;

    struct {
        unsigned system    : 1;  /* is it a system font?   */
        unsigned horiz     : 1;  /* horiz offset table?    */
        unsigned swapbytes : 1;  /* integers are reversed? */
        unsigned monospace : 1;  /* is font monospace?     */
    } flags;

    int *horztable;      /* pointer to horizontal offset table */
    int *chartable;      /* pointer to character offset table  */
    int *fonttable;      /* pointer to font bit-image data */

    int formwidth;
    int formheight;

    struct _fontform *nextfont;  /* pointer to next font def */
} fontform;


/*
    Text Data Structure
*/
typedef struct  {
    int      xdda;          /* drawing work variable                */
```

E

```
    int      ddainc;        /* drawing work variable          */
    int      scaledir;      /* drawing work variable          */
    int      mono;          /* monospaced font flag           */
    int      fontx;         /* character (x, y) in font def   */
    int      fonty;
    int      scrnx;         /* charcter (x, y) on screen      */
    int      scrny;
    int      charheight;    /* width of character             */
    int      charwidth;     /* height of character            */
    char    *fontdata;      /* pointer to font bit-image data */
    int      fontwidth;     /* width of font form             */
    int      fontstyle;     /* font style                     */
    int      litemask;      /* mask for dehilited text        */
    int      skewmask;      /* mask for italics text          */
    int      boldmask;      /* mask for bold text             */
    int      fsuper;        /* offset for superscript text    */
    int      fsub;          /* offset for subscript text      */
    int      scaleflag;     /* 0 = no scaling                 */
    int      textdir;       /* text orientation flag          */
    int      forecolor;     /* foreground text color          */
    int      textefx;       /* pointer to start of text special */
                            /* effects buffer                 */
    int      scalebuf;      /* offset for scale buffer in textefx */
    int      backcolor      /* background text color          */
} textblock;


typedef struct {
    /*
        Drawing Environment
    */
    int  vplanes;           /* Number of video planes */
    int  vwrap;             /* Number of bytes per video scan */
    int *cntrl;             /* pointer to VDI contrl array */
    int *intin;             /* pointer to VDI intin array */
    int *ptsin;             /* pointer to VDI ptsin array */
    int *intout;            /* pointer to VDI intout array */
    int *ptsout;            /* pointer to VDI ptsout array */
    int  plane0;            /* color bit mask for plane 0 */
    int  plane1;            /* color bit mask for plane 1 */
    int  plane2;            /* color bit mask for plane 2 */
    int  plane3;            /* color bit mask for plane 3 */
    int  minusone;          /* -1 used in XOR mode */
    int  linemask;          /* VDI line style */
    int  writemode;         /* VDI write mode */
    int  x1, y1, x2, y2;    /* drawing rectangle */
    int *patptr;            /* pointer to current VDI fill patter */
    int  patmask;           /* size of fill pattern mask */
    int  planefill;         /* number of planes to fill (0 = 1 plane) */
    int  clipflag;          /* clipping flag (0 = no clipping) */
```

E

```
    int  xminclip, yminclip;    /* clipping rectangle */
    int  xmaxclip, ymaxclip;

    /*
        Font Information
    */
    textblock thetext;      /* Text Drawing Block                  */

    /*
        Miscellaneous Drawing Variables
    */
    int  copymode;          /* copy mode for raster operations */
    int (*seedabort)();     /* pointer to seed fill abort routine */
} lineaport;


typedef struct {
    rect source;
    rect destin;
} copyblock;


typedef struct {
    int   x;
    int   y;
    int  *base;
    int   offset;
    int   width;
    int   plane_offset;
} bitblock;


typedef struct {
    int   width;        /* width of bit block   */
    int   height;       /* height of bit block  */
    int   planecount;   /* number of planes     */
    int   ForeColor;
    int   BackColor;
    char  table[4];

    /*
        Bit blocks to Blit
    */
    bitblock source;
    bitblock destin;

    /*
        Pattern Information
    */
    int  *patbuf;
```

E

```
    int   pat_offset;
    int   pat_width;
    int   pat_plane_offset;
    int   pat_mask;

    /*
        Temp Work space
    */
    int   work[12];
} blitblock;


typedef struct {
    int x;              /* x offset of hot spot */
    int y;              /* y offset of hot spot */
    int format;         /* 0 = Copy, 1 = XOR    */
    int forecolor;      /* background color     */
    int backcolor;      /* foreground color     */
    int image[32];      /* bit-image of sprite  */
} sprite;
typedef sprite mouse;

/*
    Save area for area behind Sprite.  Needs to be
        4 * sizeof(Sprite) so that all four color
        planes can be saved.
*/
typedef sprite spriteback[4];

extern lineaport *a_init();

/*
    Used by Line-A routines
        _lnaport  == pointer to line-a variables.
        _fonthdrs == pointer to three pointers to system font headers
*/
extern lineaport *_lnaport;
extern fontform  **_fonthdrs;

#endif
```

math.h

```
#ifndef DL_MATHSTUFF
#define DL_MATHSTUFF

extern double dc_e; /* e    */
extern double dcpi; /* pi   */
extern double dcph; /* pi/2 */
extern double dcpq; /* pi/4 */
```

E

```
extern double dcln2;    /* ln 2 */
extern double dcin; /* infimum  */
extern double dcsu; /* supremum */
extern double dchf; /* 0.5   */
extern double dc1;  /* 1.0   */
extern double dc1h; /* 1.5   */
extern double dc10; /* 10.0 */

extern double sin();
extern double cos();
extern double tan();
extern double asin();
extern double acos();
extern double atan();
extern double exp();
extern double exp_();
extern double exp2();
extern double exp_2();
extern double log();
extern double log2();
extern double mulpower2();
extern double powerd();
extern double poweri();
extern double power10();
extern double sqr();
extern double sqrt();
extern double dabs();
extern double dint();
extern double drand();
extern double fac();
extern double lngamma();
extern double matinv();

#define INF dcin
#define SUP dcsu

#endif
```

obdefs.h

```
#ifndef DL_OBDEFS
#define DL_OBDEFS

#define ROOT 0

#define MAX_LEN 81      /* max string length */

#define MAX_DEPTH 8     /* max depth of search or draw */

#define IP_HOLLOW 0     /* inside patterns */
```

E

```
#define IP_1PATT 1
#define IP_2PATT 2
#define IP_3PATT 3
#define IP_4PATT 4
#define IP_5PATT 5
#define IP_6PATT 6
#define IP_SOLID 7

#define MD_REPLACE 1    /* gsx modes */
#define MD_TRANS   2
#define MD_XOR     3
#define MD_ERASE   4

#define ALL_WHITE  0    /* bit blt rules */
#define S_AND_D    1
#define S_AND_NOTD 2
#define S_ONLY     3
#define NOTS_AND_D 4
#define D_ONLY     5
#define S_XOR_D    6
#define S_OR_D     7
#define NOT_SORD   8
#define NOT_SXORD  9
#define D_INVERT  10
#define NOT_D     11
#define S_OR_NOTD 12
#define NOTS_OR_D 13
#define NOT_SANDD 14
#define ALL_BLACK 15

#define IBM 3           /* font types */
#define SMALL 5

#define G_BOX      20   /* Graphic types of obs */
#define G_TEXT     21
#define G_BOXTEXT  22
#define G_IMAGE    23
#define G_PROGDEF  24
#define G_IBOX     25
#define G_BUTTON   26
#define G_BOXCHAR  27
#define G_STRING   28
#define G_FTEXT    29
#define G_FBOXTEXT 30
#define G_ICON     31
#define G_TITLE    32

#define NONE       0x0  /* Object flags */
#define SELECTABLE 0x1
#define DEFAULT    0x2
```

E

```
#define EXIT       0x4
#define EDITABLE   0x8
#define RBUTTON   0x10
#define LASTOB    0x20
#define TOUCHEXIT 0x40
#define HIDETREE  0x80
#define INDIRECT 0x100

#define NORMAL    0x0   /* Object states */
#define SELECTED  0x1
#define CROSSED   0x2
#define CHECKED   0x4
#define DISABLED  0x8
#define OUTLINED 0x10
#define SHADOWED 0x20

#define WHITE    0      /* Object colors */
#define BLACK    1
#define RED      2
#define GREEN    3
#define BLUE     4
#define CYAN     5
#define YELLOW   6
#define MAGENTA  7
#define LWHITE   8
#define LBLACK   9
#define LRED     10
#define LGREEN   11
#define LBLUE    12
#define LCYAN    13
#define LYELLOW  14
#define LMAGENTA 15

#define EDSTART 0   /* editable text field definitions */
#define EDINIT  1
#define EDCHAR  2
#define EDEND   3

#define TE_LEFT  0  /* editable text justification */
#define TE_RIGHT 1
#define TE_CNTR  2

/*  Structure Definitions */

typedef struct object {
    int      ob_next;    /* -> object's next sibling */
    int      ob_head;    /* -> head of object's children */
    int      ob_tail;    /* -> tail of object's children */
    unsigned int ob_type;   /* type of object- BOX, CHAR,... */
    unsigned int ob_flags;  /* flags */
```

E

```
    unsigned int ob_state;  /* state- SELECTED, OPEN, ... */
    char *ob_spec;          /* "out"- -> anything else */
    int     ob_x;           /* upper left corner of object */
    int     ob_y;           /* upper left corner of object */
    int     ob_width;       /* width of obj */
    int     ob_height;      /* height of obj */
} OBJECT;


typedef struct orect {
    struct orect *o_link;
    int o_x;
    int o_y;
    int o_w;
    int o_h;
} ORECT;


typedef struct gpoint {
    int p_x;
    int p_y;
} GPOINT;


typedef struct grect {
    int g_x;
    int g_y;
    int g_w;
    int g_h;
} GRECT;


typedef struct text_edinfo {
    char *te_ptext;         /* ptr to text (must be 1st) */
    char *te_ptmplt;        /* ptr to template */
    char *te_pvalid;        /* ptr to validation chrs. */
    int te_font;            /* font */
    int te_junk1;           /* junk word */
    int te_just;            /* justification- left, right... */
    int te_color;           /* color information word */
    int te_junk2;           /* junk word */
    int te_thickness;       /* border thickness */
    int te_txtlen;          /* length of text string */
    int te_tmplen;          /* length of template string */
} TEDINFO;


typedef struct icon_block {
    int *ib_pmask;
    int *ib_pdata;
```

```
    char *ib_ptext;
    int ib_char;
    int ib_xchar;
    int ib_ychar;
    int ib_xicon;
    int ib_yicon;
    int ib_wicon;
    int ib_hicon;
    int ib_xtext;
    int ib_ytext;
    int ib_wtext;
    int ib_htext;
} ICONBLK;

typedef struct bit_block {
    int *bi_pdata;      /* ptr to bit forms data */
    int bi_wb;          /* width of form in bytes */
    int bi_hl;          /* height in lines */
    int bi_x;           /* source x in bit form */
    int bi_y;           /* source y in bit form */
    int bi_color;       /* fg color of blt */
} BITBLK;

typedef struct appl_blk {
    int (*ub_code)();
    long ub_parm;
} APPLBLK;

typedef struct parm_blk {
    OBJECT *pb_tree;
    int pb_obj;
    int pb_prevstate;
    int pb_currstate;
    int pb_x, pb_y, pb_w, pb_h;
    int pb_xc, pb_yc, pb_wc, pb_hc;
    long pb_parm;
} PARMBLK;

#endif DL_OBDEFS
```

osbind.h

```
/*************************************************************************/
/*  OSBINDS.H   #defines for GEMDOS,BIOS & XBIOS binding    */
/*      started 5/2/85 .. Rob Zdybel                 */
/*      Copyright 1985 Atari Corp.            */
/*************************************************************************/
#ifndef DL_OSBIND
#define DL_OSBIND

extern  long    bios();
```

E

```
extern  long    xbios();
extern  long    gemdos();


/*
     These are the data structures that are used by some of the
     BIOS functions.  rpt
*/
typedef struct {
    int (*midivec)();   /* MIDI-input */
    int (*vkbderr)();   /* keyboard error */
    int (*vmiderr)();   /* MIDI error */
    int (*statvec)();   /* ikbd status packet */
    int (*mousevec)();  /* mouse packet */
    int (*clockvec)();  /* clock packet */
    int (*joyvec)();    /* joystick packet */
    int (*midisys)();   /* system MIDI vector */
    int (*ikbdsys)();   /* system IKBD vector */
} kbdvecs;

/*
    Used in function Iorec()
*/
typedef struct {
    char *ibuf; /* pointer to queue */
    int ibufsiz;    /* size of queue in bytes */
    int ibufhd; /* head index of queue */
    int ibuftl; /* tail index of queue */
    int ibuflow;    /* low water mark */
    int ibufhigh;   /* high water mark */
} iorec;


/*
    Used by function Dfree().
*/
typedef struct {
    long b_free;    /* no. of free clusters on drive */
    long b_total;   /* total no. of clusters on drive */
    long b_secsiz;  /* no. of bytes in a sector */
    long b_clsiz;   /* no. of sectors in a cluster */
} disk_info;


/*
    Used by function Getmpb().
*/
typedef struct md {
    struct md *m_link;   /* next memory block       */
    long       m_start;  /* start address of block  */
```

E

```
    long        m_length; /* No. of bytes in block   */
    long        m_own;    /* Memory block's owner ID */
} md;

typedef struct {
    md *mp_mfl;      /* memory free list       */
    md *mp_mal;      /* memory allocated list  */
    md *mp_rover;    /* roving pointer  (woof!) */
} mpb;

/*
    Used by function Getbpb().
*/
typedef struct _bpb {
    int sector_size_bytes;
    int cl_sectors;
    int cl_bytes;
    int dir_length_sectors;
    int FAT_size_sectors;
    int FAT_sector;             /*  sector number of the second FAT.        */
    int data_sector;            /*  sector number of the first data cluster */
    int total_data_clusters;    /*  number of data clusters on the disk     */
    int flags;                  /*  Miscellaneous Flags.                    */
} bpb;


/*
    This structure is a bit field that represents the different components of
    the date and time words.  A union structure was used so that a long
    could be used for the assignment from the gettime() function and the
    bit-field structure could be used to easily decode the long word.

    Note:  This data structure was designed to work with Megamax C.  Not all
           compilers allocate bit-fields in the same manner.  rpt
*/
typedef union {
    struct {
        unsigned day     : 5;
        unsigned month   : 4;
        unsigned year    : 7;
        unsigned seconds : 5;
        unsigned minutes : 6;
        unsigned hours   : 5;
    } part;
    long realtime;
} datetime;


typedef union {
    struct {
```

E

```
        unsigned day     : 5;
        unsigned month   : 4;
        unsigned year    : 7;
    } part;
    unsigned realdate;
} dateinfo;

typedef union {
    struct {
        unsigned seconds : 5;
        unsigned minutes : 6;
        unsigned hours   : 5;
    } part;
    unsigned realtime;
} timeinfo;


/*  BIOS    (trap13)    */
#define Getmpb(a) bios(0, a)
#define Bconstat(a) (int)bios(1,a)
#define Bconin(a)   bios(2,a)
#define Bconout(a,b)    bios(3,a,b)
#define Rwabs(a,b,c,d,e)    bios(4,a,b,c,d,e)
#define Setexc(a,b) bios(5,a,b)
#define Tickcal()   bios(6)
#define Getbpb(a)   (bpb *)bios(7, a)
#define Bcostat(a)  bios(8,a)
#define Mediach(a)  bios(9,a)
#define Drvmap()    bios(10)
#define Kbshift(a)  bios(11,a)

/*  XBIOS   (trap14)    */
#define Initmous(a,b,c) xbios(0,a,b,c)
#define Physbase()  xbios(2)
#define Logbase()   xbios(3)
#define Getrez()    (int)xbios(4)
#define Setscreen(a,b,c)    xbios(5,a,b,c)
#define Setpalette(a)   xbios(6,a)
#define Setcolor(a,b)   (int)xbios(7,a,b)
#define Floprd(a,b,c,d,e,f,g)   (int)xbios(8,a,b,c,d,e,f,g)
#define Flopwr(a,b,c,d,e,f,g)   (int)xbios(9,a,b,c,d,e,f,g)
#define Flopfmt(a,b,c,d,e,f,g,h,i)  (int)xbios(10,a,b,c,d,e,f,g,h,i)
#define Midiws(a,b) xbios(12,a,b)
#define Mfpint(a,b) xbios(13,a,b)
#define Iorec(a)    (iorec *)xbios(14,a)
#define Rsconf(a,b,c,d,e,f) xbios(15,a,b,c,d,e,f)
#define Keytbl(a,b,c)   xbios(16,a,b,c)
#define Random()    xbios(17)
#define Protobt(a,b,c,d)    xbios(18,a,b,c,d)
#define Flopver(a,b,c,d,e,f,g)  (int)xbios(19,a,b,c,d,e,f,g)
```

```
#define Scrdmp()    xbios(20)   /* WARNING: This Bind Incomplete */
#define Cursconf(a,b)   (int)xbios(21,a,b)
#define Settime(a)  xbios(22,a)
#define Gettime()   xbios(23)
#define Bioskeys()  xbios(24)
#define Ikbdws(a,b) xbios(25,a,b)
#define Jdisint(a)  xbios(26,a)
#define Jenabint(a) xbios(27,a)
#define Giaccess(a,b)   (char)xbios(28,a,b)
#define Offgibit(a) xbios(29,a)
#define Ongibit(a)  xbios(30,a)
#define Xbtimer(a,b,c,d)    xbios(31,a,b,c,d)
#define Dosound(a)  xbios(32,a)
#define Setprt(a)   (int) xbios(33,a)
#define Kbdvbase()  (kbdvecs *)xbios(34)
#define Kbrate(a,b) (int)xbios(35,a,b)
#define Prtblk()    xbios(36)
#define Vsync()     xbios(37)
#define Supexec(a)  xbios(38, a)
#define Puntaes()   xbios(39)

/*  GEMDOS  (trap1)     */
#define Pterm0()    gemdos(0x0)
#define Cconin()    (int)gemdos(0x1)
#define Cconout(a)  gemdos(0x2,a)
#define Cauxin()    (int)gemdos(0x3)
#define Cauxout(a)  gemdos(0x4,a)
#define Cprnout(a)  gemdos(0x5,a)
#define Crawio(a)   (int)gemdos(0x6,a)
#define Crawcin()   (int)gemdos(0x7)
#define Cnecin()    (int)gemdos(0x8)
#define Cconws(a)   gemdos(0x9,a)
#define Cconrs(a)   gemdos(0x0a,a)
#define Cconis()    (int)gemdos(0x0b)
#define Dsetdrv(a)  gemdos(0x0e,a)
#define Cconos()    (int)gemdos(0x10)
#define Cprnos()    (int)gemdos(0x11)
#define Cauxis()    (int)gemdos(0x12)
#define Cauxos()    (int)gemdos(0x13)
#define Dgetdrv()   (int)gemdos(0x19)
#define Fsetdta(a)  gemdos(0x1a,a)
#define Super(a)    gemdos(0x20,a)      /* NOTE:This name may change */
#define Tgetdate()  (int)gemdos(0x2a)
#define Tsetdate(a) gemdos(0x2b,a)
#define Tgettime()  (int)gemdos(0x2c)
#define Tsettime(a) gemdos(0x2d,a)
#define Fgetdta()   gemdos(0x2f)
#define Sversion()  (int)gemdos(0x30)
#define Ptermres(a,b)   gemdos(0x31,a,b)
#define Dfree(a,b)  gemdos(0x36,a,b)
```

```
#define Dcreate(a)  (int)gemdos(0x39,a)
#define Ddelete(a)  (int)gemdos(0x3a,a)
#define Dsetpath(a) (int)gemdos(0x3b,a)
#define Fcreate(a,b)    (int)gemdos(0x3c,a,b)
#define Fopen(a,b)  (int)gemdos(0x3d,a,b)
#define Fclose(a)   gemdos(0x3e,a)
#define Fread(a,b,c)    gemdos(0x3f,a,b,c)
#define Fwrite(a,b,c)   gemdos(0x40,a,b,c)
#define Fdelete(a)  (int)gemdos(0x41,a)
#define Fseek(a,b,c)    gemdos(0x42,a,b,c)
#define Fattrib(a,b,c)  (int)gemdos(0x43,a,b,c)
#define Fdup(a)     (int)gemdos(0x45,a)
#define Fforce(a,b) (int)gemdos(0x46,a,b)
#define Dgetpath(a,b)   (int)gemdos(0x47,a,b)
#define Malloc(a)   gemdos(0x48,a)
#define Mfree(a)    (int)gemdos(0x49,a)
#define Mshrink(a,b)    (int)gemdos(0x4a,0,a,b) /* NOTE:Null parameter added */
#define Pexec(a,b,c,d)  gemdos(0x4b,a,b,c,d)
#define Pterm(a)    gemdos(0x4c,a)
#define Fsfirst(a,b)    (int)gemdos(0x4e,a,b)
#define Fsnext()    (int)gemdos(0x4f)
#define Frename(a,b,c)  (int)gemdos(0x56,a,b,c)
#define Fdatime(a,b,c)  (int)gemdos(0x57,a,b,c)

#endif
```

portab.h

```
/*************************************************************************/
/*  PORTAB.H    Pointless redefinitions of C syntax.        */
/*      Copyright 1985 Atari Corp.          */
/*                          */
/*  WARNING: Use of this file may make your code incompatible with  */
/*       C compilers throughout the civilized world.        */
/*************************************************************************/
#ifndef DL_PORTAB
#define DL_PORTAB               /*  rpt  8-21-87    */

#define mc68k 0

#define UCHARA 1                /* if char is unsigned      */
/*
 *  Standard type definitions
 */
#define BYTE    char                /* Signed byte          */
#define BOOLEAN int             /* 2 valued (true/false)    */
#define WORD    int                 /* Signed word (16 bits)    */
#define UWORD   unsigned int            /* unsigned word        */

#define LONG    long                /* signed long (32 bits)    */
#define ULONG   long                /* Unsigned long        */
```

E

```
#define REG register            /* register variable       */
#define LOCAL   auto                /* Local var on 68000      */
#define EXTERN  extern              /* External variable       */
#define NLOCAL  static              /* Local to module     */
#define GLOBAL  /**/                /* Global variable     */
#define VOID    /**/                /* Void function return    */

#ifndef DEFAULT         /*  This means that default is defined in obdefs.h  */
#define DEFAULT int             /* Default size        */
#endif

#ifdef UCHARA
#define UBYTE   char                /* Unsigned byte       */
#else
#define UBYTE   unsigned char           /* Unsigned byte       */
#endif

#ifndef FAILURE
/****************************************************************************/
/*  Miscellaneous Definitions:                          */
/****************************************************************************/
#define FAILURE (-1)            /*  Function failure return val */
#define SUCCESS (0)         /*  Function success return val */
#define YES 1           /*  "TRUE"              */
#define NO  0           /*  "FALSE"             */
#define FOREVER for(;;)         /*  Infinite loop declaration   */
#define NULL    0           /*  Null pointer value      */
#define NULLPTR (char *) 0      /*                  */
#define EOF (-1)            /*  EOF Value           */
#define TRUE    (1)         /*  Function TRUE  value        */
#define FALSE   (0)         /*  Function FALSE value        */
#endif

#endif DL_PORTAB
```

stdio.h

```
#ifndef DL_STDIO
#define DL_STDIO

#define _BUFSIZE 512
#define BUFSIZ  _BUFSIZE        /* Unix compatable */
#define _NFILE 20

typedef struct _iobuf {
    char *_ptr;
    int _cnt;
    char *_base;
    int _flag;
    int _fd;
```

E

```
    int _bufsize;   /* buffer size for this file */
} FILE;
extern FILE _iob[_NFILE];

#define stdin (&_iob[0])
#define stdout (&_iob[1])
#define stderr (&_iob[2])
#define STDIN   0
#define STDOUT  1
#define STDERR  STDOUT
#define STDAUX  2
#define STDPRT  3

#define _READ     01
#define _WRITE    02
#define _APPEND   04
#define _UNBUF    010
#define _BIGBUF   020
#define _EOF      040
#define _ERR      0100
#define _DIRTY    0200   /* buffer was changed */
#define _LINBUF   0400
#define _IFLUSH   01000  /* (ONLY STDIN) Flush stdout when filling */
#define _RDWR     02000

#define NULL 0L    /* must be long since it can be passed as a parameter */
#define EOF (-1)

#define getc(p) (--(p)->_cnt >= 0 ? *(p)->_ptr++ & 0377 : _fillbuf(p))
#define getchar() getc(stdin)
#define putc(x,p) (--(p)->_cnt >= 0 ? (*(p)->_ptr++ = (x)) & 0377 : \
    _flushbuf((x),p))
#define putchar(x) putc(x,stdout)
#define feof(p) ((p)->_flag&_EOF)
#define ferror(p) ((p)->_flag&_ERR)
#define clearerr(p) ((p)->_flag &= ~(_ERR | _EOF))
#define fileno(p) ((p)->_fd)
#define rand() (int)(_seed = _seed * 6907 + 130253)
#define srand(x) _seed = x;

extern FILE *fopen(), *fdopen(), *freopen();
extern long ftell();
extern char *gets();
extern char *fgets();
extern long _seed;

/*
 * These are not normally part of stdio.h, but are included here to help
 * reduce errors made by beginning programmers.
 */
```

E

```
extern char *sprintf(), *malloc(), *lmalloc(), *calloc(), *lcalloc();
extern char *alloca(), *realloc(), *lrealloc();
extern long labs(), lseek();

extern int errno;   /* defined in exit.c */

typedef long jmp_buf[10];

#endif /* DL_STDIO */
```

strings.h

```
/*  string.h    4.1 83/05/26    */
#ifndef DL_STRINGS
#define DL_STRINGS

/*
 * External function definitions
 * for routines described in string(3).
 */

extern char *index();
extern char *rindex();
extern char *strcat();
extern char *strcpy();
extern char *strncat();
extern char *strncpy();
extern char *xtrcat();
extern char *xtrcpy();
extern char *xtrncpy();
extern int  strcmp();
extern int  strlen();
extern int  strncmp();

#endif /* DL_STRINGS */
```

E

Index

preprocessor, 14
68901 MFP, 493, 506, 525

A

.A extension, 8
a_bitblit(), 540
abs(), 110
.ACC extension, 8
AC_CLOSE, 183
AC_OPEN, 164, 183
a_copyraster(), 541
acos(), 134
address registers, 24
a_drawsprite(), 542
AES, 163
a_fillpoly(), 543
a_fillrect(), 545
a_getpixel(), 546
a_hidemouse(), 547
a_hline(), 548
a_init(), 549
a_line(), 550
alloca(), 132
angles, 309
APPLBLK, 245
appl_exit(), 165, 172
appl_find(), 173
application ID, 165
appl_init(), 165, 174
appl_read(), 175
appl_tplay(), 176
appl_trecord(), 177
appl_write(), 178
a_putpixel(), 551
argc, 106, 107
argv, 106, 107
a_showmouse(), 552
asin(), 134
asm, 17
assembly language, 17
atan(), 134
a_textblit(), 553
atof(), 111
atoi(), 159
atol(), 159
a_transformmouse(), 554
a_undrawsprite(), 556
auto, 18
auto variables, 23
AUX:, 106, 422

B

base page variable, 419
batch, 45
bcmp(), 112
Bconin(), 422
Bconout(), 422
Bconstat(), 422
bcopy(), 112
Bcostat(), 422
BEG_MCTRL, 282
binary mode, 105, 124

I

BIOS I/O, 106
Bioskeys(), 489
Bit fields, 18
BITBLK, 244
BSS segment, 26, 577
buffered I/O, 106
BUFSIZ, 153
bzero(), 112

C

.C extension, 8
calloc(), 107, 132
carriage return, 105
cat utility, 99
Causout(), 420
Cauxin(), 420
Cauxis(), 420
Cauxos(), 420
CC environment variable, 30
CCOM environment variable, 30
Cconin(), 425
Cconis(), 425
Cconos(), 425
Cconout(), 425
Cconrs(), 425
Cconws(), 425
.CFG extension, 8
change_aux(), 559
change_item(), 559
char, 18
char type, 13
character constants, 15
CHECKED, 250
choose, 5
CINCLUDE environment variable, 30
CINIT environment variable, 31
clearerr(), 122
clear_tree(), 559
CLIB environment variable, 31
click, 5
clipboard, 271
close(), 113
Cnecin(), 425
command line execution, 45
comments, 21
CON:, 106, 422
constant expression, 22
control-click, 5
control-drag, 5
cos(), 134
cp utility, 99
Cprnos(), 499
Cprnout(), 499
Crawcin(), 425
Crawio(), 425
creat(), 115, 147
CROSSED, 250
ctype.h, 593
Cursconf(), 427
cursor, 5

D

dabs(), 134
data registers, 24
DATA segment, 26, 577
DC pseudo op, 22
Dcreate(), 428
Ddelete(), 429
.DEF extension, 8
DEFAULT, 248
#define, 14, 24
define value option, 55
define.h, 594
desk accessory, 61, 164
desktop window, 282
development cycle, 8
device I/O, 106
Dfree(), 430
Dgetdrv(), 435
Dgetpath(), 436

I

dint(), 134
DISABLED, 250
disk cache, 32
Dosound(), 432
double type, 13
double-click, 5
drag, 5
Drvmap(), 434
Dsetdrv(), 435
Dsetpath(), 436
DTA, 460
DTA (Disk Transfer Address), 446
dump utility, 99

E

EDITABLE, 249
effective address, 21
enum type, 13
enumeration types, 15
environ, 118
event_multi(), 164
evnt_button(), 186
evnt_dclick(), 188
evnt_keybd(), 189
evnt_mesag(), 190
evnt_mouse(), 191
evnt_multi(), 193
evnt_timer(), 196
execv(), 118
execve(), 118
exit(), 120, 121
EXIT, 249
_exit(), 120
exp(), 134
exp10(), 134
exp2(), 134
extern, 18, 24
external reference, 57
external variables, 23

F

fac(), 134
Fattrib(), 437
fclose(), 121
Fclose(), 439
fcntl.h, 594
Fcreate(), 440
Fdatime(), 442
Fdelete(), 441
fdopen(), 123
Fdup(), 444
feof(), 122
ferror(), 122, 143
fflush(), 121
Fforce(), 445
fgetc(), 127
Fgetdta(), 446
fgets(), 129
filefmt.h, 594
fileno(), 122
float type, 13
Flopfmt(), 447
Floprd(), 447
Flopver(), 447
Flopwr(), 447
FMD_FINISH, 287
fopen(), 123
Fopen(), 452
form_alert(), 199, 200, 202
form_center(), 203
form_dial(), 198, 204, 287
form_do(), 198, 206
form_error(), 200, 207
forward pointer reference, 16
_fpreg0, 20, 23
fprintf(), 140
_fprintf(), 140
fputc(), 143
fputs(), 144
fread(), 125
Fread(), 456

free(), 107, 132
Frename(), 457
freopen(), 123
fscanf(), 150
fseek(), 126
Fseek(), 458
fsel_input(), 211
Fsetdta(), 446
Fsfirst(), 460
Fsnext(), 460
ftell(), 126
function arguments, 23
fwrite(), 125
Fwrite(), 456

G

G_BOX, 247
G_BOXCHAR, 247
G_BOXTEXT, 241, 247
G_BUTTON, 247
gembind.h, 597
gemdefs.h, 281, 607
Getbpb(), 463
getc(), 127
getchar(), 127
getenv(), 128
Getmpb(), 464
Getrez(), 510
gets(), 129
Gettime(), 466
getw(), 127
G_FBOXTEXT, 241, 248
G_FTEXT, 241, 248
Giaccess(), 469
G_IBOX, 247
G_ICON, 242, 248
G_IMAGE, 244, 247
global array, 261
G_PROGDEF, 245, 247
graf_dragbox(), 214
graf_growbox(), 215
graf_handle(), 216
graf_mkstate(), 217
graf_mouse(), 218
graf_movebox(), 219
graf_rubberbox(), 220
graf_shrinkbox(), 221
graf_slidebox(), 222
graf_watchbox(), 223
G_STRING, 247
G_TEXT, 241, 247
G_TITLE, 248

H

.H extension, 8
handle, 307
heap, 418
HIDETREE, 249

I

ICONBLK, 242
identifiers, 14
IEEE, 18
Ikbdws(), 475
#include, 14
include path option, 56
index(), 157
INDIRECT, 249
Initmous(), 480
in-line assembler, 17
insertion point, 5
int type, 13
I/O redirection, 106
Iorec(), 482
isalnum(), 116
isalpha(), 116
isascii(), 116
isatty(), 130
iscntrl(), 116
isdigit(), 116

islower(), 116
isprint(), 116
ispunct(), 116
isspace(), 116
isupper(), 116
isxdigit(), 116

J

Jdisint(), 493
Jenabint(), 493
Jump table, 19

K

Kbdvbase(), 485
Kbrate, 484
Kbshift(), 488
Keyboard port, 422
Keytbl(), 489
K&R, 3

L

label, assembly, 22
labels, 21
labs(), 110
LASER.CFG, 27
LASTOB, 249
LaTeX, i
lcalloc(), 132
LIBPATH environment variable, 31
line feed, 105
line separator, 105
line-A graphics routines, 531
linea.h, 611
LINKER environment variable, 30
lmalloc(), 132
.LNK extension, 8
log(), 134
log10(), 134
log2(), 134
Logbase(), 510
long type, 13
longjmp(), 155
lrealloc(), 132
ls utility, 99
lsbrk(), 149
lseek(), 131

M

main(), 490
MAKE environment variable, 30
malloc(), 107, 132
Malloc(), 490
math.h, 615
matinv(), 134
max(), 560
Mediach(), 492
menu_bar(), 225, 226, 230
menu_icheck(), 231
menu_ienable(), 232
menu_register(), 164, 183, 233
menu_text(), 234
menu_tnormal(), 235
message event, 179
MFDB, 308, 375, 378, 380
MFP, 493
Mfpint(), 493
Mfree(), 490
MIDI port, 422, 485
Midiws(), 495
min(), 560
mkdir utility, 99
MN_REDRAW, 180
MN_SELECTED, 180
Mshrink(), 490
mulpower2(), 134
mv utility, 99

N

name scoping, 16
NDC coordinates, 307
NORMAL, 250

I

O

.O extension, 8
obdefs.h, 616
O_BINARY, 107, 115, 138, 147, 162
objc_add(), 251
objc_change(), 252
objc_delete(), 254
objc_draw(), 255
objc_edit(), 256
objc_find(), 258
objc_offset(), 259
objc_order(), 260
obj_draw(), 198
OBJECT, 237, 239
object format, 577
object trees, 81
O_CREAT, 138
Offgibit(), 469
onexit(), 137
Ongibit(), 469
Opcodes, 21
open(), 138, 147
optimizations, 18
O_RDONLY, 138
O_RDWR, 138
osbind.h, 620
O_TRUNC, 138
OUTLINED, 250
O_WRONLY, 138

P

parameters, 20
PARMBLK, 245
PATH environment variable, 31
perror(), 139
Pexec(), 497
Physbase(), 510
portab.h, 625
powerd(), 134
poweri(), 134
Preprocessor, 13
press, 5
.PRG extension, 8
printf(), 140
Protobt(), 500
PRT:, 106, 422
pt_2rect(), 561
pt_add(), 562
pt_equal(), 563
Pterm(), 502
Pterm0(), 502
Ptermres(), 502
pt_inrect(), 564
pt_set(), 565
pt_sub(), 566
Puntaes(), 503
putc(), 143
putchar(), 143
puts(), 144
putw(), 143

Q

qsort(), 145

R

RAM resident, 28
RAM resident list, 28
rand(), 146
Random(), 504
RBUTTON, 249
RC coordinates, 307
RCP, 198
rcp.prg, 81
read(), 106, 147
realloc(), 132
rect_empty(), 567
rect_equal(), 568
rect_inset(), 569
rect_intersect(), 570
rect_offset(), 571

I

rect_set(), 572
rect_union(), 573
register names, 21
register variables, 21, 23, 24
regular expressions, 95
rename(), 148
resource, 81
resource file, 81
rewind(), 126
rindex(), 157
rm utility, 99
rmdir utility, 99
RS232 port, 420
.RSC extension, 8
Rsconf(), 505
rsrc_free(), 263
rsrc_gaddr(), 167, 198, 264
rsrc_load(), 165, 167, 225, 226, 261, 266
rsrc_obfix(), 267
rsrc_saddr(), 268
Rwabs(), 507

S

sbrk(), 149
scanf(), 150
Scrdmp(), 509
scroll bars, 6
scrp_read(), 272
scrp_write(), 273
select, 5
SELECTABLE, 248
SELECTED, 250
selection range, 90
selector box, 7
setbuf(), 106, 153
setbuffer(), 153
Setcolor(), 513
Setexc(), 514
setjmp(), 155
setlinebuf(), 153
Setpalette(), 515
Setprt(), 516
Setscreen(), 510
Settime(), 466
SHADOWED, 250
shel_envrn(), 276
shel_find(), 277
shel_read(), 278
shel_write(), 279
shift-click, 5
shift-drag, 5
short type, 13
sin(), 134
size utility, 99
sprintf(), 140
_sprintf(), 140
sqr(), 134
sqrt(), 134
srand(), 146
sscanf(), 150
stack dump, 49
stack space, 107
static, 18, 24
static variables, 23
stderr, 106, 123
stdin, 106, 123
stdio.h, 626
stdout, 106, 123
_stksize, 107, 133
strcat(), 157
strcmp(), 157
strcpy(), 157
stream files, 106
stream I/O, 106
strings.h, 628
strlen(), 157
strncat(), 157
strncmp(), 157
strncpy(), 157

I

strtol(), 159
structs, 18
structure assignment, 15
structure member names, 16
stuffbits(), 574
stuffhex(), 575
Super(), 517
Supexec(), 519
Sversion(), 520
sys_errlist, 139
sys_nerr, 139

T

tan(), 134
TEDINFO, 240
text marks, 42
TEXT segment, 26, 577
Tgetdate(), 522
Tgettime(), 522
Tickcal(), 521
timer event, 183
toascii(), 114
tolower(), 114
_tolower(), 114
tools (Laser Shell), 28
.TOS extension, 8
TOUCHEXIT, 249
toupper(), 114
_toupper(), 114
TPA, 490
TPA (Transient Program Area), 418
Tsetdate(), 522
Tsettime(), 522
.TTP extension, 8
type, 5

U

unbuffered I/O, 106
underscore, 14
ungetc(), 160
union assignment, 15
unions, 18
UNIX, 149
unlink(), 161
unsigned, 18
unsigned char type, 13
unsigned long type, 13
unsigned type, 13
untranslated, 105
untranslated mode, 124
update policy, 2

V

v_arc(), 312
variable names, 14
v_bar(), 314
v_circle(), 315
v_clrwk(), 316
v_clsvwk(), 317
v_clswk(), 318
v_contourfill(), 319
v_curdown(), 320
v_curhome(), 320
v_curleft(), 320
v_curright(), 320
v_curtext(), 321
v_curup(), 320
VDI, 307
v_eeol(), 322
v_eeos(), 323
v_ellarc(), 324
v_ellipse(), 325
v_ellpie(), 326
v_enter_cur(), 327
vex_butv(), 328
vex_curv(), 330
v_exit_cur(), 332
vex_motv(), 333
vex_timv(), 335
v_fillarea(), 337

I

v_get_pixel(), 338
v_gtext(), 339
v_hide_c(), 340
virtual workstation, 307, 342
v_justified(), 341
void type, 13
v_opnvwk(), 342
v_opnwk(), 345
v_pieslice(), 350
v_pline(), 351
v_pmarker(), 352
vq_chcells(), 354
vq_color(), 355
vq_curaddress(), 357
vq_extnd(), 358
vqf_attributes(), 360
vq_key_s(), 361
vql_attributes(), 363
vqm_attributes(), 364
vq_mouse(), 365
vqt_attributes(), 366
vqt_extent(), 367
vqt_fontinfo(), 368
vqt_name(), 370
vqt_width(), 371
v_rbox(), 373
v_rfbox(), 374
vro_cpyfm(), 375
vr_recfl(), 377
vrt_cpyfm(), 378
vr_trnfm(), 380
v_rvoff(), 381
v_rvon(), 381
vsc_form(), 382
vs_clip(), 287, 384
vs_color(), 385
vs_curaddress(), 386
vsf_color(), 387
vsf_interior(), 388
vsf_perimeter(), 389
vsf_style(), 390
vsf_udpat(), 392
v_show_c(), 394
vsl_color(), 395
vsl_ends(), 396
vsl_type(), 397
vsl_udsty(), 398
vsl_width(), 399
vsm_color(), 400
vsm_height(), 401
vsm_type(), 402
vsm_valuator(), 403
vst_alignment(), 404
vst_color(), 406
vst_effects(), 407
vst_font(), 409
vst_height(), 410
vst_load_fonts(), 411
vst_point(), 412
vst_rotation(), 413
vst_unload_fonts(), 414
vswr_mode(), 415
Vsync(), 524
v_updwk(), 416

W

WF_NEWDESK, 282
WF_WORKXYWH, 282
wind_calc(), 283, 292
wind_close(), 294
wind_create(), 295
wind_delete(), 297
wind_find(), 298
wind_get(), 282, 299
wind_open(), 302
wind_set(), 282, 303
wind_update(), 163, 282, 305
WM_ARROWED, 181
WM_CLOSED, 181
WM_FULLED, 181

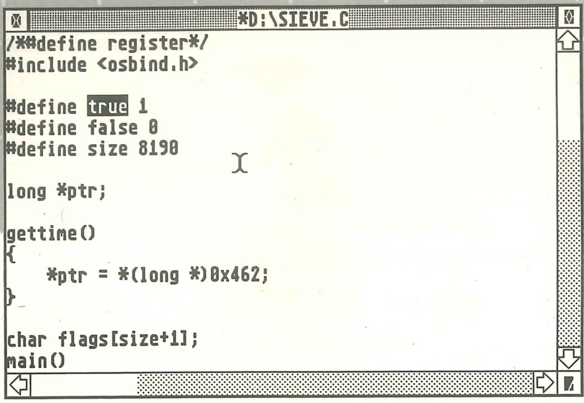
I

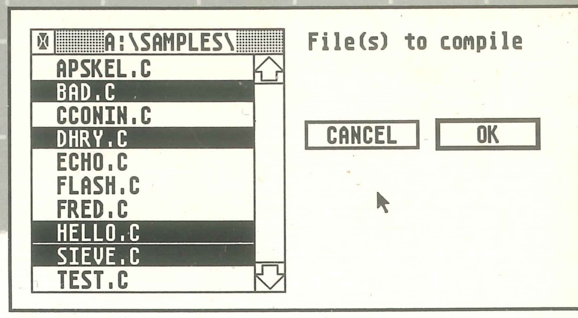
WM_HSLID, 182
WM_MOVED, 182
WM_NEWTOP, 183
WM_REDRAW, 283, 287
WM_SIZED, 182
WM_TOPPED, 181
WM_VSLID, 182
workstation, 307, 345
write(), 106, 162
write through, 30

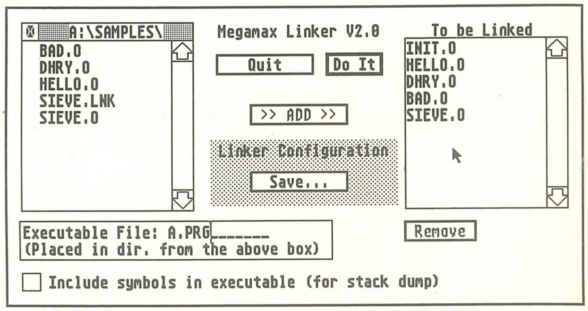
X

Xbtimer(), 525
xstrcat(), 157
xstrcpy(), 157
xstrncpy(), 157

I







"Don't even think about another C compiler" said Mike Fleischman in a review of Megamax C for ANTIC magazine. Many others shared his enthusiasm, making Megamax C the most popular development system for serious Atari ST programmers. Now Megamax brings you the next generation—Laser C.

Laser C provides the fastest and most complete C development system for the Atari ST. With compile and link speeds averaging over 15 times faster than its closest competitor, *Laser C is unsurpassed for maximizing programmer productivity. The system tools, tightly integrated within the Laser Shell, provide fast and powerful editing and debugging facilities. Laser C also includes tools to completely automate the development cycle. With one key stroke a single program or an entire project can be produced in a flash.

The entire system is designed for easy and intuitive use. Laser C fully utilizes the ST's GEM user interface, eliminating the need for a long and frustrating learning period. Just one more reason why Laser C is the only development system you will ever need.

*times based on sieve, apskel, and hello world benchmarks compiled on a 1040 ST. Competitors systems utilize ram disks.

Features Include:

- RAM resident graphical shell
- Absolute code production compiler (no 32k limits)
- Variable size RAM cache
- Full-featured Make facility
- Graphical Resource Construction Program
- DRI compatible linker
- Powerful debugging facilities
- Fast and accurate floating point
- Mouse-based multi-window editor
- Complete technical documentation
- Complete GEM documentation
- Examples using GEM routines
- Telephone technical support
- Full support of GEM functions
- Large complement of Unix™ routines

Megamax, Inc. ■ Box 851521 ■ Richardson, Texas 75085-1521 ■ (214) 987-4931