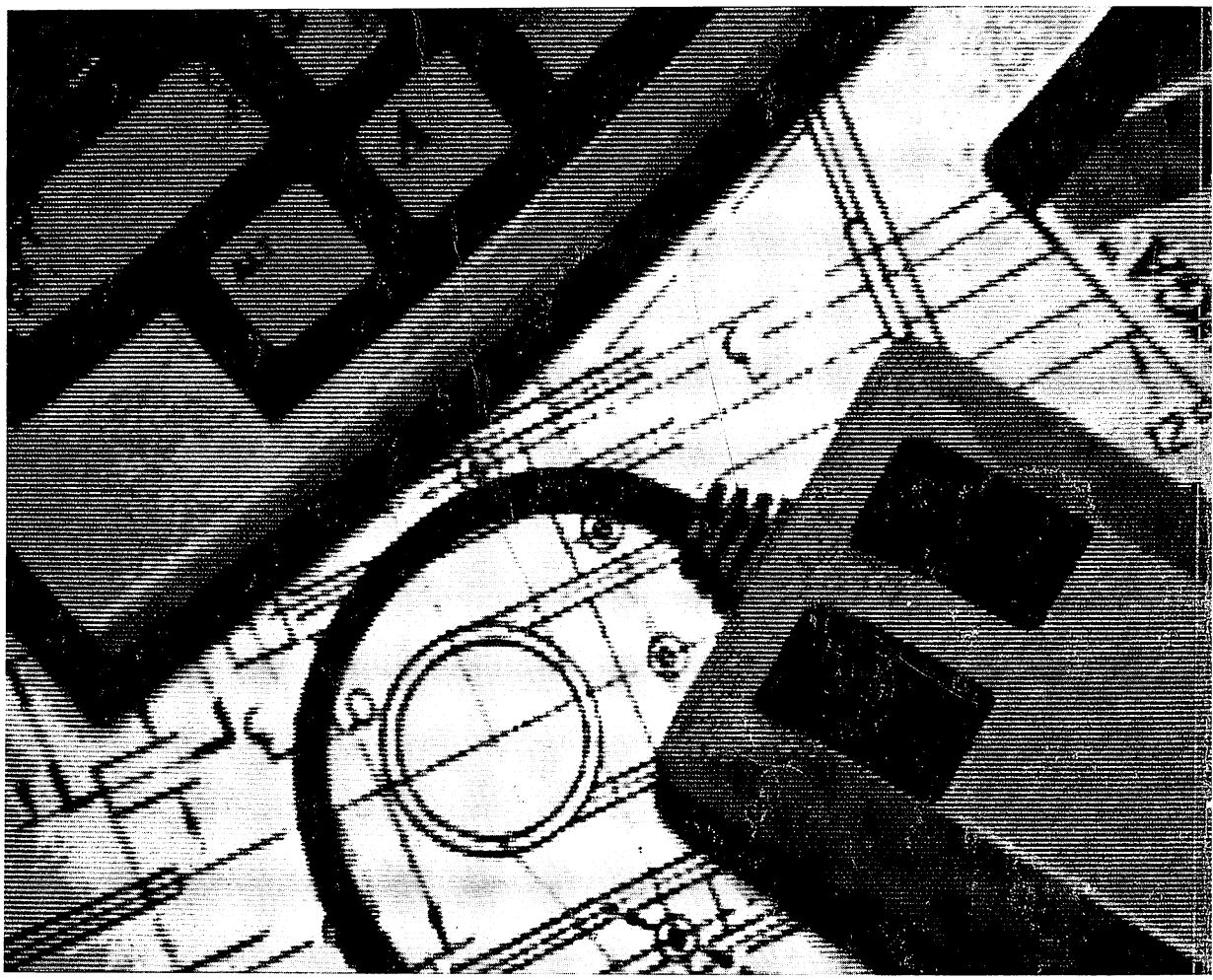
AMIGA™



ROM Kernel Manual

Volume 1

AMIGA

ROM KERNEL MANUAL

Authors: Rob Peck, Carl Sassenrath, Susan Deyl

Program Examples by: Rob Peck, Carl Sassenrath, Sam Dicker, Tom Pohorsky, Larry Hildenbrand

Acknowledgements:

The following personnel have contributed significantly to the contents of this manual:

. . . . . Bruce Barrett, Dave Berezowski, Bob Burns, Sam Dicker, Andy Finkel, Larry Hildebrand, Neil Katin, Dale Luck,
. . . . . Dave Lucas, Jim Macraz, R. J. Mical, Bob Pariseau, Tom Pohorsky, Stan Shepard, and Barry Whitebook.

This edition of the Amiga ROM Kernel Manual corresponds to version 1.1 of the Amiga kernel system software.

Printed in U.S.A.

CBM Product Number 327271-02 rev 2 12.9.85

Preface

This preface introduces kernel programming on the Amiga and gives a brief overview of the contents of this manual.

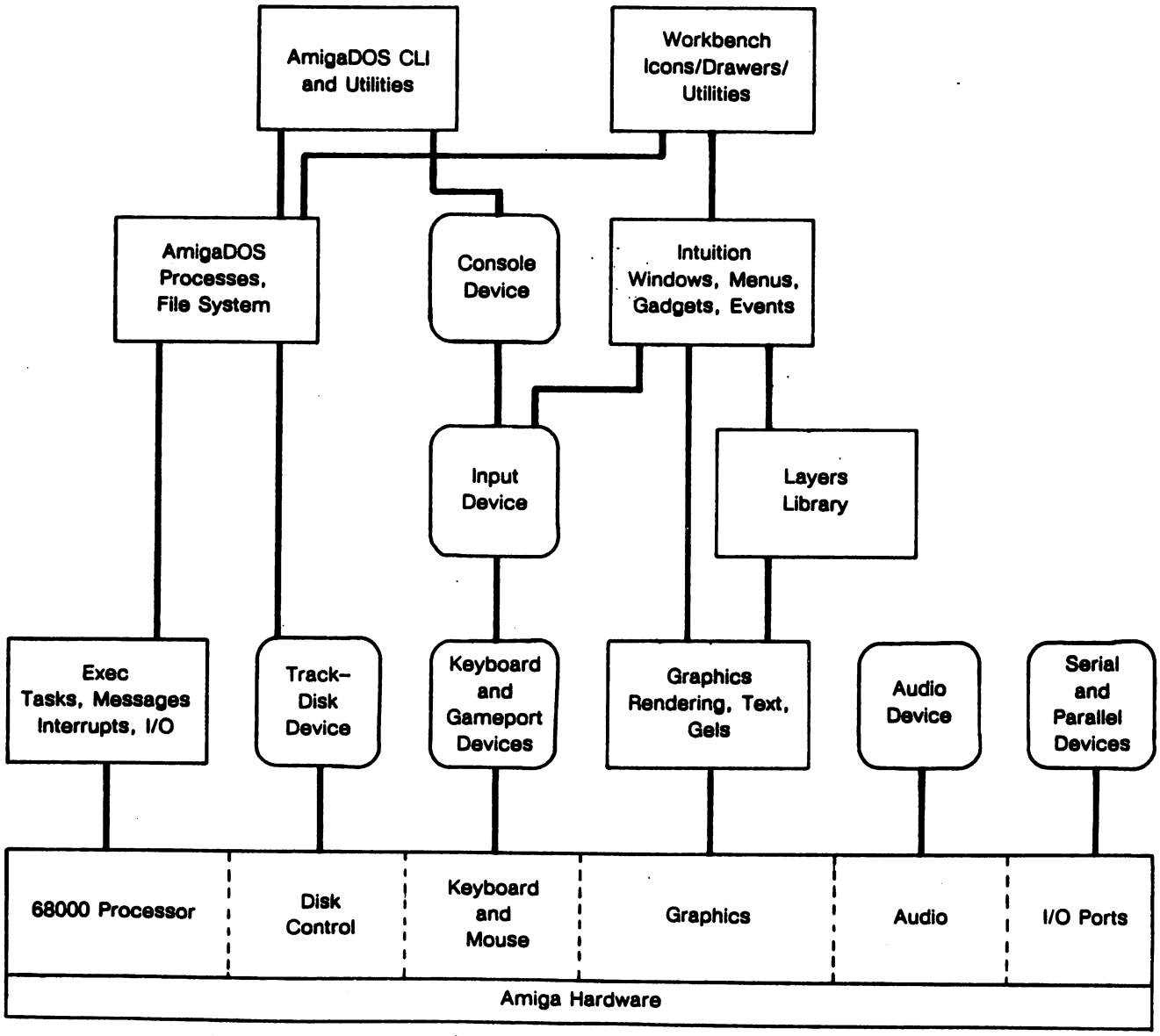
System Software Architecture

The Amiga kernel consists of a number of system modules some of which reside permanently in the protected *kickstart* memory and others that are loaded on a demand basis from the system disk. These modules form a hierarchy as illustrated in figure 1. As you look at the figure, you can see how the various modules interact with each other.

At the top of the chain are Workbench and the CLI (Command Line Interface), the user-visible portions of the system. Workbench utilizes Intuition to produce its displays and AmigaDOS to interact with the filing system. Intuition, in turn, uses the input device to retrieve its input and the graphics and layers library routines to produce its output.

AmigaDOS controls processes and maintains the filing system, and is in turn built on Exec, which manages tasks, task switching, interrupt scheduling, message passing, I/O and many other functions.

At the lowest level is the Amiga hardware itself. Just above the hardware are the modules that control the hardware directly. Exec controls the 68000, scheduling its time among tasks and maintaining its interrupt vectors, among other things. The trackdisk device is the lowest-level interface to the disk hardware, performing disk head movement and raw disk I/O. The keyboard and gameport devices handle the keyboard and gameport hardware, queueing up input events for the input device to process. The audio device, serial device, and parallel device handle their respective hardware. Finally, the routines in the graphics library handle the interface to the graphics hardware.



Amiga System Software Modules

Programming

The functions of the kernel were designed to be accessed from any language that follows our standard interface conventions. These conventions define the proper naming of symbols, the correct usage of processor registers, and the format of public data structures.

Register Conventions

All system functions follow a simple set of register conventions. The conventions apply when calling any system function, and we also encourage programmers to use the same conventions in their own code.

The registers D0, D1, A0, and A1 are always scratch; they are free to be modified at any time. They may be used by a function without first saving their previous contents.

All other data and address registers must have their values preserved. If any of these registers are used by a function, their contents must be saved and restored appropriately.

If you are using assembly code, function parameters may be passed in registers. The conventions in the preceding paragraphs apply to this use of registers as well. Parameters passed in D0, D1, A0, or A1 may be destroyed. All other registers will be preserved.

If a function returns a result, it is passed back to the caller in D0. If a function returns more than one result, the primary result is returned in D0 and all other results are returned by accessing reference parameters.

The A6 register has a special use within the system, and it may not be used as a parameter to system functions. It is normally used as a pointer to the base of a function vector table. All kernel functions are accessed by jumping to an address relative to this base.

Data Structures

In addition to the naming of public data structures, the format and initial values of these structures must also be consistent. The conventions are quite simple and are summarized below.

1. All non-byte fields must be word-aligned. This may require that you pad certain fields with an extra byte.

2. All address pointers should be 32 bits (not 24 bits) in size. The upper byte must never be used for data.

3. Fields that are not defined to contain particular initial values must be initialized to zero. This includes pointer fields.

4. All reserved fields must be initialized to zero (for future compatibility).

5. Data structures to be accessed by custom hardware must not be allocated on your program stack.

6. Public data structures must not be allocated on your program stack (a task control structure for example).

7. When data structures are dynamically allocated, items 3 and 4 above can be satisfied by specifying that the structure is to be cleared upon allocation.

Other Practices

A few other general programming practices should be noted.

1. Never use absolute addresses. All hardware registers and special addresses have symbolic names (see the include files and amiga.lib).

2. Because this is a multitasking system, you must never directly modify the processor exception vectors (including traps) or the processor priority level.

3. Do not assume that you can access hardware resources directly. Most hardware is controlled by system software that will not respond well to interference. Shared hardware requires you to use the proper sharing protocols.

4. Do not access shared data structures directly without the proper mutual exclusion. Remember, it's a multitasking system and other tasks may also be accessing the same structures.

5. Most system functions require a particular execution environment. For example, DOS functions can only be executed from within a process; execution from within a task is not sufficient. As another example, most kernel functions can be executed from within tasks, but cannot be executed from within interrupts.

6. The system does not monitor the size of your program stack. You should take care not to overflow it.

7. Tasks always execute in the 68000 processor user mode. Supervisor mode is reserved for interrupts, traps, and task dispatching. Take extreme care if your code executes in supervisor mode. Exceptions while in supervisor mode are deadly.

8. Do not disable interrupts or multitasking for long periods of time.

68010 and 68020 Compatibility

If you wish your code to be upwardly compatible with the 68010/68020 processors, you must avoid certain instructions and you must not make assumptions about the format of the supervisor stack frame.

In particular, the **MOVE SR,<ea>** instruction is a privileged instruction on the 68010 and 68020. If you want your code to work correctly on all 680x0 processors, you should use the **GetCC()** function instead (see the Exec library function descriptions in the appendices to this manual).

Using Amiga Exec Functions

The following guidelines will be helpful when you are trying to determine which functions may be run from within a task or from within interrupt code, when to forbid or permit task switching, and when to disable or enable interrupts.

Functions that Tasks Can Perform

Amiga system software distinguishes between tasks and processes. Figure 1 in this preface showed this difference. Specifically, the information in a task control block is a subset of the information contained in a process control block. Consequently, any functions that expect to use process control information will not function correctly if provided with a pointer to a task. Generally speaking, tasks can perform any function that is described in this manual.

A task cannot, however, perform any function that is related to AmigaDOS (such as printf, file-read, file-write, and so on). If you want a task to perform DOS-related functions, you should arrange for the task to send a message to a "process," which in turn can perform the function (filling a buffer that is passed to the task, for example) and signal that the job has been done. The alternative is to use the DOS function **CreateProc()** instead of the Exec support function **CreateTask()** for tasks that you spawn yourself. A process can call all functions, including DOS functions.

More information about tasks can be found in the "Tasks" chapter.

Functions that Interrupt Code Can Perform

The following Exec functions can be safely performed during interrupts:

Alert()	**FindPort()**
Disable()	**FindTask()**
Cause()	**PutMsg()**
Enable()	**ReplyMsg()**
FindName()	**Signal()**

In addition, if you are manipulating *your own* list structures during interrupt code, you can also use the following functions:

AddHead()
AddTail()
Enqueue()
RemHead()
RemTail()

General Information about Synchronization

The system functions **Enable**() and **Disable**() are provided to enable and disable interrupts. The system functions **Forbid**() and **Permit**() disallow or allow task switching. You need only determine with what you are trying to synchronize before deciding if you must wrap an **Enable**()/**Disable**() pair around a function call, use **Forbid**()/**Permit**(), or simply allow the system to interrupt or switch tasks at its whim.

If you are trying to modify a data structure common to two tasks, you must assure that your access to these structures is consistent. One method is to put **Forbid**()/**Permit**() around anything that modifies (or reads) that structure. This makes the function atomic; that is, the structure is stable and consistent after each full operation by either task. If you are trying to synchronize with something that might happen as a result of interrupt code (for example, Exec data structures), you put **Disable**()/**Enable**() around any of your own operations that might interact with such operations. There are other methods (sending messages, using semaphores, and so on) but they are somewhat more involved.

Note that if you are trying to read the contents of a data structure while it is being changed, it is possible to generate an address error that will be sensed by the 68000, causing an exception. This is caused by reading a pointer which is supposed to point to where the data is located. If the pointer value is no longer valid, it may point to a nonexistent memory location that, when read, causes an exception.

Contents of this Manual

This manual describes the Amiga's system software. For the most part, the software described here is ROM-resident. It includes the multi-tasking executive (Exec), the graphics support routines (including text and animation), and the I/O devices. Also included are the Workbench, an environment for running programs, and the floating point mathematics library.

For all parts of the system software, the discussion of the data structures and routines is reinforced through numerous C-language examples. The examples are kept as simple as possible. Whenever possible, each example demonstrates a single function. Where appropriate, there are complete sample programs.

This book is organized into four parts, which are largely tutorial, and eight appendices, which contain reference material.

Part I describes the functions of Exec. The chapters in this part are:

- *Chapter 1: Lists and Queues*—the basic elements of lists and queues, node structure of lists, linkage and initialization of list structures, and list support functions and macros.

- *Chapter 2: Tasks*—the management of tasks, task creation and termination, event signals, traps, exceptions, and mutual exclusion.

- *Chapter 3: Messages and Ports*—inter-system communication in the kernel, structure of messages and ports, message exchange methods, arrival notification actions, and various support functions.

- *Chapter 4: I/O*—concepts of I/O on system devices, form of I/O requests, device interface functions and Exec support functions, standard device commands, and how to perform I/O

- *Chapter 5: Interrupts*—the software interface to interrupts, normal interrupt sequence of events, interrupt priorities, interrupt handlers, interrupt servers, software interrupts, and interrupt exclusion.

- *Chapter 6: Memory Allocation*—routines for dynamic memory allocation and deallocation, how to specify memory allocation according to the actual needs of a task and the hardware it expects to use.

- *Chapter 7: Libraries*—how libraries are designed and used.

- *Chapter 8: ROM-Wack*—how to enter and use the ROM-resident version of the Amiga debugger.

Part II covers the graphics, text, and animation routines. Because this part is organized in the form of a tutorial about the graphics, you should read each chapter in sequence. Part I contains the following chapters:

- *Chapter 1: Graphics Primitives*—how to use the basic graphics tools: support structures, display routines, and drawing routines.

- *Chapter 2: Layers*—how to use the layers library, which allows video display to be split into overlapping, independently controllable areas.

- *Chapter 3: Animation*—how to use the animation routines to produce the two kinds of animated graphics images: sprites and playfield animation.

- *Chapter 4: Text*—how to use the text support routines to print text either in the default text fonts or your own fonts.

Part III contains a chapter for each of the Amiga I/O devices. For general information about I/O, see the chapter called "I/O" in Part I. Also, the *Amiga Hardware Manual* specifies a direct hardware interface for many of the devices covered here. The chapters in this part are:

- *Chapter 1: Audio Device*—how to use the Amiga's four audio channels to produce sound and some considerations for producing clear, quality audio effects. The audio software is implemented as a standard Amiga input/output device with commands and functions that allocate audio channels and control the sound output.

- *Chapter 2: Timer Device*—how to use the timer device to produce a semi-precise time delay.

- *Chapter 3: Trackdisk Device*—how to use the device that actually directly drives the disk, controls the disk motors, and reads and write raw data to the tracks. The trackdisk driver is the lowest level software access to the disk data, and is used by AmigaDOS.

- *Chapter 4: Console Device*—describes how to do keyboard and screen I/O.

- *Chapter 5: Input Device*—the input device is a combination of three other devices in the system: keyboard device, gameport device, and timer device. The input device merges together separate input event streams from the keyboard, mouse and timer into a single stream which can then be interpreted by input handlers.

- *Chapter 6: Keyboard Device*—describes the keyboard device, which gives system access to the Amiga keyboard.

- *Chapter 7: Gameport Device*—shows you how to use the gameport device, which is the access to the Amiga gameports.

- *Chapter 8: Narrator Device*—how to use the speech synthesizer.

- *Chapter 9: Serial Device*—describes software access to the serial port.

- *Chapter 10: Parallel Device*—describes software access to the parallel port.

- *Chapter 11: Printer Device*—describes the various ways of doing output to a printer, including graphics, and how to create your own printer device driver.

Part IV contains the following chapters:

- *Chapter 1: Math Functions*—describes the structure and calling sequences required to access the Motorola fast floating point library.

- *Chapter 2: Workbench*—shows how to interface your program to the program that (1) provides a screen where other applications can run, (2) gives users an icon interface to the Amiga file system, and (3) gives the programmer access to library functions for manipulating objects and icons.

The **Appendixes** contain reference material:

- *Appendix A* contains the summarized references for the built-in libraries of routines.
- *Appendix B* contains the summarized references for all device commands.
- *Appendix C* contains the summarized references for resources.
- *Appendix D* contains printouts of the C-language include files.
- *Appendix E* contains printouts of the assembly-language include files.
- *Appendix F* is a printout of the Exec support library.
- *Appendix G* covers some AmigaDOS topics that are not included in the series of AmigaDOS manuals.
- *Appendix H* describes the IFF standard for interchange format files.
- *Appendix I* contains the printer-dependent sample code referenced in the "Printer Device" chapter.
- *Appendix J* is the hardware memory map.
- *Appendix K* contains source code for a skeleton device and a skeleton library.
- *Appendix L* contains information about the Amiga disk format.

Other Manuals

See also *Intuition: The Amiga User Interface*, *AmigaDOS User's Manual*, *AmigaDOS Developer's Manual*, and *AmigaDOS Technical Reference Manual*.

Text Conventions

Boldface type is used for the names of functions, data structures, macros, and variables.

System header files and other system file names are shown in italics, and italics are also used for emphasis.

Table of Contents

PART I

PART II

PART III

PART IV

Part I

Chapter 1

Lists and Queues

This chapter describes the basic elements of Exec lists and queues. It discusses the node structure of lists, the linkage and initialization of list structures, and the list support functions and macros. Queues and priority sorted lists are achieved through the use of the list functions applied in a certain order and are also discussed.

A thorough understanding of this chapter is necessary to properly write programs that deal with Exec.

1.1. INTRODUCTION

The Amiga system software operates in a highly dynamic environment of control data structures. An early design goal of Exec was to keep the system flexible and open-ended by not creating artificial boundaries on the number of system structures used. Rather than using static sized system tables, Exec uses dynamically created structures that are attached to the system as needed. This concept is central to the design of Exec.

Exec uses lists to maintain its internal data base of system structures. Tasks, interrupts, libraries, devices, messages, I/O requests, and all other Exec data structures are supported and serviced through the consistent application of Exec's list mechanism.

Lists have a common data structure and a common set of functions for manipulating them. Because all of these structures are treated in a similar manner, only a small number of list handling functions need be supported by Exec.

1.2. LIST STRUCTURE

A list is composed of a *header* and a chain of linked elements called *nodes*.

The header maintains memory pointers to the first and last nodes of the linked chain of nodes. The address of the header serves as the handle to the entire list. When referring to a list, you refer to the address of its header. In addition, the header specifies the data type of the nodes in a list. We'll discuss node data typing later.

Node Structure

A *node* is divided into two parts: list linkage and node content. The linkage part contains memory pointers to the node's successor and predecessor nodes, the node data type, and the node priority. The content part stores the actual data structure of interest.

As a C language structure, the linkage part of a node is defined:

```
struct Node {
   struct  Node *ln_Succ;
   struct  Node *ln_Pred;
   UBYTE   ln_Type;
   BYTE    ln_Pri;
   char    *ln_Name;
};
```

where

ln_Succ

points to the next node in the list (successor),

ln_Pred

points to the previous node in the list (predecessor),

ln_Type

defines the type of the node,

ln_Pri

specifies the priority of the node, and

ln_Name

points to a printable name for the node.

As an example of a complete node, the Exec **Interrupt** structure is defined as:

```
struct Interrupt {
   struct  Node is_Node;
   APTR    is_Data;
   VOID    (*is_Code)();
};
```

Here the **is_Data** and **is_Code** fields represent the useful content of the node.

Node Initialization

Before you link a node into a list, you should initialize it first. The initialization consists of setting the **ln_Type**, **ln_Pri**, and **ln_Name** fields to their appropriate values. The **ln_Succ** and **ln_Pred** fields do not require initialization.

The **ln_Type** field contains the data type of the node. This indicates to Exec (and other interested subsystems) the type and hence the structure, of the content portion of the node. A few of the standard system types are defined in the *exec/nodes.i* and */exec/nodes.h* include files. A few examples are: **NT_TASK, NT_INTERRUPT, NT_DEVICE, NT_MSGPORT,** etc.

The **ln_Pri** field indicates the priority of the node relative to other nodes in the same list. This is a signed numerical value ranging from +127 to -128. Higher priority nodes have more positive values; for example, 127 is the highest priority, zero is nominal priority, -128 is the lowest priority.

Some Exec lists are kept sorted by priority order. In such lists the highest priority node is at the head of the list; the lowest priority node is at the tail of the list. For most Exec node types, priority is not used. In such cases it is a good practice to initialize the priority field to zero.

The **ln_Name** field is a pointer to a null-terminated string of characters. Node names are used mostly to bind symbolic names to actual nodes. They are also useful for debugging purposes. It is always a good idea to provide every node with a name.

Here is a C example showing how you might initialize a node called **myInt**, which is an instance of the interrupt structure defined above:

```
myInt.ln_Type = NT_INTERRUPT;
myInt.ln_Pri = 20;
myInt.ln_Name = "sample.interrupt"
```

Header Structure

As mentioned earlier the header maintains memory pointers to the first and last nodes of the linked chain of nodes. This header also serves as a handle for referencing the entire list.

Here is the C-structure of a list header:

```
struct List {
   struct Node *lh_Head;
   struct Node *lh_Tail;
   struct Node *lh_TailPred;
   UBYTE  lh_Type;
   UBYTE  lh_pad;
};
```

where:

lh_Head
: points to the first node in the list,

lh_Tail
: is always zero,

lh_TailPred
: points to the last node in the list,

lh_Type
: defines the type of nodes within the list, and

lh_pad
: is just a structure alignment byte (not used).

There is one subtlety here that should be further explained. The head and tail portions of the header actually overlap. This is best understood if you think of the head and tail as two separate nodes. The **lh_Head** field is the **ln_Succ** field of the first node in the list and the **lh_Tail** field is its **ln_Pred**. The **lh_Tail** is set permanently zero to indicate that this node is the first on the list; that is, it has no successors.

A similar method is used for the tail node. The **lh_Tail** field is the **lh_Succ** field of the last node in the list and the **lh_TailPred** field is its **ln_Pred**. In this case, the zero **lh_Tail** indicates that the node is the last on the list; that is, it has no predecessors.

Header Initialization

List headers must be properly initialized before use. It *is not* adequate to initialize the entire header to zero. The head and tail entries must be set up correctly.

Here is how the header should be initialized:

1. Assign the **lh_Head** field to the address of **lh_Tail**.
2. Assign the **lh_TailPred** field to the address of **lh_Head**.
3. Clear the **lh_Tail** field.
4. Set **lh_Type** to the same data type as that of the nodes to be kept in this list.

In C an example initialization might look like:

```
list.lh_Head = &list.lh_Tail;
list.lh_TailPred = &list.lh_Head;
list.lh_Tail = 0;
list.lh_Type = NT_INTERRUPTS;
```

In assembly code only four instructions are necessary to initialize the header:

```
MOVE.L A0,(A0)
ADDQ.L #LH_TAIL,(A0)
CLR.L  LH_TAIL(A0)
MOVE.L A0 ,LH_TAILPRED(A0).
```

Note that this is the same as the macro **NEWLIST,** contained in the file *exec/lists.i*. It performs its function without destroying the pointer to the list header in A0 (which is why **ADDQ.L** is used). This function may also be accessed from C as a call to **NewList**(lh) where **lh** is the address of the list header. See the source code for **CreatePort**() in chapter 3, "Messages and Ports", for one instance of its use.

1.3. LIST FUNCTIONS

Exec provides a number of symmetric functions for handling lists. There are functions for inserting and removing nodes in lists, for adding and removing tail and head nodes in lists, for inserting nodes in a priority order, and for searching a list for a node with a particular name.

Insertion and Removal

The **Insert()** function is used for inserting a new node into any position in a list. It always inserts the node following a specified node that is already part of the list.

For example, **Insert(list,node,pred)** inserts the node after **pred** in the specified list. If the **pred** node points to the list header or is null, the new node will be inserted at the head of the list. Similarly, if the **pred** node points to the list **lh_Tail** field, the new node will be inserted at the tail of the list. However, both of these actions can be better accomplished with the functions mentioned in the "Special Case Insertion" section below.

The **Remove()** function is used to remove a specified node from a list. For example **Remove(node)** will remove the specified node from whatever list it's in. *Please note:* removing a node requires that it actually be in the list. If you attempt to remove a node that is not in a list, you will cause serious system internal problems.

Special Case Insertion

Although the **Insert()** function allows new nodes to be inserted at the head and the tail of a list, the **AddHead()** and **AddTail()** functions will do so with higher efficiency. Adding to the head or tail of a list is common practice in queuing type operations, as in first-in-first-out (FIFO) or last-in-last-out (LIFO or stack) operations.

For example **AddHead(list,node)** would insert the node at the head of the specified list.

Special Case Removal

The two functions **RemHead()** and **RemTail()** are used in combination with **AddHead()** and **AddTail()** to create special list ordering. When you combine **AddTail()** and **RemHead()**, you produce a first-in-first-out (FIFO) list. When you combine **AddHead()** and **RemHead()** a last-in-first-out (LIFO or stack) list is produced. **RemTail()** exists for symmetry. Other combinations of these functions can also be used productively.

For example **RemTail(list)** removes the first node from the specified list and returns a pointer to it as a result. If the list is empty, it returns a zero result.

Prioritized Insertion

None of the list functions discussed so far make use of the priority field in the list data structure. The **Enqueue()** function makes use of this field and is equivalent to **Insert()** for a priority sorted list. It performs an insert on a priority basis, keeping the higher priority nodes towards the head of the list. All nodes passed to this function must have their priority assigned prior to the call.

For example, **Enqueue(list,node)** inserts the node into the prioritized list after the last node of same or higher priority.

As mentioned earlier, the highest priority node is at the head of the list, the lowest priority node is at the tail of the list. Using the **RemHead()** function will return the highest priority

node, and **RemTail()** the lowest priority node.

Note that if you insert a node that has the same priority as another node in the list, **Enqueue()** will use FIFO ordering. The new node is inserted following the last node of equal priority.

Searching by Name

Since most lists contain nodes with symbolic names attached (via the **ln_Name** field), it is possible to find a node by its name. This naming technique is used throughout Exec for such nodes as tasks, libraries, devices, and resources.

The **FindName()** function is provided to search a list for the first node with a given name. For example, **FindName(list,"Furrbol")** returns a pointer to the first node named "Furrbol". If no such node exists, a zero is returned. The case of the name characters is significatn; "foo" is different from "Foo."

To find multiple occurrences of nodes with identical names, the **FindName()** function is called multiple times. For example, if you want to find the second node with the "Furrbol" name:

```
struct Node node;
node = FindName(list,"Furrbol");
if (node != 0) {
    node = FindName(node,"Furrbol");
}
```

Notice that the second search uses the node found by the first search. The **FindName()** function never compares the specified name with that of the starting node. It always begins the search with the successor of the starting node.

1.4. LIST MACROS

Assembly code programmers may want to optimize their code by using assembly code list macros. Because these macros actually embed the specified list operation into the code, they result in slightly faster operations. The file *exec/lists.i* contains the recommended set of macros.

For example, the following instructions implement the **REMOVE** macro:

```
MOVE.L (A1),A0              * get successor
MOVE.L LN_PRED(A1),A1       * get predecessor
MOVE.L A0,(A1)              * fixup predecessor's succ pointer
MOVE.L A1,LN_PRED(A0)       * fixup successor's pred pointer
```

1.5. EMPTY LISTS

It is often important to determine if a list is empty. This can be done in many ways, but only two are worth mentioning. If either the **lh_TailPred** field is pointing to the list header, or the **ln_Succ** field of the **lh_Head** is zero, then the list is empty.

In C, for example, these would be:

```
if (list.lh_TailPred == &list) {
    printf ("list is empty");
}
```

or

```
if (list.lh_Head->ln_Succ == 0) {
    printf ("list is empty");
}
```

In assembly code if **A0** points to the list header, these would be:

```
CMP.L   LH_TAILPRED(A0),A0
BEQ     list_is_empty
```

and

```
MOVE.L LH_HEAD(A0),A1
MOVE.L LN_SUCC(A1),D0
BEQ    list_is_empty
```

Because **LH_HEAD** and **LN_SUCC** are both zero offsets, the second case can be simplified.

1.6. SCANNING A LIST

Occasionally a program may need to scan a list to locate a particular node, find a node that has a field with a particular value, or just print the list. Because lists are linked in both the forward and backward directions, the list can be scanned either from either the head or tail.

Here is an example of C code that uses a **for** loop to print the names of all nodes in a list:

```
for (node = list -> lh_Head; node -> ln_Succ; node = node -> ln_Succ) {
    printf ("node %lx is named %s", node, node -> ln_name);
}
```

In assembly code it is more efficient to use a lookahead cache pointer when scanning a list. In this example the list is scanned until the first zero priority node is reached:

```
    MOVE.L  (A1),D1         * first node
scan:
    MOVE.L  D1,A1
    MOVE.L  (A1),D1         * lookahead to next
    BEQ.S   not_found       * end of list
    TST.B   LN_PRI(A1)
    BNE.S   scan
    ...                     * found one

not_found:
```

Important Note: It is possible to collide with other tasks when manipulating shared system lists. For example, if some other task happens to be modifying a list while your task scans it, an inconsistent view of the list may be formed. This can result in a corrupted system. Generally it is not permissible to read or write a shared system list without first locking out access from other tasks (and in some cases locking out access from interrupts). This technique of mutual-exclusion is discussed in the "Tasks" chapter.

Chapter 2

Tasks

This chapter describes the management of tasks on the Amiga. It includes explanations of task creation, termination, event signals, traps, exceptions, and mutual exclusion. This chapter assumes that the reader has a basic understanding of lists (from the previous chapter) and some understanding of multitasking principles.

2.1. INTRODUCTION

Multitasking is one of the primary features supported by Exec. Multitasking is the ability of an operating system to manage the simultaneous execution of multiple independent processor contexts. In addition, good multitasking does this in a transparent fashion; a task is not forced to recognize the existence of other tasks. In Exec this involves sharing the 68000 processor among a number of concurrent programs, providing each with its own virtual processor.

Scheduling

Exec accomplishes multitasking by *multiplexing* the 68000 processor among a number of task contexts. Every task has an assigned priority and tasks are scheduled to use the processor on a priority basis. The highest priority ready task is selected and receives processing until:

1. a higher priority task becomes active,

2. the running task exceeds a preset time period (*a quantum*) and there is another equal priority task ready to run, or

3. the task needs to wait for an external event before it can continue.

Task scheduling is normally preemptive in nature. The running task may lose the processor at nearly any moment by being displaced by another more urgent task. Later, when the preempted task regains the processor it continues from where it left off.

It is also possible to run a task in a non-preemptive manner. This mode of execution is generally reserved for system data structure access. It is discussed in the "Exclusion" section toward the end of this chapter.

In addition to the prioritized scheduling of tasks, *time-slicing* also occurs for tasks with the same priority. In this scheme a task is allowed to execute for a quantum (a preset time period). If the task exceeds this period, the system will preempt it and give other tasks of the same priority a chance to run. This will result in a time-sequenced *round robin* scheduling of all equal priority tasks.

Due to the prioritized nature of task scheduling, tasks must avoid performing the *busy wait* technique of polling. This is where a piece of code loops endlessly waiting for a change in state of some external condition. Tasks that use the busy wait technique waste the processor and eat up all its spare power. In most cases this prevents lower priority tasks from receiving any processor time. Because certain devices such as the keyboard and the disk depend on their associated tasks, using a busy wait at a high priority may defer important system services. Busy waiting can even cause system deadlocks.

When there are no ready tasks, the processor is halted and only interrupts will be serviced. Since task multiplexing often occurs as a result of events triggered by system interrupts, this is not a problem. Halting the processor often helps improve the performance of other system bus devices.

Task States

For every task Exec maintains state information to indicate its status. A normally operating task will exist in one of three states:

running indicates that a task currently owns the processor. This usually means that the task is actually executing, but it is also possible that it has been temporarily displaced by a system interrupt.

ready indicates that a task is not currently executing but is scheduled for the processor. The task will receive processor time based on its priority relative to the priorities of other running and ready tasks.

waiting marks a task as waiting for an external event to occur. It is not scheduled to use the processor. The task will only be made ready when one of its external events occurs (see "Signals" section below).

It is also possible for a task to exist in a few transient states:

added indicates that a task has just been added to Exec and has not yet been scheduled for processing.

removed
marks that the task is being removed. Tasks in this state are effectively terminated and are usually undergoing cleanup operations.

exception
indicates that a task is scheduled for special exception processing.

Task Queues

Tasks that are not in the running state are linked into one of two system queues. Tasks that are marked as ready to run but awaiting an opportunity to do so are kept in the *ready queue*. This queue is always kept in a priority sorted order with the highest priority task at the head of the queue.

A *waiting queue* accounts for tasks that are awaiting external events. Unlike the ready queue, there is no reason to keep this queue sorted by priority. New entries are appended to the tail of the queue. A task will remain in the waiting queue until it is awakened by an event (at which time it is placed into the ready queue).

Priority

A task's priority indicates its importance relative to other tasks. Higher priority tasks receive the processor before lower priority tasks.

Task priority is stored as a signed number ranging from -128 to +127. Higher priorities are represented by more positive values and zero is considered the neutral priority. Normally system tasks execute somewhere in the range of +20 to -20.

It is not wise to needlessly raise a task's priority. Sometimes it may be necessary to carefully select a priority so that the task can properly interact with various system tasks. The **ChangePri()** Exec function is provided for this purpose.

Structure

Exec maintains task context and state information in a task control data structure. As with most Exec structures these are dynamically linked onto various task queues through the use of a prepended list **Node** structure. The C-language form of this structure is defined in the *exec/task.h* include file as:

```
extern struct Task {
    struct   Node tc_Node;
    UBYTE tc_Flags;
    UBYTE tc_State;
    BYTE   tc_IDNestCnt;        /* intr disabled nesting */
    BYTE   tc_TDNestCnt;        /* task disabled nesting */
    ULONG tc_SigAlloc;          /* sigs allocated */
    ULONG tc_SigWait;           /* sigs we are waiting for */
    ULONG tc_SigRecvd;          /* sigs we have received */
    ULONG tc_SigExcept;         /* sigs we will take excepts for */
    UWORDtc_TrapAlloc;          /* traps allocated */
    UWORDtc_TrapAble;           /* traps enabled */
    APTR   tc_ExceptData;       /* points to except data */
    APTR   tc_ExceptCode;       /* points to except code */
    APTR   tc_TrapData;         /* points to trap code */
    APTR   tc_TrapCode;         /* points to trap data */
    APTR   tc_SPReg;            /* stack pointer */
    APTR   tc_SPLower;          /* stack lower bound */
    APTR   tc_SPUpper;          /* stack upper bound + 2*/
    VOID   (*tc_Switch)();      /* task losing CPU */
    VOID   (*tc_Launch)();      /* task getting CPU */
    struct   List tc_MemEntry;  /* allocated memory */
    APTR   tc_UserData;         /* per task data */
};
```

A similar assembly code structure is available in the *exec/tasks.i* include file.

Most of these fields are not relevant for simple tasks; they are used by Exec for state and administrative purposes. A few fields, however, are provided for the advanced programs that support higher level environments (as in the case of *processes*) or require precise control (as in *devices*). The following sections explain these fields in more detail.

2.2. CREATION

To create a new task you must allocate a task structure, initialize its various fields, and then link it into Exec with a call to **AddTask()**.

The task structure may be allocated by calling the **AllocMem()** function with the **MEMF_CLEAR** and **MEMF_PUBLIC** allocation attributes. These attributes indicate that the data structure is to be pre-initialized to zero and that the structure is shared.

The **Task** fields that require initialization depend on how you intend to use the task. For the simplest of tasks, only a few fields must be initialized:

tc_Node
: the task list node structure. This includes the task's priority, its type, and its name (refer to the "Lists" chapter).

tc_SPLower
: the lower memory bound of the task's stack

tc_SPUpper
: the upper memory bound of the task's stack

tc_SPReg
: the initial stack pointer. Since task stacks grow *downward* in memory, this field is usually set to the same value as **tc_SPUpper**.

Zeroing all other unused fields will cause Exec to supply the appropriate system default values. Allocating the structure with the **MEMF_CLEAR** attribute is an easy way to be sure that this happens.

Once the structure has been initialized, it must be linked to Exec. This is done with a call to **AddTask()** with the following parameters:

task is a pointer to an initialized task structure.

initialPC
: is the entry point of your task code. This is the address of the first instruction the new task will execute.

finalPC
: is the finalization code for your task. This is a code fragment that will receive control if the **initialPC** routine ever performs a return (**RTS**). This exists to prevent your task from being launched into random memory upon an accidental return. The **finalPC** routine should usually perform various program related clean-up and remove the task. If a zero is supplied as this parameter, Exec will use its default finalization code (which simply calls the **RemTask()** function).

Depending on the priority of the new task and the priorities of other tasks in the system, the newly added task may immediately begin execution.

Here is an example of simple task creation:

```
#define STACK_SIZE 1000
extern APTR AllocMem();
extern EntryPoint();

SimpleTask()
{
     struct Task *task;

     stack = AllocMem (STACK_SIZE, MEMF_CLEAR );
     if (stack == 0) {
          printf ("not enough memory for task stack");
          return;
     }

     task = AllocMem (sizeof(struct Task *), MEMF_CLEAR | MEMF_PUBLIC);
     if (newTask == 0) {
          printf ("not enough memory for task control structure");
          FreeMem (stack, STACK_SIZE);
          return;
     }

     task -> tc_SPLower = stack;
     task -> tc_SPUpper = stack + STACK_SIZE;
     task -> tc_SPReg = stack + STACK_SIZE;

     task -> tc_Node.ln_Type = NT_TASK;
     task -> tc_Node.ln_Pri = 0;
     task -> tc_Node.ln_Name = "example.task";

     AddTask (task, EntryPoint, 0);
}
```

Stack

Every task requires a stack. All task stacks are *user mode* stacks (in the language of the 68000) and are addressed through the **A7** CPU register. All normal code execution occurs on this task stack. Special modes of execution (processor traps and system interrupts for example) execute on a single *supervisor mode* stack and do not directly affect task stacks.

Task stacks are normally used to store local variables, subroutine return addresses, and saved register values. Additionally, when a task loses the processor all of its current registers are preserved on this stack (with the exception of the stack pointer itself, which must be saved in the task structure).

The amount of stack used by a task can vary widely. The minimum stack size is that required to save 17 CPU registers and a single return address. This totals to 70 bytes. Of course, a stack of this size would not give you adequate space to perform any subroutine calls (because the return address occupies stack space). On the other hand a stack size of 1K would suffice to call most system functions but would not allow much in the way of local variable storage.

Because stack bounds checking is not provided as a service of Exec, it is very important to provide enough space for your task stack. Stack overflows are always difficult to debug and may result not only in the erratic failure of your task but also in the mysterious malfunction of other Amiga subsystems.

2.3. TERMINATION

Task termination may occur as the result of:

1. a program returning from its **initialPC** routine and dropping into its **finalPC** routine or the system default finalizer.

2. a task trap that is too serious for a recovery action. This includes traps like processor bus error, odd address access errors, etc.

3. a trap that is not handled by the task. For example, this might occur if your code happened to encounter a processor **TRAP** instruction and you did not provide a trap handling routine.

4. an explicit call to the Exec **RemTask()** function.

Task termination involves the deallocation of system resources and the removal of the task structure from the Exec.

The most important part of task termination is the deallocation of system resources. A task must return all memory that it allocated for its private use, it must terminate any outstanding IO commands, and it must close access to any system libraries or devices that it has open.

It is wise to adopt a strategy for task cleanup responsibility. You should decide whether resource allocation and deallocation is the duty of the creator task or the newly created task. Sometimes it is easier and safer for the creator to handle the necessary resource allocation and deallocation on behalf of its offspring. On the other hand, if you expect the creator to terminate before its offspring, it would not be able to handle resource deallocation. In such a case each of its child tasks would need to deallocate its own resources.

2.4. SIGNALS

Tasks often need to coordinate with other concurrent system activities (other tasks and interrupts). Such coordination is achieved through the synchronized exchange of specific event indicators called *signals*. This is the primary mechanism responsible for all inter-task communication and synchronization on the Amiga.

The signal mechanism operates at a low level and is designed for high performance. Signals often remain hidden from the user program. The message system, for instance, may use signals to indicate the arrival of a new message. The message system is described in more detail in the next chapter.

The signal system is designed to support independent simultaneous events. Signals may be thought of as occurring in parallel. Each task may define up to 32 independent signals. These signals are stored as single bits in a few fields of the task control structure and one or more signals can occur at the same time.

All of these signals are considered *task relative*: a task may assign its own significance to a particular signal. Signals are not broadcast to all tasks; they are directed only to individual tasks. A signal has meaning to the task that defined it and to those tasks that have been informed of its meaning. For example, signal bit 12 may indicate a timeout event to one task but to another task it may indicate a message arrival event.

Allocation

As mentioned above, a task assigns its own meaning to a particular signal. Because certain system libraries may occasionally require the use of a signal, there is a convention for signal allocation. It is unwise to ever make assumptions about which signals are actually in use.

Before a signal can be used it must first be allocated with the **AllocSignal()** function. This marks the signal as being in use and prevents the accidental use of the same signal for more than one event. You may ask for either a specific signal number or just the next free signal. The state of the newly allocated signal is cleared (ready for use).

Generally it is best to let the system assign you the next free signal. Of the 32 available signals the lower 16 are usually reserved for system use only. This leaves the upper 16 free for the user. Other subsystems that you may call depend on **AllocSignal()**.

This C example asks for the next free signal to be allocated for its use:

```
signal = AllocSignal(-1);
if (signal == -1) {
        printf("no signal bits available");
        return;
}
else {
        printf("allocated signal number %ld", signal);
}
```

Take note of the fact that the value returned by **AllocSignal()** is a signal bit number. This value cannot be used directly in calls to signal-related functions without first converting it to a mask:

```
mask = 1 << signal;
```

When a signal is no longer needed, it should be freed for reuse with **FreeSignal()**.

It is important to realize that signal bit allocation is only relevant to the running task. You cannot allocate a signal from another task.

Waiting for a Signal

Signals are most often used to wake up a task upon the occurrence of some external event. This happens when a task is in its wait state and another task (or a system interrupt) causes a signal.

The **Wait()** function specifies the set of signals that will wake up the task and then puts the task to sleep (into the waiting state). Any one signal or any combination of signals from this set are sufficient to awake the task. **Wait()** returns a mask indicating which signals from this set satisfied the wait.

The **Wait()** function implicitly clears those signals that satisfied the wait. This effectively resets those signals for reuse.

Because tasks (and interrupts) normally execute asynchronously it is often possible to receive a particular signal before a task actually waits for it. To avoid missing any events, signals will be held until the **Wait()** function is called, or until it is explicitly cleared (with **SetSignal()**). In such cases a wait will be immediately satisfied, and the task will not be put to sleep.

As mentioned earlier, a task may wait for more than one signal. When the task returns from the wait, the actual signal mask is returned. Usually the program must check which signals occurred and take the appropriate action. The order in which these bits are checked is often important. Here is a hypothetical example:

```
signals = Wait (newCharSig | cancelSig | timeOutSig);
if (signals & cancelSig) {
      printf ("canceled");
}
if (signals & newCharSig) {
      printf ("new character");
}
if (signals & timeOutSig) {
      printf ("timeout");
}
```

This will put the task to sleep waiting for either a new character, a cancel event, or the expiration of a time period. Notice that we check for a cancel signal before checking for a new character or a timeout.

Although a program can check for the occurrence of a particular event by checking whether its signal has occurred, this may lead to busy wait type polling. Such polling is wasteful of the processor and is usually detrimental to the proper function of the system.

Generating a Signal

Signals may be generated from both tasks and system interrupts with the **Signal()** function. For example **Signal(task,mask)** would signal the task with the mask signals. More than one signal can be specified in the mask.

2.5. EXCLUSION

From time to time the advanced system program may find it necessary to access global system data structures. Because these structures are shared by the system and by other tasks which execute *asynchronously* to your task, it is wise for you to exclude simultaneous access to these structures. This can be accomplished by *forbidding*, *disabling*, or with the use of *semaphores*. A section of code that requires the use of any of these mechanisms to lock out access by others is termed a *critical section*.

Forbidding

Forbidding is used when a task is accessing shared structures that might also be accessed at the same time from another task. It effectively eliminates the possibility of simultaneous access by imposing *non-preemptive* task scheduling. This has the net effect of disabling multitasking for as long as your task remains in its *running* state.

While forbidden, your task will continue running until it performs a call to **Wait()** or exits from the forbidden state. Interrupts will occur normally, but no new tasks will be dispatched *regardless of their priorities.*

When a task running in the forbidden state calls the **Wait()** function, it implies a temporary exit from its forbidden state. While the task is waiting, the system will perform normally. When the tasks receives one of the signals it's waiting for, it will again re-enter the forbidden state.

To become forbidden, a task calls the **Forbid()** function. To escape, the **Permit()** function is used. The use of these functions may be nested with the expected affects; you will not exit the forbidden mode until you call the outermost **Permit()**.

As an example, Exec memory region lists should be accessed only when forbidden. To access these lists without forbidding jeopardizes the integrity of the entire system.

```
Forbid();
for (mem = (struct MemHeader *) eb -> MemList.lh_Head;
          mem -> mh_Node.ln_Succ; mem = mem -> mh_Node.ln_Succ) {
     firsts[count++] = mem -> mh_First;
}
Permit();
```

As this program traverses down the memory region list, it remains disabled to prevent the list from changing as it is being accessed.

Disabling

Disabling is similar to forbidding but, in addition it prevents interrupts from occurring during a critical section. It is required when a task accesses structures that are shared by interrupt code. Disabling, eliminates the possibility of an interrupt accessing shared structures by preventing interrupts from occurring.

To disable interrupts you can call the **Disable()** function. If you're writing in assembly code, the **DISABLE** macro is more efficient (but consumes more code space). To enable interrupts again the **Enable()** function and **ENABLE** macros are provided.

Like forbidden sections, disabled sections can be nested. Also like forbidden sections, the **Wait()** function implies an **Enable()** until the task again regains the processor.

It is important to realize that there is a danger in using disabled sections. Because the software on the Amiga depends heavily on its interrupts occurring in nearly real time, you cannot disable for more than a very brief instant. A rule of thumb is to never disable for more than 250 microseconds.

Masking interrupts by changing the 68000 processor interrupt priority levels with the **MOVESR** instruction can also be very dangerous and is generally discouraged. The disable and enable related functions and macros control interrupts through the 4703 custom chip and *not* through the 68000 priority level. In addition, the processor priority level can only be altered from supervisor mode (which means it is a lot less efficient).

It is never necessary to both disable and forbid. Because disable prevents interrupts, it also prevents preemptory task scheduling.

Many Exec lists can only be accessed while disabled. Suppose you want to print the names of all waiting tasks. You would need to access the task list from a disabled section. In addition you must avoid calling certain system functions that require multitasking to function properly (**printf()** for example). In this example we gather the names into a name array first while disabled, then we enable and print the names.

```
Disable();
for (task = execbase -> TaskWait.tc_Node.lh_Head; task -> tc_Node.ln_Succ;
                task = task -> tc_Node.ln_Succ) {
     names[count++] = task -> tc_Node.ln_Name;
}
Enable();

for (i = 0; i < count; i++) {
     printf (" %s ", names[i]);
}
```

Of course this example will have problems if a waiting task is removed before its name is printed. If this were to happen, the name string pointer would no longer be valid. To avoid such problems it is good programming practice to copy the entire name string into a temporary buffer.

Semaphores

Messages and message ports can be used as semaphores for the purposes of mutual exclusion. With this method of locking, all tasks agree on a locking *convention* before accessing shared data structures. Tasks that do not require access are not affected and will run normally, so this type of exclusion is considered preferable to forbidding and disabling. Unfortunately, semaphores also represent a considerable amount of overhead for simple system operations and are not used internal to Exec for efficiency reasons. This form of exclusion is explained in more detail in the "Messages and Ports" chapter.

2.6. EXCEPTIONS

Tasks can specify that certain asynchronous events cause *exceptions*, which are sort of *task-private* interrupts that redirect a task's flow of control. The task essentially suspends what it is doing and enters a special routine to process its exceptional event.

Exceptions are driven by the task signal mechanism described earlier in this chapter. Instead of waiting for a signal to occur, you indicate that it is an exception signal with the **SigExcept()** function. When the signal occurs, your task will be "interrupted" from its normal execution and placed in a special exception handler.

The **tc_ExceptCode** and **tc_ExceptData** task fields are used to establish your exception handler. **tc_ExceptCode** points to the routine that will handle the initial processing of all exceptions. If this field is zero, Exec will ignore all exceptions. The **tc_ExceptData** field can be used to provide a pointer to related data structure.

On entry to your exception code the system passes certain parameters in the processor registers. **D0** contains a signal mask indicating which exception has just occurred, and **A1** points to your related exception data (from **tc_ExceptData**). In addition your previous task context is pushed onto the task's stack. This includes the previous **PC**, **SR**, **D0-D7**, and **A0-A6** registers. You can think of an exception as a subtask outside of your normal task. Because task exception code executes in *user* mode, however, the task stack must be large enough to supply the extra space consumed during an exception.

While processing a given exception, Exec prevents it from occurring recursively. At exit from your exception processing code you should return the same value in **D0** to re-enable that

exception signal. When your task executes the RTS at the end of your handler, the system restores the previous contents of all of your task registers, and resumes the task at the point where it was interrupted by the exception signal. Your exception-processing code determines the order of handling exception signals that occur simultaneously by the order in which you examine the signal bits.

2.7. TRAPS

Task *traps* are synchronous exceptions to the normal flow of program control. They are always generated as a direct result of an operation performed by your program's code. Whether they are accidental or purposely generated, they will result in your program being forced into a special condition in which it must immediately handle the trap. Address error, privilege violation, zero divide, and trap instructions all result in task traps. They may be generated directly by the 68000 processor (Motorola calls them "exceptions") or simulated by software.

A task that incurs a trap has no choice but to respond immediately. The task must have a module of code to properly handle the trap. Your task may be aborted if you get a trap and have not provided a means of handling it.

You may choose to do your own processing of traps. **tc_TrapCode** is the address of the handler that you have designed to process the trap. **tc_TrapData** is the address of the data area for use by your trap handler.

The 68000 traps of interest are:

2	bus error
3	address error
4	illegal instruction
5	zero divide
6	CHK instruction
7	TRAPV instruction
8	privilege violation
9	trace
10	line 1010 emulator

11	line 1111 emulator
32-47	trap instructions

The actual stack frames generated for these traps are processor dependent. The 68010 and 68020 processors will generate a different type of stack frame than the 68000. If you plan on handling your own traps, you should not make assumptions about the format of the supervisor stack frame. Check the flags in the **AttnFlags** field of the **ExecBase** structure for the type of processor in use, and process the stack frame accordingly.

Handlers

For compatibility with the 68000, Exec performs trap handling in supervisor mode. This means that all task switching is disabled during trap handling.

The system stack does, at entry to the task's trap handler, contain the trap frame as defined in the 68000 manual. A long word exception number is added at the bottom of this frame. That is, when a handler gains control the top of stack contains the exception number and the 68000 frame immediately follows.

To return from trap processing, remove the exception number from the stack (note that this is the supervisor stack, not the user stack) and then perform a return from exception (**RTE**).

Because trap processing takes place in supervisor mode, with task dispatching disabled, it is strongly urged that you keep trap processing as short as possible or switch back to user mode from within your trap handler.

If a trap handler already exists when you add your own trap handler, it is smart to propagate any traps that you do not handle down to the previous handler. This can be done by saving the previous **tc_TrapCode** and **tc_TrapData** for use by your handler.

Trap Instructions

The **TRAP** instructions in the 68000 generate traps 32-47. Since many independent pieces of system code may desire to use these traps, the **AllocTrap()** and **FreeTrap()** functions are provided. These work in a fashion similar to **AllocSignal()** and **FreeSignal()** mentioned above.

Allocating traps is simply a bookkeeping job within your task. It does not affect how the system calls your trap handler; it helps coordinate who owns what traps. Exec does nothing to determine whether or not your task is prepared to handle this particular trap. It simply calls your code. It is up to you to properly handle the trap.

To allocate any trap you can use:

```
trap = AllocTrap(-1);
if (trap == -1) {
        printf("all trap instructions are in use");
        return;
}
```

or to select a specific trap:

```
trap = AllocTrap(3);
if (trap == -1) {
        printf("trap #3 is in use");
        return;
}
```

To free a trap you use **FreeTrap()**.

Chapter 3

Messages and Ports

This chapter describes Exec support for inter-system communication in the Amiga kernel. It discusses the structure of messages and ports, message exchange methods, arrival notification actions, and various support functions.

3.1. INTRODUCTION

For inter-system communication, Exec provides a consistent, high performance mechanism of messages and ports. This mechanism is used to pass arbitrary size message structures from task to task, interrupt to task, or task to software interrupt. In addition, messages are often used to coordinate operations between a number of cooperating tasks.

A *message* data structure has two parts: system linkage and message body. The system linkage is used by Exec to attach a given message to its destination. The message body contains the actual data of interest. The message body is any arbitrary data block less than 64K bytes in size.

Messages are always sent to a predetermined destination *port*. At a port incoming messages are queued in a first-in-first-out (FIFO) order. There are no system restrictions on the number of ports or the number of messages that may be queued to a port (other than the amount of available system memory).

Messages are always queued by *reference*. For performance reasons message copying is not performed. In essence, a message between two tasks is a temporary license for the receiving task to use a portion of the memory space of the sending task — that portion being the message itself. This means that if task A sends a message to task B, the message is still part of the task A context; however, task A should not access the message until it has been *replied* (explained below). This technique of message exchange imposes important restrictions on message access.

3.2. PORTS

Ports are rendezvous points where messages are collected. A port may contain any number of outstanding messages from many different originators. When a message arrives at a port, the message is appended to the end of the list of messages for that port, and a pre-specified arrival action is invoked. This action may do nothing, or it may cause a pre-defined task signal or software interrupt (see the "Interrupts" chapter).

As with many Exec structures, ports may be given a symbolic name. This is particularly useful for tasks that must rendezvous with dynamically created ports. It is also useful for debugging purposes.

Structure

A message port consists of a **MsgPort** structure as defined in the *exec/ports.h* and *exec/ports.i* include files. The C structure for a port is:

```
struct MsgPort {
    struct   Node mp_Node;
    UBYTE    mp_Flags;
    UBYTE    mp_SigBit;
    struct   Task *mp_SigTask;
    struct   List mp_MsgList;
};
```

where

mp_Node
: is a standard **Node** structure. This is useful for tasks which might want to rendezvous with a particular message port by name.

mp_Flags
: are used to indicate message arrival actions. See the explanation below.

mp_SigBit
: is the signal bit *number* when a port is used with the task signal arrival action.

mp_SigTask
: is a pointer to the task to be signaled, or if a software interrupt arrival action is specified this is a pointer to the interrupt structure.

mp_MsgList
the list header for all messages queued to this port. (See the "Lists" chapter).

The **mp_Flags** field contains a sub-field indicated by the **PF_ACTION** mask. This sub-field specifies the message arrival action that occurs when a port receives a new message. The possibilities are:

PA_SIGNAL
signal the specified task on the arrival of a new message. Every time a message is put to the port another signal will occur regardless of how many messages have been queued to the port.

PA_SOFTINT
cause the specified software interrupt. Just like with **PA_SIGNAL**, every message will cause the software interrupt to again be posted.

PA_IGNORE
perform no operation other than queuing the message. This action is often used to stop signaling or software interrupts without disturbing the contents of the **mp_SigTask** field.

It is important to realize that a port's arrival action will occur for each new message queued, and that there is not a one-to-one correspondence between messages and signals. Task signals are only single-bit flags so there is no record of how many times a particular signal occurred. There may be many messages queued and only a single task signal. All of this has certain implications when designing code that deals with these actions. Your code should not depend on receiving a signal for every message at your port. All of this is also true for software interrupts.

Creation

To create a new message port you must allocate and initialize a **MsgPort** structure. If you desire to make the port *public* you will also need to call the **AddPort()** function.

Port structure initialization involves setting up a **Node** structure, establishing the message arrival action with its parameters, and initializing the list header.

The following example of port creation is equivalent to the **CreatePort()** function as supplied in *amiga.lib*:

```
extern APTR AllocMem();
extern UBYTE AllocSignal();
extern struct Task *FindTask();

struct MsgPort *
CreatePort (name, pri)
   char *name;
   BYTE  pri;
{
      UBYTE sigBit;
      struct MsgPort *port;

      if ((sigBit = AllocSignal (-1)) == -1)
            return ((struct MsgPort *) 0);

      port = AllocMem (sizeof(*port), MEMF_CLEAR | MEMF_PUBLIC);
      if (port == 0) {
            FreeSignal (sigBit);
            return ((struct MsgPort *) (0));
      }

      port->mp_Node.ln_Name = name;
      port->mp_Node.ln_Pri = pri;
      port->mp_Node.ln_Type = NT_MSGPORT;

      port->mp_Flags = PA_SIGNAL;
      port->mp_SigBit = sigBit;
      port->mp_SigTask = FindTask (0);

      if (name != 0)
            AddPort (port);
      else
            NewList (&port->mp_MsgList);

      return (port);
}
```

Deletion

Before deleting a message port, all outstanding messages from other tasks must be returned. This is done by replying to each message until the message queue is empty. Of course there is no need to reply to messages owned by the current task (the task performing the port deletion).

Public ports attached to the system with **AddPort()** must be removed from the system with **RemPort()**.

Rendezvous

The **FindPort()** function provides a means of finding the address of a public port given its symbolic name. For example, **FindPort("Spyder")** will return either the address of the message port or a zero indicating that no such public port exists. Names should be made rather unique to prevent collisions among multiple applications. It is a good idea to use your application name as a prefix for your port name.

3.3. MESSAGES

As mentioned earlier, a message contains both system header information and the actual message content. The system header is of the **Message** form defined in *exec/ports.h* and *exec/ports.i*. In C this structure is:

```
struct Message {
    struct    Node mn_Node;
    struct    MsgPort *mn_ReplyPort;
    UWORD     mn_Length;
};
```

where

mn_Node
is a standard **Node** structure used for port linkage.

mn_ReplyPort
is used to indicate a port to which this message will be returned when a reply is necessary.

mn_Length
indicates the length of the message body in bytes.

This structure is always attached to the head of all messages. Assume that you want a message structure which contains the x and y coordinates of a point on the screen. It could be defined as:

```
struct XYMessage {
    struct    Message xy_Msg;
    UWORD x,y;
}
```

For this structure the **mn_Length** field would be set to two times the size of UWORD, or four bytes.

Putting a Message

A message is delivered to a given destination port with the **PutMsg()** function. The message is queued to the port, and that port's arrival action is invoked. If the action specifies a task signal or a software interrupt, the originating task may temporarily lose the processor while the destination processes the message.

If you require a reply to the message, the **mn_ReplyPort** field must be setup prior to the call to **PutMsg()**.

Here is a simple example of putting a message to a public port:

```
struct MsgPort *port, *replyport;
struct XYMessage *xymsg;

xymsg = AllocMem (sizeof(*xymsg), MEMF_PUBLIC);
if (msg == 0) {
      printf ("not enough memory for message");
      return;
}

replyport = CreatePort ("xyreplyport",0);  /* as defined earlier in this chapter */
if (replyport == 0) {
      printf ("could not create the reply port");
      FreeMem (msg, sizeof(*xymsg));
      return;
}

xymsg -> xy_Msg.mn_Node.ln_Type = NT_MESSAGE;
xymsg -> xy_Msg.mn_ReplyPort = replyport;

port = FindPort ("Spyder");
if (port == 0) {
      printf ("Spyder port not found");
      return;
}

PutMsg (port, xymsg);
```

Waiting for a Message

A task may go to sleep waiting for a message to arrive at one or more ports. This technique is widely used on the Amiga as a general form of event notification. For example, it is used extensively by tasks for I/O request completion.

Waiting for the arrival of a message requires that the message port be properly initialized. In particular, the **mp_SigTask** field must contain the address of the task to be signaled and **mp_SigBit** must contain a pre-allocated signal number (as described in the "Tasks" chapter).

You can call the **WaitPort()** function to wait for a message to arrive at a port. This function will return the first message queued to a port. If the port is empty, your task goes to sleep waiting for the first message. If the port is not empty, then your task will not go to sleep.

A more general form of waiting for a message involves the use of the **Wait()** function (see the "Tasks" chapter). This function waits for task event signals directly. If the signal assigned to the message port occurs, the task will awaken. Using the **Wait()** function is more general because you can wait for more than just a single message port. For example, you may want to wait for a message and a timeout signal. The **Wait()** function lets you specify a mask containing the signals associated with your message port and your timeout signal.

Here's an example using **WaitPort()**:

```
signal = AllocSignal (-1);
if (signal == -1) {
      printf ("no free signal bits");
      return;
}

port -> mp_Flags |= PA_signal;
port -> mp_SigBit = signal;
port -> mp_SigTask = FindTask (0);          /* self */

msg = WaitPort (port);
```

Note that **WaitPort()** only returns a pointer to the first message in a port. It does not actually remove the message from the port queue. That's described in the next section.

Getting a Message

Messages are usually removed from ports with the **GetMsg()** function. This function removes the next message at the head of the port queue and returns a pointer to it. If there are no messages in a port, this function returns a zero.

The example below illustrates the use of **GetMsg()** to print the contents of all messages in a port:

```
while ((msg = GetMsg (port)) != 0) {
      printf ("x=%ld y=%ld", msg->x, msg->y);
}
```

Certain messages may be more important than other messages. Because ports impose FIFO ordering, these important messages may get queued behind other messages regardless of their priority. If it is necessary to recognize more important messages, it is easiest to create

another port for these special messages.

Replying

When the operations associated with receiving a new message are finished, it is usually necessary to send the message back to the originator. This is important because it notifies the originator that the message can be reused or deallocated.

The **ReplyMsg()** function is provided to serve this purpose. It will return the message to the port specified in the **mn_ReplyPort** field of the message. If this field is zero, then no reply is done.

The previous example can be enhanced to reply to each of its messages:

```
while ((msg = GetMsg (port)) != 0) {
    printf ("x=%ld y=%ld", msg->x, msg->y);
    ReplyMsg (msg);
}
```

Notice that the reply doesn't occur until *after* the message values have been used.

Often the operations associated with receiving a message involve returning *results* to the originator. Typically this is done within the message itself. The receiver places the results in fields defined (or perhaps reused) within the message body before replying the message back to the originator. Receipt of the replied message back at the originator's reply port indicates it is once again safe for the originator to use or change the values found within the message.

Chapter 4

I/O

This chapter presents the key concepts that must be understood before performing input and output on system devices. It describes the standard form of an IO Request, device interface functions, Exec support functions, standard device commands, and how to actually perform I/O on the Amiga. This chapter does not discuss how to create your own device driver. Appendix K contains source assembler code for a disk-resident device driver with its own task to handle I/O requests.

4.1. INTRODUCTION

One of the primary purposes of Exec is to provide a standard form for all device I/O. This includes the definition of a standard device interface, the format for I/O requests, and the establishment of rules for normal device/task interaction. In addition, the guidelines for non-standard device I/O are also defined. In the design of the Amiga I/O system great care has been taken to avoid dictating the form of implementation or the internal operational characteristics of a device.

A *device* in its purest sense is an abstraction that represents a set of well defined interactions with some form of physical media. This abstraction is supported by a standard Exec data structure and an independent system code module. The data structure provides the external interface and maintains the current device state. The code module supplies the operations necessary to make the device functional. (In many operating systems this code module is referred to as a device *driver*).

A device *unit* is an instance of a device. It shares the same device data structure and code module with all other units of the same device; however, it operates in an independent fashion. Often units correspond to separate physical sub-systems of the same general device class. For example, each Amiga floppy disk drive is an independent unit of the same device. There is only one device data structure and one code module to support all of these units.

Exec I/O is often performed using the message system described in the previous chapter.

often Most aspects of message passing are concealed within the Exec I/O support routines. However, it is important to realize that I/O request blocks, once issued, must *not* be modified or reused until they are returned to your control by Exec.

4.2. REQUEST STRUCTURE

An I/O *request* is always directed to a device unit. This request is organized as a control block and contains a *command* to be performed on a specified unit. It is passed through a standard device interface function where it is then processed and executed by the device's code module. All request parameters are included in the request control block, and the I/O request results are also returned in the same control block.

Every device unit responds to a standard set of commands, and may optionally provide a non-standard set of commands as well. The standard commands are reset, read, write, update, clear, stop, start, and flush. They will be explained later in this chapter. Non-standard commands are discussed in the documentation pertaining to the particular device of interest.

An I/O request always includes at least an **IORequest** data structure. This is a standard header used for all I/O requests. It is defined in the *exec/io.h* and *exec/io.i* include files as:

```
struct IORequest {
    struct   Message io_Message;
    struct   Device  *io_Device;
    struct   Unit*io_Unit;
    UWORDio_Command;
    UBYTE io_Flags;
    BYTE   io_Error;
};
```

where

io_Message
: is a message header (see "Messages and Ports" chapter). This is used by the device to return your I/O request upon completion. It is also used by devices internally for I/O request queuing. This header must be properly initialized for I/O to work correctly.

io_Device
: is a pointer to the device data structure node. This field is automatically setup by an Exec function when the device is opened.

io_Unit
specifies a unit to the device internally. This is a device *private* field and should not be accessed by the user. The format of this field is device dependent and is setup by the device during the open sequence.

io_Command
is the command requested. This may be either one of the system standard commands or a device specific command.

io_Flags
is used to indicate special request options and state. It is divided into two subfields of four bits each. The lower four bits are for use by Exec and the upper four bits are available to the device.

io_Error
is an error or warning number returned upon request completion.

The **io_Device**, **io_Unit**, and **io_Command** fields are not affected by the servicing of the request. This permits repeated I/O using the same request.

The standard I/O requests use an expanded form of the **IORequest** structure:

```
struct IOStdReq {
    struct    Message  io_Message;
    struct    Device   *io_Device;
    struct    Unit *io_Unit;
    UWORD  io_Command;
    UBYTE  io_Flags;
    BYTE   io_Error;
    ULONG  io_Actual;
    ULONG  io_Length;
    APTR   io_Data;
    ULONG  io_Offset;
}
```

where the additional fields

io_Actual
indicates the actual number of bytes transferred. This field is only valid upon completion.

io_Length
is the requested number bytes to transfer. This field must be setup prior to the request. A special length of -1 is often used to indicate variable length transfers.

io_Data
is a pointer to the transfer data buffer.

io_Offset
indicates a byte offset (for structured devices). For block structured devices (like the floppy-disk) this number must be a multiple of the block size.

Devices with non-standard commands may add their own special fields to the I/O request structure as needed. Such extensions are device specific.

4.3. INTERFACE FUNCTIONS

Four Exec functions are responsible for interfacing IO requests to actual device drivers. These functions operate independently of the particular device command requested. They deal with the request block as a whole, ignoring its command and its command parameters.

DoIO() is the most commonly used I/O function. It initiates an IO request and waits for its completion. This is a *synchronous* form of device I/O; control is not returned to the caller until completion.

SendIO()
is used to initiate an IO request without waiting for completion. This is an *asynchronous* form of device I/O; control is returned even if the request has not completed.

WaitIO()
is used to wait for the completion of a previous initiated asynchronous IO request. This function will not return control until the request has completed (either successfully or not).

CheckIO()
is used to see if an asynchronous I/O request has completed.

In addition to the above Exec functions there are two I/O related functions that are actually direct entries into the device driver itself. These functions are part of the actual device driver interface to the system, and should be used with care. They incur slightly less overhead but require more knowledge of the I/O system internals (how QuickIO works for instance):

BeginIO()
initiates an IO request. The request will be synchronous or asynchronous depending on the device driver.

AbortIO()
attempts to cancel a previous I/O request. This function is easily accessed as an assembly code macro **ABORTIO**, or through the C library Exec support function **AbortIO()**.

4.4. STANDARD COMMANDS

There are eight standard commands to which all devices are expected to respond. If the device is not capable of performing one of these commands, it will at least return an error indication that the command is not supported. These commands are defined in the *exec/io.h* and *exec/io.i* include files.

CMD_RESET
reset the device unit. This command competely initializes the device unit, returning it to its default configuration, aborting all of its pending I/O, cleaning up any internal data structures, and reseting any related hardware.

CMD_READ
read a specified number of bytes from a device unit into the data buffer. The number of bytes to read is specified in the **io_Length** field. The number of bytes actually read is returned in the **io_Actual** field.

CMD_WRITE
write a specified number of bytes to a device unit from a data buffer. The number of bytes to write is specified in the **io_Length** field. The number of bytes actually written is returned in the **io_Actual** field.

CMD_UPDATE
force out all internal buffers. This command will cause device internal memory buffers to be written out to the physical device unit. A device will transparently perform this operation when necessary, but this command causes it to occur explicitly. It is useful for devices that maintain internal caches, such as the floppy disk device.

CMD_CLEAR
clear all internal buffers. This command will delete the entire content of a device unit's internal buffers. No update is performed; all data is lost.

CMD_STOP
: immediately stop the device unit. This command stops a device unit at its first opportunity. All I/O requests continue to queue, but the device unit stops servicing them. This command is useful for devices that may require user intervention (printers, plotters, data networks, etc).

CMD_START
: continue after a previous stop command. The device resumes from where it was stopped.

CMD_FLUSH
: abort all I/O requests. This command will return all pending I/O requests with an error.

4.5. PERFORMING I/O

I/O in Exec is always performed using I/O request blocks. Before performing I/O the request block must be properly initialized by both the system and the user. Once this has been done, normal I/O may commence.

Preparation

Devices are identified within the system by name (a null-terminated character string). Device units are usually identified by number. The **OpenDevice()** function maps both the device name to an actual device and then calls the device to perform its initialization. The device will map the unit number into an internal form for later use. Both Exec and the device driver will initialize the I/O request passed to **OpenDevice()**.

For example, **OpenDevice("trackdisk.device",1,request,0)** will attempt to open unit one of the floppy disk device, mapping its symbolic name into the address of a device data structure. It also sets up a few internal fields of the request. **OpenDevice()** will return a zero if it was successful, or a nonzero error number if it was not.

Synchronous Requests

Synchronous I/O requests are initiated with the **DoIO()** function mentioned earlier. **DoIO()** will not return control until the request has completed. Since the device may respond to a request immediately or queue it for later action, an undetermined amount of time may pass before control is returned.

To perform synchronous I/O requires that you prepare the I/O request block as described in the previous section. In addition you must initialized the **io_Message**, **io_Command**, and perhaps other fields.

The **io_Message** field is setup in the same manner as a message. This is described in the "Messages and Ports" chapter.

The **io_Command** field is set to the desired command. For example:

```
request->io_Command = CMD_RESET;
DoIO (request);
```

performs a reset command.

More involved commands require yet other fields to be initialized. To read a sector from a disk might look something like:

```
request->io_Command = CMD_READ;
request->io_Length = TD_SECTOR;
request->io_Offset = 20 * TD_SECTOR;
request->io_Data = buffer;
DoIO (request);
```

When the request has completed the request block is returned with the command results. If an error occurred, **DoIO()** will return the error number. The error number is also indicated in the **io_Error** field of the request.

With this type of I/O only one request is serviced at a time. Sometimes it is necessary to perform many requests at the same time. This is the subject of the next section.

Asynchronous Requests

More efficient programs can take advantage of the multitasking characteristics of the I/O system by using asynchronous I/O. This type of I/O is supported by the **SendIO()**, **WaitIO()**, **CheckIO()**, **BeginIO()**, and **AbortIO()** functions.

Asynchronous I/O requests will return almost immediately to the user regardless of whether the request has actually completed. This lets the user maintain control while the I/O is being performed. Multiple I/O requests can be posted in this fashion.

In the disk read example above, asynchronous I/O could be performed by changing the **DoIO()** call to a **SendIO()**.

```
request->io_Command = CMD_READ;
request->io_Length = TD_SECTOR;
request->io_Offset = 20 * TD_SECTOR;
request->io_Data = buffer;
SendIO (request);
```

From the time the I/O has been initiated to the time it completes the request block should not be directly accessed by the user. The device can be said to "own" the request block. Only after the request has completed or successfully aborted should you access it.

When the I/O completes, the device will return the I/O request block to the reply port specified in its **io_Message** field. When this has happened you know that the device has finished the I/O.

The reply port used to receive the returned request can be setup to cause a task signal when the reply arrives. This technique lets a task sleep until the the request is complete. The **WaitIO()** can be called to wait for the completion of a previously initiated request.

WaitIO() will handle all of the interaction with the message reply port automatically. If you are using just the **Wait()** function, do not forget to remove the I/O request from your reply port with **GetMsg()**. Once this is done, the request may be reused.

The **CheckIO()** function is handy to determine if a particular I/O request has been satisfied. This function deals with some of the subtleties of I/O in the proper manner.

If you wish to queue several I/O requests to a device, you must issue multiple **SendIO()** requests, each with its own separately-opened request structure. This type of I/O is supported by most devices. A task can also request I/O from a number of devices then check later for their completion.

Exec also allows for certain types of optimization in device communication. One form of

optimization, in which you call the device driver directly, is called "quickIO". This concept is discussed later in this chapter.

Conclusion

When a request has completed its I/O, access to the device should be concluded with **CloseDevice()**. This function will inform the device that no further I/O is to be performed with this request. For every **OpenDevice()** there must be a corresponding **CloseDevice()**.

Quick I/O

For some types of I/O the normal internal mechanisms of I/O may present a large amount of overhead. This is mostly true for character oriented I/O, where each characters might be transferred with a separate I/O request. The overhead for such requests could significantly overload the I/O system, resulting in a loss of efficiency for the system overall.

To allow devices to optimize their I/O handling a mechanism of *QuickIO* was created. In the **IORequest** data structure one of the **io_flags** is reserved for QuickIO. When set prior to an I/O request, this flag indicates that the device is allowed to handle the I/O in a special manner. This enables some devices to take certain "short-cuts" when it comes to performing and completing the request.

The **DoIO()** function normally requests the QuickIO option, whereas the **SendIO()** function does not. Complete control over the mode for QuickIO is possible by calling a device's **BeginIO()** entry directly.

It is up to the device to determine whether it can actually handle a request marked as QuickIO. When the request has completed, if the QuickIO flag is still set, then the I/O was performed quickly. This means that no message reply occurred, so the message has not been queued to the reply port.

4.6. STANDARD DEVICES

The following standard system devices are normally available at boot-time. Each of these devices is described in a separate chapter in Part 3 of this manual.

Timer — Provides a flexible way of causing task signals or interrupts at second and microsecond intervals.

TrackDisk — Provides direct access to the 3 1/2-inch floppy disks. Among the functions provided are format, seek, read, and write. Normally trackdisk is only used by AmigaDos, however its functions are enumerated here for direct access where required. Note that the trackdisk driver is associated with the disk.resource.

Keyboard — Handles raw information from the keyboard and converts it into input events which you can retrieve and interpret. Keyboard input events are queued so that you won't miss any keystrokes.

Gameport — Handles raw information from the mouse, or a joystick device. Gameport events are queued so that you won't miss any movements. You can tell the system what type of device is connected, and also tell it how often to check and report the current status of the device.

Input — The console device, as an input device, combines both the keyboard and the gameport device. Input events from both are merged together into a single input event stream on a first-in first-out basis.

Console — The console device receives its input from the input device. The input portion of the console device is simply a handler for input events filtered by Intuition. It provides what one might call the "traditional" user interface.

Audio — The audio device is provided to control the use of the audio channels. A separate chapter in this manual is dedicated to the audio device.

Narrator — The narrator device is loaded from disk and uses the audio device to produce human-like synthesized speech. The "Narrator Device" chapter also describes the text-to-phoneme routines in the translator library.

Serial — The serial device is loaded from disk and initialized on being loaded. It controls serial communications buffering of the input/output, baud rate and so on.

Parallel — The parallel device is loaded from disk and initialized on being loaded. It controls parallel communications. The parallel device is most often used by a parallel printer driver.

Printer — The printer device driver is loaded from disk. Printers that are supported as of this writing are specified in appendix I. In addition to showing how to use the device, the "Printer Device" chapter describes the creation of a printer driver. Source code for four different printer drivers is also included (in appendix I).

Chapter 5

Interrupts

This chapter discusses the software interface to interrupts. It describes the normal interrupt sequence of events, interrupt priorities, interrupt handlers, interrupt servers, software interrupts and interrupt exclusion.

5.1. INTRODUCTION

Exec manages the decoding, dispatching, and sharing of all system interrupts. This includes control of hardware interrupts, software interrupts, task-relative interrupts (see the "Tasks" chapter), and interrupt disabling/enabling. In addition, Exec supports a more extended prioritization of interrupts than that provided in the 68000.

The proper operation of multitasking depends heavily on the consistent management of the interrupt system. Task activities are often driven by inter-system communication originated by various interrupts.

Sequence of Events

Before useful interrupt handling code can be executed, a considerable amount of hardware and software activity must occur. Each interrupt must propagate through several hardware and software interfaces before application code is finally dispatched.

1. A hardware device decides to cause an interrupt, and sends a signal to the interrupt control portions of the 4703 custom chip.

2. The 4703 interrupt control logic notices this new signal, and performs two primary operations. First, it records that the interrupt has been requested by setting a flag bit in the INTREQ register. Second, it examines the INTENA register to determine whether the corresponding interrupt and the interrupt master are enabled. If both are enabled, the 4703 generates a set of three 68000 interrupt request signals. See the *Amiga Hardware Reference Manual* for a more complete explanation of how this is done.

3. These three signals correspond to 7 interrupt priority levels in the 68000. If the priority of the new interrupt is *greater* than the current processor priority, an interrupt sequence is initiated. The priority level of the new interrupt is used to index into the top 7 words of the processor address space. The odd byte (a vector number) of the indexed word is fetched and then shifted left by two to create a low memory vector address.

4. The 68000 then switches into *supervisor* mode (if it isn't in that mode already), and saves copies of the status register and program counter (PC) onto the top of the *system* stack. The processor priority is then raised to the level of the active interrupt.

5. From the low memory vector address (calculated in step three above), a 32-bit *autovector* address is fetched and loaded into the program counter. This is an entry point into Exec's interrupt dispatcher.

6. Exec must now further decode the interrupt by examining the INTREQ and INTENA 4703 chip registers. Once the active interrupt has been determined, Exec indexes into an **ExecBase** array to fetch the interrupt's handler entry point and handler data pointer addresses.

7. Exec now turns control over to the interrupt handler by calling it as if it is a subroutine. This handler may deal with the interrupt directly, or propagate control further by invoking interrupt server chain processing.

You can see from the above discussion that the interrupt autovectors *should never be altered by the user*. If you wish to provide your own interrupt handler, you must use the Exec **SetIntVector()** function. To change the content of any autovector location violates the design rules of the Multitasking Executive.

Task multiplexing usually occurs as the result of an interrupt. When an interrupt has finished and the processor is about to return to user mode, Exec determines whether task scheduling attention is required. If a task was signaled during interrupt processing, then the task scheduler will be invoked.

Since Exec uses pre-emptive task scheduling, it can be said that the interrupt subsystem is the heart of task multiplexing. If for some reason interrupts do not occur, then a task might execute forever because it cannot be forced to relinquish the CPU.

Interrupt Priorities

Interrupts are prioritized in hardware and software. The 68000 CPU priority at which an interrupt executes is determined strictly by hardware. In addition to this, the software imposes a finer level of *pseudo-priorities* on interrupts with the same CPU priority. These pseudo-priorities determine the order in which simultaneous interrupts of the same CPU priority are processed. Multiple interrupts with the same CPU priority but a different pseudo-priority will not interrupt one another.

This table summarizes all interrupts by priority:

Table 5-1: Interrupts by Priority

4703 Name	CPU Priority	Pseudo Priority	Purpose
NMI	7	15	nonmaskable
INTEN	6	14	special (copper)
EXTER	6	13	8520B, external6
DSKSYNC	5	12	disk byte
RBF	5	11	serial input
AUD1	4	10	audio channel 1
AUD3	4	9	audio channel 3
AUD0	4	8	audio channel 0
AUD2	4	7	audio channel 2
BLIT	3	6	blitter done
VERTB	3	5	vertical blank
COPER	3	4	copper
PORTS	2	3	8520A, external2
TBE	1	2	serial output
DSKBLK	1	1	disk block done
SOFTINT	1	0	software interrupts

As described in the Motorola 68000 programmer's manual, interrupts may only nest in the direction of higher priority. Because of the time-critical nature of many interrupts on the Amiga, the CPU priority level *must never be lowered* by user or system code. When the system is running in user mode (multitasking) the CPU priority level must remain set at zero. When an interrupt occurs, the CPU priority is raised to the level appropriate for that interrupt. Lowering the CPU priority would permit unlimited interrupt recursion on the system stack and would "short-circuit" the interrupt priority scheme.

Because it is dangerous on the Amiga to hold off interrupts for any period of time, higher level interrupt code must perform its business and exit promptly. If it is necessary to perform a time-consuming operation as the result of a high priority interrupt, the operation should be deferred to a lower priority by using a *software interrupt.* In this way interrupt response time is kept to a minimum. Software interrupts are described in a later section.

Non-maskable Interrupt

The 68000 provides a non-maskable interrupt (NMI) of CPU priority 7. Although this interrupt cannot be generated by the Amiga hardware itself, it can be generated on the expansion bus by external hardware. Since this interrupt does not pass through the 4703 interrupt controller circuitry, *it is capable of violating system code critical sections.* In particular it short-circuits the **DISABLE** mutual-exclusion mechanism. Code that uses NMI must not assume that it can access system data structures.

5.2. SERVICING INTERRUPTS

Interrupts are serviced on the Amiga through the use of interrupt *handlers* and *servers.*

An interrupt handler is a system routine that exclusively handles all processing related to a particular 4703 interrupt.

An interrupt server is one of possibly many system routines that get invoked as the result of a single 4703 interrupt. Interrupt servers provide a means of interrupt sharing. This concept is useful for general purpose interrupts like vertical blanking.

At system start, Exec designates certain 4703 interrupts as handlers and others as server chains. The PORTS, COPER, VERTB, BLIT, EXTER, and NMI interrupts are initialized as server chains; hence, each of these may execute multiple interrupt routines per each interrupt. All other interrupts are designated as handlers and are always used exclusively.

Data Structure

Interrupt handlers and servers are defined by the Exec **Interrupt** structure. This structure specifies an interrupt routine entry point and data pointer. The C definition of this structure is:

```
struct Interrupt {
   struct  Node is_Node;
   APTR is_Data;
   VOID  (*is_Code)();
};
```

Once this structure has been properly initialized, it can be used for either a handler or server.

Environment

Interrupts execute in an environment unique from that of tasks. All interrupts execute in *supervisor mode* and utilize a single *system stack*. This stack is large enough to handle extreme cases of nested interrupts (of higher priorities). Obviously, interrupt processing has no effect on task stack usage.

All interrupt processing code, both handlers and servers, is invoked as assembly code subroutines. Normal assembly code CPU register conventions dictate that the D0, D1, A0, and A1 registers are free for scratch use. In the case of an interrupt handler, some of these registers also contain data which may be useful to the handler code. See the section on handlers below.

Because interrupt processing executes outside the context of most system activities, certain data structures will not be self-consistent and must be considered off limits for all practical purposes. This happens because certain system operations *are not atomic* in nature and may be interrupted after only executing part of an important instruction sequence. Take the memory allocation and deallocation routines as an example. These routines disable task switching but do not disable interrupts. This results in the finite possibility of interrupting a memory related routine. In such a case, a memory linked list may be inconsistent when examined from the interrupt code itself. To avoid serious problems, the interrupt routine must not use any of the memory allocation or deallocation functions.

Interrupt Handlers

As described above an interrupt handler is a system routine that exclusively handles all processing related to a particular 4703 interrupt. There can only be one handler per 4703 interrupt.

Every interrupt handler consists of an **Interrupt** structure (as defined above and a single assembly code routine. Optionally, a data structure pointer may also be provided. This is particularly useful for ROM-resident interrupt code.

An interrupt handler is passed control as if it were a subroutine of Exec. Once the handler has finished its business it must return to Exec by executing an **RTS** (return from subroutine) instruction rather than an **RTE** (return from exception) instruction.

Interrupt handlers should be kept very short to minimize service time overhead and thus minimize the possibilities of interrupt overruns.

As described above, an interrupt handler has the normal scratch registers at its disposal. In addition **A5** and **A6** are also free for use. These registers are saved by Exec as part of the interrupt initiation cycle.

For the sake of efficiency, Exec passes certain register parameters to the handler. These register values may be utilized to trim a few microseconds off the execution time of a handler.

D0 is scratch and contains garbage.

D1 is scratch but contains the 4703 INTENAR and INTREQR registers values ANDed together. This results in an indication of which interrupts are enabled *and* active.

A0 points to the base address of the Amiga custom chips. This is useful for performing indexed instruction access to the chip registers.

A1 points to the data area specified by the **is_Data** field of the **Interrupt** structure. Since this pointer is always fetched (regardless of whether you use it), it is to your advantage to make some use of it.

A5 is used as a vector to your interrupt code. It is free to be used as a scratch register, and it is not necessary to restore its value prior to returning.

A6 points to the Exec Library base (SysBase). You may use this register to call Exec functions or set it up as a base register to access your own library or device. It is *not* necessary to restore this register prior to returning.

Interrupt handlers are established by passing the Exec function **SetIntVector()** your initialized **Interrupt** structure and the 4703 interrupt bit number of interest. See the appendix for a complete description of this function. Keep in mind that certain interrupts are established as server chains and *should not be accessed as handlers.*

Here is a C code example of proper handler initialization and setup:

```
struct Interrupt *RBFInterrupt, *PriorInterrupt;

setup()
{
     extern void RBFHandler();
     short  *Buffer;

     /* allocate an Interrupt node structure: */
     RBFInterrupt = AllocMem (sizeof(struct Interrupt *), MEMF_PUBLIC);
     if (RBFInterrupt == 0) {
          printf ("not enough memory for interrupt handler");
          exit (100);
     }

     /* allocate an input buffer: */
     Buffer = AllocMem (512, MEMF_PUBLIC);
     if (Buffer == 0) {
          FreeMem (RBFInterrupt, sizeof(struct Interrupt *));
          printf ("not enough memory for data buffer");
          exit (100);
     }

     /* initialize the Interrupt node: */
     RBFinterrupt->is_Node.ln_Type = NT_INTERRUPT;
     RBFinterrupt->is_Node.ln_Pri = 0;
     RBFinterrupt->is_Node.ln_Name = "RBF-example";
     RBFInterrupt->is_Data = Buffer;
     RBFInterrupt->is_Code = RBFHandler;

     /* put the new interrupt handler into action: */
     PriorInterrupt = SetIntVector (INTB_RBF, RBFInterrupt);

     if (PriorInterrupt != 0) {
          printf ("we just replaced the %s interrupt handler",
                    PriorInterrupt->is_Node.ln_Name);
     }
}
```

In this example note the correct initialization of the **Node** structure.

The external interrupt handler code used above, **RBFHandler**, grabs the input character from the serial port and stores it into the buffer. Notice that the address of the buffer is

passed to the handler (shown below) via the **is_Data** pointer. This pointer is updated for every character stored.

```
        XDEF    _RBFHandler

_RBFHandler:
                move.l    (a1),a5
                move.w    serdatr(a0),(a5)+
                move.w    #INTF_RBF,intreq(a0)
                move.l    a5,(a1)
                rts
```

In this example the buffer holds complete 4703 serial data *words* which contain not only the input character, but special serial input flags as well (e.g. data overrun). This data word is deposited directly into the buffer, and the 4703 RBF interrupt request is cleared.

A more sophisticated example might perform various tests on the input word prior to storing it into the buffer.

Interrupt Servers

As mentioned above, an interrupt server is one of possibly many system interrupt routines that get invoked as the result of a single 4703 interrupt. Interrupt servers provide an essential mechanism for interrupt sharing.

Interrupt servers must be used used for PORTS, COPER, VERTB, BLIT, EXTER, or NMI interrupts. For these interrupts, all servers are linked together in a chain. Every server in the chain will be called until one returns a value of **TRUE** (nonzero) in register D0 or until the end of the chain is reached. Normally interrupt servers return a value of zero in D0 which indicates that the chain should not be *prematurely terminated.*

The same Exec **Interrupt** structure used for handlers is also used for servers. Also, like interrupt handlers, servers must terminate their code with an **RTS** instruction.

Interrupt servers are called in priority order. The priority of a server is specified in its **is_Node.ln_Pri** field. Higher priority servers get called earlier than lower priority servers.

Adding and removing interrupt servers from a particular chain is accomplished with the Exec **AddIntServer()** and **RemIntServer()** functions. These functions require you to specify both the 4703 interrupt number and a properly initialized **Interrupt** structure.

Servers have different register values passed than handlers. A server cannot count on the **D0**, **D1**, or **A6** registers containing any useful information. A server is free to use **D0-D1**

and **A0-A1/A5** as scratch.

In a server chain the interrupt is cleared automatically by the system. It is not recommended (and not necessary) that a server clear its interrupt (clearing could cause the loss of an interrupt on PORTS or EXTERN.

Here is an example of a program to setup and cleanup a low priority vertical blank interrupt server:

```
struct Interrupt *VertBIntr;
long count;

main()
{
   extern void VertBServer();

   /* allocate an Interrupt node structure: */
   VertBIntr = AllocMem (sizeof(struct Interrupt *), MEMF_PUBLIC);
   if (VertBIntr == 0) {
      printf ("not enough memory for interrupt server");
      exit (100);
   }

   /* initialize the Interrupt node: */
   VertBIntr->is_Node.ln_Type = NT_INTERRUPT;
   VertBIntr->is_Node.ln_Pri = -60;
   VertBIntr->is_Node.ln_Name = "VertB-example";
   VertBIntr->is_Data = &count;
   VertBIntr->is_Code = VertBServer;

   /* put the new interrupt server into action: */
   AddIntServer (INTB_VERTB, VertBIntr);

   while (getchar () != 'q');    /* wait for user to type 'q' */

   RemIntServer (INTB_VERTB, VertBIntr);
   printf ("%ld vertical blanks occurred", count);
   FreeMem (VertBIntr, sizeof(struct Interrupt *));
}
```

The **VertBServer** might look something like:

```
        XDEF    _VertBServer

_VertBServer:
                move.l    (a1),a0    get address of count
                addq.l    #1,(a0)    bump value of count
                moveq.l   #0,d0      continue server chain
                rts
```

5.3. SOFTWARE INTERRUPTS

Exec provides a means of generating *software interrupts*. This type of interrupt is useful for creating special purpose asynchronous system contexts. Software interrupts execute at a priority higher than that of tasks but lower than normal interrupts, so they are often used to defer normal interrupt processing to a lower priority.

Software interrupts use the same **Interrupt** data structure as normal hardware interrupts. As described above, this structure contains pointers to both interrupt code and data.

A software interrupt is usually activated with the **Cause()** function. If this function is called from a task, the task will be interrupted and the software interrupt will occur. If it is called from a hardware interrupt, the software interrupt will not be processed until the system exits from its last hardware interrupt. If a software interrupt occurs from within another software interrupt, it does not get processed until the current one completes.

Software interrupts are prioritized. Unlike interrupt servers, there are only five priority levels for software interrupts: -32, -16, 0, +16, and +32. The priority should be put into the **ln_Pri** field prior to calling **Cause()**.

Software interrupts can also be caused by message port arrival actions. See the "Messages and Ports" chapter.

5.4. DISABLING INTERRUPTS

As mentioned in the "Tasks" chapter, it is sometimes necessary to disable all interrupts when examining or modifying certain shared system data structures.

Interrupt disabling is controlled with the **DISABLE** and **ENABLE** macros and the **Disable()** and **Enable()** C functions.

In some system code, there are nested disabled sections. This type of code requires that interrupts be disabled with the first **DISABLE** and not re-enabled until the *last* **ENABLE**. The system enable/disable macros and functions are designed to permit this sort of nesting.

For example, if there is a section of system code that should not be interrupted, the **DISABLE** macro is used at the head and the **ENABLE** macro is used at the end.

Here is an assembly code macro definition for **DISABLE**:

```
DISABLE     MACRO
            MOVE.W#$4000,_intena
            ADDQ.B #1,IDNestCnt(A6)
            ENDM
```

DISABLE increments a counter, **IDNestCnt** that keeps track of how many levels of disable have been issued up to now. Only 126 levels of nesting are permitted. Notice that interrupts are disabled *before* the **IDNestCnt** variable is incremented.

Similarly, the **ENABLE** macro will re-enable macros if the last disable level has just been exited:

```
ENABLE      MACRO
            SUBQ.B  #1,IDNestCnt(A6)
            BGE.S   ENABLE@
            MOVE.W#$C000,_intena
ENABLE@:
            MEND
```

ENABLE decrements the same counter that **DISABLE** increments. Notice that interrupts are enabled *after* the **IDNestCnt** variable is decremented.

See the "Tasks" chapter for a better explanation of mutual exclusion using interrupt disabling.

Chapter 6

Memory Allocation

This chapter describes the routines used for dynamic memory allocation and deallocation on the Amiga. These routines allow the user to specify memory allocation according to the actual needs of a task and the hardware it expects to use.

6.1. INTRODUCTION

Areas of free memory are maintained as a special linked list of free regions. Each memory allocation function returns the starting address of a block of memory at least as large as the size that you requested to be allocated. Any memory that is linked into this system free list can be allocated by the memory allocation routines.

The allocated memory is *not* tagged or initialized in any way unless you have specified, for example, **MEMF_CLEAR**. Only the free memory area is tagged to reflect the size of the chunk that has been freed.

You should return allocated memory to the system when your task completes. As noted above, the system only keeps track of available system memory and has no idea which task may have allocated memory and not returned it to the system free list. If you don't return allocated memory when your task exits, that memory is unavailable until the system is powered down or reset.

This can be very critical, especially when using graphics routines that often need large blocks of contiguous RAM space. Therefore, if you dynamically allocate RAM, make sure to return it to the system by using the **FreeMem()** or **FreeEntry()** routines described below.

When you ask for memory to be allocated, the system always allocates blocks of memory in even multiples of 8 bytes. If you request more or less than 8 bytes, your request is always rounded up to the nearest multiple of 8. In addition, the address at which the memory deallocation is made is always rounded down to the nearest even multiple of 8 bytes.

COMPATIBILITY NOTE: Don't depend on this size! Future revisions of the system may require a different size to guarantee alignment of the requested area to a specific boundary. You *can* depend upon allocation being aligned to at least a longword boundary.

6.2. USING MEMORY ALLOCATION ROUTINES

NOTE: Do *not* attempt to allocate or deallocate system memory from within interrupt code. The "Interrupts" chapter explains that an interrupt may occur at any time, even during a memory allocation process. As a result, system data structures may not necessarily be internally consistent.

Memory Requirements

You must tell the system about your memory requirements when requesting a chunk of memory. There are four memory requirement possibilities. Three of these tell where within the hardware address map memory is to be allocated. The fourth, **MEMF_CLEAR**, tells the allocator that this memory space is to be zeroed before the allocator returns the starting address of that space.

The memory requirements that you can specify are:

MEMF_CHIP
: Indicates a memory block that is within the range that the special-purpose chips can access. As of this writing, this is the lowest 512K of the Amiga.

MEMF_FAST
: Indicates a memory block that is outside of the range that the special purpose chips can access. "FAST" means that the special-purpose chips cannot cause processor bus contention and therefore processor access will likely be faster. The special-purpose chips cannot use memory allocated in this way.

MEMF_PUBLIC
: Indicates that the memory requested is to be used for different tasks or interrupt code. This would be for task control blocks, messages, ports and so on. The designation **MEMF_PUBLIC** should be used to assure compatibility with future versions of the system.

MEMF_CLEAR
: Indicates clear memory to zero before returning.

If no preferences are specified, **MEMF_FAST** is assumed first, then **MEMF_CHIP**.

Memory Handling Routines

Exec has the following memory allocation routines:

AllocMem() and **FreeMem()**
: System-wide memory allocation and deallocation routines. These routines use a memory free-list owned and managed by the system.

AllocEntry() and **FreeEntry()**
: Routines for allocating and freeing different size, different type memory blocks with a single call.

Allocate() and **Deallocate()**
: Routines that may be used within a user-task to locally manage a system-allocated memory block. You use these routines to manage memory yourself, using your own memory free lists.

Sample Calls for Allocating System Memory

The following examples show how to allocate memory.

```
struct APTR mypointer,anotherptr;
mypointer = AllocMem(100, 0);
```

AllocMem() returns the address of the first byte of a memory block that is at least 100 bytes in size or null if there is not that much free memory. Since the requirement field is specified as 0, memory will be allocated from any one of the system-managed memory regions.

```
anotherptr = AllocMem(1000,MEMF_CHIP | MEMF_CLEAR);
```

Memory is allocated only out of chip-accessible memory; zeroes are filled into memory space before the address is returned.

If the system free-list does not contain enough contiguous memory bytes in an area matching your requirements and of the size you have requested, **AllocMem()** or **Allocate()** returns a zero.

Sample Function Calls for Freeing System Memory

The following examples free the memory chunks shown in the earlier call to the system allocation routines.

```
FreeMem(mypointer,100);

FreeMem(anotherptr,1000);
```

NOTE: Due to the internal operations of the allocator, your allocation request may result in an allocation larger than the number of bytes you requested in the first place. However, the **FreeMem()** routine adjusts the request to free memory in the same way as **AllocMem()** adjusts the size, thereby maintaining a consistent memory free-list.

The routine **FreeMem()** doesn't return any status. However, if you attempt to free a memory block in the middle of a chunk that the system already believes is free, you will cause a system crash.

Partial blocks can be deallocated, but note again that **FreeMem()** rounds your address down to the nearest even multiple of MEM_BLOCKSIZE and the size up to the nearest multiple before the **FreeMem()** request is performed.

Allocating Multiple Memory Blocks

Exec provides the routines **AllocEntry()** and **FreeEntry()** to allocate multiple memory blocks in a single call. **AllocEntry()** accepts a data structure called a **MemList**, which contains the information about the size of the memory blocks to be allocated and the requirements, if any, that you have regarding the allocation. The **MemList** structure is found in the include-file *exec/memory.h* and is defined as:

```
struct MemList {
    struct Node ml_Node;
    UWORD  ml_NumEntries;  /* number of MemEntries */
    struct MemEntry ml_me[1]; /*where the MemEntries */
    begin
    };
```

where:

Node
allows you to link together multiple **MemLists**. However, the node is ignored by the routines **AllocEntry()** and **FreeEntry()**.

ml_NumEntries
tells the system how many **MemEntry** sets are contained in this **MemList**. Notice that a **MemList** is a variable-length structure and can contain as many sets of entries as you wish.

For the purposes of **AllocEntry()**, the **MemEntry** structure looks like this:

```
struct MemEntry {
    union {
        ULONG meu_Reqs;   /* the AllocMem requirements */
        APTR meu_Addr;
    }me_Un;
    ULONG meu_Length;     /* the size of this request */
};
```

Sample Code for Allocating Multiple Memory Blocks

```
#define me_Reqs me_Un.meu_Reqs
#define me_Addr me_Un.meu_Addr

struct MemList *mymemlist;        /* pointer to a MemList */
struct myneeds {
     struct MemList mn_head;           /* one entry in the header */
     struct MemEntry mn_body[2];       /* additional entries follow
                                       * directly as part of
                                       * same data structure */
};

myneeds.mn_head.ml_NumEntries = 3;
myneeds.mn_head.me[0].me_Reqs = MEMF_PUBLIC;
myneeds.mn_head.me[0].me_Length = 104;
myneeds.mn_head.me[1].me_Reqs=MEMF_FAST|MEMF_CLEAR;
myneeds.mn_head.me[1].me_Length = 8000;
myneeds.mn_head.me[2].me_Reqs=MEMF_CHIP | MEMF_CLEAR;
myneeds.mn_head.me[2].me_Length = 256;

mymemlist = AllocEntry( &myneeds );

/* saying "struct MemEntry mn_body[2]" is simply a way of
 * adding extra MemEntry structures contiguously at the end
 * of the first such structure at the end of the MemList.
 * Thus members of the MemList of type MemEntry can be
 * referenced in C as additional members of the "me[ ]"
 * data structure
 */
```

AllocEntry() returns a pointer to a new **MemList** of the same size as the **MemList** that you passed to it. For example, ROM code can provide a **MemList** containing the requirements of a task and create a RAM-resident copy of the list containing the addresses of the allocated entries.

Result of Allocating Multiple Memory Blocks

The **MemList** created by **AllocEntry()** contains **MemEntry** entries of the second possible form (**MemEntrys** are defined by a union statement, which allows one memory space to be defined in more than one way.)

```
struct MemEntry {
        APTR meu_Addr;  /* the address of the region */
        ULONG meu_Length; /* the size of this request */
};
```

If **AllocEntry()** returns a value with bit 31 clear, then all of the **meu_Addr** positions in your **MemList** will contain valid memory addresses meeting the requirements which you have provided.

To use this memory area, you would use code similar to the following:

```
APTR mydata, moredata;

if (((mymemlist & (1<<31)) < 0)
{
        mydata = mymemlist->ml_me[0].me_Addr;
        moremydata = mymemlist->ml_me[1].me_Addr;
}
else
        exit (200);       /* error during AllocEntry */
```

If **AllocEntry()** has problems while trying to allocate the memory you have requested, instead of the address of a new **MemList**, it will return the memory requirements value with which it had the problem. Bit 31 of the value returned will be set, and no memory will be allocated. Entries in the list that were already allocated will be freed.

6.3. MEMORY ALLOCATION AND TASKS

If you want your task to be fully cooperative with Exec, use the **MemList** and **AllocEntry()** facility to do your dynamic memory allocation.

In the task control block structure, there is a list header named **tc_MemEntry**. This is the list header that you initialize to point to the **MemLists** that your task has created by call(s) to **AllocEntry()**.

Here is a short program segment that handles task memory list header initialization only. It assumes that you have already run **AllocEntry()** as shown in the simple **AllocEntry()** example above.

```
NewList( &mytask.tc_MemEntry ); /* Initialize the task's
                                 * memory list header */

AddTail( &mytask.tc_MemEntry, mymemlist );
```

Assuming that you have only used the **AllocEntry()** method (or **AllocMem()** and built your own custom **MemList**), your task now knows where to find the blocks of memory that your task has dynamically allocated. If your cleanup routine (the task's **finalPC** routine) finds items on the **tc_MemEntry** list when **RemTask(&mytask)** is executed, your routine can wind through all linked lists of **MemLists** and return all allocated memory to the system free-list.

Memory Allocation and the Multi-Tasking System

To ensure that you are effectively working in the multi-tasking system as a cooperating task, you can either:

- Globally allocate and free memory blocks by using **AllocMem()** and **FreeMem()**; adding each block when allocated and deleting each when it is freed, on your task's **MemList**, or
- Allocate one or more blocks of memory from the system global pool using **AllocEntry()** when your task begins, then manage those blocks internally using **Allocate()** and **Deallocate()**.

Managing Memory with Allocate() and Deallocate()

Allocate() and **Deallocate**() use a memory region header, called **MemHead**, as part of the calling sequence. You can build your own local header to manage memory locally. This structure takes the form:

```
struct MemHead {
    UWORD mh_Attributes;  /* characteristics */
    APTR mh_First;        /* first free region */
    APTR mh_Lower;        /* lower memory bounds */
    APTR mh_Upper;        /* upper memory bounds + 1 */
    ULONG mh_Free;          /* number of free bytes */
};
```

where:

mh_Attributes
is ignored by **Allocate**() and **Deallocate**().

mh_First
is the address of the first free region in the memory block.

mh_Lower
is the lowest address within the memory block, must be a multiple of 8 bytes.

mh_Upper
is the highest address within the memory block + 1. The highest address will itself be a multiple of 8 if the block was allocated to you by **AllocMem**().

mh_Free
is the total free space.

This structure is included in the include-files *exec/memory.h* and *exec/memory.i*.

The following sample program fragment shows the correct initialization of a **MemHead** structure. It assumes that you wish to allocate a block of memory from the global pool and thereafter manage it yourself using **Allocate**() and **Deallocate**().

```
struct MemHead mymemhead;
APTR myblock;

struct MemChunk {
      APTR next;
      ULONG size;
};

struct MemChunk *m;

myblock = AllocMem( 8000, MEMF_PUBLIC | MEMF_CLEAR );
      /* get a block from the system */

mymemhead.mh_First   = myblock;
mymemhead.mh_Lower   = myblock;
mymemhead.mh_Upper   = (int)myblock + 8000 + 1;
mymemhead.mh_Free    = 8000 - (sizeof MemChunk);
      /* takes 8 bytes for the memory chunk headers
       * which tag free memory */
m = myblock;
m->next = NULL; /* initialize the free memory list */
m->size = mymemhead.mh_Free;

      /* now mymemhead is ready to use with calls to:
       *
       *    Allocate( &mymemhead, size );
       *          or
       *    Deallocate( &mymemhead, size );
       */
```

Note that only free memory is "tagged" using a **MemChunk** linked list. Once memory is allocated, the system has no way of determining which task now has control of that memory.

If you allocate a large chunk from the system, you can assure that in your "final PC" routine (specified when you perform **AddTask()**) you deallocate this large chunk as your task exits. Thus, local memory allocation and deallocation from a single large block can perhaps save some bookkeeping which might otherwise be required if you had extensively used **AllocMem()** and **FreeMem()** instead.

Chapter 7

Libraries

Using a properly designed machine code interface, it is possible to call any of the system routines without knowing in advance its absolute location in the system. This chapter shows how libraries are designed and used but does not dover the internal library structure. For more information, see appendix K, which contains source for a two-routine, disk-loadable library.

7.1. WHAT IS A LIBRARY?

A library is a collection of jump instructions, a system library node, and a data segment. System library conventions require that each code vector occupy six bytes. The size and content of a library node is specified below in the topic titled "Structure of a Library Node". The data segment is of variable size and depends on the needs of the library itself.

7.2. HOW TO ACCESS A LIBRARY

There are two steps that you must perform to access a library that is already initialized. The first step is to open the library. The second step is to access the jump instructions or data by specifying an offset (negative or positive) from the library base pointer returned by **OpenLibrary()**.

This form of indirection allows you to develop code which is not dependent on the absolute locations of the system routines. Note that in the same release of an Exec kernel, it is possible that different routines can have different addresses. This depends, for example, on whether the hardware options are different or if the user asks for a different configuration. Therefore, accessing the system routines through library calls is the most expedient way of assuring that your code will work on different machines.

Opening a Library

You prepare a library for use by calling the routine **OpenLibrary()**. This call takes the form:

```
LibPtr = OpenLibrary(LibName, Version)
 D0                   A1       D0
```

where:

LibPtr
is a pointer value which is non-zero if the requested library has been located. Be sure to check that the returned value is non-zero **before** attempting to use **LibPtr**. If it is zero, the open failed.

LibName
is a pointer to a string variable (null-terminated) which contains the name of the library that you wish to open.

Version
is the version number of the library that you expect to use. Libraries of the same name will be compatible with previous versions. However, if the user expects a newer version than is present, the open will fail. Use the value 0 if you simply want "any" version of the named library.

The routine **OpenLibrary()** causes the system to search for a library of that name within the system library list. If such an entry is found, the library's open-entry routine is called. If the library is not currently RAM-resident, AmigaDOS will search the directory currently assigned to DEVS:. If that library is present, it will be loaded, initialized, and added to the system library list.

If the library allows you access, the library pointer will be returned in **LibPtr**.

Using a Library to Call a Routine

A typical way to use the library interface once a library has been opened is to use assembly language code as follows. Note that this save/restore is only necessary if A6 does not already contain the correct value.

```
move.l    A6,-(SP)          ; save current contents of A6
move.l    <libptr>,A6       ; move library pointer into A6
jsr       <_LVO<routineName>(A6) ; through library vector table
          move.l    (SP)+,A6        ; restore A6 to original value
```

This is the actual assembly code generated by the use of a machine language macro named LINKLIB as in:

```
LINKLIB functionOffset, libraryBase
```

where:

functionOffset
is "_LVO" followed by the name of the routine as called from C.

libraryBase
is the address of the base of the library.

For example:

```
LINKLIB _LVODisplayBeep,IntuitionBase
```

produces the same code sequence as shown above. This macro is located in the file *exec/libraries.h*. Notice that is handles *only* the linkage to the routine. It does not save any registers or preload any registers for passing values to the routine.

Negative offsets, in multiples of six bytes, access the code vectors within the library.

By convention A6 must contain the library pointer when a library routine is called. This allows any library routine to locate the library and access its data or any of its other entry points.

Registers A0, A1, D0, and D1 may be used as scratch registers by any routine. All other registers, both address and data, if used in a routine, should be saved and restored before exit.

Using A Library To Reference Data

You can use the **LibPtr** to reference a data segment associated with a library by specifying a positive offset from of **LibPtr**, such as:

```
move.l    <libptr>,A1        ; Move library base
move.l    <offset>(A1),D0    ; Retrieve data located at <offset>
```

Library data is not usually accessed directly from outside of a library, but rather is accessed by the routines which are part of the library itself. The sample code retrieves data specifically associated with that library. Note that different languages will have different interface requirements. This example shows only a typical assembly language interface. When you design your own libraries, you may decide on how the associated data segment is to be used. The system itself places no restrictions on its use.

Caching Library Pointers

To make your library calls more efficient, there are various pointers that you may cache if you wish. These are:

a) the **libPtr** itself (since the library node, while it is open, may not be moved, and

b) the address within the library at which a jump instruction is located (since offsets from the **libPtr** do not change).

You should not, however, cache the jump vector from within the library. You will always expect to be calling the current library routine and therefore should not cache the jump vector.

Closing A Library

When your task has finished using a specific library, you should call the routine **CloseLibrary()**. This call takes the form:

```
CloseLibrary(libPtr)
             A1
```

where **libPtr** is the value returned to you by the call to **OpenLibrary()**.

You close a library to tell the library manager that there is one less task currently using that library. If there are no tasks using a library, it is possible for the system, on request, to purge that library and free up the memory resources which it is currently using.

Each successful open should be matched by exactly one close. Do not attempt to use a library pointer after you have closed that library.

7.3. ADDING A LIBRARY

You can add your own library to the system library list, provided that it is constructed as indicated. You add a library to the system by using the **AddLibrary()** function. The format of the call to this function is:

```
AddLibrary(libPtr)
           A1
```

This command links a new library to the system and makes it available to all tasks.

Making a New Library

A function called **MakeLibrary()** is a convenient way for you to construct a library. After running **MakeLibrary()** you will normally add that library to the system library list.

```
libAddr = MakeLibrary(vectors, structure, init, dataSize, SegList )
  D0                    A0         A1         A2        D0        D1

          AddLibrary(libAddr)
                       A1
```

MakeLibrary() allocates space for the code vectors and data area, initializes the library node, and initializes the data area according to your specifications. Its parameters have the following meanings:

vectors
: a pointer to a table of code pointers terminated with a -1. **vectors** must specify a valid table address.

structure
: points to the base of an **InitStruct()** data region. That is, it points to the first location within a table which the **InitStruct()** routine can use to initialize various memory areas. **InitStruct()** will typically be used to initialize the data segment of the library, perhaps forming data tables, task control blocks, I/O control blocks and the like. If this entry is a 0, then **InitStruct()** is not called.

init
: points to a routine which is to be executed after the library node has been allocated, and the code and data areas have been initialized. When this routine is called, the **libAddr** (address of this library) is placed into data register D0. If **init** is zero, no init routine is called.

dataSize
: this variable specifies the size of the data area to be reserved for the library. It includes the standard library node data as well as the reserved data area itself.

SegList
: a pointer to the AmigaDOS memory segment list (for libraries loaded by DOS).

Minimum Subset of Library Code Vectors

The code vectors of a library must at least include the following entries: OPEN, CLOSE, EXPUNGE, and one reserved entry.

OPEN is the entry point called when you use the command **OpenLibrary()**. In the system libraries, OPEN increments the library variable **OpenCnt**. This variable is also used by CLOSE and EXPUNGE.

CLOSE is the entry point called when you use the command **CloseLibrary()**. It decrements the library variable **OpenCnt** and may do a delayed EXPUNGE.

EXPUNGE
prepares the library for removal from the system. This often includes deallocating memory resources which were reserved during initialization. EXPUNGE not only frees the memory allocated for data structures, but also the areas reserved for the library node itself.

The remaining vector is reserved for future use. It should always return zero.

Structure of a Library Node

A library node contains all of the information which the system needs to manage a library. Here is the library structure as it appears in the *exec/libraries.h* include file:

```
struct Library {
    struct   Node libNode;     /* link into the system library list */
    UBYTE lib_Flags;           /* flag variables */
    UBYTE lib_Pad;       /* unused */
    UWORDlib_NegSize;  /* size of jump vectors in bytes. */
    UWORDlib_PosSize;  /* data size */
    UWORDlib_Version;
    UWORDlib_Revision;
    ULONG lib_Sum;       /* checksum */
    UWORDlib_OpenCnt;/* count how many tasks have this library OPEN */
};

/* meaning of the library flag bits>: */

#define LIBF_SUMMING (1 << 0)       /* bit position says some task is
            currently running a checksum on this library */

#define LIBF_CHANGED (1 << 1)/* bit position says one or more
            entries have been changed in the library
            code vectors, used by SumLibrary */

#define LIBF_SUMUSED        /* bit position says user wants a
            checksum fault to cause a system panic */

#define LIBF_DELEXP         /* says there is a delayed expunge
            some user has requested expunge but
            another user still has the library open */
```

Changing The Contents Of A Library

After a library has been constructed and linked to the system library list, you can use the routine **SetFunction()** to either add or replace the contents of one of the library vectors. The format of this routine is as follows:

```
SetFunction( Library, FuncOffset, FuncEntry)
                A1          A0          D0
```

where:

Library
is a pointer to the library in which a function entry is to be changed.

FuncOffset
is the offset (negative) at which the entry to be changed is located.

FuncEntry
is a longword value which is the absolute address of the routine which is to be inserted at the selected position in the library code vectors.

When you use **SetFunction()** to modify a function entry in a library, it automatically recalculates the checksum of the library.

7.4. RELATION TO DEVICES

A device is an interface specification and an internal data structure based on the library structure. The interface specification defines a means of device control. The structures of libraries and devices are so similar that the routine **MakeLibrary()** is used to construct both libraries and devices. Devices require the same basic four code vectors, but have additional code vectors which must be located in specific positions in the code vector table. The functions that devices are expected to perform, at minimum, are shown in chapter 4, "I/O". Also, a skeleton device (source code) is provided in appendix F.

Chapter 8

ROM-Wack

This chapter describes the ROM resident version of the Amiga debugger. It discusses how to enter and use this debugger.

8.1. INTRODUCTION

Wack is a keystroke-interactive bug exterminator used with Amiga hardware and software. *ROM-Wack* is a small, ROM-resident version primarily useful for system crash data structure examination. Its command syntax and display formats are identical to *Grand-Wack*[1] of which it is functionally a subset.

8.2. GETTING TO WACK

ROM-Wack will be automatically invoked by Exec upon a fatal system error, or it can be explicitly invoked through the Exec **Debug()** function. Once invoked, communication is performed through the serial RS-232 data port at 9600 baud.

When a fatal system error occurs, Wack can be used to examine memory in an attempt to locate the source of the failure. The state of the machine will be frozen at the point in which the error occurred and Wack will not disturb the state of system beyond using a small amount of supervisor stack, memory between 200 and 400 hex, and the serial data port.

A program may explicitly invoke wack by calling the Exec **Debug()** function. This is useful during the debug phase of development for establishing program breakpoints. For future compatibility, **Debug** should be called with a single, null parameter; for example, **Debug(0)**.

[1] The RAM resident and remote versions of Wack.

Please note however, that calling the **Debug()** function does not necessarily invoke ROM-Wack. If Grand-Wack or a user supplied debugger has been installed, it will be invoked in place of ROM-Wack.

When Wack is called from a program, system interrupts continue to process, but multitasking is disabled. Generally this is not harmful to the system. Your graphics will still display, keys may be typed, the mouse can be moved, and so on. However, many interrupts deposit raw data into bounded or circular buffers. These interrupts often signal related device tasks to further process these buffers. If too many interrupts occur, device buffers may begin to overflow or wrap-around. You should limit the number of interrupt actions (typing keys on the Amiga keyboard for example) you perform while executing in Wack.

8.3. KEYSTROKES, NUMBERS, AND SYMBOLS

Wack performs a function upon every keyboard keystroke. In ROM-Wack, these functions are permanently bound to certain keys. For example, typing ">" will immediately result in the execution of the **next-word** function. This type of operation gives a "keystroke interactive" feel to most of the common Wack commands.

Whenever a key is pressed, it is mapped through a *KeyMap* which translates it into an action. This action is context-dependent. A key can have different meanings in different contexts. For simplicity, ROM-Wack applies keys consistently in all contexts[2].

In the default keymap most punctuation marks are bound to simple actions, such as displaying a memory frame, moving the frame pointer, or altering a single word. These actions are always performed immediately. In contrast, the keys A-Z, a-z, and 0-9 are bound to a function that collects the keys as a string. When such a string is terminated with <RETURN>, the keys are interpreted as a single *symbol* or *number*.

In ROM-Wack, symbols are only treated as *intrinsic functions*. Macros, constants, offsets, and bases are not supported. Hence, typing a symbol name will always result in the invocation of the symbol's statically bound function.

If a string of keys forms a number, that number is treated as a hexadecimal value. If a string of keys is neither a number nor a known symbol, the message "unknown symbol" is presented.

During the "collection" of a symbol or number string, typing a backspace deletes the previous character. Typing <CTRL-X> deletes the entire line.

[2] The Grand-Wack feature of arbitrary key binding is not available in ROM-Wack.

8.4. REGISTER FRAME

When Wack is invoked for any reason, a *Register Frame* is displayed:

```
ROM-Wack
PC: F00AB4  SR: 0000  USP: 001268  SSP: 07FFE8  TRAP: 0000  TASK: 0008B8
DR: 00000001 00000004 0000000C 00000AB4 00000001 0000001C 00000914 00000914
AR: 00000AB4 00F0D348 00011A80 00000B9C 00F20770 00F20380 00000804
SF: 0000 00F0 0AB4 0014 00F0 0AB4 0014 00F0 0AB4 0004 00F0 0AB4 0000 0004 0000
```

This frame displays the current processor state and system context from which you entered Wack. If you are familiar with the M68000 processor, most of this frame should be obvious: USP for user stack pointer, SSP for system stack pointer, etc.

The TRAP field indicates the trap[3] number which forced us into Wack. The standard TRAP numbers are:

0 normal entry

2 bus error

3 address error

4 illegal instruction

5 zero divide

6 CHK instruction (should not happen ..)

7 TRAPV instruction (should not happen ..)

8 privilege violation

9 trace (single step)

A line 1010 emulator

B line 1111 emulator

2N trap instruction N (2F normally for breakpoint)

The TASK field indicates the task from which the system entered Wack. If this field is zero, the system entered Wack from supervisor mode.

[3] Motorola calls these exceptions. We use the word "exception" for asynchronous task events.

The SF line provides a backtrace of the current stack frame. This is often useful for determining the current execution context (last function called, for example). The user stack[4] is displayed for entry from a task; the system stack for entry from supervisor mode.

8.5. DISPLAY FRAMES

Wack displays memory in fixed size *frames*. A frame may vary in size from 0 to 64K bytes. Frames normally show addresses, word size hex data, and ASCII equivalent characters:

```
F000C4 6578 6563 2E6C 6962 7261 7279 0000 4AFC  e x e c . l i b r a r y...
F000D4 00F0 00D2 00F0 2918 0019 0978 00F0 00C4 ............ )^X..^Y^I x...
```

By default, Wack will pack as much memory content as it can onto a single line. Sometimes it is preferable to see more or less than this default frame size. The frame size may be modified with :n. Here "n" represents the number of bytes (rounded to the next unit size) that will be displayed.

```
:4
F000C4 6578 6563  e x e c
:20
F000C4 6578 6563 2E6C 6962 7261 7279 0000 4AFC  e x e c . l i b r a r y...
F000D4 00F0 00D2 00F0 2918 0019 0978 00F0 00C4 ............ )^X..^Y^I.....
```

A ":0" frame size is useful for altering the write-only custom chip registers.

8.6. RELATIVE POSITIONING

Wack functions like a *memory editor*; nearly all commands are performed relative to your current position in memory. The following commands cause relative movement:

. forward a frame

[4] Version 25.1 always shows the system stack, never the user stack. This will change.

, backward a frame

> forward a word

< backward a word

+n forward n bytes

-n backward n bytes

<RETURN>
redisplay current frame

<SPACE>
forward a word

<BKSP>
backward a word

For example:

```
< RETURN >
f00200 7072 6573 656e 7429 0d0a 0000 2028 6372  p r e s e n t ) ^M ^J . . .
.
f00210 6173 6820 2d20 6361 6e6e 6f74 2072 6563  a s h   -   c a n n o t
,
f00200 7072 6573 656e 7429 0d0a 0000 2028 6372  p r e s e n t ) ^M ^J . . .
>
f00202 6573 656e 7429 0d0a 0000 2028 6372 6173  e s e n t ) ^M ^J . . . .   (
<
f00200 7072 6573 656e 7429 0d0a 0000 2028 6372  p r e s e n t ) ^M ^J . . .
+24
f00224 290d 0a00 2028 626f 6f74 2064 6576 6963  ) ^M ^J . .   ( b o o t   d
-38
f001ec 6c65 290d 0a00 2028 6e6f 2064 6562 7567  l e ) ^M ^J . .   ( n o   d
```

8.7. ABSOLUTE POSITIONING

There are a few commands that perform absolute positioning. Typing a hex number moves you to that position in memory:

```
10ec
0010ec 00f0 17c0 4ef9 00f0 179a 4ef9 00f0 1786 ....^W..N......^W..
```

Also, Wack maintains an indirection stack to help you walk down linked lists of absolute pointers:

```
4
000004 0000 11ec 00f0 0a8e 00f0 0a90 00f0 0a92 ....^Q......^J......^J.....
[      (use current longword as the next address)
0011ec 0000 18f8 0000 1332 0900 00f0 086a 0000 ....^X......^S 2^I.........
]      (return to the previous "indirected" address)
000004 0000 11ec 00f0 0a8e 00f0 0a90 00f0 0a92 ....^Q......^J......^J.....
```

The "find" command finds a given pattern in memory, and the "limit" command determines the upper bound of the search. The pattern may be from one to four bytes in length. The pattern is not affected by the alignment of memory; that is, byte alignment is used for all searches regardless of the pattern size.

To set the upper bound for a "find" command, type an address followed by "limit" or "^". The default bound is 1000000 hex.

8.8. ALTERING MEMORY

The = command lets you modify your current memory word:

```
20134
020134 0000 0000 0000 ............
020134 0000 = 767
020134 0767 0000 0000 ^G g........
```

If framesize is zero, the contents of the word will not be displayed prior to letting you modify it:

```
:0
dff09c
DFF09C xxxx = 7fff
```

If you decide not to modify the contents after typing an =, press <RETURN> without typing a number. If you've already typed a number, type <CTRL-X>.

The **alter** command performs a repeated = which is handy for setting up tables. While in this mode, the > and < will move you forward or backward one word. To exit from this mode, type a <RETURN> with no preceding number.

```
alter
001400 0280 = 222
001402 00C8 = <
001400 0222 = 333
001402 00C8 = 444
001404 0000 = 0
001406 3700 = >
001408 0000 = 666
00140A 0000 = < RETURN >
```

You can modify registers when single-stepping or breakpointing. Typing "!" followed by the register name (D0-D7, A0-A6), U) lets you make modifications. SR and SSP cannot be modified.

The "fill" command fills memory with a given pattern from the current location to an upper bound. The "limit" command determines the upper bound of the fill. The size of the fill pattern determines the number of bytes the pattern occupies in memory. For example, typing:

```
fill <RETURN>
45
```

fills individual bytes with the value 45. Typing:

```
fill <RETURN>
045
```

fills words, and

```
fill <RETURN>
0000045
```

fills longwords.

CAUTION: Using the fill command without properly setting the limit can destroy data in memory. To set the upper bound for a fill, type an address followed by "limit" or a "^".

8.9. EXECUTION CONTROL

These commands control program execution and system reset:

go	execute from current address
resume	resume at current PC address
^D	resume at current PC address
^I (tab)	single instruction step
boot	reboot system (cold-reset)
ig	reboot system (cold-reset)

8.10. BREAKPOINTS

ROM-Wack has the ability to perform limited program breakpoints. Up to 16 breakpoints may be set. The breakpoint commands are:

set	set breakpoint at current address
clear	clear breakpoint at current address
show	show all breakpoint addresses
reset	clear all breakpoints

To set a breakpoint, position the address pointer to the break address and type **set**. Resume program execution with **go** or **resume**. When your breakpoint has been reached, Wack will display a register frame. The breakpoint is automatically cleared once the breakpoint is reached.

8.11. RETURNING TO MULTI-TASKING AFTER A CRASH

The "user" command forces the machine back into multi-tasking after a crash that invoked ROM-Wack. This gives your system a chance to flush disk buffers before you reset, thus securing your disk's super-structures.

Once you type "user", you cannot exit from ROM-Wack, so you should use this command only when you want to reboot after debugging. Give your disk a few seconds to write out its buffers. If your machine is in a bad way, the "user" command may not work.

Part II

Chapter 1

Graphics Primitives

This chapter describes the basic graphics tools. It covers the graphics support structures, display routines, and drawing routines.

Many of the operations described in this section are also performed by the Intuition software. See the book called *Intuition: The Amiga User Interface* for more information.

1.1. INTRODUCTION

The Amiga has two basic types of graphics support routines: display routines and drawing routines. These routines are very versatile and allow you to define any combination of drawing and display area you may wish to use.

Section 1.2 of this chapter defines the display routines. These routines show you how to form and manipulate a display, including the following:

- how to identify the memory area that you wish to have displayed
- how to position the display area window to show only a certain portion of a larger drawing area
- how to split the screen into as many vertically stacked slices as you wish
- whether to use high-resolution (640 pixels across) or low-resolution (320 pixels across) display mode for a particular screen segment, and whether to use interlaced (400 lines top to bottom) or noninterlaced (200 lines) mode
- how to specify how many color choices per pixel are to be available in a specific section of the display

Section 1.3 explains all of the available modes of drawing supported by the system software, including how to:

- reserve memory space for use by the drawing routines
- define the colors that can be drawn into a drawing area
- define the colors of the drawing pens (foreground pen, background pen for patterns, and outline pen for area-fill outlines)
- define the pen position in the drawing area
- draw lines, define vertex points for area-filling, and specify the area-fill color and pattern
- define a pattern for patterned line drawing
- change drawing modes
- read or write individual pixels in a drawing area
- copy rectangular blocks of drawing area data from one drawing area to another
- use a template (predefined shape) to draw an object into a drawing area

Components of a Display

In producing a display, you are concerned with two primary components: sprites and the playfield. Sprites are the easily movable parts of the display. The playfield is the static part of the display and forms a backdrop against which the sprites can move and with which the sprites can interact.

This chapter covers the creation of the background. Sprites are described in Chapter 3, "Animation".

Introduction to Raster Displays

The Amiga produces its video displays on standard television or video monitors by using raster display techniques. The picture you see on the video display screen is made up of a series of horizontal video lines stacked one on top of another, as illustrated in Figure 1-1. Each line represents one sweep of an electronic video beam, which "paints" the picture as it moves along. The beam sweeps from left to right, producing the full screen one line at a time. After producing the full screen, the beam returns to the top of the display screen.

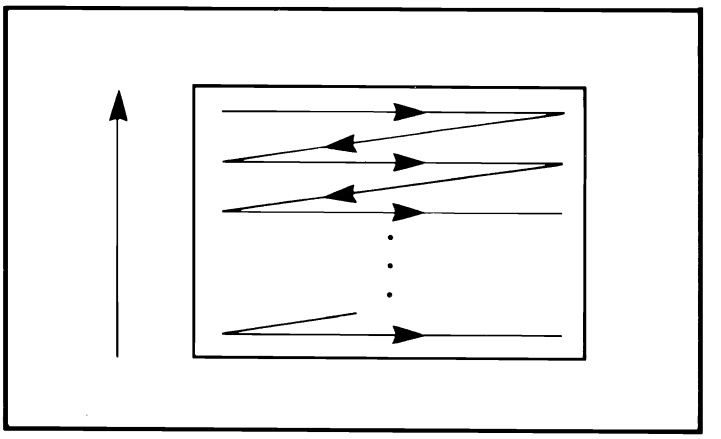


Figure 1-1: How the Video Display Picture is Produced

The diagonal lines in the figure show how the video beam returns to the start of each horizontal line.

Affect of Display Overscan on the Viewing Area

To assure that the picture entirely fills the viewable region of the screen, the manufacturer of the video display usually creates a deliberate *overscan*. That is, the video beam is swept across a larger section than the front face of the screen can actually display. The video beam actually covers 262 vertical lines. The user, however, only sees the portion of the picture that is within the center region of the display, which is is about 200 rows, as illustrated in Figure 1-2 below. The graphics system software lets you specify more than 200 rows.

Overscan also restricts the amount of video data that can appear on each display line. The system software allows you to specify a display width of up to 352 pixels (or 704 in high-resolution mode) per horizontal line. You should generally, however, use the standard values of 320 (or 640 in high-resolution mode) for most applications.

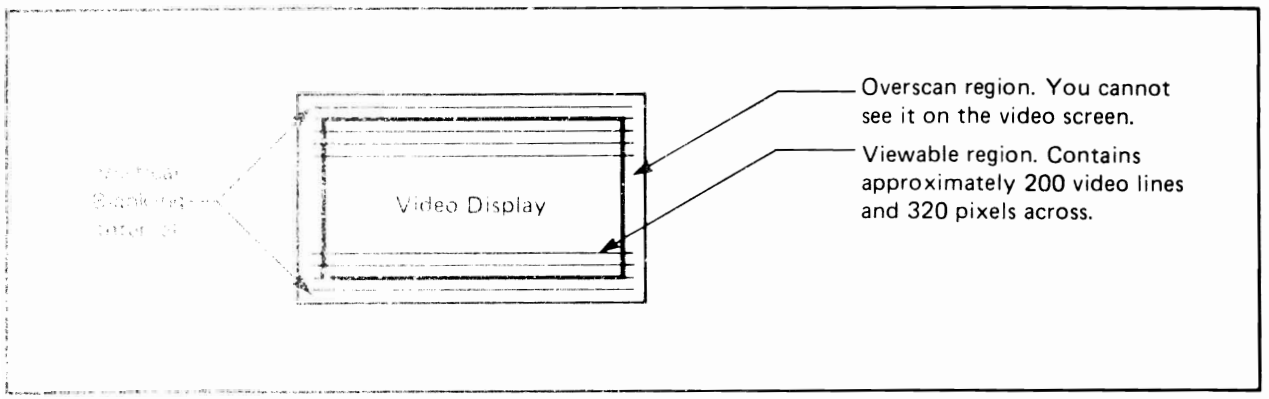


Figure 1-2: Display Overscan Restricts Usable Picture Area

The time during which the video beam is in the region below the bottom line of the viewable area and above the top line of the next display field is called the ***vertical blanking interval.***

Color Information for the Video Lines

The hardware reads the system display memory to obtain the color information for each line. As the video display beam sweeps across the screen producing the display line, it changes color, producing the images you have defined.

Interlaced and Non-Interlaced Mode

In producing the complete display (262 video lines), the video display device produces the top line, then the next lower line, then the next until it reaches the bottom of the screen. When it reaches the bottom, it returns to the top to start a new scan of the screen. Each complete set of 262 lines is called a *display field*. It takes about 1/60th of a second to produce a complete display field.

The Amiga has two vertical display modes: *interlaced* and *non-interlaced.* In non-interlaced mode, the video display produces the same picture for each successive display field. A non-interlaced display normally has about 200 lines in the viewable area (for a full-screen size display).

To make the display more precise in the vertical direction, you use interlaced mode, which displays twice as much data in the same vertical area as non-interlaced mode. Within the same amount of viewable area, you can display 400 video lines instead of 200.

For interlaced mode, the video beam scans the screen at the same rate (1/60th of a second per complete video display field); however, it takes two display fields to form a complete video display picture. During the first of each pair of display fields, the system hardware shows the odd numbered lines of an interlaced display (1, 3, 5, and so on). During the second display field, it shows the even numbered lines (2, 4, 6 and so on). These sets of lines are taken from data defining 400 lines. During the display, the hardware moves the second display field's lines downward slightly from the position of the first, so that the lines in the second field are "interlaced" with those of the first field, giving the higher vertical resolution of this mode. For an interlaced display, the data in memory defines twice as many lines as for a non-interlaced display as shown in Figure 1-3.

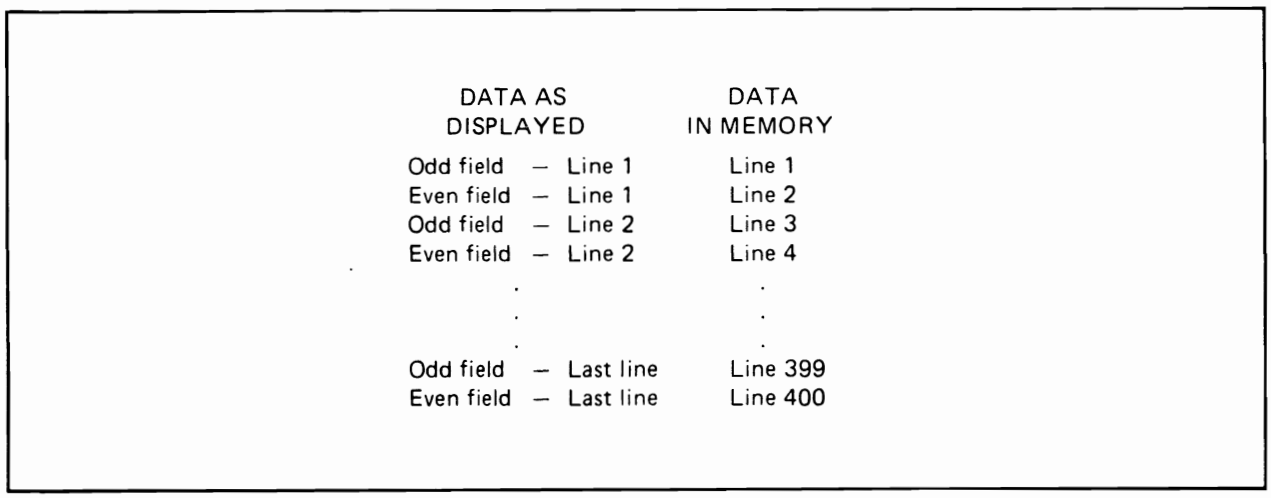


Figure 1-3: Interlaced Mode — Display Fields and Data in Memory

Figure 1-4 shows a display formed as display lines 1, 2, 3, 4, ... 400. The 400-line interlaced display uses the same physical display area as a 200-line non-interlaced display.

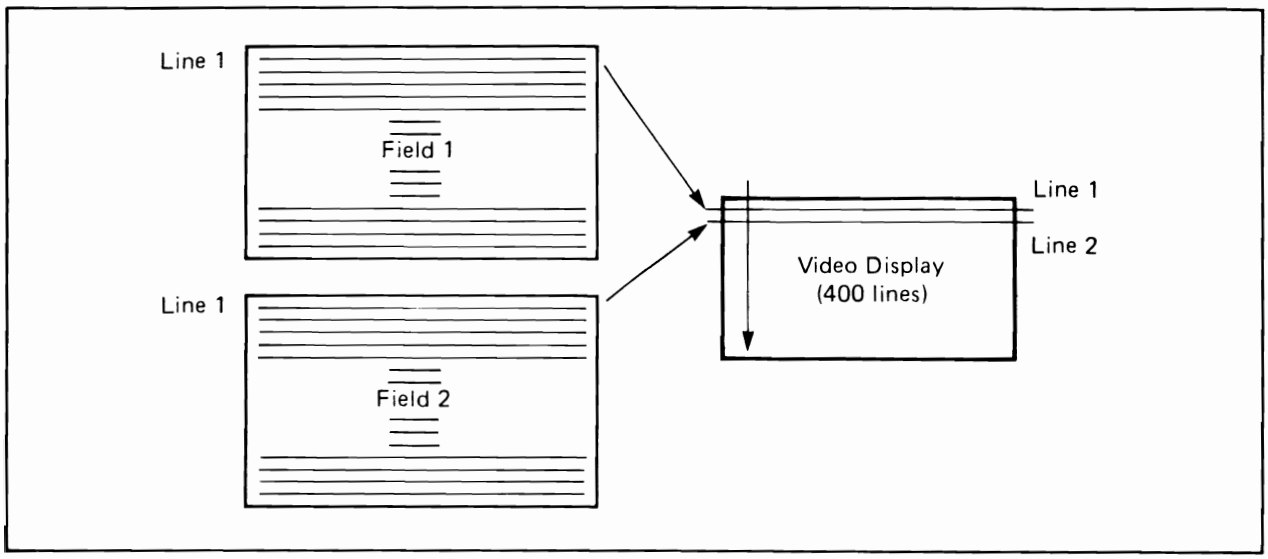


Figure 1-4: Interlaced Mode Doubles Vertical Resolution

During an interlaced display, it appears that both display fields are present on the screen at the same time and form one complete picture. This phenomenon is called *video persistence.*

High and Low Resolution Modes

The Amiga also has two horizontal display modes: *high-resolution* and *low-resolution.* High-resolution mode provides (nominally) 640 distinct pixels (picture elements) across a horizontal line. Low-resolution provides (nominally) 320 pixels across each line.

Low-resolution mode allows up to 32 colors at one time, and high-resolution mode allows 16 colors (out of 4,096 choices) at one time.

One other display mode affects the number of colors you can display at one time: *hold-and-modify.* Hold-and-modify mode allows you to display all 4,096 colors on-screen at once.

Forming an Image

To create an image, you write data ("draw") into a memory area in the computer. From this memory area, the system can retrieve the image for display. You tell the system exactly how the memory area is organized, so that the display is correctly produced. You use a block of memory words at sequentially increasing addresses to represent a rectangular region of data bits. Figure 1-5 shows the contents of three example memory words; 0-bits are shown as blank rectangles, and 1-bits as filled-in rectangles.

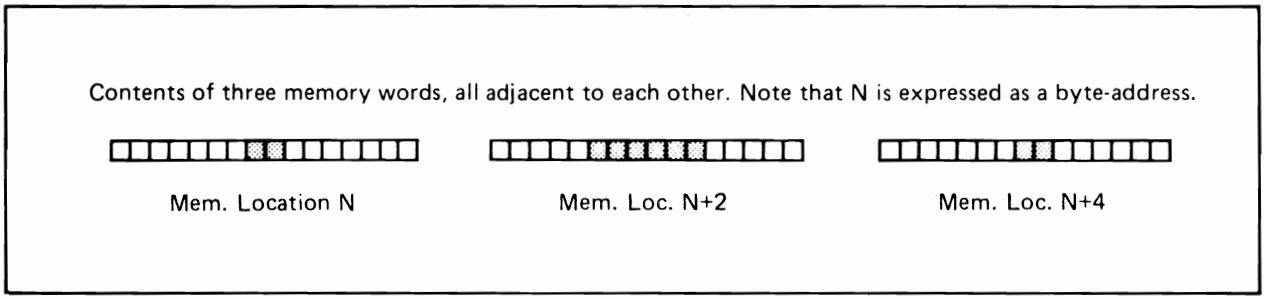


Figure 1-5: Sample Memory Words

The system software lets you define linear memory as rectangular regions, called *bit-planes*. Figure 1-6 shows how the system views the same 3 words as a bit-plane, wherein the data bits form an X-Y plane.

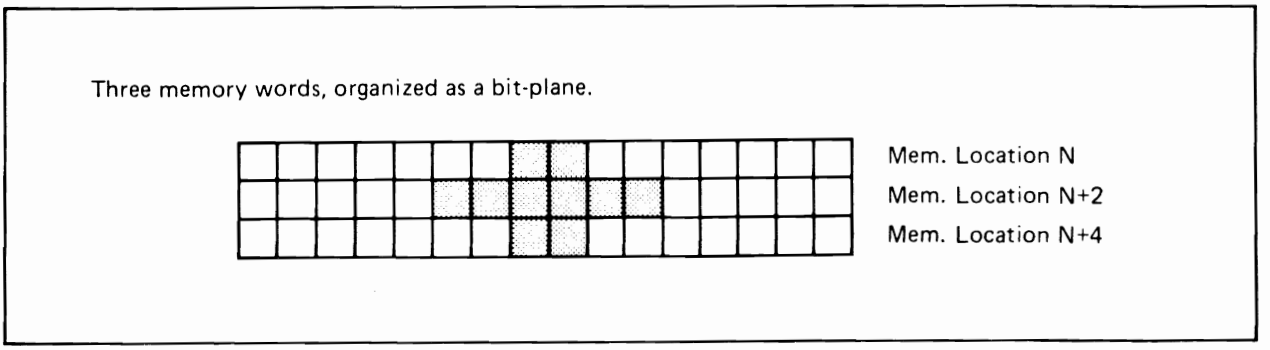


Figure 1-6: A Rectangular "Look" at the Sample Memory Words

Figure 1-7 shows how 4,000 words (8,000 bytes) of memory can be organized to provide enough bits to define a single bit-plane of a full-screen low-resolution video display (320 x 200).

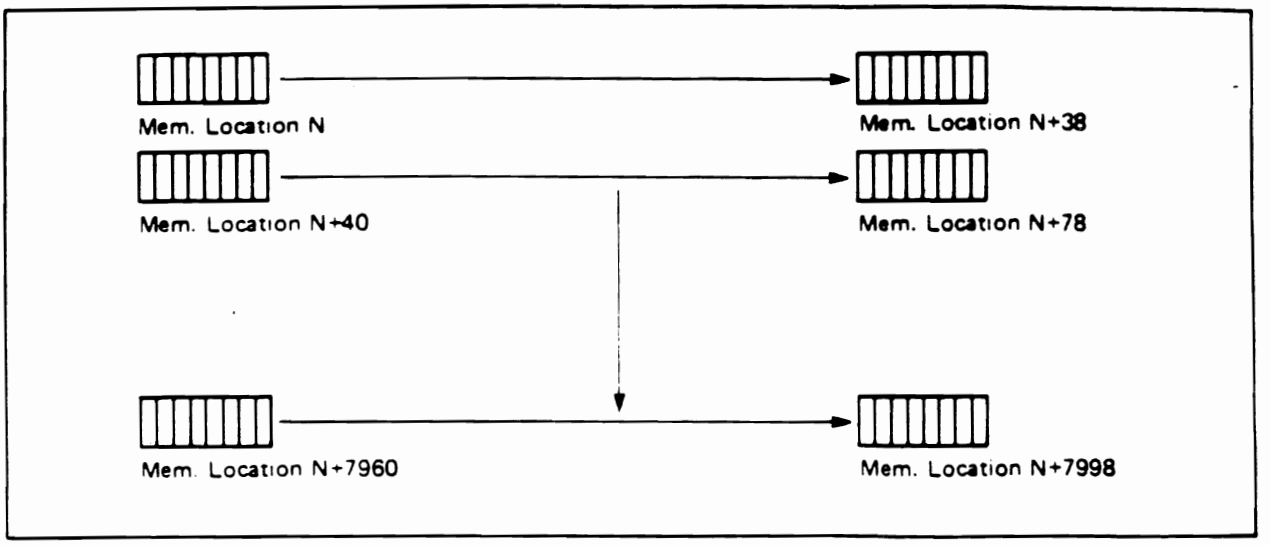


Figure 1-7: Bit-Plane for a Full-Screen Low-Resolution Display

Each memory data word contains 16 data-bits. The color of each pixel on a video display line is directly related to the value of one or more data-bits in memory as follows:

- If you create a display where each pixel is related to only one data-bit, then you can only select from one of two possible colors, because each bit can only have a value of 0 or 1.

- If you use two bits per pixel, there is a choice of 4 different colors because there are 4 possible combinations of the values of 0 and 1 from each of the two bits.

- If you specify 3, 4 or 5 bits per pixel, you will have 8, 16 or 32 possible choices of a color for each pixel respectively.

To create multi-colored images, you must tell the system how many bits are to be used per pixel. The number of bits per pixel is the same as the number of bit-planes used to define the image.

As the video beam sweeps across the screen, the system retrieves one data bit from each bit-plane. Each of the data bits is taken from a different bit-plane, and one or more bit-planes are used to fully define the video display screen. For each pixel, data-bits in the same x,y position in each bit-plane are combined by the system hardware to create a binary value. This value determines the color that appears on the video display for that pixel.

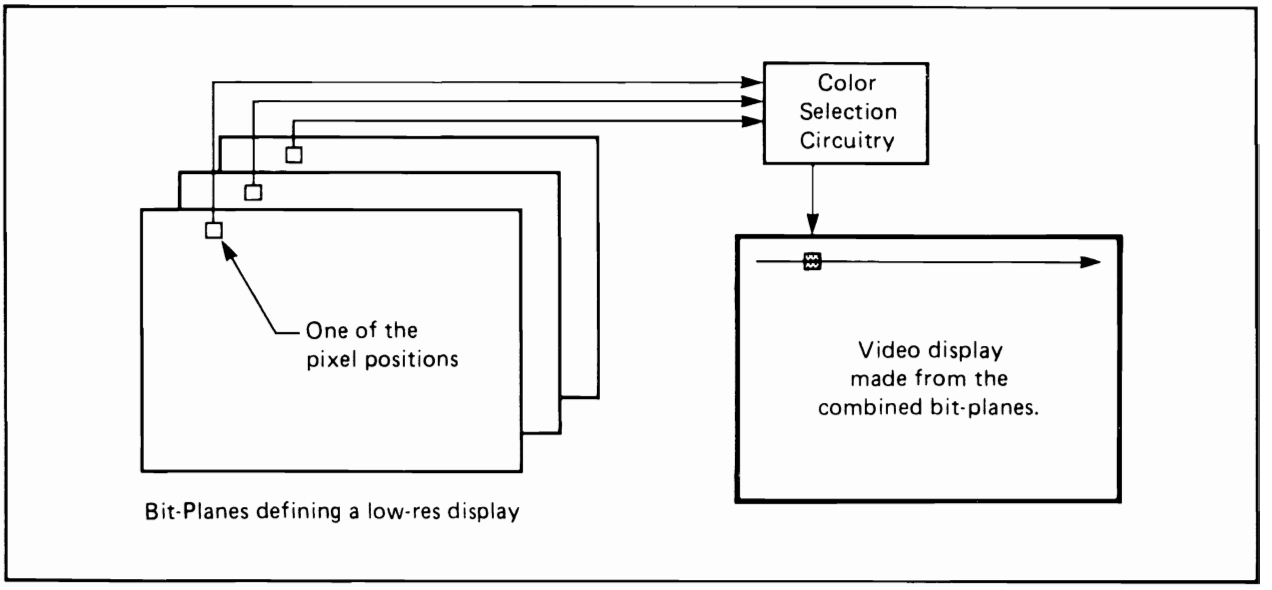


Figure 1-8: Bits From Each Bit-Plane Select Pixel Color

You will find more information showing how the data bits actually select the color of the displayed pixel in the section called "ViewPort Color Selection".

Role of the Copper (Coprocessor)

The Amiga has a special-purpose coprocessor, called the *Copper*, that can control nearly the entire graphics system. The Copper can control register updates, reposition sprites, change the color palette, and update the blitter. The graphics and animation routines use the Copper to set up lists of instructions for handling displays, and advanced users can write their own "user Copper lists".

1.2. DISPLAY ROUTINES AND STRUCTURES

CAUTION

> Section 1.2 describes the lowest level graphics interface to the system hardware. If you use any of the routines and the data structures described in these sections, your program will essentially take over the entire display. It will not, therefore, be compatible with the multi-window operating environment, known as Intuition, which is used by AmigaDOS.

The descriptions of the display routines, as well as those of the drawing routines, occasionally use the same terminology as that in *Intuition: The Amiga User Interface*. These routines and data structures are the same ones that Intuition software uses to produce its displays.

The computer produces a display from a set of instructions you define. You organize the instructions as a set of parameters known as the **View** structure.

Figure 1-9 shows how the system interprets the contents of a **View** structure. This drawing shows a complete display composed of two different component parts, which could, for example, be a low-resolution, multi-colored part and a high-resolution, two-colored part.

A complete display consists of one or more **ViewPorts**, whose display sections are separated from each other by at least one blank line. The viewable area defined by each **ViewPort** is a rectangular cut from the same size (or larger) raster. You are essentially defining a display consisting of a number of vertically stacked display areas in which separate sections of graphics rasters can be shown.

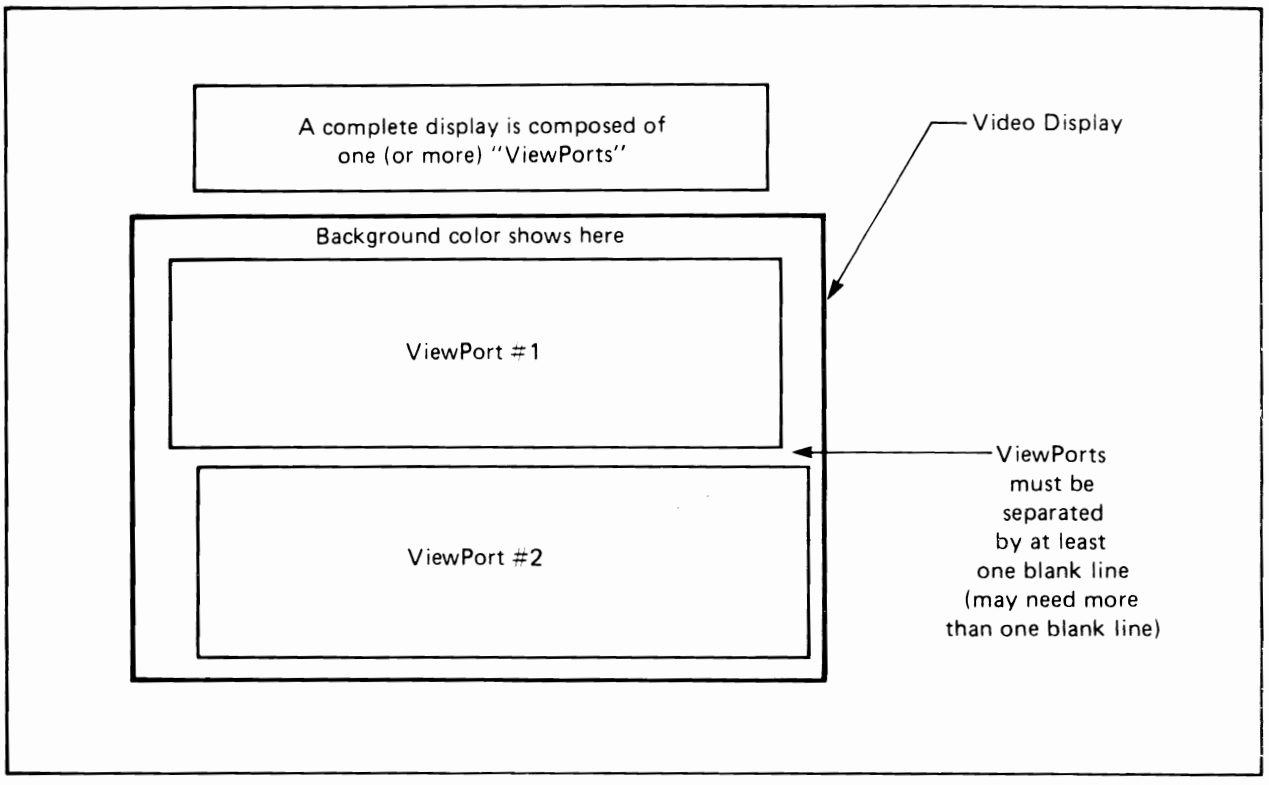


Figure 1-9: The Display is Composed of ViewPorts

Limitations on the Use of ViewPorts

The system software for defining **ViewPorts** allows only vertically stacked fields to be defined. Figure 1-10 shows acceptable and unacceptable display configurations. If you want to create overlapping windows, define a single **ViewPort** and manage the windows yourself within that **ViewPort**.

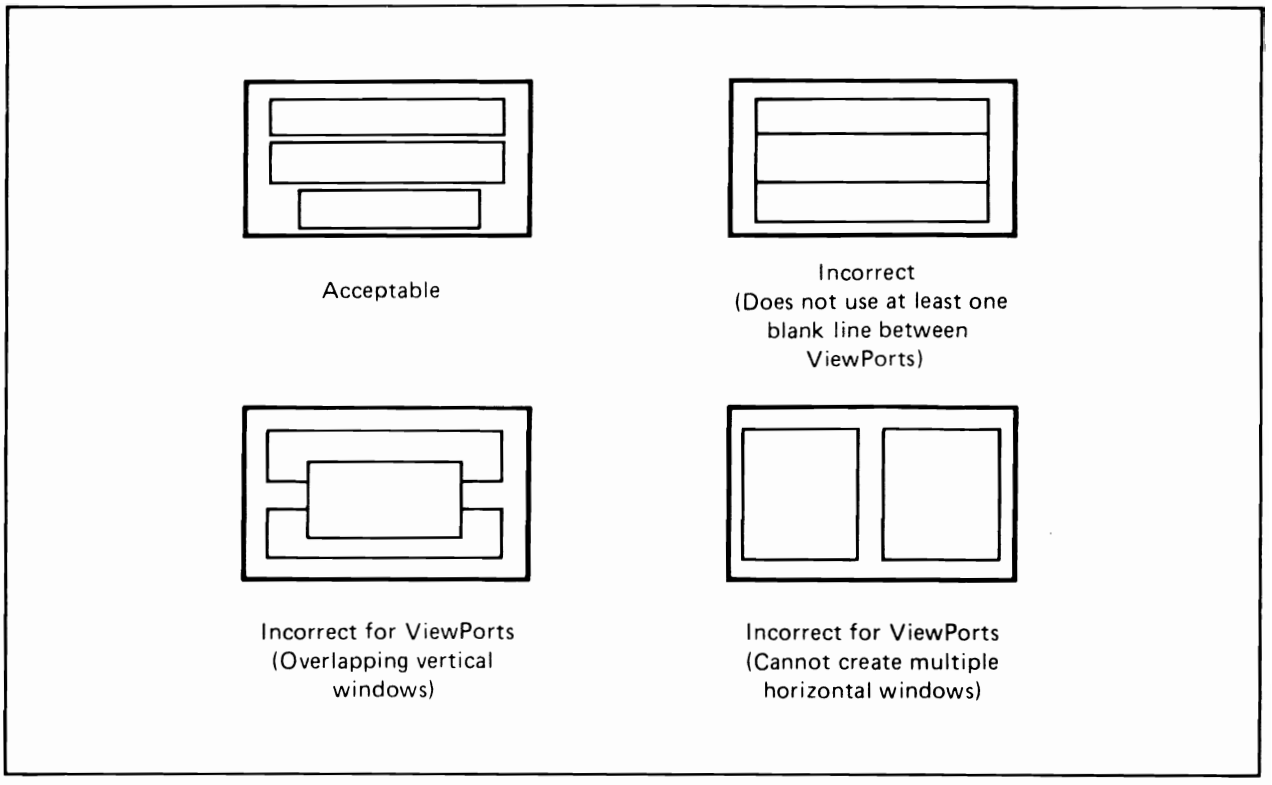


Figure 1-10: Correct and Incorrect Uses of ViewPorts

A **ViewPort** is related to the custom screen option of Intuition. In a custom screen, you can split the screen into slices as shown in the "correct" illustration of Figure 1-10. Each custom screen can have its own set of colors, its own resolution, and show its own display area. Within a **ViewPort**, actually within its associated **RastPort** (drawing area definition), it is possible to split the display into separate drawing areas called windows. The **ViewPort** is simply an indivisible window onto a possibly larger complex drawing area.

Characteristics of a ViewPort

To describe a **ViewPort** fully, you need to set the following parameters:

- height

- width
- display mode

In addition to these parameters, you must also tell the system

- from where in memory to retrieve the data for the **ViewPort** display, and
- how to position the final **ViewPort** display on the screen.

ViewPort Size Specifications

Figure 1-11 illustrates that the variables **DHeight**, and **DWidth** specify the size of a **ViewPort**.

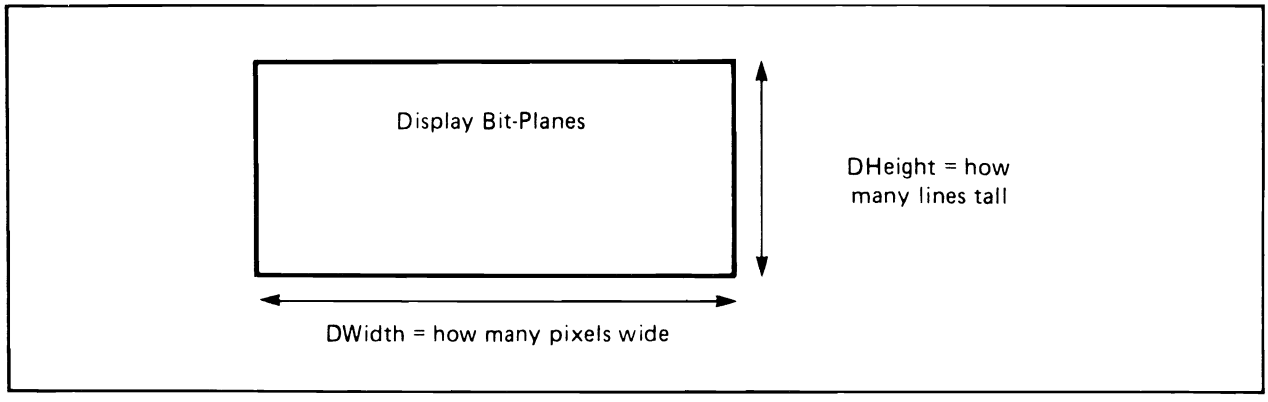


Figure 1-11: Size Definition for a ViewPort

ViewPort Height

The variable **DHeight** determines how many video lines will be reserved to show the height of this display segment. The size of the actual segment depends on whether you define a non-interlaced or an interlaced display. An interlaced display is half as tall as a non-interlaced display of the same number of lines.

For example, a **View** consisting of two **ViewPorts** might be defined as follows:

- **ViewPort** #1 is 150 lines, high-resolution mode (uses the top 3/4 of the display)
- **ViewPort** #2 is 49 lines of low-resolution mode (uses the bottom 1/4 of the display, and allows the space for the one blank line between **ViewPorts** which is required by the system)

The user interface software (Intuition) assumes a standard configuration of 200 rows (400 in interlaced mode).

ViewPort Width

The **DWidth** variable determines how wide, in current pixels, the display segment will be. If you are using low-resolution mode, you should specify a width of 320 pixels per horizontal line. If you are using high-resolution mode, you should specify a width of 640 pixels. You may specify a smaller value of pixels per line to produce a narrower display segment.

Although the system software allows you define low-resolution displays as wide as 352 pixels and high-resolution displays of 704 pixels, you should not exceed the normal values of 320 or 640, respectively. Because of display overscan, many video displays will not be able to show all of a wider display and sprite display may be affected. If you are using hardware sprites or VSprites with your display, and you specify **ViewPort** widths exceeding 320 or 640 pixels (for low- or high-resolution, respectively), it is likely that hardware sprites 5, 6, and 7 will not be rendered on-screen. These sprites may not be rendered because playfield DMA (direct memory access) takes precedence over sprite DMA when an extra-wide display is produced.

ViewPort Color Selection

The maximum number of colors that a **ViewPort** can display is determined by the depth of the **BitMap** that the **ViewPort** displays. The depth is specified when the **BitMap** is initialized. See the section below called "Preparing the BitMap Structure".

Depth determines the number of bit-planes used to define the colors of the rectangular image you are trying to build (the raster image) and the number of different colors that can be displayed at the same time within a **ViewPort**. For any single pixel, the system can display any one of 4,096 possible colors.

Table 1-1 shows depth values and the corresponding number of possible colors for each value.

Table 1-1: Depth Values and Number of Colors in the ViewPort

Colors	Depth Value	
2	1	
4	2	
8	3	
16	4	(Note 1)
32	5	(Notes 1,2)
4096	6	(Notes 1,2,3)
64	6	(Note 1,2,)

NOTES:

1. Single-playfield mode only—**ViewPort** mode not DUALPF
2. Low-resolution mode only—**ViewPort** mode not HIRES
3. Hold-and-modify mode only—**ViewPort** mode = HAM

The color palette used by a **ViewPort** is specified in a **ColorMap**. See the "Preparing the ColorMap" below for more information.

Depending on whether single- or dual-playfield mode is used, the system will use different color register groupings for interpreting the on-screen colors. Table 1-2 below details how the depth and the **Modes** variable in the **ViewPort** structure affect the registers the system uses.

Table 1-2: Single Playfield Mode (**Modes** variable not equal to DUALPF)

Depth	Color Registers Used
1	0,1
2	0-3
3	0-7
4	0-15
5	0-31
6	0-16 (if modes = HAM)

Table 1-3 shows the five possible combinations when the **Modes** variable is set to DUALPF.

Table 1-3: Dual Playfield Mode (**Modes** variable = DUALPF)

Depth (PF-1)	Color Registers	Depth (PF-2)	Color Registers
1	0,1	1	8,9
2	0-3	1	8,9
2	0-3	2	8-11
3	0-7	2	8-11
3	0-7	3	8-15

ViewPort Display Modes

The system has eight different display modes that you can specify for each **ViewPort**. The 8 bits that control the modes are DUALPF, PFBA, HIRES, LACE, HAM, SPRITES, and VP_HIDE. A mode becomes active if you set the corresponding bit to 1 in the **Modes** variable of the **ViewPort** structure. After you initialize the **ViewPort**, you can set the bit(s) for the modes you want. (See the section below called "Preparing the ViewPort Structure" for more information about initializing a ViewPort).

Modes DUALPF and PFBA are related. DUALPF tells the system to treat the raster specified by this **ViewPort** as the first of two independent and separately controllable playfields. It also modifies the manner in which the pixel colors are selected for this raster.

When PFBA is a 1, it specifies that a second playfield has video priority over the first one. Playfield relative priorities can be controlled when the playfield is split into two overlapping regions. Single-playfield and dual-playfield modes are discussed in "Advanced Topics" below.

HIRES tells the system that the raster specified by this **ViewPort** is to be displayed with 640 horizontal pixels rather than 320 horizontal pixels.

LACE tells the system that the raster specified by this **ViewPort** is to be displayed in interlaced mode. If the **ViewPort** is non-interlaced and the **View** is interlaced, the ViewPort will be displayed at its specified height and will look only slightly different than it would look when displayed in a non-interlaced **View**. See "Interlaced Mode vs. Non-Interlaced Mode" below for more information.

HAM tells the system to use "hold-and-modify" mode, a special mode that lets you display up to 4096 colors on screen at the same time. It is described in the "Advanced Topics" section.

SPRITES tells the system that you are using VSprites or Simple Sprites in this display. This bit, when a 1, tells the software to load color registers for sprites. See Chapter 3 "Animation", for more information about sprites.

VP_HIDE tells the system that this **ViewPort** is obscured by other ViewPorts. When a **View** is constructed, no display instructions are generated for this ViewPort.

EXTRA_HALFBRITE is reserved for future use.

Single-Playfield Mode vs Dual-Playfield Mode

When you specify single-playfield mode, you ask that the system treat all bit-planes as part of the definition of a single playfield image. Each of the bit-planes defined as part of this **ViewPort** contributes data bits that determine the color of the pixels in a single playfield.

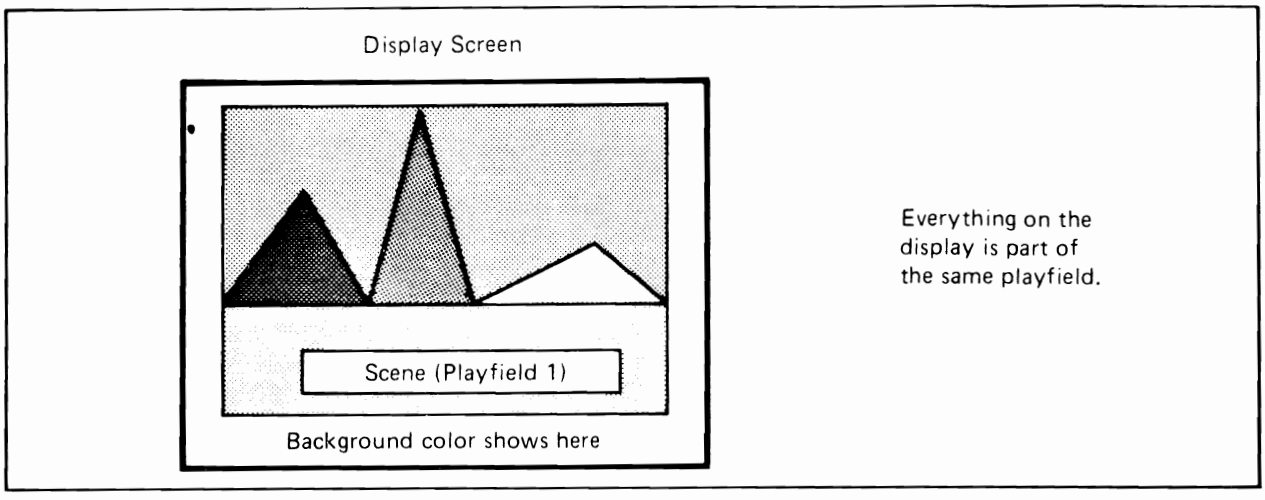


Figure 1-12: A Single Playfield Display

If you use dual-playfield mode (**ViewPort.Modes** = DUALPF), you can define two independent, separately controllable playfield areas.

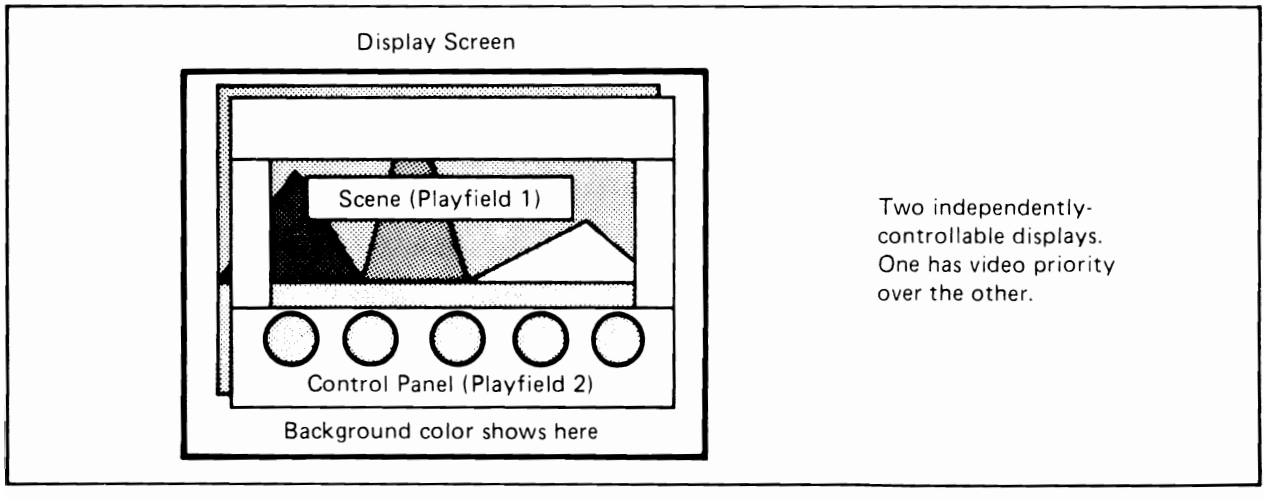


Figure 1-13: A Dual Playfield Display

In Figure 1-13, the display mode bit PFBA is set to 1. If PFBA = 0, the relative priorities will be reversed; playfield 2 will appear to be behind playfield 1.

Low-Resolution Mode vs High-Resolution Mode

In low-resolution mode, horizontal lines of 320 pixels fill most of the ordinary viewing area. The system software lets you define a screen segment width up to 352 pixels in this mode, or you can define a screen segment as narrow as you desire. In high-resolution mode (also called "normal" resolution), 640 pixels fill a horizontal line. In this mode you can specify any range from 0 to 704 pixels wide. Overscan normally limits you to showing only 0 to 320 pixels per line in low-resolution mode or 0 to 640 pixels per line in high-resolution mode. Intuition assumes the nominal 320-pixel or 640-pixel width.

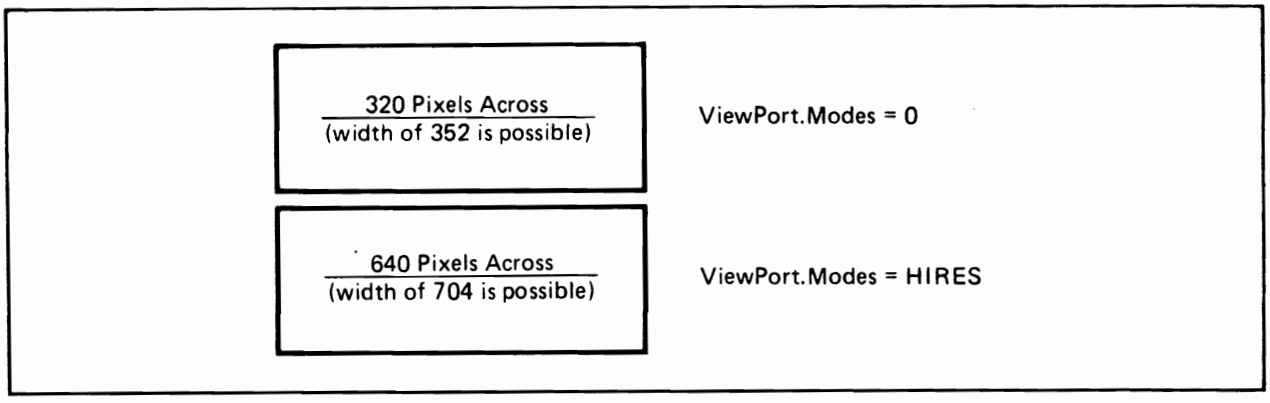


Figure 1-14: How HIRES Affects Width of Pixels

Interlaced Mode vs Non-Interlaced Mode

In interlaced mode, there are twice as many lines available, providing better vertical resolution in the same display area.

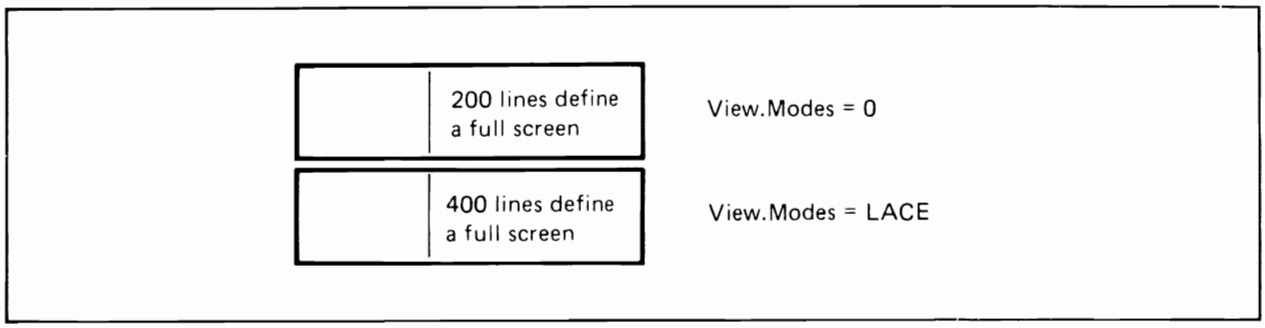


Figure 1-15: How LACE Affects Vertical Resolution

If the **View** structure does not specify LACE, and the **ViewPort** specifies LACE, you may only see every other line of the **ViewPort** data. If the **View** structure specifies LACE and the **ViewPort** is non-interlaced, then the same **ViewPort** data is repeated in both fields. The height of the **ViewPort** display is the height specified in the **ViewPort** structure. If both the **View** and the **ViewPort** are interlaced, then the **ViewPort** is built with double the normal vertical resolution. That means it will need twice as much data space in memory as a non-interlaced picture for this display.

ViewPort Display Memory

The picture you create in memory can be larger than the screen image that can be displayed within your **ViewPort**. This big picture (called a raster and represented by the **BitMap** structure) can have a maximum size of 1024 by 1024. Because a picture this large cannot fit fully on the display, you specify which piece of it to display. Once you have selected the piece to be shown, you can specify where it is to appear on the screen.

The example in Figure 1-16 introduces terms that tell the system how to find the display data and how to display it in the **ViewPort**. These terms are RHeight, RWidth, **RyOffset**, **RxOffset**, **DHeight**, **DWidth**, **DyOffset** and **DxOffset**.

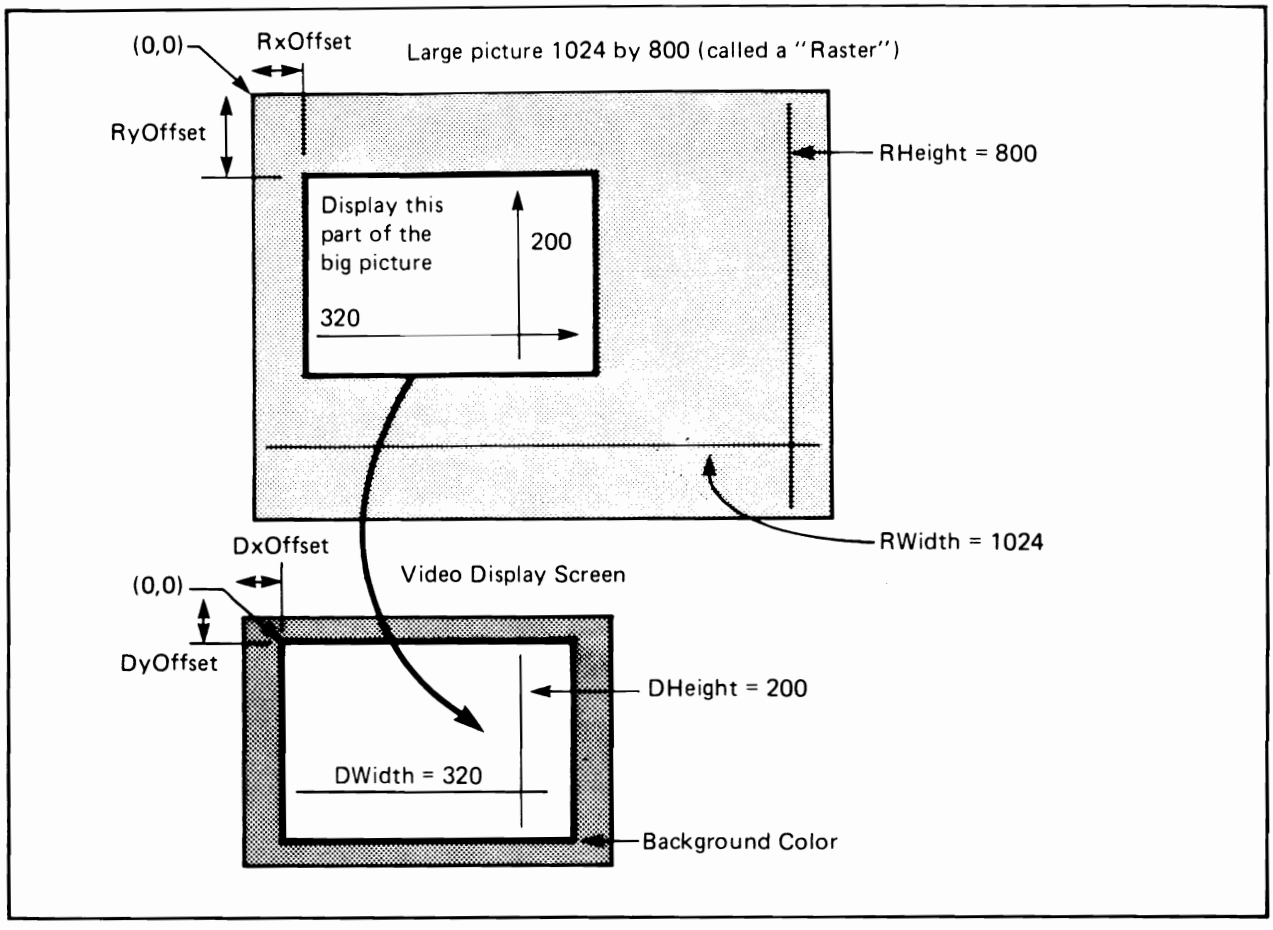


Figure 1-16: ViewPort Data Area Parameters

The terms RHeight and RWidth do not appear in actual system data structures. They refer to the dimensions of the raster and are used here to relate the size of the raster to the size of the display area. RHeight is the number of rows in the raster, and RWidth is bytes per row times 8. The raster shown in the figure is too big to fit entirely in the display area, so you tell the system which pixel of the raster should appear in the upper left corner of the display segment specified by your **ViewPort**. The variables that control that placement are **RyOffset** and **RxOffset**.

To compute **RyOffset** and **RxOffset**, you need RHeight and RWidth and **DHeight** and **DWidth**. The **DHeight** and **DWidth** variables define the height and width in pixels of the portion of the display that you want to appear in the **ViewPort**. The example shows a full-screen, low-resolution mode (320 pixel), non-interlaced (200 line) display formed from the larger overall picture.

Normal values for **RyOffset** and **RxOffset** are defined by the formulas:

$0 <= RyOffset <= (RHeight - DHeight)$
$0 <= RxOffset <= (RWidth - DWidth)$

Once you have defined the size of the raster and the section of that raster that you wish to display, you need only specify where on-screen to put this **ViewPort**. This is controlled by the variables **DyOffset** and **DxOffset**. A value of 0 for each of these offsets places a normal-sized picture in a centered position at the top, bottom, left and right on the display screen. Possible values for **DyOffset** range from -16 to +200 (-32 to +400 if **View.Modes** includes LACE). Possible values for **DxOffset** range from -16 to +352 (-32 to +704 if **ViewPort.Modes** includes HIRES).

The parameters shown in the figure above are distributed in the following data structures:

- **RasInfo** (information about the raster) contains the variables **RxOffset** and **RyOffset**. It also contains a pointer to the **BitMap** structure.

- The **View** (information about the whole display) includes the variables that you use to position the whole display on the screen.

  The **View** structure contains a **Modes** variable used to determine if the whole display is to be interlaced or non-interlaced. It also contains pointers to its list of **ViewPorts** and pointers to the Copper instructions produced by the system to create the display you have defined.

- **ViewPort** (information about this piece of the display) includes the values **DxOffset** and **DyOffset** that are used to position this slice relative to the overall **View**.

  The **ViewPort** also contains the variables **DHeight** and **DWidth**, which define the size of this slice, a **Modes** variable, and a pointer to the local **ColorMap**.

  Each **ViewPort** also contains a pointer to the next **ViewPort**. You create a linked list of **ViewPorts** to define the complete display.

- **BitMap** (information about memory usage) tells the system where to find the display and drawing area memory and shows how this memory space is organized.

You must allocate enough memory for the display you define. The memory you use for the display may be shared with the area control structures used for drawing. This allows you to draw into the same areas that you are currently displaying on-screen.

As an alternative, you can define two **BitMaps**. One of them can be the active structure (that being displayed) and the other can be the inactive structure. If you draw into one **BitMap** while displaying another, the user cannot see the drawing taking place. This is called *double-buffering* of the display. See "Advanced Topics" below for an explanation of the steps required for double-buffering. Double-buffering takes twice as much memory as single-buffering because two full displays are produced.

To determine the amount of required memory for each **ViewPort** for single-buffering, you can use the following formula.

```
bytes_per_ViewPort = Depth * RASSIZE (Width, Height);
```

RASSIZE is a system macro attuned to the current design of the system memory allocation for display rasters.

For example, a 32-color **ViewPort** (depth = 5), 320 pixels wide by 200 lines high (as of this writing) uses 40,000 bytes. A 16-color **ViewPort** (depth = 4), 640 pixels wide by 400 lines high (as of this writing) uses 128,000 bytes.

Forming a Basic Display

This section begins an example that shows how to create a single **ViewPort** with a size of 200 lines, where the area displayed is the same size as the big picture (raster) stored in memory. It also shows how this **ViewPort** becomes the single display segment of a **View** structure. Following the description of the individual operations, the "Graphics Example Program" section pulls all of the pieces into a complete executable program. Instead of linking these routines to drawing routines, the example allocates memory specifically and only for the display (instead of sharing the memory with the drawing routines) and writes data directly to this memory. This keeps the display and the drawing routines separate for purposes of discussion.

Here are the data structures that you need to define to create a basic display:

```
struct View v;        /* The name used here for a View is v, */
struct ViewPort vp;   /* for a ViewPort is vp, */
struct BitMap b;      /* for a BitMap is b, */
struct RasInfo ri;    /* and for a RasInfo is ri. */
```

Opening the Graphics Library

Most of the system routines used here are located in the graphics library. When you compile your program, you must provide a way to tell the compiler to link your calling sequences into the routine library in which they are located. You accomplish this by declaring the variable

called **GfxBase**. Then, by opening the graphics library, you provide the value (address of the library) that the system needs for linking with your program. See the "Libraries" chapter in Part I for more information.

Here is a typical sequence:

```
struct GfxBase *GfxBase;    /* declare the name *GfxBase as a
                               pointer to the corresponding library */
```

Preparing the View Structure

The following code section prepares the **View** structure for further use:

```
InitView( &v );     /* initialize the View structure */

v.ViewPort = &vp;     /* tell the View structure where to find the
                         first ViewPort in a possible list of Viewports */
```

Preparing the ViewPort Structure

The following code section prepares the **ViewPort** structure for further use:

```
InitVPort( &vp );     /* initialize the structure (set up default values) */

vp.DWidth = WIDTH;        /* how wide is the display */
vp.DHeight = HEIGHT;      /* how tall is the display for this viewport */
vp.RasInfo = &ri;         /* pointer to a RasInfo structure */
vp.ColorMap = GetColorMap(32);       /* using a 32-color map */
```

The **InitVPort()** routine presets certain default values. The defaults include:

- **Modes** variable set to zero—this means you select a low-resolution display.

- **Next** variable set to zero—no other **ViewPort** linked to this one. If you want to have multiple **ViewPorts** in a single **View**, you must create the link yourself. The last **ViewPort** in the chain must have a **Next** value of 0.

If you have defined two **ViewPorts**, such as

```
struct ViewPort vpA;
struct ViewPort vpB;
```

and you want them to both be part of the same display, you must create a link between them, and a NULL link at the end of the chain of **ViewPorts**:

```
vpA.Next = &vpB;     /* tell first one the address of the second */
vpB.Next = NULL;     /* after this one, there are no others */
```

Preparing the BitMap Structure

The **BitMap** structure tells the system where to find the display and drawing memory and how this memory space is organized. The following code section prepares a **BitMap** structure, including allocation of memory for the bit-map. This memory is, for this example, used only for the display and is not shared with any drawing routines. The example below writes directly to the display area.

```
                    /* initialize the bitmap structure */
InitBitMap( &b, DEPTH, WIDTH, HEIGHT );
                    /* now allocate some memory that can
                    be linked into the bitmap for display purposes */
for( i=0; i<DEPTH, i++)
    {
    b.Planes[i] = (PLANEPTR)AllocRaster(WIDTH, HEIGHT);
```

This code allocates enough memory to handle the display area for as many bit-planes as the depth you have defined. This code segment does not include the error checking that is present in the full example later on.

Preparing the RasInfo Structure

The **RasInfo** structure provides information to the system about the location of the **BitMap** as well as the positioning of the display area as a window against a larger drawing area. Use the following steps to prepare the **RasInfo** structure:

```
ri.BitMap = &b;    /* specify address of the bitmap structure */
ri.RxOffset = 0;
ri.RyOffset = 0;   /* match the upper left-hand corner of the
                      display area with the upper left corner of
                      the drawing area - see Figure 1-16 */
ri.next = NULL;    /* for a single playfield display, there
                      is only one RasInfo structure present */
```

Preparing the ColorMap Structure

The **ColorMap** structure tells the system which real colors should be used to display this **ViewPort**. When the **View** is created, Copper instructions are generated to change the current contents of each color register just before the topmost line of a **ViewPort** so that this ViewPort's color registers will be used for interpreting its display.

Here are the steps normally used for initializing a **ColorMap**:

```
UWORD colortable [] = { 0, 0xf00, 0x0f0, 0x00f }
            /* define some colors in an array of words */
vp.ColorMap = GetColorMap (4);
            /* allocate space and get a pointer to it */
            /* 4 colors in this table (4 registers for
               Copper to reload before this ViewPort
               is displayed */
LoadRGB4( vp, ColorTable, 4 )
```

NOTE: The 4 in **LoadRGB4()** refers to the fact that each of the red, green, and blue values in a color table entry consists of four bits. It has nothing to do with the fact that this particular color table contains four entries, which is a consequence of the choice of DEPTH

= 2 for this example.

From the section called "ViewPort Color Selection", notice that you might need to specify more colors in the color map than you think. Namely, if you use a dual-playfield display (covered later in this chapter) with a depth of 1 for each of the two playfields, this means a total of four colors (two for each playfield). However, because playfield 2 uses color registers starting from number 8 on up when in dual-playfield mode, the color map must be initialized to contain at least 10 entries. That is, it must contain entries for colors 0 and 1 (for playfield 1), color numbers 8 and 9 (for playfield 2). Space for sprite colors must be allocated as well.

Creating the Display Instructions

Now that you have initialized the system data structures, you can request that the system prepare a set of display instructions for the Copper using these structures as input data. During the one or more blank vertical lines that precede each **ViewPort**, the Copper is busy changing the characteristics of the display hardware to match the characteristics you expect for this **ViewPort**. This may include a change in display resolution, a change in the colors to be used or other user-defined modifications to system registers.

Here is the code that creates the display instructions:

```
MakeVPort( &v, &vp );
```

where **&v** is the address of the **View** structure and **&vp** is the address of the first **ViewPort** structure. Using these structures, the system has enough information to build the instruction stream that defines your display.

MakeVPort() creates a special set of instructions that controls the appearance of the display. If you are using animation, the graphics animation routines create a special set of instructions to control the hardware sprites and the system color registers. In addition, the advanced user can create special instructions to change system operations based on the position of the video beam on-screen (user Copper instructions).

All of these special instructions must be merged together before the system can use them to produce the display you have designed. This is done by the system routine **MrgCop()** (stands for "Merge Coprocessor Instructions"). Here is a typical call:

```
MrgCop (&v); /* merge this View's Copper
                instructions into a single instruction list */
```

Loading and Displaying the View

To display the **View**, you need to load it, using **LoadView()**, and turn on the direct memory access (DMA).

A typical call is shown below.

```
LoadView( &v );
```

where &v is the address of the **View** structure defined in the example above.

Two macros control display DMA: ON_DISPLAY and OFF_DISPLAY. They simply turn the display DMA control bit in the DMA control register on or off. After you have loaded a new **View**, you use ON_DISPLAY to allow the system DMA to display it on-screen.

If you are drawing to the display area and don't want the user to see intermediate steps in the drawing, you can turn off the display. Because OFF_DISPLAY shuts down the display DMA and possibly speeds up other system operations, it can be used to provide additional memory cycles to the blitter or the 68000. The distribution of system DMA, however, allows 4-channel sound, disk read/write, a 16-color, low-resolution display (or 4-color, high-resolution display) to operate at the same time with no slowdown (7.1 megahertz effective rate) in the operation of the 68000.

Graphics Example Program

The program below creates and displays a single-playfield display that is 320 pixels wide, 200 lines high, and two bit-planes deep.

```
#include "exec/types.h"
#include "graphics/gfx.h"
#include "hardware/dmabits.h"
#include "hardware/custom.h"
#include "hardware/blit.h"
#include "graphics/gfxmacros.h"
#include "graphics/copper.h"
#include "graphics/view.h"
#include "graphics/gels.h"
#include "graphics/regions.h"
#include "graphics/clip.h"
```

```
#include "exec/exec.h"
#include "graphics/text.h"
#include "graphics/gfxbase.h"

#define DEPTH 2
#define WIDTH 320
#define HEIGHT 200
#define NOT_ENOUGH_MEMORY -1000
/* construct a simple display */

struct View v;
struct ViewPort vp;
struct ColorMap *cm; /* pointer to colormap structure, dynamic alloc */
struct RasInfo ri;
struct BitMap b; /* note:  Due to the static allocation of a
                  * structure accessed directly by the custom
                  * chips, this program will only work if it
                  * resides entirely within the lower 512k
                  * bytes of memory (CHIP memory)
                  */
struct RastPort rp;

LONG i;
SHORT j,k,n;

extern struct ColorMap *GetColorMap();
struct GfxBase *GfxBase;

struct View *oldview;        /* save pointer to old view so can restore */

USHORT colortable[] = { 0x000, 0xf00, 0x0f0, 0x00f }; /* my own colors */
                                  /* black, red, green, blue */
SHORT  boxoffsets[] = { 802, 2010, 3218 };  /* where to draw boxes */

UBYTE *displaymem;
UWORD *colorpalette;

main()
{
      GfxBase = (struct GfxBase *)OpenLibrary("graphics.library",0);
      if (GfxBase == NULL) exit(1);
      oldview = GfxBase->ActiView; /* save current view to restore later */
      /* example steals screen from Intuition if started from WBench */

      InitView(&v);          /* initialize view */
      InitVPort(&vp); /* init view port */
      v.ViewPort = &vp;    /* link view into viewport */

      /* init bit map (for rasinfo and rastport) */
      InitBitMap(&b,DEPTH,WIDTH,HEIGHT);
```

```
/* (init RasInfo) */
ri.BitMap = &b;
ri.RxOffset = 0;
ri.RyOffset = 0;
ri.Next = NULL;

/* now specify critical characteristics */
vp.DWidth = WIDTH;
vp.DHeight = HEIGHT;
vp.RasInfo = &ri;

/* (init color table) */
cm = GetColorMap(4);      /* 4 entries, since only 2 planes deep */
colorpalette = (UWORD *)cm->ColorTable;
for(i=0; i<4; i++)
      *colorpalette++ = colortable[i];

/* copy my colors into this data structure */
vp.ColorMap = cm;   /* link it with the viewport */

/* allocate space for bitmap */
for(i=0; i<DEPTH; i++)
{
      b.Planes[i] = (PLANEPTR)AllocRaster(WIDTH,HEIGHT);
      if(b.Planes[i] == NULL) exit(NOT_ENOUGH_MEMORY);
}

MakeVPort( &v, &vp );      /* construct copper instr (prelim) list */
MrgCop( &v );          /* merge prelim lists together into a real
                  * copper list in the view structure. */

for(i=0; i<2; i++)
{
      displaymem = (UBYTE *)b.Planes[i];
      for(j=0; j<RASSIZE(WIDTH,HEIGHT); j++) {
            *displaymem++ = 0;
      }
      /* zeros to all bytes of the display area */
}

LoadView(&v);
/* now fill some boxes so that user can see something */
/* always draw into both planes to assure true colors */
for(n=1; n<4; n++)      /* three boxes */
{
      for(k=0; k<2; k++)
      {
            /* boxes will be in red, green and blue */
            displaymem = b.Planes[k] + boxoffsets[n-1];
            DrawFilledBox(n,k);
```

```
            }
        }

        for(i=0; i<100000;i++) ;    /* do nothing for a while */

        LoadView(oldview);          /* put back the old view  */

        FreeMemory();               /* exit gracefully */
        CloseLibrary(GfxBase);      /* since opened library, close it */

}       /* end of main() */

/* return user and system-allocated memory to sys manager */
FreeMemory()
{
        /* free drawing area */
        for(i=0; i<DEPTH; i++)
                FreeRaster(b.Planes[i],WIDTH,HEIGHT);
        /* free the color map created by GetColorMap() */
        FreeColorMap(cm);
        /* free dynamically created structures */
        FreeVPortCopLists(&vp);
        FreeCprList(v.LOFCprList);
        return(0);
}
DrawFilledBox(fillcolor,plane)
SHORT fillcolor,plane;
{
        UBYTE value;
        for(j=0; j<100; j++)
        {
                if((fillcolor & (1 << plane)) != 0)
                        value = 0xff;
                else
                        value = 0;
                for(i=0; i<20; i++)
                {
                        *displaymem++ = value;
                }
                displaymem += (b.BytesPerRow - 20);
        }
        return(0);
}
```

Exiting Gracefully

The sample program above provides a way of exiting gracefully, returning to the memory manager all dynamically-allocated memory chunks.

Notice the calls to **FreeRaster()** and **FreeColorMap()**. These calls correspond directly to the allocation calls **AllocRaster()** and **GetColorMap()** located in the body of the program.

Now look at the calls within FreeMemory() to **FreeVPortCopLists()** and **FreeCprList()**.

When you call **MakeVPort()**, the graphics system dynamically allocates some space to hold intermediate instructions from which a final Copper instruction list is created. When you call **MrgCop()**, these intermediate Copper lists are merged together into the final Copper list, which is then given to the hardware for interpretation. It is this list that provides the stable display on-screen, split into separate **ViewPorts** with their own colors and resolutions and so on.

When your program completes, you must see that it returns all of the memory resources that it used, so that those memory areas are again available to the system for reassignment to other projects. Therefore, if you use the routines **MakeVPort()** or **MrgCop()**, you must also arrange to use **FreeCprList()** (pointing to each of those lists in the **View** structure) and **FreeVPortCopLists()** (pointing to the **ViewPort** that is about to be deallocated). If your view is interlaced, you will also have to call **FreeCprList(&v.SHFCprList)** because an interlaced view has a separate copper list for each of the two fields displayed.

As a final caveat, notice that when you do free everything, the memory manager or other programs may immediately change the contents of the freed memory. Therefore, if the Copper is still executing an instruction stream (as a result of a previous **LoadView()**) when you free that memory, the display will go "south". when you free that memory, You will probably want to turn off the display, or provide an alternate Copper list when this one is to be deallocated.

Advanced Topics

Creating a Dual-Playfield Display

In dual-playfield mode, you have two separately controllable playfields. In this mode, you always define two **RasInfo** data structures. Each of these structures defines one of the playfields. There are five different ways you can configure a dual-playfield display, because there are five different distributions of the bit-planes which the system hardware allows. Table 1-4 shows these distributions.

Table 1-4: Bit-Plane Assignment in Dual-playfield Mode

Number of Bit-planes	Playfield 1 Depth	Playfield 2 Depth
0	0	0
1	1	0
2	1	1
3	2	1
4	2	2
5	3	2
6	3	3

Recall that if you set PFBA in the **ViewPort Modes** variable to 1, you can swap playfield priority and display Playfield 2 in front of Playfield 1. In this way, you can get more bit-planes in the background playfield than you have in the foreground playfield. If you create a display with multiple **ViewPorts**, only for this **ViewPort** will the playfield priority be changed.

Playfield 1 is defined by the first of the two **RasInfo** structures. Playfield 2 is defined by the second of the two **RasInfo** structures.

When you call **MakeVPort()**, you use parameters as follows:

```
MakeVPort( &view, &viewport );
```

The **ViewPort Modes** variable must include the DUALPF bit. This tells the graphics system that there are two **RasInfo** structures to be used.

In summary, to create a dual-playfield display you must:

- o allocate one **View** structure
- o allocate two **BitMap** structures
- o allocate two **RasInfo** structures (linked together), each pointing to different **BitMaps**
- o allocate one **ViewPort** structure
- o set up a pointer in the **ViewPort** structure to the playfield 1 **RasInfo**
- o initialize each **BitMap** structure to describe one playfield, using one of the permissible bit-plane distributions shown in Table 1-4 and allocate memory for the bit-planes themselves.

  Note that **BitMap** 1 and **BitMap** 2 need *not* be the same width and height.
- o initialize the **ViewPort** structure
- o set the DUALPF (and possibly the PFBA) bit in the ViewPort **Modes** variable
- o call **MakeVPort()**
- o call **MrgCop()**

For display purposes, each of the two **BitMaps** is assigned to a separate playfield display.

To draw separately into the **BitMaps**, you must also assign these **BitMaps** to two separate **RastPorts**. The section called "Initializing the RastPort" shows you how to use a **RastPort** data structure to control your drawing routines.

Creating a Double-Buffered Display

To produce smooth animation or other such effects, it is occasionally necessary to double-buffer your display. To prevent the user from seeing your graphics rendering while it is in progress, you will want to draw into one memory area while actually displaying a different area.

Double-buffering consists of creating two separate display areas and two sets of pointers to those areas for a single **View**.

To create a double-buffered display, you must:

- Allocate two **BitMap** structures.
- Allocate one **RasInfo** structure.
- Allocate one **ViewPort** structure.
- Allocate one **View** structure.
- Initialize each **BitMap** structure to describe one drawing area and allocate memory for the bit-planes themselves.
- Create a pointer for each **BitMap**.
- Create a pointer for the **View** long-frame Copper list (**LOFCprList**) and short-frame Copper list (**SHFCprList**) for each of two alternate display fields. The **SHFCprList** is for interlaced displays.
- Initialize the **RasInfo** structure, setting the **BitMap** pointer to point to one of the two **BitMaps** you have created.
- Call **MakeVPort()**.
- Call **MrgCop()**.

When you call **MrgCop()**, the system uses all of the information you have provided in the various data structures to create a list of instructions for the Copper to execute. This list tells the Copper how to split the display and how to specify colors for the various portions of the display. When the steps shown above have been completed, the system will have allocated memory for a long-frame (LOF) Copper list list) and a short-frame (SHF) Copper list and set pointers called **LOFCprList** and **SOFCprList** in the **View** structure. The long-frame Copper list is normally used for all non-interlaced displays, and the short-frame Copper list is used only when interlaced mode is turned on. The pointers point to the two sets of Copper instructions.

The **LOFCprList** and **SHFCprList** pointers are initialized when **MrgCop()** is called. The instruction stream referenced by these pointers includes references to the first **BitMap**.

You must now do the following:

- Save the current values in backup pointers and set the values of **LOFCprList** and **SHFCprlist** in the **View** structure to zero. When you next perform **MrgCop()**, the system automatically allocates another memory area to hold a new list of

instructions for the Copper.

- Install the pointer to the other **BitMap** structure in the **RasInfo** structure before your call to **MakeVPort()**, and then call **MakeVPort** and **MrgCop**.

Now you have created two sets of instruction streams for the Copper, one of which you have saved in a pair of pointer variables. The other has been newly created and is in the **View** structure. You can save this new set of pointers as well, swapping in the set which you want to use for display, and meanwhile drawing into the **BitMap** which is not on the display. Remember that you will have to call **FreeCprList()** on both sets of copper lists when you have finished.

Hold-and-modify Mode

In hold-and-modify mode you can create a single-playfield display in which 4,096 different colors can be displayed simultaneously. This requires that your **ViewPort** be defined using 6 bit-planes and that you set the HAM bit in the **ViewPort Modes** variable.

When you draw into the **BitMap** associated with this **ViewPort**, you can choose one of four different ways of drawing into the **BitMap**. (Drawing into a **BitMap** is shown in Section 1-3, "Drawing Routines".) If you draw using color numbers 0-15, the pixel you draw will appear in the color specified in that particular system color register.

If you draw with any other color value from 16-31, the color displayed depends on the color of the pixel which is to the immediate left of this pixel on-screen. For example, hold constant the contents of the red and the green parts of the previously produced color, and take the rest of the bits of this new pixel's color register number as the new contents for the blue part of the color. Hold-and-modify means hold part and modify part of the preceding defined pixel's color.

Note that a particular hold-and-modify pixel can only change one of the three color values at a time. Thus, the effect has a limited control.

In hold and modify mode, you use all six bit-planes. planes 5 and 6 are used to modify the way bits from planes 1 - 4 are treated, as follows:

- If the 6-5 bit combination from planes 6 and 5 for any given pixel is 00, normal color selection procedure is followed. Thus, the bit combinations from planes 4-1, in that order of significance, are used to choose one of 16 color registers (registers 0 - 15).

  If only 5 bit-planes are used, the data from the 6th plane is automatically supplied

with the value as 0.

- If the 6-5 bit combination is 01, the color of the pixel immediately to the left of this pixel is duplicated and then modified. The bit-combinations from planes 4 - 1 are used to replace the 4 "blue" bits in the pixel color without changing the value in any color register.

- If the 6-5 bit combination is 10, the color of the pixel immediately to the left of this pixel is duplicated and then modified. The bit-combinations from planes 4 - 1 are used to replace the 4 "red" bits.

- If the 6-5 bit combination is 11, the color of the pixel immediately to the left of this pixel is duplicated and then modified. The bit-combinations from planes 4 - 1 are used to replace the 4 "green" bits.

1.3. DRAWING ROUTINES

Most of the graphics drawing routines require information about how the drawing is to take place. For this reason, the graphics support routines provide a data structure called a **RastPort**, which contains information essential to the graphics drawing functions. You must pass a pointer to your **RastPort** structure to most of the drawing functions. Associated with the **RastPort** is another data structure called a **BitMap**, which contains a description of the organization of the data in the drawing area.

Initializing a BitMap Structure

The **RastPort** contains information for controlling the drawing. In order to use the graphics, you also need to tell the system the memory area location where the drawing will occur. You do this by initializing a **BitMap** structure, defining the characteristics of the drawing area, as shown in the following example. This was already shown in the section called "Forming a Basic Display", but is repeated here because it relates to drawing as well as to display routines. You need not necessarily use the same **BitMap** for both the drawing and the display.

```
struct BitMap myBitMap;
SHORT depth = 3;    /* max of 8 colors ... going to
                    * need 3 bit planes to represent
                    * this number of colors */
SHORT width = 320;
SHORT height = 200;

InitBitMap( &myBitMap, depth, width, height);
```

Initializing a RastPort Structure

Before you can use a **RastPort** for drawing, you must initialize it. Here is a sample initialization sequence:

```
struct RastPort myRastPort;
InitRastPort(&myRastPort);

     /* now link together the bitmap and the rastport */
myRastPort.BitMap = &myBitMap;
```

Note that you *cannot* perform the link until *after* the **RastPort** has been initialized.

The **RastPort** data structure can be found in the include files *rastport.h* and *rastport.i*. It contains the following information:

- Drawing pens
- Drawing modes
- Patterns
- Text attributes and font information
- Area filling information
- Graphics elements information for animation
- Current pen position

- A write mask
- Some graphics private data
- A pointer for user extensions

The following sections explain each of the items in the **RastPort** structure.

Drawing Pens

The Amiga has three different drawing "pens" associated with the graphics drawing routines. These are:

- **FgPen** — the foreground or primary drawing pen. For historical reasons, it's also called the A-Pen.
- **BgPen** — the background or secondary drawing pen. For historical reasons, it's also called the B-Pen.
- **AOlPen** — the area outline pen. For historical reasons, it's also called the O-Pen.

A drawing pen variable in the **RastPort** contains the current value (range 0-255) for a particular color choice. This value represents a color register number whose contents are to be used in rendering a particular type of image. In essence, the bits of a "pen" determine which bit-planes are affected when a color is written into a pixel (as determined by the drawing mode and modified by the pattern variables and the write mask as described below). The drawing routines support **BitMaps** up to 8 planes deep, allowing for future expansion in the hardware.

NOTE

> The Amiga 1000 contains only 32 color registers. Any range beyond that repeats the colors in 0-31. For example, pen numbers 32-63 refer to the colors in registers 0-31.

The color in **FgPen** is used as the primary drawing color for rendering lines and areas. This pen is used when the drawing mode is JAM1 (see the next section for drawing modes). JAM1 specifies that only one color is to be "jammed" into the drawing area.

You establish the color for **FgPen** by the statement:

```
SetAPen( &myRastPort, newcolor );
```

The color in **BgPen** is used as the secondary drawing color for rendering lines and areas. If you specify that the drawing mode is JAM2, (jamming 2 colors) and a pattern is being drawn, the primary drawing color (**FgPen**) is used where there are 1's in the pattern. The secondary drawing color (**BgPen**) is used where there are 0's in the pattern.

You establish the drawing color for **BgPen** by the statement:

```
SetBPen( &myRastPort, newcolor );
```

The area outline pen **AOlPen** is used in two applications: area fill and flood fill. (See "Area Fill Operations" below.) In area fill, you can specify that an area, once filled, can be outlined in this **AOlPen** color. In flood fill (in one of its operating modes) you can fill until the flood-filler hits a pixel of the color specified in this pen variable.

You establish the drawing color for **AOlPen** by the statement:

```
SetOPen( &myRastPort, newcolor );
```

Drawing Modes

There are four drawing modes that you can specify:

JAM1 — Whenever you execute a graphics drawing command, one color is jammed into the target drawing area. You use only the primary drawing pen color and for each pixel drawn, you *replace* the color at that location with the **FgPen** color.

JAM2 — Whenever you execute a graphics drawing command, two colors are jammed into the target drawing area. This mode tells the system that the pattern (both line pattern and area pattern—see the next section) variables are to be used for the drawing. Wherever there is a 1-bit in the pattern variable, the **FgPen** color replaces the color of the pixel at the drawing position. Wherever there is a 0-bit in the pattern variable, the **BgPen** color is used.

COMPLEMENT

For each 1-bit in the primary drawing pen (**FgPen**) the corresponding bit in the target drawing area is complemented; that is, its state is reversed. Complement mode is often used for drawing, then erasing, lines.

INVERSEVID

This is the drawing mode used primarily for text. If the drawing mode is (JAM1 | INVERSEVID), the text appears as a transparent letter surrounded by the **FgPen** color. If the drawing mode is (JAM2|INVERSEVID), the text appears as in (JAM1|INVERSEVID) except that the **BgPen** color is used to draw the text character itself. In this mode, the roles of **FgPen** and **BgPen** are effectively reversed.

You set the drawing modes with the statement:

```
SetDrMd( &myRastPort, newmode );
```

Patterns

The **RastPort** data structure provides two different pattern variables which it uses during the various drawing functions: a line pattern, and an area pattern.

The line pattern is 16-bits wide and is applied to all lines. When you initialize a **RastPort**, this line pattern value is set to all 1's (hex FFFF), so that solid lines are drawn.

You can also set this pattern to other values to draw dotted lines if you wish. For example, you can establish a dotted line pattern with the statement:

```
SetDrPt( &myRastPort, 0xcccc );
```

where "cccc" is a bit-pattern, 1100110011001100, to be applied to all lines drawn. If you draw multiple, connected lines, the pattern cleanly connects all the points.

The area pattern is 16 bits wide and its height is some power of two. This means that you can define patterns in heights of 1, 2, 4, 8, 16, and so on. To tell the system how large a pattern you are providing, include this statement:

```
SetAfPt( &myRastPort, &myAreaPattern, power_of_two );
```

where **&myAreaPattern** is the address of the first word of the area pattern and **power_of_two** specifies how many words are in the pattern. For example:

```
USHORT myAreaPattern[ ] = {
        0xff00,
        0xff00,
        0x00ff,
        0x00ff,
        0xf0f0,
        0xf0f0,
        0x0f0f,
        0x0f0f  };


SetAfPt( &myRastPort, &myAreaPattern, 3 );
```

This example produces a pattern which is a large checkerboard above a small checkerboard. Because **power_of_two** is set to 3, the pattern is 2 to the 3rd, or 8 rows high.

Pattern Positioning

The pattern is always positioned with respect to the upper left corner of the **RastPort** drawing area (the 0,0 coordinate). If you draw two rectangles whose edges are adjacent, the pattern will be continuous across the rectangle boundaries.

Multi-Colored Patterns

The last example above produces a two-color pattern with one color where there are 1's and the other color where there are 0's in the pattern. A special mode allows you to develop a pattern having up to 256 colors. To create this effect, specify **power_of_two** as a negative value instead of a positive value.

The following initialization establishes an 8-color checkerboard pattern where each square in the checkerboard has a different color. The checkerboard is 2 squares wide by 4 squares high.

```
USHORT myAreaPattern[ ] = {

        0x0000,
        0x0000,
        0xffff,
        0xffff,
        0x0000,
        0x0000,
        0xffff,
        0xffff,


        0x0000,       /* plane 2 pattern */
        0x0000,
        0x0000,
        0x0000,
        0xffff,
        0xffff,
        0xffff,
        0xffff,

        0xff00,       /* plane 3 pattern */
        0xff00,
        0xff00,
        0xff00,
        0xff00,
        0xff00,
        0xff00,
        0xff00     };

    SetAfPt( &myRastPort, &myAreaPattern, -3 );

  /* when doing this, best to set three other parameters as follows: */
     SetAPen( &myRastPort, 255);
     SetBPen( &myRastPort, 0);
     SetDrMd( &myRastPort, JAM2);
```

If you use this multi-colored pattern mode, you must provide as many planes of pattern data as there are planes in your **BitMap**.

Text Attributes

Text attributes and font information are set by calls to the font routines. These are covered separately in Chapter 4, "Text".

Area Fill Information

Two structures in the **RastPort**, **AreaInfo** and **TmpRas**, define certain information for area filling operations. The **AreaInfo** pointer is initialized by a call to the routine **InitArea()**.

```
InitArea (&myRastPort, &areabuffer, count);
```

To use area fill, you must first provide a work space in memory for the system to store the list of points that define your area. You must allow a storage space of 5 bytes per vertex. To create the areas in the work space, you use the functions **AreaMove()**, **AreaDraw()**, and **AreaEnd()**.

Typically, you prepare the **RastPort** for area filling by a sequence like the following.

```
UWORD areabuffer [250];
    /* allow up to 100 vertices in the definition of an area */
InitArea (&myRastPort, &areabuffer[0], 100);
```

The area buffer *must* start on a WORD boundary. That is why the sample declaration shows **areabuffer** as composed of unsigned words (250), rather than unsigned bytes (500). It still reserves the same amount of space, but aligns the data space correctly.

In addition to the **AreaInfo** structure in the **RastPort**, you must also provide the system with some work space to build the object whose vertices you are going to define. This requires that you initialize a **TmpRas** structure, then point to that structure for your **RastPort** to use.

Here is sample code that builds and initializes a **TmpRas**. Note that the area to which **TmpRas.RasPtr** points must be at least as large as the area (width times height) of the largest rectangular region you plan to fill. Typically, you allocate a space as large as a single

bit-plane (usually 320 by 200 bits for low-resolution mode, 640 by 200 bits for high-resolution mode).

```
PLANEPTR myplane;
myplane = AllocRaster(320,200);        /* get some space */
if (myplane == 0) exit(1);             /* stop if no space */
myRastPort.TmpRas= InitTmpRas(&myTmpRas,
      myplane,RASSIZE(320,200));
```

When you use functions that dynamically allocate memory from the system, you must remember to return these memory blocks to the system before your program exits. See the description of **FreeRaster()** in the appendixes.

Graphics Element Pointer

The graphics element pointer in the **RastPort** structure is called **GelsInfo**. If you are doing graphics animation using the GELS system, this pointer must refer to a properly initialized **GelsInfo** structure. See Chapter 3, "Animation", for more information.

Current Pen Position

The graphics drawing routines keep the current position of the drawing pen in the variables **cp_x** and **cp_y**, for the horizontal and vertical positions, respectively. The coordinate location 0,0 is in the upper left corner of the drawing area. The x value increases proceeding to the right; the y value increases proceeding toward the bottom of the drawing area.

Write Mask

The write mask is a **RastPort** variable that determines which of the bit-planes are currently writable. For most applications, this variable contains all 1's (hex ff). This means that all bit-planes defined in the **BitMap** are affected by a graphics writing operation. You can

selectively disable one or more bit-planes by simply specifying a 0-bit in that specific position in the control byte. For example:

```
/* disable bitplane 2 */

myRastPort.Mask = 0xFB;
```

Using the Graphics Drawing Routines

This section shows you how to use the Amiga drawing routines. All of these routines work either on their own or with the windowing system and layer library. See Chapter 2, "Layers" or *Intuition: The Amiga User Interface* for details about using the layer library and windows.

As you read this section, keep in mind that to use the drawing routines, you need to pass them a pointer to a **RastPort**. You can define the **RastPort** directly, as shown in the sample program segments in preceding sections, or you can get a **RastPort** from your window structure with code like the following.

```
struct Window *w;
struct RastPort *usableRastPort;
      /* and then, after your Window is initialized... */
usableRastPort = w->RastPort;
```

You can also get the **RastPort** from the layer structure, if you are not using Intuition.

Drawing Individual Pixels

You can set a specific pixel to a desired color by using a statement like this:

```
INT result;
result = WritePixel( &myRastPort, x, y);
```

WritePixel() uses the primary drawing pen and changes the pixel at that x,y position to the desired color if the x,y coordinate falls within the boundaries of the **RastPort**. A value of 0 is returned if the write was successful; a value of -1 is returned if x,y was outside the range

of the **RastPort**.

Reading Individual Pixels

You can determine the color of a specific pixel with a statement like this:

```
INT    result;
result = ReadPixel( &myRastPort, x, y);
```

ReadPixel() returns the value of the pixel color selector (from 0 to 255) at the specified x,y location. If you specify an x,y outside the range of your **RastPort**, this function returns a value of -1.

Drawing Lines

Two functions are associated with line drawing: **Move()** and **Draw()**. **Move()** simply moves the cursor to a new position. It is like picking up a drawing pen and placing it at a new location. This function is executed by the statement:

```
Move( &myRastPort, x, y);
```

Draw() draws a line from the current x,y position to a new x,y position specified in the statement itself. The drawing pen is left at the new position. This is done by the statement:

```
Draw( &myRastPort, x, y);
```

Draw() uses the pen color specified for **FgPen**. Here is a sample sequence that draws a red line from location (0,0) to (100,50). Assume that the value in color register 2 represents red.

```
SetAPen( &myRastPort, 2);    /* make primary pen red */
Move( &myRastPort, 0, 0);    /* move to new location */
Draw( &myRastPort, 100,50);  /* draw to a new location */
```

CAUTION

If you attempt to draw a line outside the bounds of the **BitMap**, using the basic initialized **RastPort**, you may possibly crash the system. You must either do your own software clipping to assure that the line is in range, or use the layer library. Software clipping means that you need to determine if the line will fall outside your **BitMap** before you draw it.

Drawing Patterned Lines

To turn the example above into a patterned line draw, simply add the following statement:

```
SetDrPt( &myRastPort, 0xaaaa);
```

Now all lines drawn appear as dotted lines. To resume drawing solid lines, execute the statement:

```
SetDrPt( &myRastPort, -1);
```

Drawing Multiple Lines with a Single Command

You can use multiple **Draw()** statements to draw connected line figures. If the shapes are all definable as interconnected, continuous lines, you can use a simpler function, called **PolyDraw()**. **PolyDraw()** takes a set of line endpoints and draws a shape using these points. You call **PolyDraw()** with the statement:

```
PolyDraw( &myRastPort, count, arraypointer);
```

PolyDraw() reads an array of points and draws a line from the current pen position to the first, then a connecting line to each succeeding position in the array until **count** points have been drawn. This function uses the current drawing mode, pens, line pattern and write mask specified in the target **RastPort**; for example:

```
SHORT linearray[ ] = {
    3,3,
    15,3,
    15,15,
    3,15,
    3,3     };
PolyDraw( &myRastPort, 5, &linearray[0]);
```

draws a rectangle, using the 5 defined pairs of x,y coordinates.

Area Fill Operations

Assuming that you have properly initialized your **RastPort** structure to include a properly initialized **AreaInfo**, you can perform area fill by using the functions described in this section.

AreaMove() tells the system to begin a new polygon, closing off any other polygon which may already be in process by connecting the end-point of the previous polygon to its starting point. **AreaMove()** is executed with the statement:

```
AreaMove( &myRastPort, x, y);
```

AreaDraw() tells the system to add a new vertex to a list which it is building. No drawing takes place when **AreaDraw()** is executed. It is executed with the statement:

```
AreaDraw( &myRastPort, x, y);
```

AreaEnd() tells the system to draw all of the defined shapes and fill them. When this function is executed, it obeys the drawing mode and uses the line pattern and area pattern specified in your **RastPort** to render the objects you have defined. Note that to fill an area, you do not have to **AreaDraw()** back to the first point before calling **AreaEnd()**. **AreaEnd()** automatically closes the polygon. **AreaEnd()** is executed with the following statement:

```
AreaEnd( &myRastPort);
```

Here is a sample program segment that includes the **AreaInfo** initialization. It draws a pair of disconnected triangles, using the currently defined **FgPen**, **BgPen**, **AOlPen**,

DrawMode, **LinePtrn**, and **AreaPtrn**:

```
WORD areabuffer[250];
struct RastPort *rp;
struct TmpRas tmpras;
struct AreaInfo myAreaInfo;

InitArea(myAreaInfo, areabuffer, 100);
rp->AreaInfo = &myAreaInfo;
rp->TmpRas = InitTmpRas( &tmpras, AllocRaster(320,200), RASSIZE(320,200) );

/* area routines need a temporary raster buffer at least
 * as large as the largest object to be drawn
 * If a single task uses multiple rastports, it is sometimes
 * possible to share the same TmpRas structure among multiple rastports
 * Multiple tasks, however, cannot share a TmpRas,
 * as each task won't know when
 * another task has a drawing partially completed.
 */

AreaMove( rp, 0,0 );
AreaDraw( rp, 0,100);
AreaDraw( rp, 100,100);

AreaMove( rp, 50,10);
AreaDraw( rp, 50,50);
AreaDraw( rp, 100,50);

AreaEnd ( rp );
```

If you had executed the statement "SetOPen(&myRastPort, 3)" in the area fill example, then the areas that you had defined would have been outlined in pen color 3. To turn off the outline function, you have to set the **RastPort Flags** variable back to 0 by:

```
BOUNDARY_OFF(rp);
```

Otherwise, every subsequent area fill or rectangle fill operation will use the outline pen.

CAUTION

If you attempt to fill an area outside the bounds of the **BitMap**, using the basic initialized **RastPort**, it may possibly crash the system. You must either do your own software clipping to assure that the area is in range, or use the layer library.

Flood Fill Operations

Flood fill is a technique for filling an arbitrary shape with a color. The Amiga flood fill routines can use either a plain color or combine the drawing mode, **FgPen**, **BgPen**, and the area pattern to do the fill.

There are two different modes for flood fill:

- In *outline mode* you specify an x,y coordinate, and from that point the system searches outward in all directions for a pixel whose color is the same as that specified in the area outline pen. All horizontally or vertically adjacent pixels *not* of that color are filled with a colored pattern or plain color. The fill stops at the outline color. Outline mode is selected when the **mode** variable is a "0".

- In *color mode* you specify an x,y coordinate, and whatever pixel color is found at that position defines the area to be filled. The system searches for all horizontally or vertically adjacent pixels whose color is the same as this one and replaces them with the colored pattern or plain color. Color mode is selected when the **mode** variable is a "1".

You use the **Flood()** routine for flood fill. The syntax for this routine follows.

```
Flood( rp, mode, x, y);
```

where:

rp is a pointer to the **RastPort**

x,y
is the starting coordinate in the **BitMap**

mode
tells how to do the fill

The following sample program fragment creates and then flood fills a triangular region. The overall effect is exactly the same as shown in the preceding area fill example above except that flood fill is slightly slower than area fill. Mode 0 (fill to a pixel that has the color of the outline pen) is used in the example.

```
oldFgPen = myRastPort.FgPen;
SetAPen( &myRastPort, myRastPort.AOlPen);
      /* using mode 0 */
Move( &myRastPort, 0, 0);
Draw( &myRastPort, 0, 100);
Draw( &myRastPort, 100, 100);
    /* triangular shape */
Draw( &myRastPort, 0, 0);      /* close it */

SetAPen( &myRastPort, oldFgPen);
Flood(&myRastPort, 0, 10, 50);
```

This example saves the current **FgPen** value and draws the shape in the same color as **AOlPen**. Then **FgPen** is restored to its original color so that **FgPen**, **BgPen**, **DrawMode**, and **AreaPtrn** can be used to define the fill within the outline.

Rectangle Fill Operations

The final fill function, **RectFill()**, is for filling rectangular areas. The form of this function follows.

```
RectFill( rp, xmin, ymin, xmax, ymax);
```

where:

xmin and **ymin**
represent the upper left corner of the rectangle

xmax and **ymax**
represent the lower right corner of the rectangle

rp points to the **RastPort** that receives the filled rectangle

Rectangle fill uses **FgPen**, **BgPen**, **AOlPen**, **DrawMode** and **AreaPtrn** to fill the area you specify. Remember that the fill can be multi-colored as well as single- or two-colored.

The following three sets of statements perform exactly the same function:

```
/* areafill a rectangular area */
SetAPen(rp,1);
SetOPen(rp,3);
AreaMove(rp,0,0);
AreaDraw(rp,0,100);
AreaDraw(rp,100,100);
AreaDraw(rp,100,0);
AreaEnd(rp);

/* floodfill a rectangular area */
SetAPen(rp,3);
SetOPen(rp,3);
Move(rp,0,0);
Draw(rp,0,100);
Draw(rp,100,100);
Draw(rp,100,0);
Draw(rp,0,0);
SetAPen(rp,1);
Flood(rp,0,50,50);

/* rectfill a rectangular area */
SetAPen(rp,1);
SetOPen(rp,3);
Rectfill(rp,0,0,100,100);
```

Not only is the **RectFill()** routine the shortest, it is also the fastest to execute.

Data Move Operations

The graphics support functions include several routines for simplifying the handling of the rectangularly organized data that you would encounter when doing raster-based graphics. These routines:

- Clear an entire segment of memory
- Set a raster to a specific color
- Scroll a subrectangle of a raster
- Draw a pattern "through a stencil"
- Extract a pattern from a bit-packed array and draw it into a raster
- Copy rectangular regions from one bit-map to another
- Control and utilize the hardware-based data mover, the blitter.

The following sections cover these routines in detail.

Clearing a Memory Area

For memory that is accessible to the blitter (that is, internal CHIP memory), the most efficient way to clear a range of memory is to use the the blitter.

You use the blitter to clear a block of memory with the statement:

```
BltClear( memblock, bytecount, flags);
```

where **memblock** is a pointer to the location of the first byte to be cleared, and **bytecount** is the number of bytes to set to zero.

This command accepts the starting location and count and clears that block to zeros. For the meanings of settings of the **flags** variable, see the summary page for this routine in the appendixes.

Setting a Whole Raster to a Color

You can preset a whole raster to a single color by using the function **SetRast()**. A call to this function takes the following form.

```
SetRast( RastPort, pen);
```

where:

RastPort
is a pointer to the **RastPort** you wish to use

pen
is the pen value that you wish to fill that **RastPort**

Scrolling a Sub-Rectangle of a Raster

You can scroll a sub-rectangle of a raster in any direction—up, down, left, right, or diagonally. To perform a scroll, you use the **ScrollRaster()** routine and specify a dx and dy (delta-x, delta-y) by which the rectangle image should be moved towards the (0,0) location.

As a result of this operation, the data within the rectangle will become physically smaller by the size of delta-x and delta-y, and the area vacated by the data when it has been cropped and moved is filled with the background color (color in **BgPen**).

Here is the syntax of the **ScrollRaster()** function:

```
ScrollRaster( rp, dx, dy, xmin, ymin, xmas, ymax );
```

where:

rp is a pointer to a **RastPort**.

dx, **dy**
are the distances (positive, 0, or negative) to move the rectangle

xmin, **xmax**, **ymin**, **ymax**
specify the outer bounds of the sub-rectangle

Here are some examples that scroll a sub-rectangle:

```
ScrollRaster(&myRastPort,0,2,10,10,50,50);
    /* scroll up 2 */

ScrollRaster(&myRastPort,1,0,10,10,50,50);
    /* scroll left 1 */
```

Drawing Through a Stencil

The routine **BltPattern()** allows you to change only a very selective portion of a drawing area. Basically, this routine lets you define the rectangular region to be affected by this drawing operation and a mask of the same size that defines how that area will be affected.

Figure 1-17 shows an example of what you can do with **BltPattern()**. The 0-bits are represented by blank rectangles, the 1-bits by filled-in rectangles.

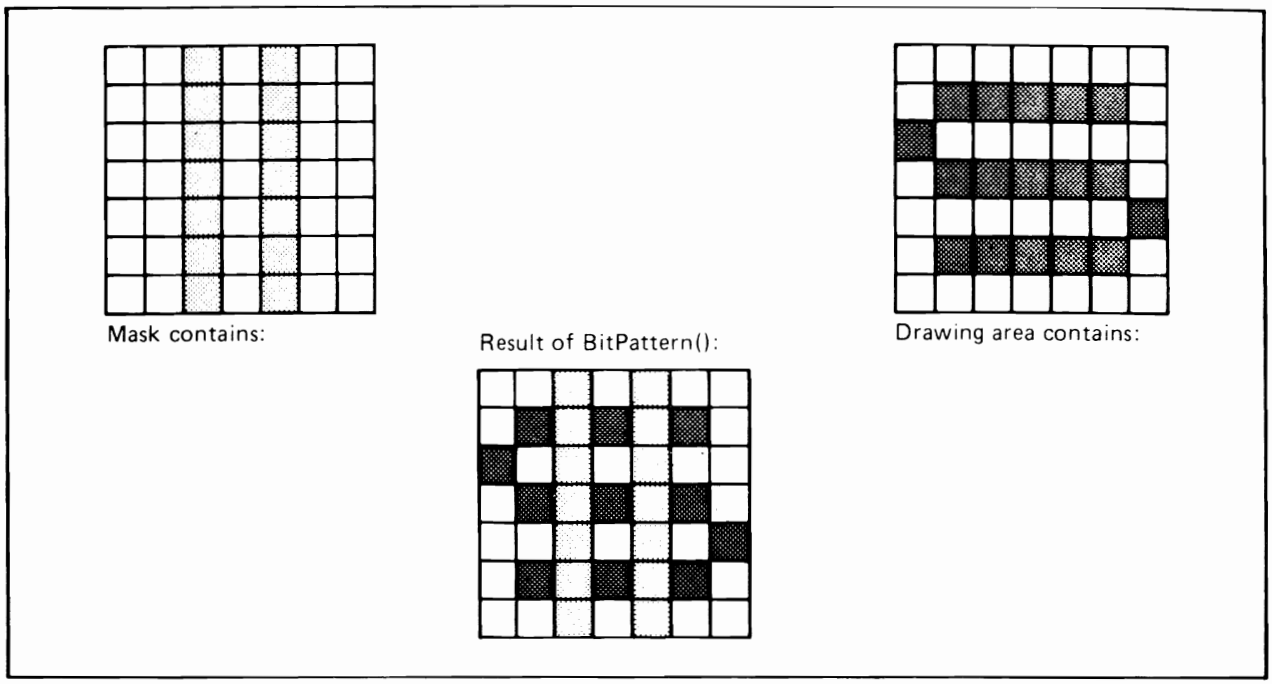


Figure 1-17: Example of Drawing Through a Stencil

In the area where the x's have been substituted, the target drawing area has been affected. Exactly *what* goes into the drawing area where the mask has 1's is determined by your **FgPen**, **BgPen**, **DrawMode**, and **AreaPtrn**.

The variables that control this function are:

rastport a pointer to the drawing area.

mask a pointer to the mask (mask layout explained below).

xl, maxx upper left corner x, and lower right corner x.

yl, maxy upper left corner y, and lower right corner y.

bytecnt number of bytes per row for the mask (*must* be an even number of bytes).

You call **BltPattern()** with:

```
BltPattern( rastport, mask, xl, yl, maxx, maxy, bytecnt)
```

The **mask** parameter is a rectangularly organized, contiguously stored pattern. This means that the pattern is stored in linearly increasing memory locations stored as (**maxy - yl**) rows of **bytecnt** bytes per row.

NOTE

These patterns must obey the same rules as **BitMaps**. This means that they must consist of an even number of bytes per row. For example, a mask such as:

0100001000000000
0010010000000000
0001100000000000
0010010000000000

is stored in memory beginning at a legal *word* address.

Extracting From a Bit-Packed Array

You use the routine **BltTemplate()** to extract a rectangular area from a source area and place it into a destination area. Figure 1-18 shows an example.

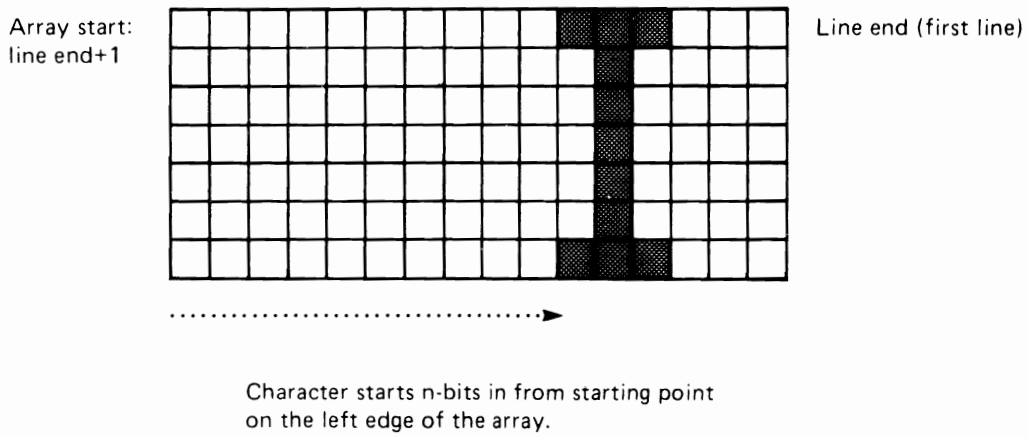


Figure 1-18: Example of Extracting from a Bit-Packed Array

If the character is to be represented as a rectangle within a larger, rectangularly organized bit-array, the system must know how the larger array is organized. This allows the system to extract each line of the object properly. For this extraction to occur properly, you need to

tell the system the modulo for the array. The modulo value is the value that must be added to the address pointer so that it points to the correct word in the next line in this rectangularly organized array.

Figure 1-19 represents a single bit-plane and the smaller rectangle to be extracted. The modulo in this instance is 4, because at the end of each line, you must add 4 to the address pointer to make it point to the first word in the smaller rectangle.

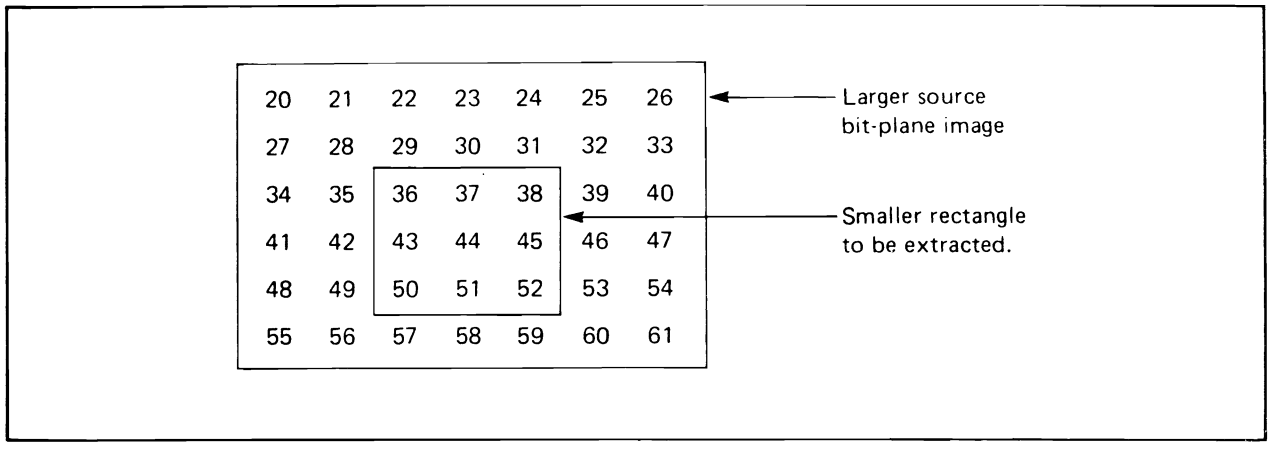


Figure 1-19: Modulo

Note that the modulo value must be an even number of bytes.

BltTemplate() takes the following arguments:

source	the source pointer for the array.
srcX	source X (bit position) in the array at which the rectangle begins.
srcMod	source modulo so it can find the next part of the source rectangle.
destRastPort	the destination **RastPort**.
destX, destY	destination x and y showing where to put the rectangle.
sizeX, sizeY	size x and y so it knows how much data to move.

You call **BltTemplate()** with:

```
BltTemplate( source, srcX, srcMod, destRastPort, destX, destY, sizeX, sizeY );
```

BltTemplate() uses **FgPen**, **BgPen**, **DrawMode** and **Mask** to place the template into the destination area. This routine differs from **BltPattern()** in that only a solid color is deposited in the destination drawing area, with or without a second solid color as the background (as in the case of text). Also, the template can be arbitrarily bit-aligned and sized in x.

Copying Rectangular Areas

Two routines copy rectangular areas from one section of chip-memory to another: **BltBitMap()** and **ClipBlit()**. **BltBitMap()** is the basic routine, taking **BitMaps** as part of its arguments. It allows you to define a rectangle in a source region and copy it to a destination area of the same size elsewhere in memory. This routine is often used in graphics rendering.

ClipBlit() takes most of the same arguments, but it works with the **RastPorts** and layers. Before **ClipBlit()** moves data, it looks at the area from which and to which the data is being copied (**RastPorts**, not **BitMaps**) and determines if there are overlapping areas involved. It then splits up the overall operation into a number of bit maps to move the data in the way you request.

Here is a sample call to **ClipBlit()**. This call is used in an image editor to transfer a rectangular block of data from the screen to a backup area.

```
ClipBlit( &rastport,          /* on-screen area */
        x,y,              /* upper left corner of rectangle */
        &undorastport,        /* screen editor can undo
                           * things, has a rastport
                           * specifically for undo */
        0,0,              /* upper left corner of destination */
        SIZEx,SIZEy           /* how big is the rectangle */
        minterm);
```

The **minterm** variable is an unsigned byte value whose leftmost 4 bits represent the action to be performed during the move. This routine uses the blitter device to move the data and can therefore logically combine or change the data as the move is made. The most common operation is a direct copy from source area to destination, which is the hex value C0.

You can determine how to set the **minterm** variable by using the logic equations shown in Table 1-5.

Table 1-5: Minterm Logic Equations

Logic Term in Leftmost 4 Bits	Logic Term Included in Final Output
8	BC
4	$B\bar{C}$
2	$\bar{B}C$
1	$\bar{B}\bar{C}$

Source B contains the data from the source rectangle, and source C contains the data from the destination area. If you choose bits 8 and 4 from the logic terms (C0), in the final destination area you will have data that occurs in source B only. Thus, C0 means a direct copy. The logic equation for this is:

$$BC + B\bar{C} = B(C + \bar{C}) = B$$

Logic equations may be used to decide on a number of different ways of moving the data. For your convenience, a few of the most common ones are listed in Table 1-6.

Table 1-6: Some common Logic Equations for Copying

Hex Value	Mode
30	Replace destination area with inverted source B.
50	Replace destination area with inverted version of original of destination.
60	Put B where C is not, put C where B is not (cookie cut).
80	Only put bits into destination where there is a bit in the same position for both source and destination (sieve operation).

Refer to the summary in the appendixes for **BltBitMap()**.

Controlling the Blitter

To use the blitter, you must first be familiar with how its registers control its operation. This topic is covered thoroughly in the *Amiga Hardware Manual* and is not repeated here.

There are four routines that you can use to control the blitter:

- **OwnBlitter()** allows your task to obtain exclusive use of this device.
- **DisownBlitter()** returns the device to shared operation.
- **QBlit()** and **QBSBlit()** let your task queue up requests for the use of the blitter on a non-exclusive basis

You provide a data structure called a **bltnode** (blitter node). The system can use this structure to link blitter usage requests into a first-in, first-out (FIFO) queue. When your turn comes, your own blitter routine can be repeatedly called until your routine says it is finished using the blitter.

Two separate queues are formed. One queue is for the **QBlit()** routine. You use **QBlit()** when you simply want something done and you don't necessarily care when it happens. This may be the case when you are moving data in a memory area which is not currently being displayed.

The second queue is maintained for **QBSBlit()**. QBS stands for "queue beam synchronized" blitter operations. **QBSBlit()** forms a beam-synchronized FIFO. When the video beam gets to a predetermined position, your routine is called. Beam synchronization takes precedence over the simple FIFO. This means that if the beam sync matches, the beam-synchronous blit will be done first, then the non-synchronous blit in the first position in the queue. You might use **QBSBlit()** to draw into an area of memory that is currently being displayed to modify memory that has already been "passed-over" by the video beam. This avoids display flicker as an area is being updated.

The input to each routine is a pointer to a **bltnode** data structure. The required items of the data structure are:

- a pointer to a **bltnode**
- a pointer to a function to perform

- a **beamsync** value (used if this a a **beamsync** blit)
- a status flag indicating whether the blitter control should perform a "cleanup" routine when the last blit is finished
- the address of the cleanup routine if the status flag states that it should be used

The **bltnode** data structure is contained in the include file *blit.h*. Here is a copy of that data structure, followed by details about the items which you must initialize.

```
struct bltnode
    {
    struct  bltnode *n;
    int     (*function)( );
    char    stat;
    short   beamsync;
    int     (*cleanup)( );
    };
```

The contents of **bltnode** are as follows:

struct bltnode *n;

a pointer to the next **bltnode**, which, for most applications will be zero

You should not link **bltnodes** together. This is to be performed by the system by way of a separate call to **QBlit()** or **QBSBlit()**.

int (*function)( );

This position is occupied by the address of a function which the blitter queuer will call when your turn comes up. Your routine must be formed as a subroutine, with an **RTS** at the end. Using the C language convention, the returned value will be in D0 (C returns its value by the **return(value)** statement).

If you return a nonzero value, the system will again call your routine until you finally return 0. This is to allow you to maintain control over the blitter; for example, for all 5 bit planes if you are blitting an object that spans that number, or for some other purpose. For display purposes, if you are blitting multiple objects and then saving and restoring the background, you must be sure that all planes of the object are positioned before another object is overlaid. This is the reason for the lockup in the blitter queue; it allows all work per object to be completed before going on to the next one.

Actually, the system tests the *status codes* for a condition of EQUAL or NOTEQUAL. When the C language returns the value of 0, it sets the status codes to EQUAL. When it returns a value of -1, it sets the status code to NOTEQUAL, so they would be compatible. Functions (*function)()) that are written for **QBlit()** and **QBSBlit()** are not normally written in C. They are usually written in assembly language as they then can take advantage of the ability of the queue routines to pass

them parameters in the system registers. The register passing conventions for these routines are as follows.

- Register A0 receives a pointer to the system hardware registers so that all hardware registers can be referenced as an offset from that address.

- Register A1 contains a pointer to the current **bltnode**. You may have queued up multiple blits each of which perhaps uses the same blitter routine. You can access the data for this particular operation as an offset from the value in A1. A typical user of these routines will precalculate the hardware register values that are stuffed into the registers and, during the routine, simply stuff them. For example, you can create a new structure such as the following:

```
struct myblit {
        struct bltnode;   /* make this new structure
                             compatible with the bltnode
                             by making it the first element */

        short bltcon1;    /* contents to be stuffed into
                             blitter control register 1 */
        short fwmask,lwmask;
                          /* first and last word masks */
        short bltmdc, bltmdb, bltmda;
                          /* modulos for sources a, b,and c */
        char *bltpta, *bltptb, *bltptc;
                          /* pointer to source data for sources */
                          };
```

Other forms of data structures are certainly possible, but this should give you the general idea.

char stat;

Tells the system whether or not to execute the cleanup routine at the end. This byte should be set to CLEANUP (0x40) if cleanup is to be performed. If not, then the **bltnode cleanup** variable can be zero.

short beamsync;

The value that should be in the VBEAM counter for use during a beam-synchronous blit before the **function()** is called.

The system cooperates with you in planning when to start a blit in the routine **QBSBlit()** by not calling your routine until, for example, the video beam has already passed by the area on-screen into which you are writing. This is especially useful during single buffering of your displays. There may be time enough to write the object between scans of the video display. You won't be visibly writing while

the beam is trying to scan the object. This avoids flicker (part of an old view of an object along with part of a new view of the object).

int (*cleanup)();

The address of a routine which is to be called after your last return from the **QBlit()** routine. When you return a zero finally, the queuer will call this subroutine (ends in **RTS** or **return();**) as the cleanup. Your first entry to the function may have dynamically allocated some memory or something else which must be undone to make for a clean exit. This routine must be specified.

Chapter 2

Layers

The layers library enables you to create displays containing overlapping display elements. This chapter describes the layers library routines and how you use them in creating graphics.

2.1. INTRODUCTION

The layers library contains routines that:

- multiplex a **BitMap** among various tasks by creating "layers" in the **BitMap**.
- create separate writable **BitMap** areas, some portions of which may be in the common (perhaps on-screen) **BitMap**, and some portions in an obscured area. In two modes, called smart-refresh and superbitmap, graphics are rendered into both the obscured and the non-obscured areas.
- move, size or depth-arrange the layers, bringing obscured segments into a non-obscured area.

Tasks can create layers in a common **BitMap**, then output graphics to those layers without any knowledge that there are other tasks currently using this **BitMap**.

To see what the layers library provides, you need only look at the Intuition user interface, as used by numerous applications on the Amiga. The windows that Intuition creates are based, in part, on the underlying strata of the layers library. You can find more details about Intuition in the book titled *Intuition: The Amiga User Interface.*

If you wish, you can use the layers library directly to create your own windowing system. The layers library takes care of the difficult things, that is, the bookkeeping jobs that are needed to keep track of where to put which bits. Once a layer is created, it may be moved, sized, depth-arranged or deleted using the routines provided in this library. In performing

their rendering operations, the graphics routines know how to use the layers and only draw into the correct drawing areas.

Definition of Layers

The internal definition of the layers resembles a set of *clipping rectangles* in that a drawing area is split into a set of rectangles. A clipping rectangle is a rectangular area into which the graphics routines will draw. Some of the rectangles are visible, some invisible. If a rectangle is visible, the graphics can draw directly into it. If a rectangle is obscured by an overlapping layer, the graphics routine may possibly draw into some other memory area. This memory area must be at least large enough to hold the obscured rectangle so the graphics routines can, on command, expose the obscured area.

The layers library manages interactions between the various layers by using a data structure called **Layer_Info**. Each major drawing area, called a **BitMap** (which all windows share), requires one **Layer_Info** data structure.

You may choose to split the viewing area into multiple parts by providing multiple independent **ViewPorts**. If you use the layers library to subdivide each of these parts into layers (effectively providing windows within these subdivisions), then you must provide one **Layer_Info** structure for each of these parts.

Types of Layers Supported

The layers library supports four types of layers:

- *Simple Refresh*

  No backup area is provided. Instead, when an obscured section of the layer is exposed to view, the routine using this layer is told that a "refresh" of that area is in order. This means that the program using this layer must redraw those portions of its display that are contained in the previously obscured section of the layer. All graphics rendering routines are "clipped" so that they will only draw into exposed sections of the layer.

- *Smart Refresh*

  The system provides one or more "backup" areas into which the graphics routines can draw whenever a part of this layer is obscured.

- *Superbitmap*

  There is a single backup area, which is permanently provided to store what is not in the layer. The backup area may be larger than the area that is actually shown in the on-screen **BitMap**.

- *Backdrop*

  A backdrop layer always appears behind all other layers that you create. The current implementation of backdrop layers prevents them from being moved, sized, or depth-arranged.

2.2. LAYERS LIBRARY ROUTINES

The layers library contains routines for:

Operation	**Routine**
Allocating a **Layer_Info** structure	**NewLayerInfo()**
Deallocating a **Layer_Info** structure	**DisposeLayerInfo()**
Intertask operations	**LockLayer(), UnLockLayer(), LockLayers(), UnlockLayers(), LockLayerInfo(), UnlockLayerInfo**
Creating and deleting layers	**CreateUpfrontLayer(), CreateBehindLayer(), DeleteLayer()**
Moving layers	**MoveLayer()**
Sizing layers	**SizeLayer()**
Changing a viewpoint	**ScrollLayer()**
Reordering layers	**BehindLayer, UpfrontLayer()**
Determining layer position	**WhichLayer()**
Sub-layer rectangle operations	**SwapBitsRastPortClipRect()**

Initializing and Deallocating Layers

The function **NewLayerInfo()** allocates and initializes a **Layer_Info** data structure and allocates some extra needed memory for the 1.1 release. After the call to **NewLayerInfo()**, you can use the layer operations described in the following paragraphs.

The function **DisposeLayerInfo()** deallocates a **Layer_Info** structure that was allocated with a call to **NewLayerInfo()** and frees the extra memory that was allocated.

NOTE: Prior to the current 1.1 release, **Layer_Info** structures were initialized with the **InitLayers()** function. For backwards compatibility, you can still use this function with newer software. For optimal performance, however, you should call **FattenLayerInfo()** to allocate the needed extra memory, and **ThinLayerInfo()** to return the memory to the system free-list. Failure to deallocate memory will result in loss of available memory.

Intertask Operations

This section shows the use of the routines **LockLayerInfo()**, **UnlockLayerInfo()**, **LockLayer()**, **UnlockLayer()**, **LockLayers()**, and **UnlockLayers()**.

LockLayerInfo() and UnlockLayerInfo()

You create layers by using the routines **CreateUpFrontLayer()** and **CreateBehindLayer()**. If multiple tasks are all trying to create layers on the same screen or **ViewPort**, each task will be trying to affect the same data structures while creating its layers. The **Layer_Info** data structure controls the layers. **LockLayerInfo()** ensures that the **Layer_Info** data structure remains intact and tasks can obtain this exclusive access.

LockLayerInfo() waits (sleeps) until there are no other tasks that have done a **LockLayerInfo()**. Then it grants exclusive access to the locking task.

LockLayer() and Unlocklayer()

If a task is making some changes to a particular layer, such as resizing it or moving it, the task must inhibit the graphics rendering into the layer. **LockLayer()** blocks graphics output once the current graphics function has completed. The other task goes to sleep only if it attempts to draw graphics. **LockLayer()** returns exclusive access to the layer once other tasks, including graphics, are finished with this layer.

Unlocklayer() frees the locked layer for other operations.

If more than one layer must be locked, then these **LockLayer()** calls must be surrounded by **LockLayerInfo()** and **UnLockLayerInfo()**. This is to prevent deadlock situations.

LockLayers() and UnlockLayers()

Sometimes it is necessary to lock all layers at the same time. For example, under Intuition, a rubber-band box is drawn when a window is being moved or sized. To draw such a box, Intuition must stop all graphics rendering to all windows (and associated layers) so that it can draw a line using the graphics complement drawing mode. If other graphics draw over this line, it would not be possible for Intuition to erase it again, using a subsequent complement operation over the same line. Thus **LockLayers()** is used to lock all layers in a single command. **UnlockLayers()** releases the layers.

You can simulate **LockLayers()** by calling **LockLayer()** for each layer in the **LayerList**. However, in that case, you must call **LockLayerInfo()** before and **UnlockLayerInfo()** after each **LockLayer()** call.

Creating and Deleting Layers

CreateUpFrontLayer() creates a layer that is in front of all other layers. Intuition uses this function to create certain types of new windows, as well as other Intuition components.

CreateBehindLayer() creates a layer that is behind all other layers. Intuition uses this function to create a new "Backdrop" window.

Each of the routines that create layers return a pointer to a layer data structure (shown in the include file *layers.h*).

NOTE: When you create a layer, the system automatically creates a **RastPort** to go along with it. Because a **RastPort** is specified by the drawing routines, if you use this layer's **RastPort**, you will draw into specifically and only the area that you have designated on-screen for this layer. See also the topic called "The Layer's RastPort" below.

DeleteLayer() is used to remove a layer from the layer list. It is one of the functions used by Intuition to close a window.

For these functions, you need to perform **LockLayerInfo()** and **UnlockLayerInfo()** because you need to access the **Layer_Info** structure itself.

Moving Layers

MoveLayer() moves a layer to a new location. When you move a layer, the move command affects the list of layers that is being managed by the **Layer_Info** data structure. The system locks the **Layer_Info** for you during this operation.

Sizing Layers

The **SizeLayer()** command changes the size of a layer by leaving the coordinates of the upper left corner the same and modifying the coordinates of the lower right corner of the layer. The system locks the **Layer_Info** for you during this operation.

Changing a View Point

ScrollLayer() is for superbitmap layers only. This command changes the portion of a superbitmap that is shown by a layer. An analogy is a window in a wall. If the homeowner doesn't like the view he sees from a particular window, he might either change what he sees by planting trees (that is, new graphics rendering) or he might decide to move the window to

see another part of the great outdoors (changing the portion of the superbitmap shown by a layer). You must provide a superbitmap; the **ScrollLayer()** command repositions the smaller layer against the larger superbitmap, thus showing a different part of it.

Because the layer size and on-screen position do not change while this operation is taking place, it is not necessary to lock the **Layer_Info** data structure. However it is necessary to prevent graphics rendering operations from drawing into this layer or its associated superbitmap while **ScrollLayer()** is performing the repositioning. Thus, the system locks the layer for you while this operation is taking place.

Reordering Layers

BehindLayer() and **UpfrontLayer()** are used, respectively, to move a layer behind all other layers or in front of all other layers. **BehindLayer()** also considers any backdrop layers, moving a current layer behind all others except backdrop layers. The system performs **LockLayers()** and **LockLayerInfo()** for you during this operation.

Determining Layer Position

If the viewing area has been separated into several layers, you may wish to find out which layer is topmost at a particular x,y coordinate. For example, Intuition does this while keeping track of the mouse position. When you move the mouse into one of the windows and click the left button, Intuition feeds the current x,y coordinate to **WhichLayer()**. In return, **WhichLayer()** tells Intuition which layer has been selected, and thus it knows with which window you wish to work.

If you wish to be sure that no task changes the sequence of layers (by using **UpfrontLayer()**, **BehindLayer()**, **CreateUpFrontLayer()**, **DeleteLayer()**, **MoveLayer()** or **SizeLayer()**) before your task can use this information, call **LockLayerInfo()** before calling **WhichLayer()**. Then, after receiving and using the information that **WhichLayer()** delivers, you can call **UnlockLayerInfo()**. In this way you will be acting on data that was true as of the moment it was received.

Sub-Layer Rectangle Operations

The **SwapBitsClipRectRastPort()** routine is for users who don't want to worry about clipping rectangles. The need for this routine goes a bit deeper than that. It is a routine that actually enables the menu operations of Intuition to function much more quickly than they would if this routine were not provided.

Consider the case where there are several windows open on an Intuition screen. If you wish to produce a menu, there are two ways to do it:

a) Create an up-front layer with **CreateUpfrontLayer()**, then render the menu in it. This could use lots of memory and require a lot of (very temporary) "slice-and-dice" operations to create all of the clipping rectangles for the existing windows and so on, OR

b) use **SwapBitsClipRectRastPort()**, directly on the screen drawing area:

- Render the menu in a backup area off-screen, then lock all of the on-screen layers so that no task can use graphics routines to draw over your menu area on-screen.
- Next, swap the on-screen bits with those off-screen, making the menu appear.
- When you finish with the menu, swap again, and unlock the layers.

The second rendering method is faster and leaves the clipping rectangles and most of the rest of the window data structures untouched.

Notice that all of the layers must be locked while the menu is visible. Any task that is using any of the layers for graphics output will be halted while the menu operations are taking place. If, on the other hand, the menu is rendered as a layer, no task need be halted while the menu is up because the lower layers need not be locked. It is a tradeoff decision that you must make.

2.3. THE LAYER'S RASTPORT

When you create a layer, you automatically get a **RastPort**. The pointer to the **RastPort** is contained in the layer data structure and can be retrieved typically by the statement:

```
rp = layer->rp;        /* copy the pointer from the layer structure
                        * into a local pointer for further use */
```

Using this **RastPort**, you can draw anywhere into the layer's defined rectangle. Location (0,0) is the coordinate location for the upper left corner of the rectangle, and location (**xmax, ymax**) is the lower right corner. If you try to draw to any location outside of this coordinate system, the graphics routines will clip the drawing to the inside boundaries of this area.

The type of layer you specify by the **Flags** variable determines the other facilities the layer provides. The following paragraphs describe the types of layers —simple refresh, smart refresh, superbitmap, and backdrop—and the flags you set for the type you want. Note that the three layer-type **Flags** are mutually exclusive. That is, you cannot specify more than one layer-type flag—LAYERSIMPLE, LAYERSMART, LAYERSUPER.

Simple Refresh Layer

When you draw into the layer, any portion of the layer that is visible (not obscured) will have its drawing rendered into the common **BitMap** of the viewing area.

If another layer operation is performed that causes part of a simple refresh layer to be obscured and then exposed, you must restore the lost the part of the drawing that your application rendered into the obscured area.

Simple refresh has two basic advantages:

- It uses no backup area (**BitMap**) to save drawing sections that cannot be seen anyway (and therefore saves memory).
- When an application tries to restore the layer by performing a full-layer redraw, (sandwiched between a **BeginUpdate()**, **EndUpdate()** pair), only those damaged areas are redrawn, making the operation very time efficient.

Its disadvantage is that the application needs to watch to see if its layer needs refreshing. This test can be performed, typically, by a statement set such as the following.

```
refreshstatus = layer->Flags & LAYERREFRESH;
if (refreshstatus != 0) refresh(layer);
```

NOTE: Applications using Intuition typically get their refresh notifications as event messages passed through an Intuition Direct Communications Message Port (IDCMP).

Smart Refresh Layer

If any portion of the layer is hidden by another layer, the bits for that obscured portion are rendered into a backup area.

With smart refresh layers, the system handles all of the refresh requirements except when the layer is made larger. Its disadvantage is the additional memory needed to handle this automatic refresh.

Superbitmap Layer

A superbitmap layer is similar to a smart refresh layer. It too has a backup area into which drawings are rendered for currently obscured parts of the display. However, it differs from smart refresh in that:

- The backup **BitMap** is user-supplied, rather than being allocated dynamically by the system.
- The backup **BitMap** may be larger than the area of this **BitMap** that is currently showing within the current size of this layer.

To see a larger portion of a superbitmap in the on-screen layer, you use **SizeLayer()**. To see a different portion of the superbitmap in the layer, you use **ScrollLayer()**.

When the graphics routines perform your drawing commands, part of the drawing appears in the common **BitMap** (the on-screen portion). Any drawing outside the layer itself is rendered into the superbitmap. When it is time to scroll or size the layer, the layer contents are copied into the superbitmap, the scroll or size positioning is modified, and the appropriate portions are then copied back into the layer.

Backdrop Layer

Any layer can be designated a backdrop layer. You can turn off the backdrop flag temporarily and allow a layer to be depth-arranged. Then by restoring the backdrop flag, you can again inhibit depth-arrangement operations.

You change the backdrop flag typically by the statements:

```
        /* turn off the backdrop bit */
layer->Flags &= LAYERBACKDROP;
        /* turn on the backdrop bit */
layer->Flags |= LAYERBACKDROP;
```

2.4. USING THE LAYERS LIBRARY

The following is a step-by-step example showing how the layers library can be used in your programs. Note that the Intuition software, which is part of the system as well, manages many of these items for you. The example below can be started up under Intuition, but requires that the Amiga be reset in order to exit the program.

The example program explains the individual parts separately, then merges the parts into a single working example. This simple example produces three rectangles on-screen: one red, one green, and one blue. Each rectangle is rendered as a rectangle-fill of one of three smart layers created for the example.

Opening the Layers Library

As with all library routines, before the layers library can be used, it must be opened. This is done typically by the following code:

```
struct LayersBase *LayersBase;
    ...
LayersBase = (struct LayersBase *)OpenLibrary("layers.library",0);
if(LayersBase == NULL)
    exit(NO_LAYERS_LIBRARY_FOUND);
```

Opening the Graphics Library

Because the example uses various graphics library functions as well as the layers library, you must also open the graphics library with the following code:

```
struct GfxBase *GfxBase;
    ...
GfxBase = (struct GfxBase *)OpenLibrary("graphics.library",0);
if(GfxBase == NULL)
    exit(NO_GRAPHICS_LIBRARY_FOUND);
```

Creating a Viewing Workspace

You can create a viewing workspace by using the primitives **InitVPort()**, **InitView()**, **MakeVPort()**, **MrgCop()**, and **LoadView()**. See the "Graphics Example" section in Chapter 1, "Graphics Primitives". You add the following statements.

```
struct Layer_Info *li;
li=NewLayerInfo();
```

This provides and initializes a **Layer_Info** data structure with which the system can keep track of layers that you create.

Creating the Layers

You can create layers in this common bit map by calling **CreateUpfrontLayer()** (or **CreateBehindLayer()**), with a sequence like the following. The **Flags** value in this example is LAYERSMART (see "clip.h" in the appendixes for all other flag values). This sequence requests construction of a smart refresh layer.

```
#define FLAGS LAYERSMART

struct RastPort *rp[3];
      /* allocate a rastport pointer for each layer */
struct Layer *layer[3];
      /* allocate a layer pointer for each layer */

layer[0] = CreateUpfrontLayer(&li,&b,20,20,100,80,FLAGS,NULL);
      /* layerinfo, common bitmap, x1,y1,x2,y2,
       * flags = 0 (smart refresh), null pointer to superbitmap */

layer[1] = CreateUpfrontLayer(&li,&b,30,30,110,90,FLAGS,NULL);
layer[2] = CreateUpfrontLayer(&li,&b,40,40,120,100,FLAGS,NULL);

if(layer[0]==NULL || layer[1]==NULL || layer[2]==NULL) exit(3);
      /* if not enough memory, can't continue the example */
```

Getting the Pointers to the RastPorts

Each layer pointer data structure contains a pointer to the **RastPort** that it uses. Here is the assignment from the layer structure to a set of local pointers:

```
for(i=0; i<3; i++)
      rp[i] = layer[i]->rp;
```

Using the RastPorts for Display

Here are the rectangle-fill operations that create the display:

```
for(i=0; i<3; i++)
    {
    SetAPen(rp[i],i+1);
    SetDrMd(rp[i],JAM1);
    RectFill(rp[i],0,0,80,50);
    }
```

If you perform an **UpfrontLayer()** or **BehindLayer()** command prior to the Delay shown in the complete example below, all of the data contained in each layer is retained and correctly rendered automatically by the layers library. This is because these are all smart-refresh layers. If you change the example to use a **Flags** value of LAYERSIMPLE, and then perform **UpfrontLayer()** or **BehindLayer()**, the obscured portions of the layers, now exposed, contain only the background color. This illustrates that simple-refresh layers may have to be redrawn after layer operations are performed.

Layers Example

Here is the complete example, pulled together from the complete example in Chapter 1, as well as the pieces given above. Sections of the example that differ from those shown in the Chapter 1 example are indicated through comments to show the additions adding the layers library demonstration.

```
/* ************************************************************
 * THIS EXAMPLE SHOWS HOW TO USE THE layers.library.  Certain
 * functions are not available in the system software prior to
 * the release of version 1.1.  Therefore this example can only
 * be compiled if your  C-disk supports version 1.1 or beyond.
 ************************************************************ */
#include "exec/types.h"
#include "graphics/gfx.h"
#include "hardware/dmabits.h"
#include "hardware/custom.h"
#include "hardware/blit.h"
#include "graphics/gfxmacros.h"
```

```
#include "graphics/copper.h"
#include "graphics/view.h"
#include "graphics/gels.h"
#include "graphics/regions.h"
#include "graphics/clip.h"
#include "exec/exec.h"
#include "graphics/text.h"
#include "graphics/gfxbase.h"
/* ************ added for layers support ********************** */
#include "graphics/layers.h"
#include "graphics/clip.h"
/* ************ added for layers support ********************** */


#define DEPTH 2
#define WIDTH 320
#define HEIGHT 200
#define NOT_ENOUGH_MEMORY -1000
#define FOREVER for(;;)
      /* construct a simple display */

#define FLAGS LAYERSMART
struct View *oldview; /* save pointer to old view so can go back to sys */
struct View v;
struct ViewPort vp;
struct ColorMap *cm; /* pointer to colormap structure, dynamic alloc */
struct RasInfo ri;
struct BitMap b;
/* made 3 separate rastports for layers testing ************ */
struct RastPort *rp[3];       /* rastport for each layer */
/* dynamically created RastPorts from the calls to CreateUpfrontLayer */

short i,j,k,n;
struct ColorMap *GetColorMap();
struct GfxBase *GfxBase;

SHORT  boxoffsets[] = { 802, 2010, 3218 };
USHORT colortable[] = { 0x000, 0xf00, 0x0f0, 0x00f };
                  /* black, red, green, blue */
UBYTE *displaymem;
UBYTE *colorpalette;

long LayersBase;
struct Layer_Info *li;
struct Layer *layer[3];
extern struct Layer *CreateUpfrontLayer();
extern struct Layer_Info *NewLayerInfo();

main()
{
```

```
GfxBase = (struct GfxBase *)OpenLibrary("graphics.library",0);
if (GfxBase == NULL) exit(1);

LayersBase = OpenLibrary("layers.library",0);
if(LayersBase == NULL) exit(2);

oldview = GfxBase->ActiView;  /* save current view, go back later */
/* example steals screen from Intuition if started from WBench */

li = NewLayerInfo();  /* get a LayerInfo structure */
if(li == NULL) exit(100);

/* not needed if gotten by NewLayerInfo     InitLayers(li);  */

                    /* initialize view */
InitView(&v);
                    /* link view into viewport */
v.ViewPort = &vp;
                    /* init view port */
InitVPort(&vp);
                    /* now specify critical characteristics */
vp.DWidth = WIDTH;
vp.DHeight = HEIGHT;
vp.RasInfo = &ri;
                    /* init bit map (for rasinfo and rastport) */
InitBitMap(&b,DEPTH,WIDTH,HEIGHT);
                    /* (init RasInfo) */
ri.BitMap = &b;
ri.RxOffset = 0; /* align upper left corners of display
                  * with upper left corner of drawing area */
ri.RyOffset = 0;
ri.Next = NULL;
                    /* (init color table) */
cm = GetColorMap(4);      /* 4 entries, since only 2 planes deep */
colorpalette = (UBYTE *)cm->ColorTable;
for(i=0; i<4; i++)
      *colorpalette++ = colortable[i];
                    /* copy my colors into this data structure */
vp.ColorMap = cm;   /* link it with the viewport */

                     /* allocate space for bitmap */
for(i=0; i<DEPTH; i++)
   {
   b.Planes[i] = (PLANEPTR)AllocRaster(WIDTH,HEIGHT);
   if(b.Planes[i] == NULL) exit(NOT_ENOUGH_MEMORY);
   }

MakeVPort( &v, &vp );      /* construct copper instr (prelim) list */
MrgCop( &v );          /* merge prelim lists together into a real
                        * copper list in the view structure. */
```

```
for(i=0; i<2; i++)
      {
      displaymem = (UBYTE *)b.Planes[i];
      for(j=0; j<RASSIZE(WIDTH,HEIGHT); j++)
            *displaymem++ = 0;
      /* zeros to all bytes of the display area */                    }

LoadView(&v);

/* now fill some boxes so that user can see something */

layer[0] = CreateUpfrontLayer(li,&b,5,5,85,65,FLAGS,NULL);
      /* layerinfo, common bitmap, x,y,x2,y2,
       * flags = 0 (simple refresh), null pointer to superbitmap */
if(layer[0] == NULL) goto cleanup1;

layer[1] = CreateUpfrontLayer(li,&b,20,20,100,80,FLAGS,NULL);
if(layer[1] == NULL) goto cleanup2;

layer[2] = CreateUpfrontLayer(li,&b,45,45,125,105,FLAGS,NULL);
if(layer[2] == NULL) goto cleanup3;

for(i=0; i<3; i++)    /* layers are created, now draw to them */
{
rp[i] = layer[i]->rp;
SetAPen(rp[i],i+1);
SetDrMd(rp[i],JAM1);
RectFill(rp[i],0,0,79,59);
}
SetAPen(rp[0],0);
Move(rp[0],5,30);
Text(rp[0],"Layer 0",7);

SetAPen(rp[1],0);
Move(rp[1],5,30);
Text(rp[1],"Layer 1",7);

SetAPen(rp[2],0);
Move(rp[2],5,30);
Text(rp[2],"Layer 2",7);

Delay(120);/* 2 seconds before first change */
BehindLayer(li,layer[2]);

Delay(120);/* another change 2 seconds later */

UpfrontLayer(li,layer[0]);

for(i=0; i<30; i++)
{
```

```
            MoveLayer(li,layer[1],1,3);
            Delay(10);  /* wait .16 seconds (uses DOS function) */

        }

    cleanup3:
        LoadView(oldview);              /* put back the old view  */
        DeleteLayer(li,layer[2]);
    cleanup2:
        DeleteLayer(li,layer[1]);
    cleanup1:
        DeleteLayer(li,layer[0]);

        DisposeLayerInfo(li);
        FreeMemory();
        CloseLibrary(GfxBase);

}       /* end of main() */


FreeMemory()
{                   /* return user and system-allocated memory to sys manager */

        for(i=0; i<DEPTH; i++)                  /* free the drawing area */
            FreeRaster(b.Planes[i],WIDTH,HEIGHT);
        FreeColorMap(cm);                   /* free the color map */
              /* free dynamically created structures */
        FreeVPortCopLists(&vp);
        FreeCprList(v.LOFCprList);
        return(0);
}
```

2.5. CLIPPING RECTANGLE LIST

When you perform the various graphics drawing routines, you will notice that the routines draw into Intuition windows even though the windows might be partially or totally obscured on-screen. This is because the layer library functions split the drawing area to provide lists of drawing areas which the graphics drawing can use for its operations.

In particular, the layer library functions split the windows into rectangles. You need only concern yourself with a single overall **RastPort** that contains the description of the complete area that you are managing. When either you or Intuition use the layer library, the graphics routines will be able to tell how the drawing area is split and where rendering can occur.

The set of rectangles comprising the layer is known as a clipping rectangle list (**ClipRect** structure). A clipping rectangle is a rectangular area into which the graphics routines will draw. All drawing that would fall outside of that rectangular area is clipped (not rendered).

Damage List

For a smart refresh window, the system automatically generates off-screen buffer spaces, essentially linked into the clipping rectangle list. Thus, parts of the display that are on-screen are rendered into the on-screen drawing area, and parts of the display that are obscured are drawn into a backup area. When segments are exposed, the backup area information is brought to view automatically during the routines **UpfrontLayer()** and **BehindLayer()**, as well as during **MoveLayer()**.

For a simple refresh window however, any section of a drawing area that is not covered in the clipping rectangle list is not drawn into by the graphics routines. When obscured areas are exposed, they will not contain any graphics rendering at all. As the system creates and moves layers in front of such simple refresh windows, the layers library keeps track of the rectangular segments that have not been drawn and are therefore not part of any automatically saved backup areas. This list of non-drawn areas is called a **DamageList**.

Repairing the Damage

When you receive a REFRESH event from Intuition for a simple refresh window, you are being told that Intuition, through the layers library, has done something to change the portions of your window that are exposed to view. In other words, there is likely to be a blank space where there is supposed to be some graphics.

To update only those areas that need updating, you call **BeginUpdate()**. **BeginUpdate()** saves the pointer to the current clipping rectangles. It also installs in the layer structure a pointer to the set of **ClipRects** generated from the **DamageList**. In other words, the graphics rendering routines see only those rectangular spaces that need to be updated and refuse to draw into any other spaces within this layer. If, for example, there are only one or two tiny rectangles that need to be fixed, the graphics can ignore all but these spaces and repair them very quickly and efficiently. To repair the layer, you ask the graphics routines to redraw the whole layer, but the graphics uses the new clipping rectangle list (that is, the damage list) to speed the process.

To complete the update process, call **EndUpdate()**, to restore the original **ClipRect** list.

2.6. REGIONS

Regions are rectangles that, when combined, can become part of a **DamageList**. The graphics.library contains several support routines for regions. Among these are routines for:

Operation	Routine
Creating and deleting regions	**NewRegion(), DisposeRegion()**
Changing a region	**AndRectRegion(), OrRectRegion, XorRectRegion()**
Clearing a region	**ClearRegion()**

Basically, the region commands let you construct a custom **DamageList**, which you can use with your graphics rendering routines. With this list, you can selectively update a custom-sized, custom-shaped part of your display area without disturbing any of the other layers that might be present.

Creating and Deleting Regions

NewRegion() allocates and initializes a new data structure that may be thought of as a blank painter's easel.

If this new region is to be used as the basis for a **DamageList**, and you asked the graphics routines to draw something through this **DamageList**, nothing would be drawn as there is nothing in the region. The region that you produce can be thought of as patches of canvas. A new region has no canvas.

Because a region is dynamically created by using **NewRegion()**, the procedure **DisposeRegion()** is provided to return the memory to the system when you have finished with it. Note that not only the region structure is deallocated, but also any rectangles that have been linked into it.

Changing a Region

OrRectRegion() modifies a region structure by *or*'ing a clipping rectangle into the region. This has an effect similar to adding a rectangle of canvas to the easel. If you now exercise the drawing routines, the rendering will occur in the areas where the region has been *or*'ed (canvas rectangle has been added) and inhibited elsewhere.

AndRectRegion() modifies a region structure by *and*'ing a clipping rectangle into the region. This has an effect similar to using the rectangle as an outline for a position on the easel. Any area of canvas that falls outside this outline is clipped and discarded.

XorRectRegion() applies the rectangle to the region in an exclusive-or mode. That is, wherever there is no canvas, canvas is applied to the easel. Wherever there is canvas present within the rectangle, a hole is created. Thus it is a combination of **OrRectRegion()** and **AndRectRegion()** in a single application.

Clearing a Region

While you are performing various types of selective drawing area updates, you may wish to do some of your graphics rendering with one form of region, and some with a different form of region. You can perform **ClearRegion()** to go from one form back to a fresh, empty region. Then you can begin again to compose yet another modified region for the next drawing function.

Using Regions

The region routines typically are used in a sequence like the following:

```
struct Region *r;
struct Rectangle *rect1, *rect2, rect3;

r = NewRegion();
OrRectRegion(rect1, r);       /* add a rectangle */
AndRectRegion(rect3, r); /* patch a rectangle */
XorRectRegion(rect2, r);  /* weird patch */
            . . .
/* in this section of code:
  1. Save current pointer to DamageList for the
     Layer you wish to affect.
  2. Equate the region address (r) to the DamageList
     pointer in the Layer structure.
  3. Perform whatever drawing functions you wish into
     this layer
  4. Restore the original DamageList pointer.
*/
            . . .
DisposeRegion(r);
```

The drawing will only occur in those areas of the drawing area that you have specified should be updated. Graphics rendering is often made faster because not all of the area need be updated.

A typical sequence using **ClearRegion()** might be:

```
struct Region *r;
struct Rectangle *rect1, *rect2, rect3;
struct Layer_Info *li;

r = NewRegion();
OrRectRegion(rect1, r);
OrRectRegion(rect2, r);
            . . .
      (swap in as a damaged list)
BeginUpdate(li);
      (draw, draw, draw something)
EndUpdate(li);
      (restore original damaged list)
            . . .
ClearRegion(r);
AndRectRegion(rect3, r);
            . . .
      (swap, draw, restore)
            . . .
DisposeRegion(r);
```

Sample Application for Regions

For example, assume that you are producing a display that requires a view through a fence. You can create this "slats" effect by using regions, as follows:

a) Create a new region.

b) Create several rectangles representing the open areas of the slats in the fence.

c) *Or* these into the region.

d) Save the **DamageList** pointer in the affected layer so it can be restored later.

e) Copy the region address into **DamageList** pointer.

f) Draw the scene into the entire layer using the graphics

g) Restore the original **DamageList** pointer.

h) Dispose of the region.

Here is a sample application. It is based on the sample layers library program shown above. For brevity, the comments have been stripped out except where new material, pertinent to regions, has been inserted.

```
/* SIMPLE REGIONS EXAMPLE.... DRAW BEHIND A FENCE */
/* Certain layers.library routines are used herein that aren't
 * available until Amiga C compiler version 1.1 and beyond. */

#include <exec/types.h>
#include <graphics/gfx.h>
#include <hardware/dmabits.h>
#include <hardware/custom.h>
#include <graphics/gfxmacros.h>
#include <graphics/regions.h>
#include <graphics/clip.h>
#include <graphics/text.h>
#include <hardware/blit.h>
#include <graphics/gfxbase.h>
#include <graphics/copper.h>
#include <graphics/gels.h>
#include <graphics/rastport.h>
```

```
#include <graphics/view.h>
#include <exec/exec.h>
#include <graphics/layers.h>

#define FLAGS LAYERSIMPLE
extern struct Layer *CreateUpfrontLayer();

struct GfxBase *GfxBase;

long LayersBase;

#define DEPTH 2
#define WIDTH 320
#define HEIGHT 200
#define NOT_ENOUGH_MEMORY -1000
#define FOREVER for(;;)

struct View *oldview;
struct View v;
struct ViewPort vp;
struct ColorMap *cm;
struct RasInfo ri;
struct BitMap b;
struct RastPort *rp;      /* one rastport for one layer */

short i,j,k,n;
struct ColorMap *GetColorMap();

USHORT colortable[] = { 0x000, 0xf00, 0x0f0, 0x00f };
       /* black, red, green, blue */
UBYTE *displaymem;
UWORD *colorpalette;

struct Layer_Info *li;
struct Layer *layer;      /* one layer pointer */

extern struct Region *NewRegion();
struct Region *rgn;       /* one region pointer */
struct Rectangle rect[14];   /* some rectangle structures */
struct Region *oldDamageList;

extern struct Layer_Info *NewLayerInfo();

main()
{
  SHORT x,y;

  GfxBase = (struct GfxBase *)OpenLibrary("graphics.library",0);
  if (GfxBase == NULL) exit(1);
  LayersBase = OpenLibrary("layers.library",0);
```

```
if(LayersBase == NULL) exit(2);


oldview = GfxBase->ActiView;

li = NewLayerInfo();      /* v1.1 compiler only */
InitView(&v);
v.ViewPort = &vp;
InitVPort(&vp);
vp.DWidth = WIDTH;
vp.DHeight = HEIGHT;
vp.RasInfo = &ri;
InitBitMap(&b,DEPTH,WIDTH,HEIGHT);
ri.BitMap = &b;
ri.RxOffset = 0;
ri.RyOffset = 0;
ri.Next = NULL;
cm = GetColorMap(4);
colorpalette = (UWORD *)cm->ColorTable;
for(i=0; i<4; i++)
  *colorpalette++ = colortable[i];
vp.ColorMap = cm;
for(i=0; i<DEPTH; i++)
{
      b.Planes[i] = (PLANEPTR)AllocRaster(WIDTH,HEIGHT);
      if(b.Planes[i] == NULL) exit(NOT_ENOUGH_MEMORY);
}

MakeVPort( &v, &vp );
MrgCop( &v );
for(i=0; i<2; i++)
  {
  displaymem = (UBYTE *)b.Planes[i];
  for(j=0; j<RASSIZE(WIDTH,HEIGHT); j++)
    *displaymem++ = 0;
  /* zeros to all bytes of the display area */          }

LoadView(&v);
layer = CreateUpfrontLayer(li,&b,0,0,200,140,FLAGS,NULL);
if(layer==NULL) exit(3);

rp = layer->rp;

SetAPen(rp,3);
RectFill(rp,0,0,199,139);   /* show the layer itself */

j=10;       /* initialize the rectangles */
for(i=0; i<10; i++)
  {
  rect[i].MinX = j;
```

```
    rect[i].MaxX = j + 8;
    rect[i].MinY = 20;
    rect[i].MaxY = 120;
    j += 16;
    }

  rgn = NewRegion();   /* get a new region to use */
  if(rgn == NULL) exit(4);

  for(i=0; i<14; i++)
    OrRectRegion(rgn,&rect[i]);

  oldDamageList = layer->DamageList;
  layer->DamageList = rgn;

  BeginUpdate(layer);

  /* here insert the drawing routines to draw something
       * behind the slats.
       */
  x = 4;  y = 10;
  SetAPen(rp,0);
  SetDrMd(rp,JAM1);
  RectFill(rp,0,0,199,139);
  SetAPen(rp,1);
  SetBPen(rp,0);
  SetDrMd(rp,JAM2);
  for(i=0; i<14; i++)
  {
    Move(rp, x, y);
    Text(rp,"Behind A Fence",14);
    x += 4;  y += 9;
  }
  EndUpdate(layer);
  layer->DamageList = oldDamageList;
  DisposeRegion(rgn);

  Delay(300);

  DeleteLayer(li, layer);
  DisposeLayerInfo(li);

  LoadView(oldview);

  FreeMemory();
  CloseLibrary(GfxBase);

}   /* end of main() */
```

```
FreeMemory()
{           /* return user and system-allocated memory to sys manager */

    for(i=0; i<DEPTH; i++)        /* free the drawing area */
        FreeRaster(b.Planes[i],WIDTH,HEIGHT);
  FreeColorMap(cm);          /* free the color map */
    /* free dynamically created structures */
  FreeVPortCopLists(&vp);
  FreeCprList(v.LOFCprList);
  return(0);
}
```

Chapter 3

Animation

This chapter shows how you can use the graphics animation routines to produce and animate graphics images.

3.1. INTRODUCTION

The graphics animation routines let you define images by specifying various characteristics of graphic objects, such as the following:

- height
- width
- colors
- shape
- position in the drawing area
- how to draw the object
- how to move the object
- how the object interacts with other elements

The objects you define are called GELS (for "graphic elements"). You can draw GELS into or onto a background display of some type. The graphics animation routines operate on a list of GELS to produce a list of instructions that cause the system to draw the GELS in the manner you have specified.

Preparing to Use Graphics Animation

Because the animation routines have been designed to interact with a background display, you must first make sure that such a display is already defined.

To define a display with which the GELS can interact, you define **View**, **ViewPort**, and **RastPort** structures. For details on the construction of these structures, see chapter 1, "Graphics Primitives" and chapter 2, "Layers".

The graphics animation routines described in this chapter create additional material that is linked into the **View** structure. This material consists of additional instructions for color changes and dynamic reassignment of the hardware resources that create the display animation effects you specify.

Types of Animation

Using the Amiga system tools, you can perform two different kinds of image animation:

- Sprite animation
- Playfield animation

Sprite Animation

Sprites are hardware objects that you create and move independently of the playfield display. Sprites are always 16 low-resolution pixels wide, and are as high as you specify.

To move sprites, you must define where they are on-screen. The built-in priority circuitry determines how the sprite appears on-screen relative to the playfield elements or to other sprites.

You can manipulate sprites directly through a simple sprite set of routines, or by using the graphics kernel **VSprite** routines.

Playfield Animation

Sprites are normally moved against a background. This background area is called the playfield. You may treat the playfield area as a single background or separate it into two separately controllable sections, using dual-playfield mode. See chapter 1, "Graphics Primitives", for details on how to create and control playfields.

In playfield animation, sections of the playfield are modified. You draw, erase, and redraw objects into the playfield, creating an animation effect.

To move the data quickly and efficiently, the system uses one of the specialized built-in hardware devices, the *blitter*. The system uses the blitter to move the playfield objects, while it saves and restores the background. The objects controlled by the blitter are called **Bobs**, for "blitter objects".

Playfield animation is somewhat more complicated than **VSprite** animation from the point of view of system design, but not much more complicated for you as the user of the animation routines. The hardware displays the **VSprites** over the playfield automatically, and the priority overlay circuitry assures that they will be displayed in the correct order. If you are animating multiple **Bobs**, you control their *video priority* by defining the sequence in which the system draws them. The last one drawn has the highest video priority in the sense that it appears to be in front of all other **Bobs**.

A **Bob** is physically a part of the playfield. When the system displays a **Bob**, it must first save a copy of the playfield area into which the **Bob** will be drawn. Then the system can restore the playfield to its original condition when moving the **Bob** to a new location. Once the playfield areas have been saved, the system can draw the **Bob**. To move the **Bob**, the system must first restore the playfield area (thus, erasing the object) before it saves the playfield at the new location and draws the **Bob** there.

Bobs offer more flexibility and many more features than **VSprites**. **Bob** animation is less restrictive but slower than **VSprite** animation. **VSprites** are superior to **Bobs** in speed of display, because **VSprites** are mostly hardware-driven and **Bobs** are part hardware and part software. **Bobs**, on the other hand, are superior to **VSprites** because they offer almost all of the benefits of **VSprites** but suffer none of the limitations, such as size or number of colors. Both are very powerful and useful. The requirements of your particular application determine the type of GEL to use.

The GELS System

The acronym GEL describes all of the graphic "objects" supplied by the Amiga ROM kernel. Both **VSprites** and **Bobs** are GELS, as are the more advanced animation elements known as **AnimComps** and **AnimObs**.

Initializing the GEL System

To initialize the graphics element animation system, you provide the system with the addresses of two data structures. The system uses these data structures to keep track of the GELS that you will later define. To perform this initialization, you call the system routine **InitGels()**, which takes the form:

```
InitGels( head, tail, Ginfo );
```

where:

head
is a pointer to the **VSprite** structure to be used as the GEL list head

tail
is a pointer to the **VSprite** structure to be used as the GEL list tail

Ginfo
is a pointer to the **GelsInfo** structure to be initialized

The graphics animation system uses two "dummy" **VSprites** as place holders in the list of GELS that you will construct. The dummy **VSprites** are used as the head and tail elements in the system list of GELS. You add graphics elements to or delete them from this list.

The call to **InitGels()** forms a linked list of GELS which is empty except for these two dummy elements. When the system initializes the list with the dummy **VSprite**, it automatically gives the **VSprite** at the head the maximum possible negative y and x positions and the **VSprite** at the tail the maximum possible positive y and x positions. This assures that the two dummy elements are always the outermost elements of the list.

The y,x values are coordinates that relate to the physical position of the GEL within the drawing area. The system uses the y,x values as the basis for the placement (and later sorting) of the GELS in the list.

When you add a GEL to the list of graphics elements, the system links that GEL into the list shown above. Then the system adds any new element to the list immediately ahead of the first GEL whose y,x value is greater than or equal to that of the new GEL being added.

Types of GELS

Figure 3-2 shows how you can view the components of GELS as inter-related layers of graphics elements.

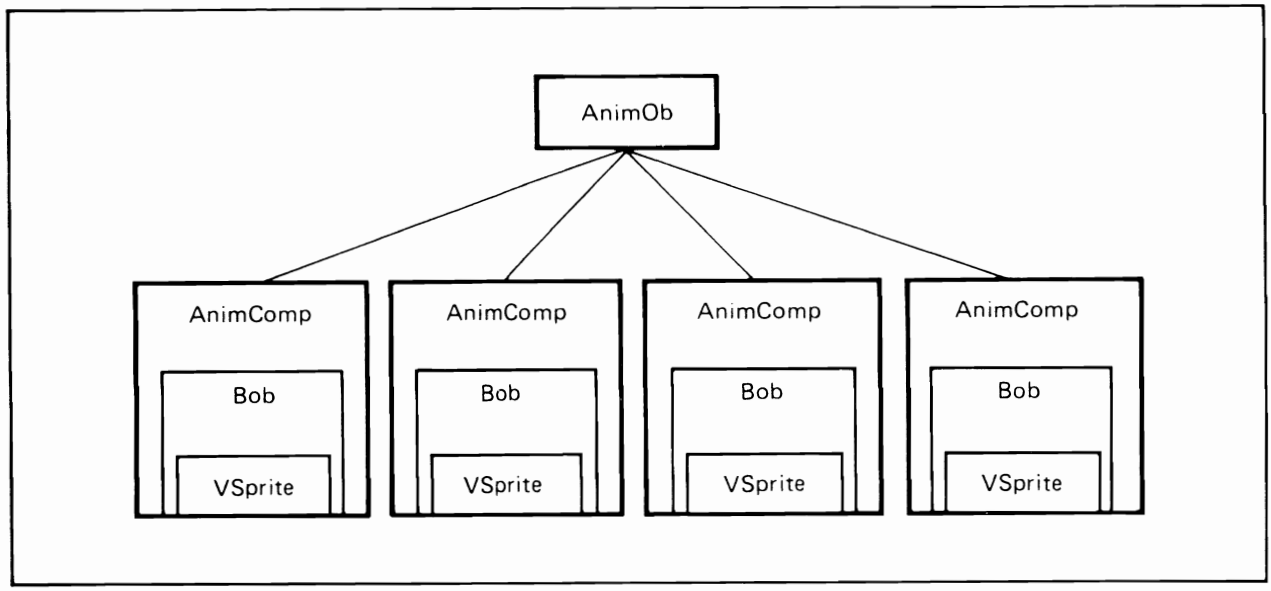


Figure 3-1: Shells of Gels

The types of GELS are listed below.

- Simple (hardware) sprites
- **VSprites**

- **Bobs**
- **AnimComps**
- **AnimObs**

VSprites and **Bobs** are the primary software-controlled animation objects. They are part of an integrated animation system. The simple sprites, on the other hand, are separate from the animation system. It is up to you to decide which type of sprite to use. The next sections describe all of these animation components.

Simple (Hardware) Sprites

The simple sprite is a special graphics element, related to the graphics animation system only in that it vies with the **VSprites** for the use of the same underlying hardware elements, the real hardware sprites.

The Amiga hardware has the ability to handle up to eight sprite objects. Each sprite is produced by one of the eight hardware sprite DMA channels. Each sprite is 16-bits wide and arbitrarily tall. The Amiga software provides a choice about how you can use these hardware elements.

You can either allocate one or more hardware sprites for your exclusive use, or you can allow all sprites to be managed by the system software and assigned as virtual sprites by the system. Using virtual sprites, it can appear as though you have an unlimited set of sprites with which to work. If you need only a few sprites, however, you may wish to use the less complex routines shown in Section 3.3, "Using Simple Sprites".

VSprites

The virtual sprite is the most elemental component. It contains a little more information than is needed to define a hardware sprite. The system temporarily assigns each **VSprite** to a hardware sprite, as needed.

The information in the **VSprite** structure allows the system to maintain the more general GEL functions, such as collision detection and double-buffering. After a sprite DMA channel has displayed the last line of a sprite, the system can reuse the channel to display a different

image lower on the screen. The system software takes advantage of this reusability to dynamically assign hardware sprites to carry **VSprite** images.

The **VSprite** is a data structure closely related to hardware sprites. The **VSprite** structure contains the following information:

- Size
- Image display data
- Screen coordinates
- Collision descriptors
- A pointer to color information

Bobs

The **Bob** (for blitter object) is the next outermost level of the GEL system. It is like an expanded hardware sprite done in software. It uses the same information defined in a **VSprite**, but adds other data that further defines this type of object. **Bobs** and **VSprites** differ in that the system draws **Bobs** into the playfield using the blitter, while it assigns **VSprites** to hardware sprites.

A **Bob** structure contains the following information:

- A pointer to a **VSprite**
- Priority descriptors
- Variables and pointers that define how and where to save the background

AnimComps

The **AnimComp** (for "animation component") is a data structure that extends the definition of a **Bob**. It allows the system to include the **Bob** as part of a total animation object. An **AnimComp** expands on the **Bob** data. **AnimComps** include the following:

- a pointer to this **AnimComp**'s **Bob**
- links that define the sequence of animation drawings
- information that describes the screen coordinates of the **AnimComp** with respect to the position of the **AnimOb**, described below.
- timing information for sequencing this **AnimComp** as part of the list of animation drawings
- a pointer to a user routine to execute in conjunction with this **AnimComp**

AnimObs

The **AnimOb** (for "animation object") is the primary animation object. It is a pseudo-object whose primary purpose is to link one or more **AnimComps** into a single overall object. As the **AnimOb** moves, so move its **AnimComps**. When the **Bobs** move with their **AnimComps**, the system sets the screen coordinates in the **VSprite** accordingly. **AnimObs** include the following:

- A pointer to this **AnimOb**'s first **AnimComp**
- Links to previous or succeeding **AnimObs**
- Information that describes the position of this **AnimOb** on the screen, as well as its velocity and acceleration

- Information for double-buffering this **AnimOb**, if desired
- A pointer to a user routine to execute in conjunction with this **AnimOb**

3.2. USING SIMPLE (HARDWARE) SPRITES

To use simple sprites, define their data structures and use the following routines:

- **ON_SPRITE** — a system macro to turn on Sprite DMA.
- **OFF_SPRITE** — a system macro to turn off Sprite DMA.
- **GetSprite()** — attempts to allocate a sprite from the virtual sprite machine for your exclusive use
- **ChangeSprite()** — modifies the sprite's appearance
- **MoveSprite()** — changes the sprite's position
- **FreeSprite()** — returns the sprite to the virtual sprite machine

These routines are described in detail in the following sections.

To use these simple sprite routines or the **VSprite** routines, you must incude the SPRITE flag in the data structure for **OpenScreen()**. Or, if you are not using Intuition, this flag must be specified in the **View** and **ViewPort** data structures before **MakeView()** is called.

Controlling Sprite DMA

You can use the graphics macros ON_SPRITE and OFF_SPRITE to control sprite DMA. OFF_SPRITE prevents the system from displaying any sprites, whether hardware or VSprite. ON_SPRITE restores the sprite data access and display. Note that the Intuition cursor is a sprite. Thus, if you use OFF_SPRITE, you make Intuition's cursor invisible as well.

Accessing a Hardware Sprite

You use **GetSprite()** to gain access to a new hardware sprite. You use a call such as:

```
status = GetSprite( sprite, number )
```

GetSprite() allocates a hardware sprite for your exclusive use. The virtual sprite allocator can no longer assign this sprite. Note that if you steal one sprite, you are effectively stealing two. The sprite pairs 0/1, 2/3, 4/5, and 6/7 share the same color registers. If you are stealing a hardware sprite, you steal its color registers as well. So you might as well ask for the other sprite in the pair. Table 3-1 shows the color registers assigned to each sprite pair.

Table 3-1: Sprite Color Registers

Color Registers	Sprite
16-19	0 or 1
20-23	2 or 3
24-27	4 or 5
28-31	6 or 7

You are not granted *exclusive* use of the color registers. If the **ViewPort** is 5 bit-planes deep, all 32 of the system color registers will still be used by the playfield display hardware.

Note, however, that registers 16, 20, 24, and 28 always generate the "transparent" color when selected by a sprite, regardless of which color is actually in them. Their true color will only be used if they are selected by a playfield. For further information, see the *Amiga Hardware Reference Manual.*

Also note that sprites and sprite colors are bound to the **ViewPort** in that you can reload the colors between **ViewPorts**. In other words, if a user in a **ViewPort** located in the top part of the screen allocates sprite 0 and a user in the a **ViewPort** at the bottom of the screen allocates sprite 1, these two sprites will not necessarily have the same color set, as the two **ViewPorts** can have totally independent sets of colors.

The inputs to the **GetSprite()** routine are:

sprite A pointer containing the address of a data structure called **SimpleSprite**

number The number (0-7) of the hardware sprite you wish to reserve. If number is -1, the system gets any sprite.

A value of 0-7 is returned in "status" if your request was granted, specifying which sprite you have allocated. A value of -1 means that this sprite is already allocated.

The structure for a simple sprite is shown below:

```
struct SimpleSprite {
      /* pointer to the definition data of the hardware
      ** sprite to be displayed
      */
      UWORD *posctldata;
      UWORD height;   /* the height of this simple sprite in rows */
      UWORD x,y;    /* current position */
      /* the number (0-7) of the hardware sprite associated
      **  with this simple sprite
      */
      UWORD num;
};
```

This data structure is found in the *sprite.h* file in the appendixes to this manual.

Changing the Appearance of a Simple Sprite

The **ChangeSprite()** routine changes the appearance of a reserved sprite. It is called by the following sequence:

```
ChangeSprite( vp, s, newdata )
```

ChangeSprite() substitutes a new data content for that currently used to display a reserved hardware sprite.

The inputs to this routine are:

vp A pointer to the **ViewPort** for this sprite or 0 if this sprite relative only to the current **View**

s A pointer to a **SimpleSprite** structure

newdata A pointer to a data structure containing the new data to be used

The structure for the new data is shown below:

```
struct userspritedata
{
        /* position and control information for this sprite */
        UWORD posctl[2];
        /* two words per line of sprite height, first of the two
        ** words contains msbit for color selection, second word
        ** contains lsbit (colors 0,1,2,3 from allowable color
        ** register selection set).  Color '0' for any sprite
        ** pixel makes it transparent.
        */
        UWORD sprdata[2][height]; /* actual sprite image */

        /* initialize to 0, 0 for unattached simple spites */
        UWORD reserved[2];
};
```

Moving a Simple Sprite

MoveSprite() repositions a reserved hardware sprite. It is called as follows:

```
MoveSprite( vp, sprite, x, y )
```

After you call this routine, the reserved sprite is moved to a new position relative to the upper left corner of the **ViewPort**.

The inputs to **MoveSprite()** are as follows:

vp A pointer to the **ViewPort** with which this sprite interacts or 0 if this sprite's position is relative only to the current View

sprite A pointer to a SimpleSprite structure

x, y Pixel position to which a sprite is to be moved. If the sprite is being moved over a high-resolution display, the system can only move the sprite in two-pixel increments. In low-resolution mode, single-pixel increments in the x direction are acceptable. For an interlaced mode display, the y direction motions are in two line increments. The same

image of the sprite is placed into both even and odd fields of the interlaced display.

The upper left corner of the **ViewPort** area has coordinates (0,0). The motion of the sprite is relative to this position.

The following example demonstrates how you move a simple sprite.

```
/* This program creates and displays a 320 by 200 by 2 bit-plane
 * single playfield display and adds one simple sprite to it.
 */

#include "exec/types.h"
#include "graphics/gfx.h"
#include "hardware/dmabits.h"
#include "hardware/custom.h"
#include "hardware/blit.h"
#include "graphics/gfxmacros.h"
#include "graphics/copper.h"
#include "graphics/view.h"
#include "graphics/gels.h"
#include "graphics/regions.h"
#include "graphics/clip.h"
#include "exec/exec.h"
#include "graphics/text.h"
#include "graphics/gfxbase.h"
#include "graphics/sprite.h"

#define DEPTH 2
#define WIDTH 320
#define HEIGHT 200
#define NOT_ENOUGH_MEMORY -1000

/* construct a simple display */

struct View view;
struct ViewPort viewport;

/* pointer to colormap structure, dynamically allocated */
struct ColorMap *cm;

struct RasInfo rasinfo;
struct BitMap bitmap;

SHORT xmove, ymove;

extern struct ColorMap *GetColorMap();
struct GfxBase *GfxBase;
```

```
/* save pointer to old view so can restore */
struct View *oldview;

USHORT colortable[] = {
        /* black, red, green, blue */
        0x000, 0xf00, 0x0f0, 0x00f,
        0,0,0,0,
        0,0,0,0,0,0,0,0,
        0,0,0,0,0,0,0,0,    /* sprites from here up */
        0,0,0,0,0,0,0,0
};

/* where to draw boxes */
SHORT  boxoffsets[] = {
        802, 2010, 3218
};

UWORD *colorpalette;

struct SimpleSprite sprite;

/* Last entry is "position control" for
** the next reuse of the hardware sprite.
** Simple sprite machine supports only one
** use of a hardware sprite per video frame.
** Any combination of binary bits from word
** 1 and word 2 per line establishes the
** color for a pixel on that line.
** Any nonzero pixels in lines 1-3 are color
** "1" of the sprite, lines 4-6 are color "2",
** lines 7-9 are color "3"
*/
UWORD sprite_data[ ] = {
        0,0,            /* position control */
        0x0fc3, 0x0000, /* image data line 1*/
        0x3ff3, 0x0000, /* image data line 2*/
        0x30c3, 0x0000, /* image data line 3*/
        0x0000, 0x3c03, /* image data line 4*/
        0x0000, 0x3fc3, /* image data line 5*/
        0x0000, 0x03c3, /* image data line 6*/
        0xc033, 0xc033, /* image data line 7*/
        0xffc0, 0xffc0, /* image data line 8*/
        0x3f03, 0x3f03, /* image data line 9*/

        /* NOTE this last line specifies unattached, simple sprites */
        0, 0            /* next sprite field */
};

/*************************************************************
** FOLLOWING IS FOR INFORMATION ONLY.... the simple-sprite machine
```

```
** directly sets these bits; the user has no need to fiddle with any
** of them.  Use the functions ChangeSprite, and MoveSprite to have
** an effect on the sprite.
**
** position control:
**
**        first UWORD:
**          bits 15-8, start vertical value, lowest 8 bits
**                     of this value contained here.
**          bits  7-0, start horizontal value, highest 8 bits
**                     of this value contained here.
**
**        second UWORD:
**          bits 15-8, end (stopping) vertical value, lowest
**                     8 bits of this value contained here.
**          bit 7 = Attach-bit (used for attaching sprites to
**                     get additional colors (15 instead
**                     of 3, supported by the hardware but
**                     NOT supported by the simple sprite
**                     machine).
**          bits 6-4 (unused)
**          bit 2 start vertical value; bit 8 of that value.
**          bit 2 end vertical value; bit 8 of that value.
**          bit 2 start horizontal value; bit 0 of that value.
**
*************************************************************************/

main()
{
      LONG i;
      SHORT j,k,n;
      SHORT spgot;
      UBYTE *displaymem;

      GfxBase = (struct GfxBase *)OpenLibrary( "graphics.library" , 0 );
      if( GfxBase == NULL ) exit(100);

      /* save current view to restore later */
      oldview = GfxBase->ActiView;

      /* example steals screen from Intuition if started from WBench */

      InitView( &view );            /* initialize view */
      InitVPort( &viewport );       /* init view port */
      view.ViewPort = &viewport;      /* link view into viewport */

      /* init bit map (for rasinfo and rastport) */
      InitBitMap( &bitmap, DEPTH, WIDTH, HEIGHT );

      /* init RasInfo */
```

```
rasinfo.BitMap = &bitmap;
rasinfo.RxOffset = 0;
rasinfo.RyOffset = 0;
rasinfo.Next = NULL;

/* now specify critical characteristics */
viewport.DWidth = WIDTH;
viewport.DHeight = HEIGHT;
viewport.RasInfo = &rasinfo;

/* initalize the color map.  It has 32 entries. Sprites take up
**the top 16 and we want to specify some sprite colors
*/
cm = GetColorMap( 32 );

/* no memory for color map */
if(cm == NULL) {
      FreeMemory();
      exit( 100 );
}

colorpalette = (UWORD *)cm->ColorTable;
for(i=0; i<32; i++) {
      *colorpalette++ = colortable[i];
}

/* copy my colors into this viewport structure */
viewport.ColorMap = cm;

/* addition for simple sprite: */
vp.Modes = SPRITES;

/* allocate space for bitmap */
for(i=0; i<DEPTH; i++) {
      bitmap.Planes[i] = (PLANEPTR) AllocRaster( WIDTH, HEIGHT );
      if( bitmap.Planes[i] == NULL ) exit( NOT_ENOUGH_MEMORY );

       /* clear the display area */
      BltClear( bitmap.Planes[i], RASSIZE(WIDTH,HEIGHT), 1 );
}

/* construct copper instr (prelim) list */
MakeVPort( &view, &viewport );

/* merge prelim lists together into a real
** copper list in the view structure.
*/
MrgCop( &view );
LoadView( &view );
```

```
/* now fill some boxes so that user can see something */
/* always draw into both planes to assure true colors */
for(n=1; n<4; n++)      /* three boxes */
{
      for(k=0; k<2; k++)
      {
            /* boxes will be in red, green and blue */
            displaymem = bitmap.Planes[k] + boxoffsets[n-1];
            DrawFilledBox( n, k, displaymem );
      }
}

/*********************************************
/* now we are ready to play with the sprites!
*********************************************/

/* Get the next available sprite.  We should do an error
** check, if returns -1, then no sprites are available
*/
spgot = GetSprite( &sprite, -1 );

sprite.x = 0;           /* initialize position and size info */
sprite.y = 0;           /* matches that shown in sprite_data */
sprite.height = 9;      /* so that system knows layout of data later */

/* now put some colors into this sprite's color registers so as
** to custom control the colors this particular sprite will display.
** NOTE: sprite pairs share color registers;  i.e. sprite 0 and 1,
** 2 and 3, 4 and 5, 6 and 7 as pairs share the same sets of color
** registers (see the Amiga Hardware manual for details).
** The code following figures out which sprite the system gave us,
** and sets that sprite's color registers to the correct value
*/
k = ((spgot & 0x06)*2) + 16;

/* convert sprite number into the base number for its color reg set */
/* value at k treated as transparent */
SetRGB4( &viewport, k+1, 12, 3, 8 );
SetRGB4( &viewport, k+2, 13, 13, 13 );
SetRGB4( &viewport, k+3, 4, 4, 15 );

/* top of sprite is red, middle is white, bottom is blue-ish */
ChangeSprite(&viewport,&sprite,sprite_data);

MoveSprite(0,&sprite,30,0);

xmove = 1;  ymove = 1;
for( n = 0; n < 4; n++ ) {
      i=0;
      while( i++ < 185 ) {
```

```
                MoveSprite( 0, &sprite, sprite. x + xmove, sprite.y + ymove );

                /* slow it down to one move per video frame */
                WaitTOF();
            }
            ymove = -ymove;
            xmove = -xmove;
        }

        /* free this sprite so others can use it also */
        FreeSprite( spgot );


        /* restore the system to its original state */
        LoadView( oldview );
        FreeMemory();
        CloseLibrary( GfxBase );

}       /* end of main() */

/* return user and system-allocated memory to sys manager */
FreeMemory()
{
        LONG i;

        /* free drawing area */
        for( i=0; i<DEPTH; i++ ) {
            if( bitmap.Planes[i] != NULL ) {
                FreeRaster( bitmap.Planes[i], WIDTH, HEIGHT );
            }
        }
        /* free the color map created by GetColorMap() */
        if( cm != NULL ) FreeColorMap( cm );

        /* free dynamically created structures */
        FreeVPortCopLists( &viewport );
        FreeCprList( view.LOFCprList );
        return( 0 );
}

DrawFilledBox( fillcolor, plane, displaymem )
SHORT fillcolor,plane;
UBYTE *displaymem;
{
        UBYTE value;
        LONG j;

        for(j=0; j<100; j++) {
            if((fillcolor & (1 << plane)) != 0) {
                value = 0xff;
```

```
            } else {
                value = 0;
            }
            for(i=0; i<20; i++) {
                *displaymem++ = value;
            }
            displaymem += (bitmap.BytesPerRow - 20);
        }
        return(0);
}
```

Relinquishing a Simple Sprite

The **FreeSprite()** routine returns an allocated sprite to the virtual sprite machine. The virtual sprite machine can now reuse this sprite to allocate virtual sprites. The syntax of this routine is:

```
FreeSprite( num )
```

where **num** is the number (0-7) of the sprite you want to return.

NOTE: You *must* free sprites after you have allocated them using **GetSprite()**. If you do not free them, and your task ends, the system will have no way of reallocating those sprites until the system is rebooted.

3.3. USING VSPRITES

This section tells how to define a **VSprite**. It describes how to:

- Specify the size of the **VSprite** object
- Select its colors
- Form its image
- Specify its position within the drawing area

- Add it to the list of GELS
- Control it after you add it to the list

The system software also provides a way to detect collisions between individual **VSprites** and other on-screen objects. Collision detection applies to both **VSprites** and to **Bobs**. It appears as a separate topic under "VSprite and Bob Topics".

The first step in defining a **VSprite** is telling its dimensions to the system.

Specifying the Size of a VSprite

A **VSprite** is always 16 pixels wide and may be any number of lines high. Each pixel is the same size as a pixel in low-resolution mode (320 pixels across a horizontal line) of the graphics display.

To specify how many lines make up the **VSprite** image, you use the **VSprite** structure **Height** variable.

If your **VSprite** is 12 lines high and the name of your **VSprite** structure is myVSprite, then you can set the height value with the following statement:

```
myVSprite.Height = 12;
```

Each line of a **VSprite** requires two data words to specify the color content of each pixel. This means that the data area containing the **VSprite** image is 12 x 2 or 24 words long.

See the next section for details on how bits of these data words select the color of the **VSprite** pixels.

Specifying the Colors of a VSprite

Because **VSprites** are so closely related to the hardware sprites, the choice of colors for **VSprites** is limited in the same way. Specifically, each pixel of a **VSprite** can be any one of three different colors, or it may be transparent. However, the system software provides a great deal of versatility in the choice of colors for the virtual sprites. Each virtual sprite may have its own set of three unique colors.

When the system assigns a hardware sprite to carry the **VSprite**'s image, it assigns that **VSprite**'s color set to the hardware sprite that will produce that image. To define which set of three colors to use for this **VSprite**, you initialize the **VSprite** structure pointer named **SprColors**, **SprColors** points to the first data item of three sequentially-stored 16-bit values. The system then jams these values into the selected hardware sprite's color registers when it is being used to display this **VSprite**.

Every time you direct the system to redraw the **VSprites**, the GEL system re-evaluates the current on-screen position of each **VSprite** and decides which hardware sprite will carry this **VSprite**'s image for this rendering. It creates a customized Copper instruction sequence including both the repositioning of hardware sprites and reloading of sprite color registers for various screen positions. Thus, during a move sequence, a **VSprite** may be represented by one or many different real hardware sprites, depending on its current position relative to other **VSprites**.

For example, if your set of colors is defined by the statement:

```
WORD spriteColors = { 0x00F, 0x0F0, 0xF00 };
```

and if your **VSprite** is named "myVSprite", then to set the **VSprite** colors you would use the following statement:

```
myVSprite.SprColors = &spriteColors;
```

How you specify the **VSprite** colors may affect how many **VSprites** you can show on-screen at any one time. For further information, see "How VSprites are Assigned" later on.

Now that you've specified the size and colors of the **VSprite**, it's time to tell the system about its form.

Specifying the Shape of a VSprite

To define the appearance of a **VSprite**, initialize the **VSprite** structure pointer called **ImageData** to point to the first word of the image data. A **VSprite** image is defined exactly as the image of a real hardware sprite. It takes two sequential 16-bit data words to define each line of a **VSprite**.

To select colors for the pixels of a **VSprite**, examine the combination of the data bits in corresponding locations in each of the two data words that define each line. The first of each pair of data words supplies the low-order bit of the color selector for that pixel; the second word of the pair supplies the high-order bit.

For example:

```
mem       0101111111111111
mem + 1   0011111111111111
```

Reading from left to right, the combinations of these two sequential memory data words form the binary values of 00, 01, 10, 11, and so on. These binary values select colors as follows.

00 - selects **VSprite** color of "transparent"
01 - selects the first of three **VSprite** colors you have defined
10 - selects the second **VSprite** color
11 - selects the third **VSprite** color

In those areas where the combination of bits yields a value of 00, the **VSprite** is transparent. Any object whose priority is lower than that of the **VSprite** will show through in transparent sections of the **VSprite**.

Thus, you might form a full three-color image, with some transparent areas, from a data set like the following sample:

VSprite Data

mem	1111111111111111	Defines top line -
mem + 1	1111111111111111	contains only Color 3
mem + 2	0011111111111100	Defines second line -
mem + 3	0011000000001100	contains colors 1 & 3 and some transparency
mem + 4	0000110000110000	Defines third line -
mem + 5	0000111111110000	contains colors 2 & 3 and some transparency
mem + 6	0000001001000000	Defines fourth line -
mem + 7	0000001111000000	contains colors 2 and 3 and some transparency
mem + 8	0000000110000000	Defines last line -
mem + 9	0000000110000000	contains color 3 and and some transparency

The **VSprite** Height for this sample image is 5.

SprColors must point to the set of three colors that are to be used to display this **VSprite**, and **ImageData** must point to the location ("mem" in the example) that contains the first word of the **VSprite** definition.

After you've specified the shape of the **VSprite**, you tell the system where to display it.

Specifying VSprite Position

To control the position of a **VSprite**, you use the y and x variables within the **VSprite** structure. You specify the position of the upper left corner of a **VSprite** relative to the upper left corner of the drawing area where you wish the **VSprite** to appear.

Assign a value of 0,0 for y,x to make the **VSprite** appear with its upper left corner against the upper left corner of the drawing area. You can use values of y and x to move the **VSprite** entirely off-screen, if you wish.

You resolve the vertical positioning for **VSprites** in terms of the non-interlaced mode of the display. When you position a **VSprite** so that its y-value is within the visible area of the screen, you can select any one of 200 possible positions down the screen at which its topmost edge can be placed.

You resolve the horizontal positioning for **VSprites** in terms of the low-resolution mode of the screen display. When you position a **VSprite** so that its x-value is within the visible area of the screen, you can select any one of 320 possible positions across the screen at which its leftmost edge can be placed. Note that if you are using **VSprites** under Intuition and within a screen, they will be positioned relative to the upper left-hand corner of the screen.

Now that you've defined the VSprite's size, colors, shape, and position, you may want to know where to add information to the data structures or where to check about the progress of the system routines. The following sections describe the functions of the **VSprite** flags, the variables that let you do some of these activities.

Using VSprite Flags

The **VSprite** data structure contains a variable named **Flags** that has information about its data and about the progress of the system routines. The following sections describe the uses of the VSPRITE, VSOVERFLOW, and GELGONE flags. You can use these flags to perform these tasks:

VSPRITE	Indicate whether the system should treat the structure as a **VSprite** or part of a **Bob**.
VSOVERFLOW	Check on the **VSprites** the system can't display. (This is a read-only system variable.)

GELGONE	Find out if the system has moved a GEL outside the clipping region of the drawing area. (This is a read-only system variable.)

VSPRITE Flag

To tell the GEL routines to treat this **VSprite** structure as a VSprite instead of a Bob, set the VSPRITE flag to 1. This affects the interpretation of the data layout and the use of various system variables. If you set the VSPRITE flag bit to zero, the GEL routines treat this **VSprite** structure as though it defined a **Bob** instead of a **VSprite**.

NOTE: Under Intuition, **VSprites** work only in screens, not in windows. **Bobs** work in both screens and in windows. Thus, if you wish to use **VSprites** and **Bobs** together, you can only do so by writing directly to the **RastPort** of a screen.

VSOVERFLOW Flag

If you have currently defined more **VSprites** at the same horizontal line than the system can possibly assign to the real hardware sprites, then the **VSprites** that the system can't display have their VSOVERFLOW flag set. This means that it is possible that one or more **VSprites** will not appear on the display for this pass of producing the GELS.

GELGONE Flag

When the GELGONE flag is set to 1, you know that the system has moved a GEL (**VSprite** or a Bob) entirely outside of the clipping region of the drawing area. You can assume that the system will fully or at least partially draw any objects within the clipping region. Because the system will not draw this object that is outside the clipping area, you may wish to use **RemVSprite()** to delete the **VSprite** from the GEL list in order to speed up processing of the rest of the list. Of course, **VSprites** that you remove from the list are no longer managed or checked by the system.

Now that you've learned how to use the **VSprite** flags to control the **VSprites** and to check on the system routines, you are ready to tell the system how many **VSprites** to handle.

Adding a VSprite

To control **VSprites**, you first describe them using the **VSprite** structure variables mentioned above. Next you tell the system (by adding the **VSprites** to the GEL list) which **VSprites** to handle. This section tells you how to add a **VSprite** to the GEL list.

To add a **VSprite** to the system GEL list, call the system routine **AddVSprite()**, and specify the address of the **VSprite** structure that controls this **VSprite** as well as the **RastPort** with which it is associated.

A typical system call for this purpose follows.

```
struct VSprite myVSprite;
    ...
    ...
AddVSprite( &myVSprite, &rastport );
```

The next section tells how to remove a **VSprite** from the system list.

Removing a VSprite

To remove a **VSprite** from the list of controlled objects, use the system routine **RemVSprite()**. This function takes the following form:

```
RemVSprite( VS );
```

where **VS** is a pointer to the **VSprite** structure to be removed from the GEL list

Once you've specified which **VSprites** you want to add to or remove from the list of controlled objects, you need to organize that list.

Getting the VSprite List in Order

When the system has displayed the last line of a **VSprite**, it reassigns the hardware sprite to another **VSprite** located at a lower, farther left position on-screen. The system allocates hardware sprites in the order in which it encounters the **VSprites** in the list. Therefore, you must sort the list of **VSprites** before the system can assign the use of the hardware sprites correctly.

When you first enter **VSprites** into the list using **AddVSprite()**, the system uses the y,x coordinates to place the **VSprites** into the correct position in the list. If you change the y,x coordinates after they are in the list, you must reorder the list before the system can use it to produce the display.

You use the routine **SortGList()** (for "sort the GEL list") to get them in the correct order before asking the system to display them. This sorting step is essential! You call this function as follows:

```
SortGList( RPort );
```

where **RPort** is a pointer to the **RastPort** structure containing the **GelsInfo**

Note that there may be a GEL list in more than one **RastPort**. You must sort all of them.

Now that you've put the list in order, you are ready to display the **VSprites** using the system functions described in the following sections.

Displaying the VSprites

The next few sections explain how to display the **VSprites**. You use the following system routines:

- **ON_DISPLAY** — to turn on the playfield display
- **ON_SPRITE** — to turn on the **VSprites** display
- **DrawGList()** — to draw the elements into the current **RastPort**

- **MrgCop()** — to install the **VSprites** into the display
- **LoadView()** — to ask the system to display the new **View**
- **WaitTOF()** — to synchronize the routines with the display

Turning on the Display

Before you can view a display on-screen, you must enable the system direct memory access for both the hardware sprites and the playfield display. To enable the display of both playfield and **VSprites**, use the system macro calls:

```
ON_DISPLAY;
ON_SPRITE;
```

Drawing the Graphics Elements

The system routine called **DrawGList()** looks through the list of controlled GELS. It prepares necessary instructions and memory areas to display the data according to your requirements. You call this routine as follows.

```
DrawGList( RPort, VPort );
```

where:

RPort
is a pointer to the **RastPort**

VPort
is a pointer to the **View**

Because the system links **VSprites** to a View, the use of a **RastPort** is not significant for them. However, you can use **DrawGList()** for **Bobs** as well as **VSprites**, so it is required that you pass the pointer to the **RastPort** to the routine. **DrawGList()** actually draws **Bobs** into that **RastPort** when you execute the instructions.

Once **DrawGList()** has prepared the necessary instructions and memory areas to display the data, you'll need to install the **VSprites** into the display with **MrgCop()**.

Merging VSprite Instructions

Recall that the call to **DrawGList()** did not actually draw the **VSprites**. It simply provided a new set of instructions that the system uses to assign the **VSprite** images to real hardware sprites, based on their positions. The **View** structure already has a set of instructions that specifies how to construct the display area. It includes pointers to the set of **VSprite** instructions that was made by the call to **DrawGList()**. To install the current **VSprites** into the display area, you call the routine **MrgCop()** to merge together all of the display-type instructions in the **View** structure. You call this routine as follows.

```
MrgCop( View );
```

where **View** is a pointer to the **view** structure whose Copper instructions are to be merged

DrawGList() handles **Bobs** as wells as **VSprites**. Therefore, the call to **DrawGList()**, although it did not really draw the VSprite images yet, *does* draw the **Bobs** into the selected **RastPort**.

Loading the New View

Now that the display instructions include the definition of the **VSprites**, you can ask that the system prepare to display this newly configured View. You do this with the following system routine:

```
LoadView( view );
```

where:

view
is a pointer to the **View** that contains the pointer to the Copper instruction list

The Copper instruction lists are double-buffered, so this instruction does not actually take effect until the next display field occurs. This avoids the possibility of some routine trying to update the Copper instruction list while the Copper is trying to use it to create the display.

Now you'll want to make sure that the timing of the routines coincides with that of the display.

Synchronizing with the Display

To synchronize your routines with the display, you use a call to the system routine **WaitTOF()**. Although your routines may possibly be capable of generating more than 60 complete display fields per second, the system itself is limited to 60 displays per second. Therefore, after generating a complete display, you may wish to wait until that display is ready to be shown on-screen before starting to generate the next one. **WaitTOF()** holds your task until the vertical blanking interval (blank area at the top of the screen) has begun. At that time, the system has retrieved the current Copper instruction list and is ready to allow generation of a new list.

The call to the vertical blanking synchronization routine takes the following form:

```
WaitTOF();
```

Now that you've learned how to add and display **VSprites**, you may want want to change some of their characteristics, as shown in the following section.

Changing VSprites

Once the **VSprite** has been added to the GEL list and is in the display, you can change some of its characteristics with the following operations:

- Pointing to a new **VSprite** image (change the **ImageData** pointer)

- Pointing to a new **VSprite** color set (change the **SprColors** pointer)
- Defining a new **VSprite** position (change the y,x values)

The section immediately following provides a summary of the **VSprite** operations in their proper sequence.

VSprite Operations Summary

- Define a **View** structure that you can later merge with the **VSprite** instructions.
- Initialize the GEL system (call **InitGels()**). This only needs to be done once.
- Define the VSprite:
  - Height
  - On-screen position
  - Where to find **ImageData** data
  - Where to find **SprColors** to use
  - Define **VSprite** structure flags to show that this is a **VSprite**.
- Add the **VSprite** to the GEL list.
- Change the **VSprite** appearance by
  - Changing the pointer to **ImageData**
  - Changing its height
- Change the **VSprite** colors by changing the pointer to **SprColors**
- Move the **VSprite** by defining a new y,x position

- Display the **VSprite** with this sequence of routines:

  - **ON_DISPLAY**;
  - **ON_SPRITE**;
  - **SortGList()**
  - **DrawGList()**
  - **MrgCop()**
  - **LoadView()**

Once you've mastered the basics of handling **VSprites**, you may want to study the next two sections to find out how to reserve hardware sprites for use outside the **VSprite** system and how to assign the **VSprites**.

VSprite Advanced Topics

This section describes advanced topics pertaining to **VSprites**. It contains details about reserving hardware sprites for use outside of the **VSprite** system, information about how **VSprites** are assigned, and more information about **VSprite** colors.

Reserving Hardware Sprites

To prevent the **VSprite** system from using specific hardware sprites, you can write into the variable named **sprRsrvd** in the **GelsInfo** structure. The pointer to the **GelsInfo** structure is contained in the **RastPort** structure. If the contents of this 8-bit value is zero, then all of the hardware sprites may be used by the **VSprite** system. If any of the bits is a 1, then the sprite corresponding to that bit will not be utilized by **VSprites**. Note that this increases the likelihood of a **VSprite** VSOVERFLOW. See the next section, "How VSprites are Assigned", for further details on this topic.

Hardware sprite usage corresponds to the bit assignment shown below:

RESERVE SPRITE NUMBER: 7 6 5 4 3 2 1 0

If this **sprRsrvd** bit is a 1: 7 6 5 4 3 2 1 0

You normally assign hardware sprites in pairs, as suggested by this example. Suppose you want to reserve sprites 0 and 1. Your program would typically include the following kinds of statements:

```
struct RastPort myRastPort;   /* the view structure is defined */
    ...
    ...
myRastPort->GelsInfo->sprRsrvd = 0x03;   /* reserve 0 and 1 */
```

If you reserve a hardware sprite for your own use, the system is unable to use that hardware sprite when it makes a **VSprite** assignment. In addition, because pairs of hardware sprites share color register sets, reserving one hardware sprite effectively eliminates two.

If you are using the simple sprite system to allocate sprites, you can look in the **GfxBase** structure to see which sprites are already in use.

NOTE: If Intuition is running, sprite 0 is already reserved for use as the cursor.

The reserved sprite status is accessible as:

```
currentreserved = GfxBase->SpritesReserved.
```

The next section presents a few trouble-shooting techniques for **VSprite** assignment.

How VSprites are Assigned

Each **VSprite** can display three possible colors plus transparent. To define colors for **VSprites**, you use the **SprColors** pointer. **SprColors** points to the first of three word quantities, representing the three possible pixel colors for that virtual sprite.

Although the **VSprites** are handled by the automatic routines, the system *may* run out of sprites. If you ask that the software display more than 4 **VSprites** on a single horizontal scan line, it is possible that one or more sprites may disappear until the conflict is resolved.

Here is the reason that the **VSprite** routines might have problems, and some suggestions on how to avoid them. There are 8 *real* sprite DMA channels. Sprites 0 and 1 share color registers 17-19; sprites 2 and 3 share registers 21-23; sprites 4 and 5 share registers 25-27; and sprites 6 and 7 share registers 29-31.

When the **VSprite** routines use the sorted list of **VSprite** elements, they build a Copper instruction list that decides when to re-use a sprite DMA channel. They also build a Copper instruction stream that stuffs the color register set for the sprite selected at that time on-screen to represent this **VSprite** image.

This process consists of the following steps:

1. Use real sprite 0 to represent the first virtual sprite. Load that virtual sprite's colors into the three color registers for sprite 0 (registers 17, 18, 19).

2. Now look at the rest of the virtual sprites the user wishes to display on this same horizontal line.

3. If the VSprite-color pointers are all different from the pointer found in the sprite 0 pointer, then it will not be possible to use the real sprite 1 DMA channel for display on this line because it shares the real sprite 0 colors.

4. Conversely, if one of the other virtual sprites to appear on this line shares the same virtual-color pointer, then the **VSprite** routines can use sprite DMA channel 1 to represent that second virtual sprite.

5. The **VSprite** routines continue to map virtual sprites against the real sprites until either of the following events occurs:

    a. All virtual sprites are assigned, or

    b. The system runs out of real sprites that it can use.

The system will run out of real sprites to use if you ask the virtual sprite system to display more than four sprites having different pointers to their color table on the same horizontal line.

During the time that there is a conflict, one or more of your virtual sprites will disappear.

You can avoid the above problems by taking the following precautions:

- Minimize the number of **VSprites** you wish to appear on a single horizontal line.

- If colors for some virtual sprites are the same, assure that the pointer for each of the **VSprite** structures for these virtual sprites points to the same memory location, rather than to a duplicate set of colors elsewhere in memory.

The following section explores what happens if you don't specify the colors of a **VSprite**.

If You Don't Specify VSprite Colors

To pick the set of colors to use, you specify the pointer named **SprColors**. If you specify a 0 value for **SprColors**, then that **VSprite** does *not* generate a color-change instruction stream for the Copper when the system displays it. Instead, the **VSprite** appears drawn in the color set that is currently written into the color registers for the hardware sprite currently selected to display this **VSprite**.

Table 3-2 shows how the hardware sprites use the color registers to select their possible range of colors:

Table 3-2: Hardware Sprite Color Registers

Hardware Sprite	Color Registers
0 and 1	17 - 19
2 and 3	21 - 23
4 and 5	25 - 27
6 and 7	29 - 31

During one screen display, the system may use hardware sprite number 1 to display a **VSprite**. In this case, the **VSprite** selects its three available colors from color register numbers 17-19. On another screen display, the system may select hardware sprite number 7 to display the same **VSprite**. In this case, the hardware sprite uses color registers 29-31.

Therefore, if you make the **SprColors** pointer a 0, specifying that color does not matter, the system may display your **VSprite** in any one of a set of four different possible color groupings as indicated in the table above.

The next section clarifies how the **VSprite** and playfield colors interact.

How VSprite and Playfield Colors Interact

The **VSprites** use system color registers 16 through 31 to hold the **VSprite** color selections. There are only 32 color registers in the system. The highest 16 color registers (16-31) are shared with the playfield color selections. If you are working in 32-color low-resolution mode, the system makes the first 16 color selections for the playfield pixels from color registers 0-15, and then makes remaining color selections from color registers 16-31.

If you are using the **VSprite** system and specifying the colors (using **SprColors**) for each **VSprite**, then the contents of color registers 16-31 will be constantly changing as the video display beam progresses down the screen. The Copper instructions change the registers to display the correct set of colors for your **VSprites** depending on their positions. If you have any part of a 32-color playfield display drawn in any of the colors shown in Table 3-2, those colors appear to flicker and change as your **VSprites** move.

This problem also affects 32-color **Bobs** because **Bobs** are actually drawn as part of the playfield display. Anything that affects the playfield affects the **Bobs** as well.

You can avoid this flickering and changing of colors by taking the following precautions:

a. Use no more than 16 colors in the playfield display whenever you use **VSprites**, or

b. If you are using a 32-color playfield display, do not use any colors other than 0-15, 16, 20, 24, and 28. The remaining color numbers are used by the **VSprite** system, or

c. Specify the **VSprite SprColors** pointer as a value of 0. This avoids changing the contents of any of the hardware sprite color registers, but may cause the **VSprites** to change colors depending on their positions relative to each other, as described in the previous section.

Alternatives a and b are the easiest to implement.

3.4. USING BOBS

Because **Bobs** and **VSprites** are both graphics objects handled by the GEL system, they share many of the same data requirements. **VSprites** and **Bobs** differ primarily in that **Bobs** are drawn into the playfield using the blitter, while **VSprites** are assigned to hardware sprites.

The following sections describe how to define a **Bob**. These sections include how to:

- specify its size
- select its colors
- form its image
- specify its on-screen position

Because a **Bob** is a more complex object than a VSprite, you must also define various other items, such as the color depth of the **Bob**, how to handle the drawing of the **Bob**, and certain other variables that the GEL system requires when **Bobs** are used.

To define a **Bob**, you must first link it to a **VSprite** structure.

Linking a Bob to a VSprite Structure

To fully define a **Bob**, you define two different structures: a **VSprite** structure and a **Bob** structure.

The graphics animation system has been designed as a set of inter-related elements, each of which builds on the information provided by the underlying structure to create additional versatility.

The common elements, such as height, collision-handling information, position in the drawing area, and pointers to the data definition are part of the **VSprite** structure. The added features, such as drawing sequence, data about saving and restoring the background, and other features not common to **VSprites** are part of the **Bob** structure instead.

The **VSprite** and **Bob** structures must point to the other, so that the system knows where all of the appropriate variables are defined. For example, suppose your program defines two structures that are to define a **Bob** named "myBob":

```
struct Bob myBob;
struct VSprite myVSprite;
```

You must create a link between the two structures with a set of program statements such as:

```
myBob.BobVSprite = &myVSprite;
myVSprite.VSBob = &myBob;
```

Now the system can go back and forth between the two structures to obtain the various elements as needed to define the **Bob**.

Now that you've linked the two structures, you're ready to define the size of the **Bob**.

Specifying the Size of a Bob

Whereas a **VSprite** was limited to 16 pixels of width, a **Bob** can be any size you wish to define. To specify the size of a **Bob**, you use not only the **Height** but also the **Width** variable. You specify these variables in a **VSprite** structure associated with the **Bob**. Specify the width as the number of 16-bit words it takes to fully contain the object.

As an example, suppose the **Bob** is 24 pixels wide and 20 lines tall. You use statements such as the following to specify the size:

```
myVSprite.Height = 20;     /* 20 lines tall */
myVSprite.Width  = 2;      /* 2 words = 24 pixels wide, rounded
                              up to the next multiple of 16 pixels. */
```

Because **Bobs** are drawn into the playfield background, the pixels of the **Bob** are the same size as the background pixels. With hardware sprites, the pixels are of a fixed size (low resolution).

The next step is to tell the system the colors you want for this **Bob**.

Specifying the Colors of a Bob

Because a **Bob** is drawn into the playfield area, it can have as many colors as the playfield area itself. Typically a 5-bit-plane, low-resolution mode display allows you to select playfield pixels (and therefore, **Bob** pixels) from any of 32 active colors out of a system palette of 4096 different color choices. The set of colors you select for the playfield area is the set of colors the system uses to display the **Bobs**.

For **Bobs**, the system ignores the **SprColors** variable in the VSprite structure. You use the **Depth** variable in the **VSprite** structure to define how much data is provided to define the **Bob**. This variable also defines how many different colors you can choose for each of the pixels of a **Bob**.

The **Depth** variable specifies how many bit-plane images the system must retrieve from the **Bob** image data area to make up the **Bob**. These are called bit-plane images as the system will write each image into a different bit-plane. The combination of bits in identical y,x positions in each bit-plane determines the color of the pixel at that position.

For example, if you specify only one plane, then the bits of that image let you select only two different colors: one color for each bit that is a 0, a second color for each bit that is a 1. Likewise, if there are 5 images stored sequentially, and you specify a depth of 5, each image contributes one bit for each position in the image to the color number selector, allowing up to 32 different choices of color for each **Bob** pixel.

You specify depth by a statement such as the following:

```
myVSprite.Depth = 5;    /* allow 32 colors,
                           requires that a 5-bit-plane-deep image
                           be present in data area. */
```

After you've specified **Bob** colors, you're ready to tell the system about **Bob** shape.

Specifying the Shape of a Bob

The organization in memory of a **Bob** is different from that of a **VSprite** because of the way the system retrieves data to draw **Bobs**. To define a **Bob**, you must still initialize the **ImageData** pointer to point to the first word of the image definition; however, the layout of the data is different for **Bobs** than for **VSprites**.

The sample image below shows the same image defined as a **VSprite** in the **VSprite** section above. The data, however, is stored in a way typical of a **Bob**.

If a shape is 2-bits "deep" and is a triangular shape, you would lay it out in memory as follows:

BOB LAYOUT

<first bit-plane data>

	mem
mem + 1	0011000000001100
mem + 2	0000111111110000
mem + 3	0000001111000000
mem + 4	0000000110000000

<second bit-plane data>

mem + 5	1111111111111111
mem + 6	0011111111111100
mem + 7	0000110000110000
mem + 8	0000001111000000
mem + 9	0000000110000000

<<third bit-plane data>

...

<<fourth bit-plane data>

...

<<fifth bit-plane data>

...

To state the width of the **Bob** image, you use 16-bit words. The width value is the number of words that fully contain the image. For example, you store a 29-bit wide image in 32 bits (2 data words of 16 bits each) for each line of its data.

You still specify the number of lines with the **Height** variable in the **VSprite** data structure. However, you treat **Height** somewhat differently for a **Bob** than for a **VSprite**. Specifically, for a **VSprite**, two adjacent data words that always occur together define the colors of each **VSprite** pixel. For a **Bob**, the **Height** variable defines how many adjacent data words it takes to define one complete bit-plane image. That is, for a **Bob** the number of adjacent data words in each bit-plane image definition is given by the following formula: **Height** x **Width**.

The **Depth** variable defines how many adjacent (end-to-end) images there are in the data area to define the shape of the **Bob**. See the example at the end of the "PlaneOnOff" section below.

The next few sections describe other variables that affect the color of **Bob** pixels.

Other Items Influencing Bob Colors

Three other variables in the **VSprite** structure affect the color of **Bob** pixels: **PlanePick**, **PlaneOnoff**, and **ImageShadow**.

PlanePick

Let's assume that you have defined a playfield composed of 5 bit-planes. The variable **PlanePick** in the **VSprite** structure lets you specify which of the bit-planes are to be affected when the system draws the **Bob**. **PlanePick** binary values affect the bit-planes according to the following pattern:

Draw **Bob** into plane:	5 4 3 2 1 0
If this **PlanePick** bit is a 1:	5 4 3 2 1 0

For example, if **PlanePick** has a binary value of:

0 0 0 1 1

then the system draws the first bit-plane image of the Bob into bit-plane 0 and the second image into plane 1.

Suppose that you still want to define an image of only 2 bit-planes, but wish to draw the **Bob** into bit-planes 1 and 4 instead of 0 and 1. Simply choose a **PlanePick** value of:

1 0 0 1 0

This value means "write first image into plane 1, second image into plane 4".

Now that you've defined the bit-planes you wish to have the system draw the **Bob** into, you may want to set the pointer for your sprite shadow mask. The next section describes what the shadow mask is and how to set a pointer to it.

Sprite Shadow Mask

The variable named **ImageShadow** is a pointer to a memory area that you have reserved for holding the shadow mask of a **Bob**. A shadow mask is the logical *or* combination of all 1-bits of a **Bob** image. There is a variable in the **VSprite** structure called **CollMask** (pointer to a collision mask, covered under "VSprite and Bob Topics") for which you reserve some memory space. The **ImageShadow** and **CollMask** pointers usually, but not necessarily, point to the same data.

Figure 3-3 shows an example of a shadow mask with only the 1-bits.

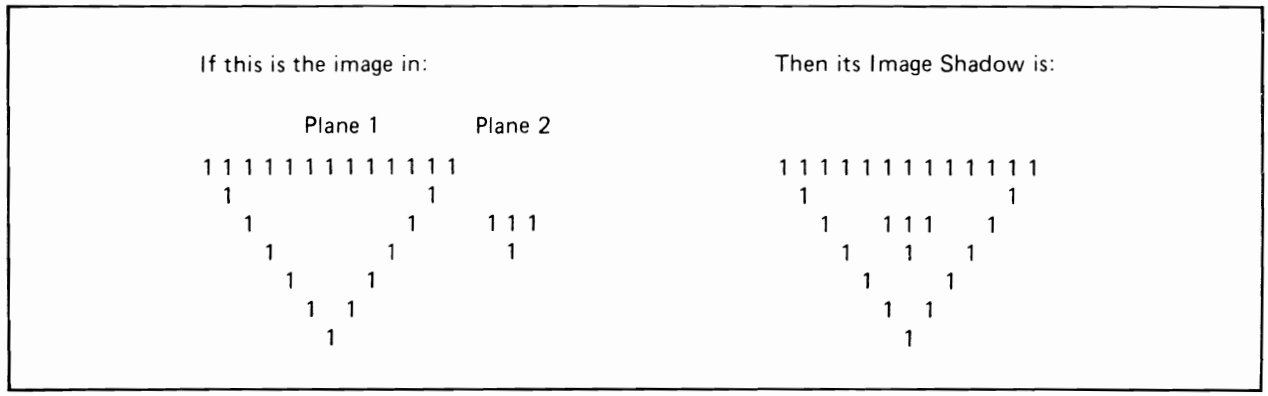


Figure 3-2: An Image and Its ImageShadow

The system uses the shadow mask along with the variable **PlaneOnOff**, discussed in the next section. Because **ImageShadow** in the **Bob** structure is a pointer to a data area containing the sprite shadow, you must provide space that the the system can use for this purpose. You must then initialize the pointer to the first location within the data area that you have set aside. You can calculate the minimum size of this area as follows:

shadow size = Height * Width

So, for example, an object 5 lines high by 32 bits wide (**VSprite** *or* Bob) requires a sprite shadow storage area of at least 5 x 2, or ten 16-bit locations. The example in the

"PlaneOnOff" section below shows how to reserve the memory for the sprite shadow and then how to tell the system where to find it.

PlaneOnOff

The variable named **PlaneOnOff** tells the system what to do with the playfields that are not "picked" (affected) by **PlanePick**. The binary bit positions for **PlaneOnOff** are the same as those for **PlanePick** (lowest bit position specifies the lowest numbered bit-plane). However, their meaning differs.

For every plane position *not* selected by **PlanePick**, parts of the non-selected plane are filled with the value shown in the corresponding position of **PlaneOnOff**. The parts that are filled are the positions where there is a 1-bit present in the sprite's image shadow.

This provides a great deal of versatility. You can use a 2-plane **VSprite** image as the source for many **Bob** images. Yet, because of the color combinations each contains, it may seem that there several different images present.

For example, assume that the data shown in the **Bob** layout above defines a 2-bit-plane Bob image that selects its colors from color registers 0, 1, 4, and 5. To initialize the **Bob** and **VSprite** structures, you need to provide the following types of statements:

```
/* data definition from example layout */
WORD BobData[]= {
      0xFFFF, 0x300C, 0x0FF0, 0x03C0, 0x0180,
      0xFFFF, 0x3FFC, 0x0C30, 0x03C0, 0x0180
};

/* reserve space for the collision mask for this Bob */
WORD BobCollision[10];

myVSprite.Width = 1;       /* sample image is 16 pixels wide (1 word) */
myVSprite.Height = 5;      /* takes 5 lines to define each image of the Bob */
myVSprite.Depth = 2;       /* only two bit-plane images are defined in BobData */

/* show the system where it can find the data image of the Bob */
myVSprite.ImageData = BobData;

/* binary = 00101, means draw into only bit-planes 0 and 2 */
myVSprite.PlanePick = 0x05;

/* binary = 00000, means for planes not picked, that is, 1, 3, and 4,
**fill those planes with 0's wherever there is a 1 in the sprite shadow mask
*/
myVSprite.PlaneOnOff = 0x00;

/* where to put collision mask */
myVSprite.CollMask = BobCollision;

/* tell the system where it can assemble a sprite shadow */
/* point to same area as CollMask */
myBob.ImageShadow = BobCollision;

/* create the sprite collision mask for this Bob's VSprite structure */
InitMasks( &myVSprite );
```

Whenever the system draws this **Bob**, it fills any position where there is a 1 in the sprite shadow with a 0 for any plane not selected by **PlanePick**. Therefore, the only binary combinations the **Bob** pixels can form are as shown below. Because of **PlanePick**, 1's can only appear at these two locations: 0 0 1 0 1.

So the color choices are limited to the following:

Color Selected	Binary Combination
color 0	0 0 0 0 0
color 1	0 0 0 0 1
color 4	0 0 1 0 0
color 5	0 0 1 0 1

These color choices fulfill the requirements specified for the example.

When you've considered and worked out the factors that influence Bob color, you're ready to specify the **Bob** position.

Specifying Bob Position

To select the position of a **Bob**, specify the y and x variables in the **VSprite** structure associated with the **Bob**. For example:

```
myVSprite.y = 100;
myVSprite.x = 100;
```

You've now defined the size, shape, color, and position of a **Bob**. You can now decide either to specify when the system draws it or to let the system choose.

Bob Priorities

This section describes the two choices you have for system priorities between **Bobs**. You can ignore the priority issue and let the system decide which **Bob** has the highest priority, or you can specify the drawing order yourself. When you specify the drawing order, you control which **Bob** the system draws *last*, and therefore, which one appears in front of other **Bobs**.

Letting the System Decide Priorities

If you want the system to decide, you set the pointers in the Bob data structure named **Before** and **After** to zero. In this case, the system draws the **Bobs** in their y,x positional order on-screen. In other words, the system draws whichever object is on-screen and is currently the highest within the drawing area (lowest y coordinate value). If two objects have the same y coordinate, then the object that has the lowest x coordinate value is drawn first.

The **Bob** drawn first has the lowest priority. The **Bob** drawn last has the highest priority because later objects overlap the objects drawn earlier.

As you use the animation system to move objects past each other on-screen, you'll notice that sometimes the objects switch priorities as they pass each other. For example, suppose you want the system to establish the priorities of the **Bobs**, and there are two **Bobs** defined in the system—myBob2, myBob3. You set the **Before** and **After** pointers as follows:

```
myBob2.Before = 0;
myBob2.After  = 0;
myBob3.Before = 0;
myBob3.After  = 0;
```

Specifying the Drawing Order

If you wish to specify the priorities, simply specify the pointers as follows. **Before** points to the **Bob** that this **Bob** should be drawn before, and **After** points to the **Bob** that this **Bob** should be drawn after. This guarantees that **Bob** objects retain their relative priorities.

For example, suppose you want to assure that myBob3 always appears in front of myBob2. You must initialize the **Before** and **After** pointers so that the system will always draw myBob3 last; that is, after myBob2.

```
myBob2.Before = &myBob3;    /* draw Bob2 before drawing Bob3 */
myBob2.After  = 0;          /* draw Bob2 after no other Bob */
myBob3.After = &myBob2;     /* draw Bob3 after drawing Bob2 */
myBob3.Before = 0;          /* draw Bob3 before no other Bob */
                            /* draw nothing in particular after this Bob */
```

If you decide to specify the **Before** and **After** pointers for any one **Bob** in a group, then you must also at least set to zero the **Before** and **After** pointers for *all* of the rest of the **Bobs** in that group.

For example, if there are 10 **Bobs**, and you only care that the system draws numbers 4, 6, and 9 in that sequence, then you must properly fill in the **Before** and **After** pointers for these three **Bobs**. If you don't care in which order the system draws the other 7 **Bobs**, you need only initialize their **Before** and **After** pointers to a value of 0 to assure correct treatment by the system.

You must properly point *all* **Before** and **After** pointers of a group to each other because the Bob which is the upper-leftmost becomes the first the system considers for drawing. The system follows the **Before** pointers until it finds one having a zero value, and draws that **Bob** first. It then draws other **Bobs** in the sequence you have specified.

In the example code sequence above, the comment "draw nothing in particular after this Bob" simply means that once the drawing sequence for this set of **Bobs** has been performed, the system still proceeds to find and draw all other **Bobs** currently linked into the GEL list. To continue the drawing operation, the system simply goes on searching the list for the next **Bob** whose **Before** pointer is 0.

Specifying Priority Between Bobs and VSprites

See "VSprite and Bob Topics" below for details.

Saving the Playfield Display

Once the system has drawn the **Bobs**, they become part of the playfield segment of the display. The image of a **Bob** overlays part of the background area. To move a **Bob** from one place to another, you must tell the system to save the background before it draws the **Bob**, and to restore the background to its original condition when it moves the **Bob**.

A variable called **sprFlag** in the **VSprite** structure contains a flag called SAVEBACK. To cause the system to save and restore the background for that **Bob**, set the SAVEBACK flag to 1.

In addition to the **sprFlag** variable, you must also tell the system where it can put this saved background area. For this, you use the **SaveBuffer** variable. For example, if the **Bob** is 48 pixels wide and 20 lines high, and the system is drawing it into a 5 bit-plane playfield, you must allocate space for storing the following:

(48 pixels/16 pixels per word) * (20 lines) * (5 bit-planes) = 300 words

To allocate this space, use the graphics function **AllocRaster()**. When you use **AllocRaster()** for this purpose, you can specify the area size in bits, so it may well be the most convenient way to reserve the space you need. For example:

```
myBob.SaveBuffer = AllocRaster(48,20 * 5);
      /* save space to store 48
      bits times 20 words times 5 bit-planes */
```

Note that the **AllocRaster()** function rounds the width value up to the next integer multiple of 16 bits.

Using Bob Flags

The following sections describe the **Bob** flags. Some of these are in the **VSprite** structure associated with the Bob; others are in the **Bob** structure itself. The description of each flag tells the structure in which the flag is located.

VSPRITE Flag

If you are using the **VSprite** structure to describe a **Bob**, set VSPRITE to zero.

The VSPRITE flag is located in the **VSprite** structure.

SAVEBACK Flag

If you want the GEL routines to save the background before the **Bob** is drawn and to restore the background after the **Bob** is removed, set the SAVEBACK (for "save the background") flag in **VSprite** structure to 1.

If you set this flag, you must have allocated the buffer named **SaveBuffer**.

OVERLAY Flag

If the system should use the sprite shadow mask when it draws the **Bob** into the background, set the OVERLAY flag in the **VSprite** structure to 1. If this flag is set, it means that the background original colors show through in any section where there are 0-bits in the sprite shadow mask. Essentially, then, those 0-bits define areas of the **Bob** that are "transparent".

If you set the OVERLAY bit to a value of 0, then the system uses the *entire rectangle* of words that define the **Bob** image and uses its contents to *replace* the playfield area at the specified y,x coordinates.

If you set this flag, you must have allocated space for the **ImageShadow** and must also have initialized the **ImageShadow**.

See the section above called "Sprite Shadow Mask" for details on the shadow mask.

GELGONE Flag

The system sets this flag in the **VSprite** structure to indicate when the **Bob** has been moved to y,x coordinates entirely outside of the "clipping region".

When an object crosses over certain specified boundaries in the drawing area, the system does not draw all of the object into the background but "clips" (truncates) it to those limits. At the time of this writing, the variables named **topmost**, **bottommost**, **leftmost** and **rightmost** define the minimum and maximum y,x coordinates of this clipping region.

When the system sets the GELGONE flag to a 1, you know that the object has passed entirely beyond those limits and that the system will not draw any part of the object into the drawing area. On the basis of that information, you may decide that the object need no longer be part of the GEL list and may decide to remove it to speed up the consideration of other objects.

SAVEBOB Flag

To tell the system not to erase the old image of the **Bob** when the **Bob** is moved, set the SAVEBOB flag in the **Bob** structure to 1. This lets you use the **Bob** like a paintbrush if you wish. It has the opposite effect of SAVEBACK.

NOTE: It takes longer to preserve and restore the raster image than simply to draw a new **Bob** image wherever required.

BOBISCOMP Flag

If this **Bob** is part of an **AnimComp**, set the BOBISCOMP flag in the **Bob** structure to 1. If the flag is a 1, you must also initialize the pointer named **BobComp**. Otherwise, the system ignores the pointer, and it may be left alone. See "Animation Structures and Controls" for a discussion of **AnimComps**.

BWAITING Flag

When a **Bob** is waiting to be drawn, the system sets the BWAITING flag in the **Bob** structure to 1. This only occurs if the system has found a **Before** pointer in this **Bob**'s structure that points to another **Bob**. Thus, the system flag BWAITING provides current draw-status

to the system. Currently, the system clears this flag on return from each call to **DrawGList()**.

BDRAWN Flag

The BDRAWN system status flag in the **Bob** structure tells the system that this **Bob** has already been drawn. Therefore, in the process of examining the various **Before** and **After** flags, the drawing routines may determine the drawing sequence. Currently, the system clears this flag on return from each call to **DrawGList()**.

BOBSAWAY Flag

To initiate the removal of a **Bob** during the next call to **DrawGList()**, set BOBSAWAY to 1. Either you or the system may set this **Bob** structure system flag. The system restores the background where it has last drawn the **Bob**. The system will unlink the **Bob** from the system GEL list the next time **DrawGList()** is called unless you are using double-buffering. In that case, the **Bob** will not be unlinked and completely removed until two calls to **DrawGList()** have occurred and the **Bob** has been removed from both buffers.

BOBNIX Flag

When a **Bob** has been completely removed, the system sets the BOBNIX flag to 1 on return from **DrawGList()**. In other words, when the background area has been fully restored and the **Bob** has been removed from the GEL list, this flag in the removed **Bob** is set to a 1. BOBNIX is significant in cases where you use double-buffering because once you ask that a **Bob** be removed, the system must remove it from the active drawing buffer and from the display buffer. Once BOBNIX has been set for a double-buffered **Bob**, it has been removed from both buffers and you are free to re-use it or de-allocate it.

This flag is in the **Bob** structure.

SAVEPRESERVE Flag

The SAVEPRESERVE flag is a double-buffer version of the SAVEBACK flag. If you are using double-buffering and wish to save and restore the background, you set SAVEBACK to 1. SAVEPRESERVE is used by the system to indicate whether the Bob in the "other" buffer has been restored; it is for system use only.

Adding a Bob

To add a **Bob** to the system GEL list (the same list you created for **VSprites** using **InitGels()**), you use the **AddBob()** routine. It is advisable that you initialize the different variables you plan to use within the **Bob** structure before you ask that the system add this **Bob** to the list.

For example:

```
struct GelsInfo myGelsInfo;
struct VSprite dummySpriteA, dummySpriteB;
struct Bob myBob;

/* done ONCE, for this GelsInfo */
InitGels( &dummySpriteA, &dummySpriteB, &myGelsInfo );

/* here initialize the Bob variables */
AddBob( &myBob, &rastport );
```

Removing a Bob

You have two methods for removing a **Bob**. This section describes the system routine for each.

The first method uses the **RemBob()** routine. You call this routine as follows:

```
RemBob ( &myBob, &rastport );
```

RemBob() causes the system to remove the **Bob** during the next call to **DrawGList()** (or two calls to **DrawGList()** if the system is double-buffered). **RemBob()** asks the system to remove the **Bob** "at its next convenience".

The second method is to call the **RemIBob()** routine. For example:

```
RemIBob ( &myBob, &rastport, &viewport );
```

RemIBob() tells the system "remove this **Bob** immediately!". It causes the system to erase the **Bob** from the drawing area and causes the immediate erasure of any other **Bob** which had been drawn subsequent to this one. The system then unlinks the **Bob** from the system GEL list. To redraw the **Bobs** that were drawn on top of the one just removed, you must make another call to **DrawGList()**.

Now that you've seen how to add and remove **Bobs** from the system GEL list, you'll need to learn how to organize that list.

Getting the List of Bobs in Order

As with **VSprites**, the list of GELS must be in the proper y,x sorted order from top of screen to bottom and from left to right. The system uses the position information to decide drawing sequences if you have not specified otherwise by using the **Before** and **After** pointers. You must therefore assure that the GEL list is sorted before you ask the system to display the **Bobs**.

To sort the GEL list, you call **SortGList()**. For example:

```
SortGList( &rastport );
```

Once the list is sorted, it's time to instruct the system to display the **Bobs**.

Displaying Bobs

This section provides the typical sequence of operations for drawing the **Bobs** on-screen. It is very similar to that shown for **VSprites** as both **Bobs** and **VSprites** are GELS and are part of the same list of controlled objects.

Specifically, the system automatically synchronizes the drawing routines to the display beam and may not require that the display be turned off during the update. If large **Bobs** or many **Bobs** are created, you may be interested in double-buffering. See the section called "Double-Buffering" in this chapter for details.

When you call **DrawGList()**, the system *actually draws* any **Bobs** on this list into the area you have specified. The system saves the backgrounds if you have provided for the save and then performs the drawing sequence in the order you requested. To initiate this drawing, call **DrawGList()**. For example:

```
struct RastPort *rp;
struct ViewPort *vp;
    ...
DrawGList(rp, vp);     /* draw the elements */
```

Now that you know how to instruct the system to draw the **Bobs** into the area you've specified, you may want to try out several ways to change the characteristics of those **Bobs**.

Changing Bobs

You can change the following characteristics of **Bobs**:

- To change their appearance, change the pointer to the **ImageData** in the associated **VSprite** structure. Note that the change in the **ImageData** pointer also requires a change in the **ImageShadow** or a recalculation of the object mask, using **InitMasks()**.

- To change their color choices, change their **PlanePick** and/or **PlaneOnOff** values, as well as the depth parameters if the sprite image has multiple planes defined.

- To change the location in the drawing area, change the y,x values in the associated **VSprite** structure.

- To change the object priorities, change the drawing sequence by altering the **Before** and **After** flags in the **Bob** structures.

- To change the **Bob** into a paintbrush, set the SAVEBOB flag to a 1 in the **Bob** structure.

NOTE: Neither these nor other changes actually happen until you call **SortGList()** and then **DrawGList()**.

Double-Buffering

Double-buffering is the technique of supplying two different memory areas in which the drawing routines may create images. The system displays one memory space while you are drawing into the other area. This assures that you never see any display fields on-screen that consist partly of old material and partly of new material.

The system animation routines use an extension that you establish to the **Bob** structure. Also, if you do not not care to use double-buffering, you need not tie up precious memory resources for unneeded variable storage space.

To find whether a **Bob** is to be double-buffered, the system examines the pointer named **DBuffer** in the **Bob** structure. If this pointer has a value of 0, the system does not use double-buffering for this **Bob**.

NOTE: If you do *not* wish to use double-buffering, you must initialize the **DBuffer** pointer to zero. For example:

```
myBob.DBuffer = 0;    /* do this if this Bob
                         is NOT double-buffered */
```

The next section discusses several other variables that you must describe if you want to use double-buffering. *NOTE*: if any of the **Bobs** are double-buffered, then *all* of them must be double-buffered.

Variables Used in Double-Buffering

To use double-buffering for a given **Bob**, you must provide a data packet for the system to store some of the variables it needs to handle double-buffering. This data packet is a structure named **DBufPacket** which consists of the following variables:

BufY, BufX
System variables that let the system keep track of where the object was located "last screen" (as compared to the **Bob** structure variables called **oldY** and **oldX** that tell where the object was two-screens ago). **BufY** and **BufX** provide for correct restoration of the background within the currently active drawing buffer.

BufPath
System variable related to the drawing order used to draw this **Bob** into the background. **BufPath** assures that the system restores the backgrounds in the correct sequence; it relates to the system variables **DrawPath** and **ClearPath** (found in this Bob's **VSprite** structure).

BufBuffer
You must set this field to point to a buffer as big as this Bob's **SaveBuffer** to allocate separate space for buffering the background on which you are drawing the **Bob**. This buffer is used to store the background for later restoration when the system moves the object.

The next section shows how to pull all these variables together to make a double-buffered **Bob**.

Creating a Double-Buffered Bob

To create a double-buffered **Bob**, you must initialize all of the normal **Bob** variables and pointers and execute a code sequence similar to the following:

```
struct DBufPacket myDBufPacket;

/* allocate a DBufPacket for myBob */
  ...
  ...
/* same size as previous example in
**"Saving the Playfield Display"
*/
myDBufPacket.BufBuffer = AllocRaster( 48, 20 * 5 );

/* tell Bob about its double buff status */
myBob.DBuffer = myDBufPacket;
```

The next section summarizes the steps you take to define, move, and display a **Bob**.

Bob Operations Summary

- Define a **RastPort** structure for the drawing routine to use.
- Initialize the GEL system (call **InitGels()**) for this **RastPort**. You only need to do this once.
- Create and link a **Bob** and a **VSprite** structure.
- Define the following **Bob** parameters:
  - Height
  - Width
  - Depth
  - Position
  - Where to find **ImageData** data
  - Which planes to pick for writing this Bob
  - How to treat the planes not picked

- **VSprite** structure flags to show that this is a Bob
- Space for the sprite shadow
- Pointer to a **DBufPacket** if you want to use double-buffering (Otherwise, make this pointer a NULL (0) value.)

o Call **InitMasks()** to create the sprite shadow.

o Add the **Bob** to the GEL list.

o Change the **Bob** appearance by

- Changing the pointer to **ImageData**
- Changing its height, width or depth.

o Change the **Bob** colors by

- Changing the playfield color set
- Changing **PlanePick** and **PlaneOnOff**.

o Move the **Bob** by defining a new y,x position.

o Display the **Bob** by calling:

- **SortGList()**;
- **DrawGList()**;

Now that you've mastered the basics of handling **VSprites** and **Bobs**, you may want to find out about some of the interactions between the two and how to cope with these interactions. Or, you may want to skip these advanced topics and read about software collisions, clipping, and adding new features in "VSprite and Bob Topics" below.

Bob Advanced Topics

This section provides information pertaining to more advanced uses of the **Bob** system.

How Bob Colors are Controlled

Bobs do not use the **SprColor** pointer. To determine the color of a **Bob**, you use the existing colors in the 32-entry color table. The lower 16 of the 32 possible color selections (registers 0-15) are always dedicated to playfield color selections, providing 16 unique colors for the **Bobs**, since they are playfield objects.

However, the playfields and the **VSprites** share the upper 16 of the 32 color entries (registers 16-31). If you are using 5 bit-planes to display the **Bobs**, then any **Bob** with a pixel whose color value exceeds 15 may change color if the virtual sprites are running at the same time.

NOTE: This also applies to *any* static part of the display area (the playfield), whether a **Bob** or simply part of the background display, for which a 5 or 6 bit-plane image is used if the color number for a specific pixel exceeds the value of 15.

To explain further, the virtual sprite routines, notably **SortGList()** and **DrawGList()**, work together to decide which real sprite will be used at any point on-screen. **DrawGList()** makes up a Copper instruction list to change the contents of the upper 16 color registers, perhaps several times within a single display field. Therefore, depending on where a **Bob** image may be on-screen relative to a virtual sprite, and depending on its color content, a **Bob** may take on different colors (perhaps even within only a part of its body).

To minimize color interactions between **Bobs** and virtual sprites, take the appropriate precautions:

- Limit the background to 4 bit-planes or less and thus limit the **Bob** color choices to 16 or less.

- Use 5 bit-planes, but specify **Bob** colors or background colors from the colors 0 through 15, or 16, 20, 24, or 28 only. Colors 16, 20, 24, and 28 are used neither by real sprites nor by virtual sprites and are treated as transparent areas. Therefore, if you use only these colors for **Bobs**, the simultaneous use of virtual sprites will not affect the **Bob** or background colors.

- Use **sprRsrvd** to "fence-off" certain sprite pairs, so you can also use their colors for **Bobs**.

Now that you've seen some of the ways to handle **VSprite** and **Bob** color interaction, you may want to read about some topics they share.

3.5. VSPRITE AND BOB TOPICS

This section explores topics common to both **VSprites** and **Bobs**. It includes a discussion of software collision detection, ways of expanding the **VSprite** and **Bob** data structures, and a way of expanding the scope of the **Bob** and **VSprite** structures.

Detecting GEL Collisions

To detect collisions between graphics elements, you use the **DoCollision()** routine. **DoCollision()** determines if there are any pixels of one graphics element currently touching those of another graphics element, or if any of the graphics elements have passed outside of specified screen boundaries.

Whenever there is a collision, the system performs one of 16 possible collision routines. The addresses of the collision routines are kept in a table called the collision handler table. **DoCollision()** examines the **HitMask** and **MeMask** of each of the **VSprite** structures in the GEL list, and determines if there is a collision between any two GELS. It then calls the collision handler routine at the table position corresponding to the bits in the **HitMask** and **MeMask**, as outlined below.

NOTE: The current form of these routines does *not* use the built-in *hardware* collision detection. You may, if you wish, reserve one or more sprites for your own use and move them using your own routines. When specific sprites have been reserved for your own use, you may choose to use the hardware collision detection to sense collisions between your own objects and other on-screen elements. See the *Amiga Hardware Reference Manual* for information about hardware collision detection.

Default Kinds of Collisions

The two kinds of software collision sensing built into the collision routines are:

- boundary hits
- GEL to GEL hits

You can set up as many as 16 different kinds of collisions using the **VSprite** structure **MeMask** and **HitMask**. When you call a collision routine, you give it certain kinds of information about the colliding elements, as described in the next two sections.

Boundary Hits

During the operation of the **DoCollision()** routines, if you have enabled boundary collisions for a GEL and it crosses a boundary, the system calls the boundary hit routine you have defined. Note that the system calls the routine once for each GEL that has gone outside of the boundary.

The system will call your routine with the following two arguments:

- A pointer to the **VSprite** structure of the GEL that hit the boundary
- A flag word containing one to four bits set, representing top, bottom, left and right boundaries, telling you which one or more boundaries it has hit or exceeded. To test these, use the names TOPHIT, BOTTOMHIT, LEFTHIT and RIGHTHIT.

GEL to GEL Collisions

If, instead of a GEL to boundary collision, **DoCollision()** senses a GEL to GEL collision, the system calls your collision routine with the following two parameters. They will be different from those in the GEL to boundary collision.

- Address of the **VSprite** structure that defines the uppermost (or leftmost if y coordinates are identical) object of a colliding pair, and
- Address of the **VSprite** structure that defines the lowermost (or rightmost if y coordinates are identical) object of a colliding pair.

Handling Multiple Collisions

When multiple elements collide within the same display field, the following set of sequential calls to the collision routines occurs:

- The system issues each call in a sorted order, for GELs starting at the upper left-hand corner of the screen and proceeding to the right and down the screen.
- For any two colliding graphics elements, the system issues only one call to the collision routine for this pair. The system bases the collision call on the object that is the highest and leftmost of the pair on-screen.

Preparing for Collision Detection

Before you can use the system to detect collisions between GELS, you must initialize the table of collision detection routines. This table points to the actual routines that you will use for the various collision types you have defined. Also, you must prepare certain variables and pointers within the **VSprite** structure: **BorderLine**, **CollMask**, **HitMask**, and **MeMask**.

Building a Table of Collision Routines

To add to or change the table entries for the collision routines, call the **SetCollision()** routine. The syntax this routine follows.

```
SetCollision( num, routine, Ginfo)
```

where:

num
is the collision vector number.

routine
is a pointer to the user collision routine.

GInfo
is a pointer to a **GelsInfo** structure.

When the **View** structure is first initialized, the system sets all of the values of the collision routine pointers to zero. You must initialize those table entries corresponding to the **HitMask** and **MeMask** bits that you have set. Only those can cause the system to call the collision routines.

You must also allocate a table, pointed to by **GelsInfo**, for vectors. The table needs to be only as large as the number of bits for which you wish to provide collision processing. For example:

```
VOID myCollisionRoutine( GELM, GELN )  /* sample collision routine */
struct VSprite *GELM;
struct VSprite *GELN;
{
        printf("GEL at %lx has hit GEL at %lx", (long)GELM, (long)GELN);
}

        /* sample initialization */
        ReadyGels(gelsinfo, rastport);      /* use exec_support function */
        SetCollision( 15, myCollisionRoutine, &gelsinfo );
```

Collision Mask

The variable named **CollMask** is a pointer to a memory area that you have reserved for holding the collision mask of a GEL. A collision mask is usually the same as the shadow mask of the GEL, formed from a *logical-or* combination of all 1-bits in all planes of the image. Figure 3-3 shows an example collision mask.

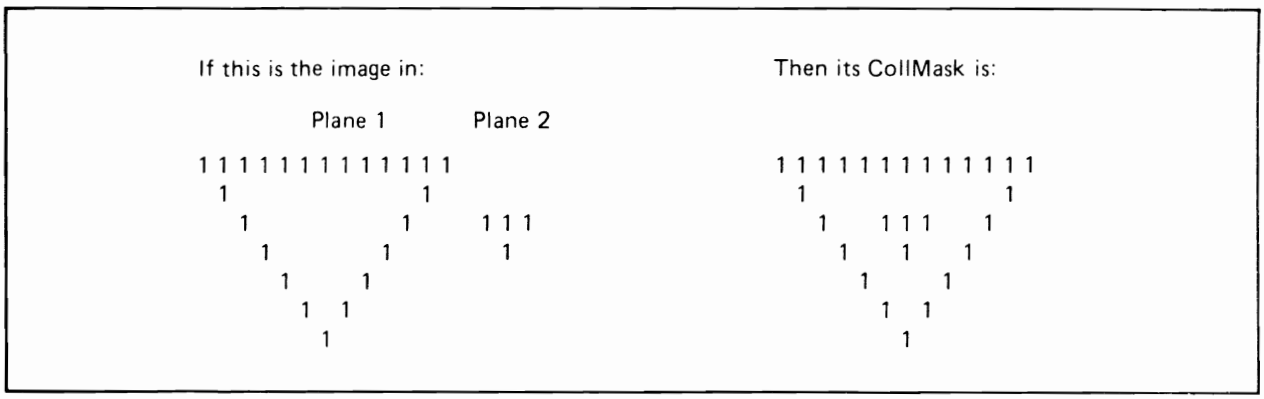


Figure 3-3: A Collision Mask

You normally use this collision mask to control drawing of the object and to define essentially the positions where there is an image-bit present. After you have defined the collision mask through the routine **InitMasks()**, you may specify that the system is to store both the shadow mask and the collision mask in the same location.

For example, here are typical program statements to reserve an area for the sprite shadow, initialize the pointer correctly, and then specify that the system uses the same mask for collisions (assumes a 2-word wide, 4-line high image):

```
/* reserve 8 16-bit locations for sprite shadow to be stored into by the system. */
WORD myShadowData[8];
myVSprite.ImageShadow = myShadowData; /* and point to it */
myVSprite.CollMask  = myShadowData;  /* collision mask is same as shadow */
```

As an alternative, for certain game-oriented applications, you may design certain objects with sensitive regions and non-sensitive regions. For example, if you have designed a ship with some kind of superstructure, a missile that strikes in the superstructure perhaps should pass through without registering a collision, but a missile that strikes into the heart of the ship should cause it to explode. The collision mask gives you the versatility to perform this kind of operation.

Suppose you have an object with an outer layer that is to be insensitive and an inner core that you want to register collisions for the overall object. Using the same ship with a superstructure, refer to Figure 3-4.

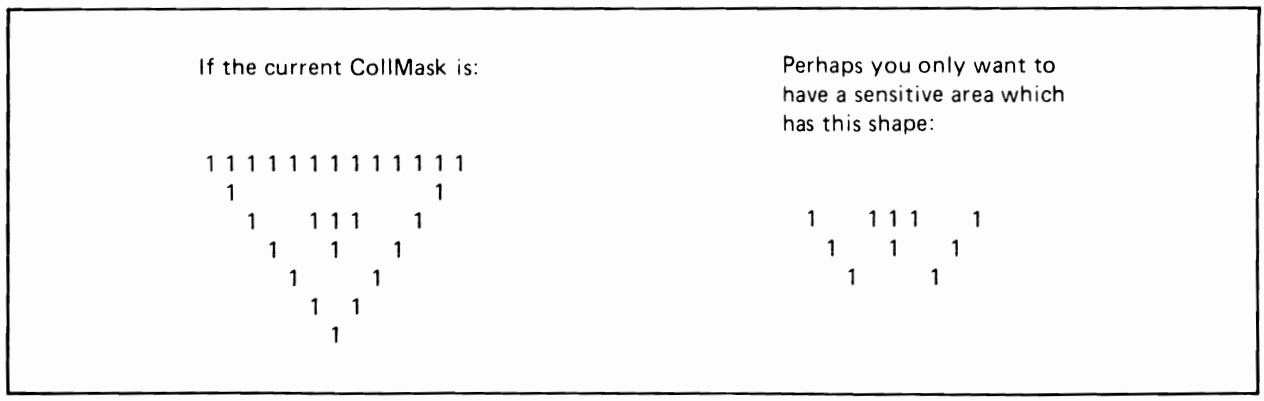


Figure 3-4: Shadow Mask for a Sensitive Area

Therefore, you would define your own shadow mask with 1-bits in the appropriate positions to define the desired sensitive area.

BorderLine Image

For fast collision detection, the system uses the pointer named **BorderLine**. **BorderLine** specifies the location of the horizontal *logical-or* combination of all of the bits of the object. It may be compared to taking the whole object and squashing it down into one single horizontal line. Here is a sample of an object and its **BorderLine** image:

OBJECT

001100001100
000110011000
000011110000
000110011000
001100001100

BORDERLINE IMAGE

001111111100

The borderline image establishes a single set of words (represented by the collision mask) that have 1-bits at the outermost edges of the object. Using this squashed image, the system can quickly determine if the image is touching the left or rightmost boundary of the drawing area.

To establish the borderline data, you make a system call to **InitMasks()**. Before calling **InitMasks()**, you provide the system with a place to store the image it creates. The size of the data area you reserve must be at least as large as the image is wide.

In other words, if it takes three 16-bit words to hold the width of a GEL, then you must reserve three words for the borderline image. For example:

```
/* reserve some space for the border image to be stored for this Bob */
WORD myBorderLineData[3];

/* tell the system where to put the BorderLine image it will form */
myVSprite.BorderLine = myBorderLineData;
```

NOTE: Both **Bobs** and **VSprites** participate in the software-collision detection.

The next section tells how to turn on the software-collision detection independently for each GEL.

Software Collision Detect Control Variables

You can enable or disable software collision detection for each GEL independently. In addition, any time the system senses a collision, you can specify which routines, out of 16 possible collision routines, you wish to have automatically executed. The **HitMask** and **MeMask** variables in the **VSprite** structure let you specify the relationships between different GELS.

By specifying the bits in these masks, you can control how and when the system senses collisions between objects. The collision testing routine, in addition to sensing an overlap between objects, also uses these masks to determine which routine(s) (if any) the system will call when a collision occurs.

When the system determines a collision, it *and*s the **HitMask** of the upper-leftmost object in the colliding pair with the **MeMask** of the lower-rightmost object of the pair. The bits that are 1's after the *and* operation choose which of the 16 possible collision routines to perform.

- If the collision is with the boundary, bit 0 is a 1 and the system calls the collision handling routine number 0. You assign bit 0 to the condition called "boundary hit". The system uses the flag called BORDERHIT to indicate that an object has landed on or moved beyond the outermost bounds of the drawing area (the edge of the clipping region).

- If you set any one of the other bits (1 to 15), then the system calls the collision handling routine corresponding to the set bit.

If more than one bit is set in both masks, the system calls the vector corresponding to the rightmost bit.

Using HitMask and MeMask

This section provides an example of the use of the **HitMask** and **MeMask** to define a new form of collision detection.

Suppose there are two classes of objects that you wish to control on-screen: ENEMYTANK and MYMISSILE. Objects of class ENEMYTANK should be able to pass across each other without registering any collisions. Objects of class MYMISSILE should also be able to pass across each other without collisions. However, when MYMISSILE collides with ENEMYTANK, or ENEMYTANK collides with MYMISSILE, then the system should process a collision routine.

Choose a pair of collision detect bits not yet assigned within **MeMask**, one to represent ENEMYTANK, the other to represent MYMISSILE. You will use the same two bits in the corresponding **HitMask**.

	MeMask	HitMask	
Bit #	2 1	2 1	
GEL #1	0 1	1 0	ENEMYTANK
GEL #2	0 1	1 0	ENEMYTANK
GEL #3	1 0	0 1	MYMISSILE

In the example, bit 1 represents ENEMYTANK objects. In the **MeMask**, bit 1 is a 1 for GEL #1 and says "I am an ENEMYTANK". Bit 2 is a zero says this object is *not* a MYMISSILE object.

In bit 1 of the **HitMask** of GEL #1, the 0-bit there says, "I will not register collisions with other ENEMYTANK objects." However, the 1-bit in Bit 2 says, "I *will* register collisions with MYMISSILE objects".

Thus when a call to **DoCollision()** occurs, for any objects that appear to be colliding, the system *and*s the **MeMask** of one object with the **HitMask** of the other object. If there are non-zero bits present, the system will call one (or more) of your collision routines.

In this example, suppose that the system senses a collision between ENEMYTANK #1 and ENEMYTANK #2. Suppose also that ENEMYTANK #1 is the top/leftmost object of the pair. Here is the way that the collision testing routine performs the test to see if the system will call any collision handling routines:

Bit #	2	1
ENEMYTANK #1 MeMask	0	1
ENEMYTANK #2 HitMask	1	0
Result of *and*	0	0

Therefore, the system doesn't call a collision routine.

Suppose that **DoCollision()** finds an overlap between ENEMYTANK #1 and MYMISSILE, and MYMISSILE is the top/leftmost of the pair:

Bit #	2	1
MYMISSILE #1 MeMask	1	0
ENEMYTANK #2 HitMask	1	0
Result of *and*	1	0

Therefore, the system calls the collision routine at position 2 in the table of collision-handling routines.

Bob/VSprite Collision Boundaries Within a RastPort

To specify a region within the **RastPort** (drawing area) that the system will use to define the outermost limits of the GEL boundaries, you use the following variables: **topmost**, **bottommost**, **leftmost**, and **rightmost**. The **DoCollision()** routine tests these boundaries when determining collisions within this **RastPort**.

Here is a typical program segment that assigns the variables correctly. It assumes that you already have a **RastPort** structure named myRastPort.

```
myRastPort->GelsInfo->topmost = 50;
myRastPort->GelsInfo->bottommost = 100;
myRastPort->GelsInfo->leftmost = 80;
myRastPort->GelsInfo->rightmost = 240;
```

The current release of the system software makes use of the clipping rectangle feature of the **RastPorts** to create clipping to the **RastPort**'s limits. However, you may base the "boundary collision" limits for this **RastPort** on the variables shown here.

Adding New Features to Bob/VSprite Data Structures

This section describes how to expand the size and scope of the **VSprite** or **Bob** data structures. In the definition for the **VSprite** and the **Bob** structures, there is an item called **UserExt** at the end of the structure. If you want to add something to these structures (specifically, a user extension), you simply specify that the **UserExt** variable is composed of a specific type.

Why would you want to add things to the structure? When the **DoCollision()** routine passes control to your collision processing function, you may wish to change some variable associated with the GEL. The example below places speed and acceleration figures with each GEL. When you perform the collision routine, it exchanges these values between the two colliding objects. The system uses additional routines during the no-collision times to calculate the new positions for the objects.

You could define a structure similar to the following:

```
struct myInfo {
        short xvelocity;
        short yvelocity;
        short xaccel;
        short yaccel;
};
```

that you want to have associated with each of the GELS. These variables are, for example, *your* user extensions.

You would also provide the following line:

For **VSprites**:

```
#define VUserStuff struct myInfo
```

For **Bobs**:

```
#define BUserStuff struct myInfo
```

For **AnimObs**:

```
#define AUserStuff struct myInfo
```

When the system is compiling the gels.h file with your program, the compiler substitutes struct myInfo everywhere that **UserExt** is used in the header. The structure is thereby customized to include the items you wish to associate with it.

NOTE: The header files include the following UserStuff variables for **VSprites**, **Bobs**, and **AnimObs**:

VSprites:	**VUserStuff**
Bobs:	**BUserStuff**
AnimObs:	**AUserStuff**

Once you've added **Bobs** and **VSprites** and worked with the collision routines and data structure extensions, you'll probably want to perform animations. The following sections outline the system animation support for **Bobs** only.

3.6. ANIMATION STRUCTURES AND CONTROLS

The Amiga currently provides animation support for **Bobs**.

In the section called "Bob Priorities" you learned how to control the priorities of **Bobs** with respect to each other by specifying the drawing sequence. The following sections explain how to link objects and how to specify an animation completely by linking different views of objects into a sequence.

To perform animation, an artist produces a series of drawings. Each drawing differs from the preceding one so that when they are sequenced, the object appears to move naturally. An animation in the Amiga consists of:

- a linked list of the components of the animation object, and
- each component as a linked list of the different drawings in its sequence.

To perform the actual animation, you make a call to a system routine called **Animate()**. When you call **Animate()**, the software follows all of your animation instructions and "moves" the objects accordingly. When you next call **DrawGList()**, the system draws all objects in the position caused by your calls to **Animate()**. Essentially, **Animate()** simply manipulates a set of instructions in a set of object lists. Only when the system draws the objects are your instructions visually displayed.

As you recall, the system draws the currently sorted objects from its GELS list.

General Characteristics of the Animation System

The animation system lets you define a series of **Bobs**, which it then links into an overall object. The combined object consists of one or more **Bobs** that comprise the overall object and additional **Bobs** that comprise alternate appearances (animation sequences) for the various component parts.

You specify:

- the initial appearance of an overall object by defining **Bobs** as its components.
- alternate views of various components by defining additional bobs.
- the drawing precedence for the initial appearance of the object among the **Bobs** that comprise the initial appearance.

The animation system:

- moves all linked objects simultaneously.
- maintains inter-object prioritization.
- sequences alternate views to provide animation through user-specified timing variables.

Keeping Track of Graphic Objects

The section called "Getting the List of Bobs in Order", described how the system maintains a list of **Bobs** to draw on-screen according to your instructions. The animation system selectively adds items to and removes items from this list of screen objects during the **Animate()** routine. The next time you call **DrawGList()**, the system will draw the current **Bobs** in the list into the selected **RastPort**.

The next few sections define the two kinds of animation objects.

Classes of Animation Objects

You have two classes of objects to consider: **AnimObs** and **AnimComps**. The AnimOb is the primary animation object. It is this object whose position you are specifying with respect to the coordinates of the drawing area. Actually, an AnimOb itself contains no imagery. It is merely the top-level data structure which organizes the components that it manages and specifies a position relative to which everything else is drawn. The **AnimComp**, on the other hand, is an animation component — for example, an arm, leg or head — of an animation object. The animation object consist of animation components that you specify.

To define an **AnimOb**, you specify several characteristics of the primary animation object, including the following:

- The initial position of this object.
- Its velocity and acceleration in the X and Y directions.
- How many calls to **DrawGList()** you have made while this object has been active.
- A pointer to a special animation routine related to this object (if desired).
- A pointer to the first of its animation components.
- Your own extensions to this structure, if desired.

Positions of Animation Objects

The next two sections tell how to specify the initial position of an **AnimOb** and its **AnimComp**.

Position of an AnimOb

To specify a registration point within the drawing area (the **RastPort**) for all components, you use the variables **AnX** and **AnY** in the **AnimOb** structure. Figure 3-5 illustrates that each component has its own offset from the object's registration point.

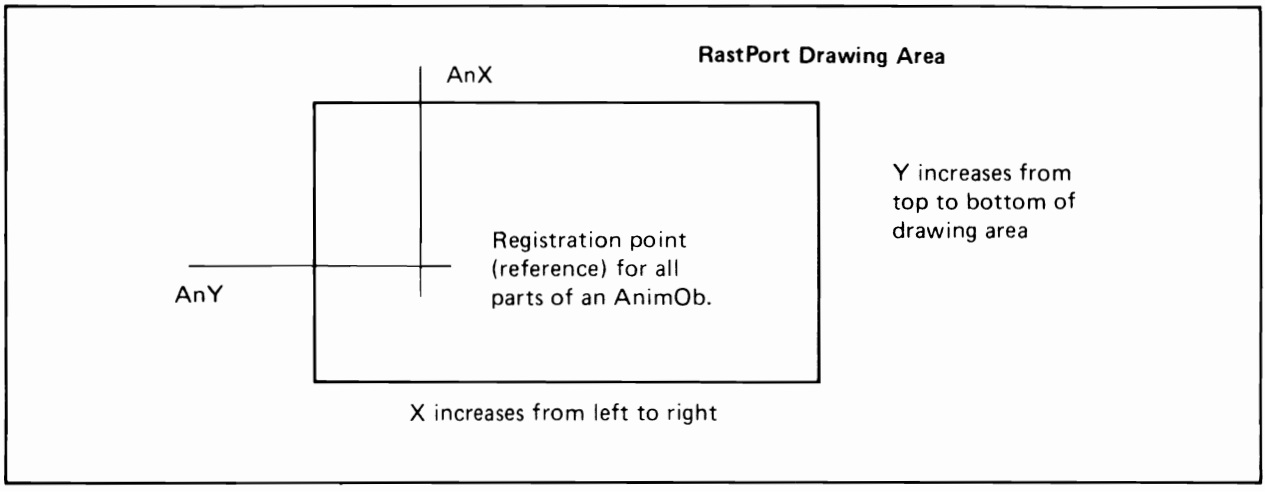


Figure 3-5: Specifying an AnimOb Position

Position of an AnimComp

To specify where the component is to be located *relative to the position of the registration point*, you use variables in the **AnimComp** structure. When you move the animation object, all of the component parts of this animation object move with it as illustrated in Figure 3-6.

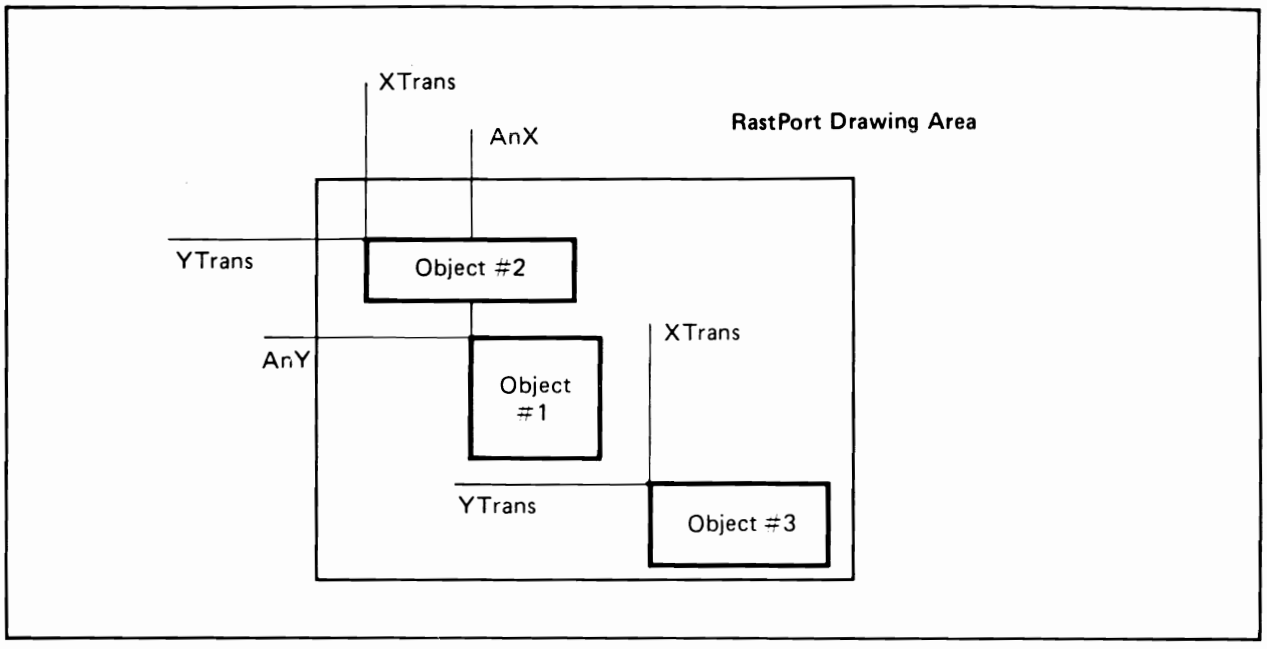


Figure 3-6: Specifying an AnimComp Position

To specify the relative placement of a component with respect to the registration point of the **AnimOb**, you assign the values of **XTrans** and **YTrans** in the **AnimComp** structure. These values can be either positive (as shown for object #3), or negative (as shown for object #2) or zero (as shown for component #1) in Figure 3-6 above.

Now that the system knows the position of the objects and components you wish to animate, you can tell the system how to animate them. The following sections describe the animation choices provided for you by the system.

Animation Types

The system software allows two forms of animation:

- sequenced drawing
- motion control

Sequenced Drawing

In sequenced drawing, an artist produces a sequence of views of an object, where each view is a modification of a preceding view. To produce apparent motion of the object, the artist draws each new view of an object at a position somewhat farther from a common reference point than the preceding view.

If an animation is to be continuous, based on a repeating sequence, then the last drawing in the series should allow the first drawing in the series to be the next-in-line, creating a continuity of motion. Figure 3-7 shows four out of a sequence of drawings that could use this technique for animation. (The other intermediate steps are not shown.)

As you will notice, each of the drawings, reading from right to left, is a little closer to its registration point (the reference point). The upper level of the figure shows the figures individually. The lower level shows the figures overlaid, demonstrating that smooth motion would be possible. To the left of the overlaid figures is a second set, drawn in gray, representing the reinitialization of the sequence of drawings again, beginning with number one.

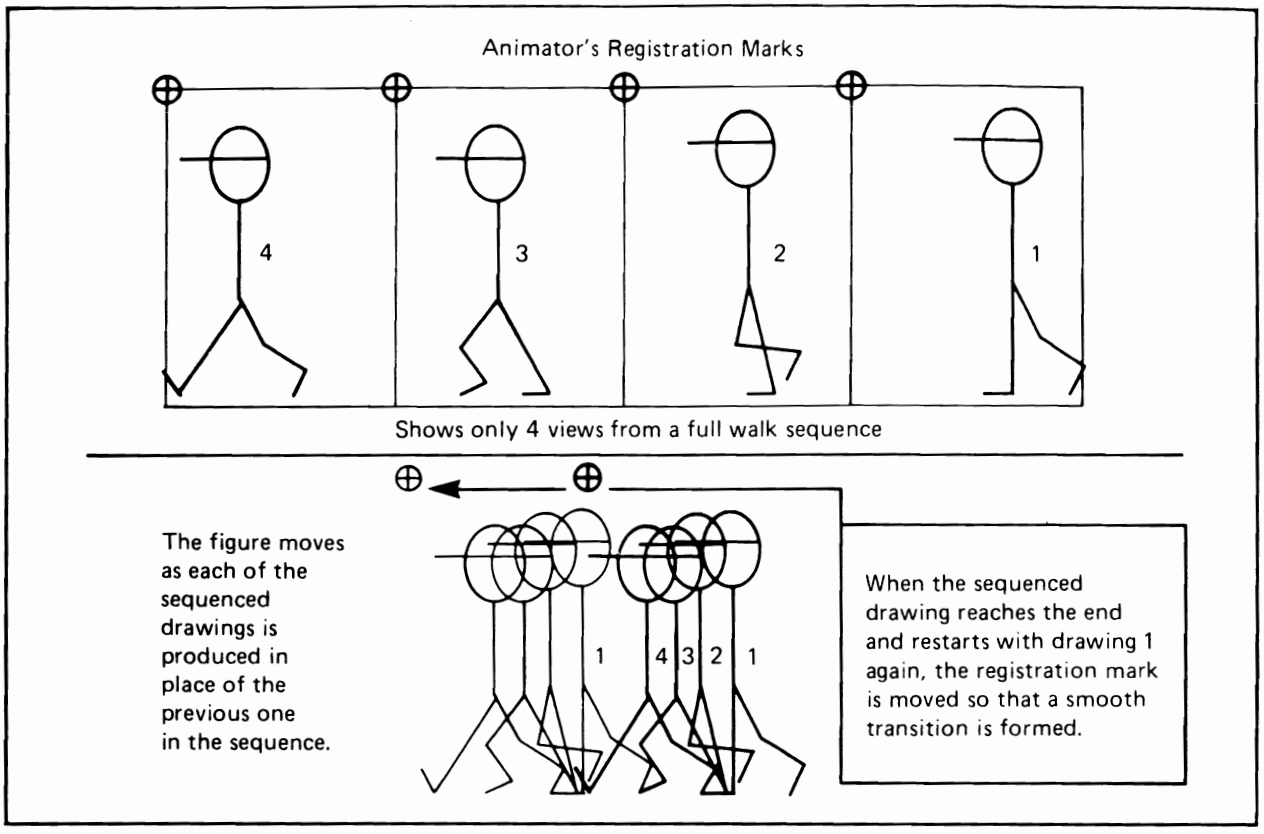


Figure 3-7: A Sequenced Drawing

Sequenced animation often consists of a closed "ring" of drawings. When the last drawing of the sequence has been completed, the first drawing in the sequence is repeated again, becoming the first in the next part of the animation, offset by a specific position in space.

To specify sequenced drawing, use the variables called **compFlags** in the **AnimComp** structure, and **RingXTrans** and **RingYTrans** in the **AnimOb** structure.

To move the registration mark to a new location, you set the RINGTRIGGER bit for a component in its **compFlags** variable. The system software adds the values of **RingXTrans** and **RingYTrans** found in the **AnimOb** structure to the values of **AnX** and **AnY** of the head object (the registration mark), thereby moving the reference point to the new location. The next time you execute **DrawGList()**, the drawing sequence starts over again at the new location, mating properly with the final drawing of the sequence at the old registration mark.

You usually set RINGTRIGGER in only one of the animation components in a sequence; however, you can choose to use this flag and the translation variables in any way you wish.

The next section shows how to tell the system to control the movement of the animation objects.

Motion Control

In the second form of animation, you can specify objects that have independently controllable velocities and accelerations in the X and Y directions. Components can still sequence. Furthermore, you can use ring and velocity simultaneously if you wish.

The variables that control this motion are located in the **AnimOb** structure, and are called:

- **YVel, XVel**—the velocities in the y and x directions
- **YAccel, XAccel**—the accelerations in the y and x directions

Velocities and accelerations can be either positive or negative.

The system treats the velocity numbers as though they are fixed-point binary fractions, with the decimal point fixed at position 6 in the word. That is:

vvvvvvvvvv.ffffff

where v stands for actual values that you add to the X or Y (**AnX**, **AnY**) positions of the object for each call to **Animate()**, and f stands for the fractional part. By using a fractional part, you can specify the speed of an object in increments as precise as 1/64th of an interval.

In other words, if you set the value of **XVel** at 0x0001, it will take 64 calls to the **Animate()** routine before the system will modify the object's X coordinate position by a step of one. The system requires a value of 0x0040 to move the object one step per call to **Animate()**.

Each call you make to **Animate()** simply adds the value of **XAccel** to the current value of **XVel**, and **YAccel** to the current value of **YVel**, modifying these values accordingly.

The following section considers the relative merits of the two animation techniques offered by the system.

Using Sequenced Drawing and Motion Control

If you are using sequenced drawing, you will probably set the velocity and acceleration variables to zero. This allows you to produce the animation exactly in the form in which the artist has designed it in the first place.

Consider an example of a person walking. As each foot falls, with sequenced drawing, it is positioned on the ground exactly as originally drawn. If you include a velocity value, then the person's foot will not be stationary with respect to the ground, and the person appears to "skate" rather than walk. If you set the velocity and acceleration variables at zero, you can avoid this problem.

Once you've specified the sequence of the drawings or the acceleration and velocity variables, it's time to initialize the animation system.

Initializing the Animation System

To initialize the system, you must define a pointer to an **AnimOb**. The system uses this pointer to keep track of all of the real **AnimObs** that you create. The following typical code sequence accomplishes this:

```
struct AnimOb *animKey;
    -
    -
animKey = NULL;
```

NOTE: Before you can use the animation system, you must call the routine **InitGels()**. Therefore, you must initialize the GEL system as well as the animation system. See the "Initializing the GEL System" section for details on **InitGels()**, the Bob-control system that eventually displays the objects that you manipulate.

Now you're ready to add animation objects and components, as described in the following two sections.

Specifying the Animation Objects

To add animation objects to the controlled object list, you use the routine **AddAnimOb()**. Figure 3-8 shows how to build a list of controlled objects using this routine. The **animKey** always points to the object most recently added to the list.

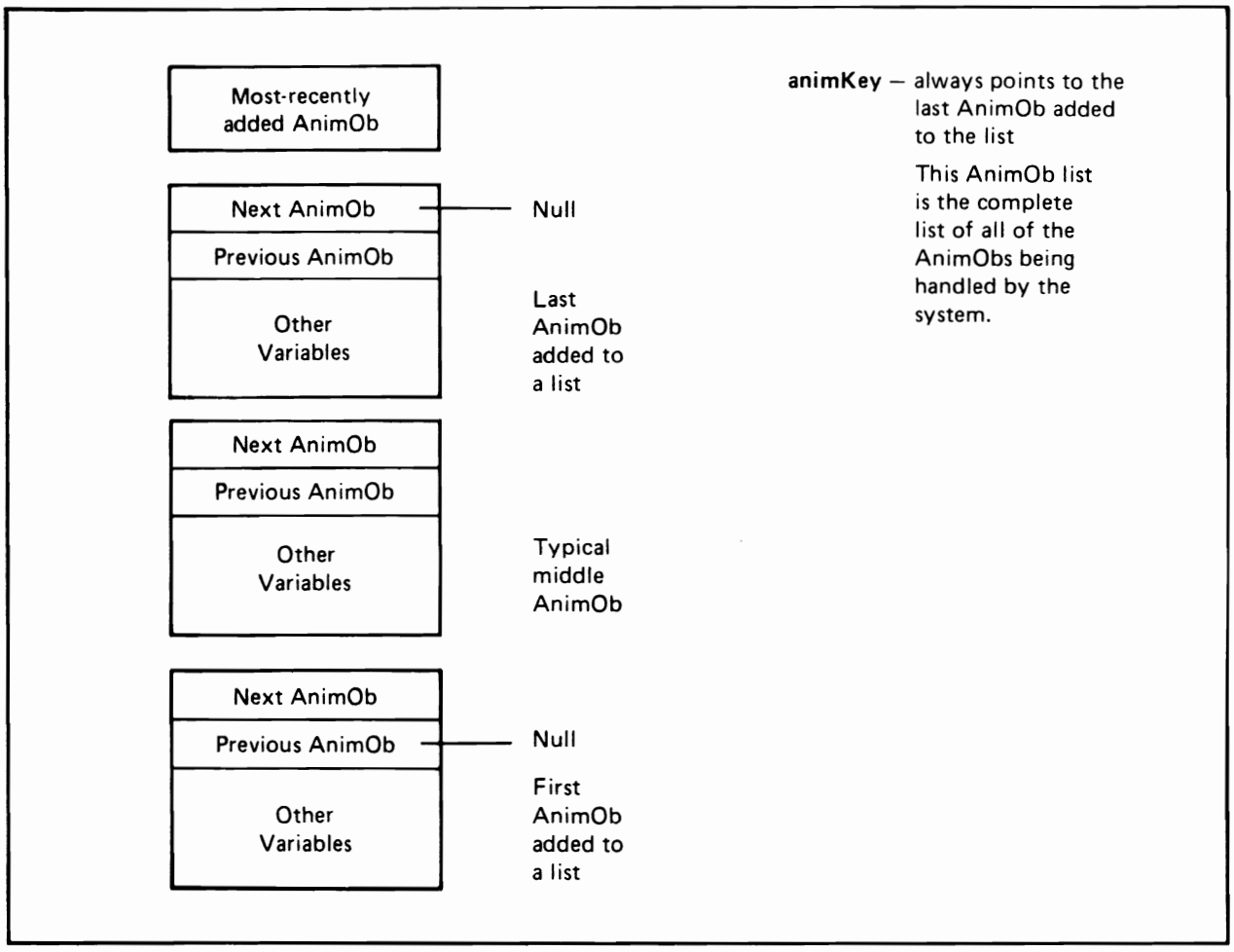


Figure 3-8: Linking AnimObs into a List

Next, you tell the system about the components that make up the object.

Specifying Animation Components

As previously stated, each animation object consists of one or more individual component parts. The parts may be, for example, the body of an object, its arms, its legs, and so on. Not only does the system animator move parts from place to place, but it also offers different views of each of the parts.

To specify the relationships between the individual parts and views of those parts, you initialize various pointers within the **AnimComp** structure.

You use the pointers called **PrevSeq** and **NextSeq** to build a doubly-linked list of a set of animation components used for sequenced drawing, as outlined above. In all cases, when you specify **AnimComps**, you must initialize these pointers to build the sequence that you wish the system to follow for drawing the various views of this component. The "Animation Sequencing" section below shows how the system uses these pointers.

To link the components together into a whole object, use the pointers called **PrevComp** and **NextComp**. When you build an animation object, you must initialize the **PrevComp** and **NextComp** pointers *for only the initial view of the animation object.* Whenever the animation system senses that one of the animation objects has "timed out" and switched to a new sequence of that component, the system automatically adjusts the **PrevComp** and **NextComp** pointers so that it retains the complete animation object.

Figure 3-9 shows an animation object built of several components. The **AnimOb** points to the head component. Notice that the "head" component may be any one of the components of the object. A pointer in the structure of the head component, in turn, points to the next one, and so on (building the initial view of the object).

To point around the ring for each of the component sequenced views (although the objects do not necessarily have to form a ring), you initialize the sequence pointers **NextSeq** and **PrevSeq**. The animation system ignores the **PrevComp** and **NextComp** pointers for each of the *non-current* components.

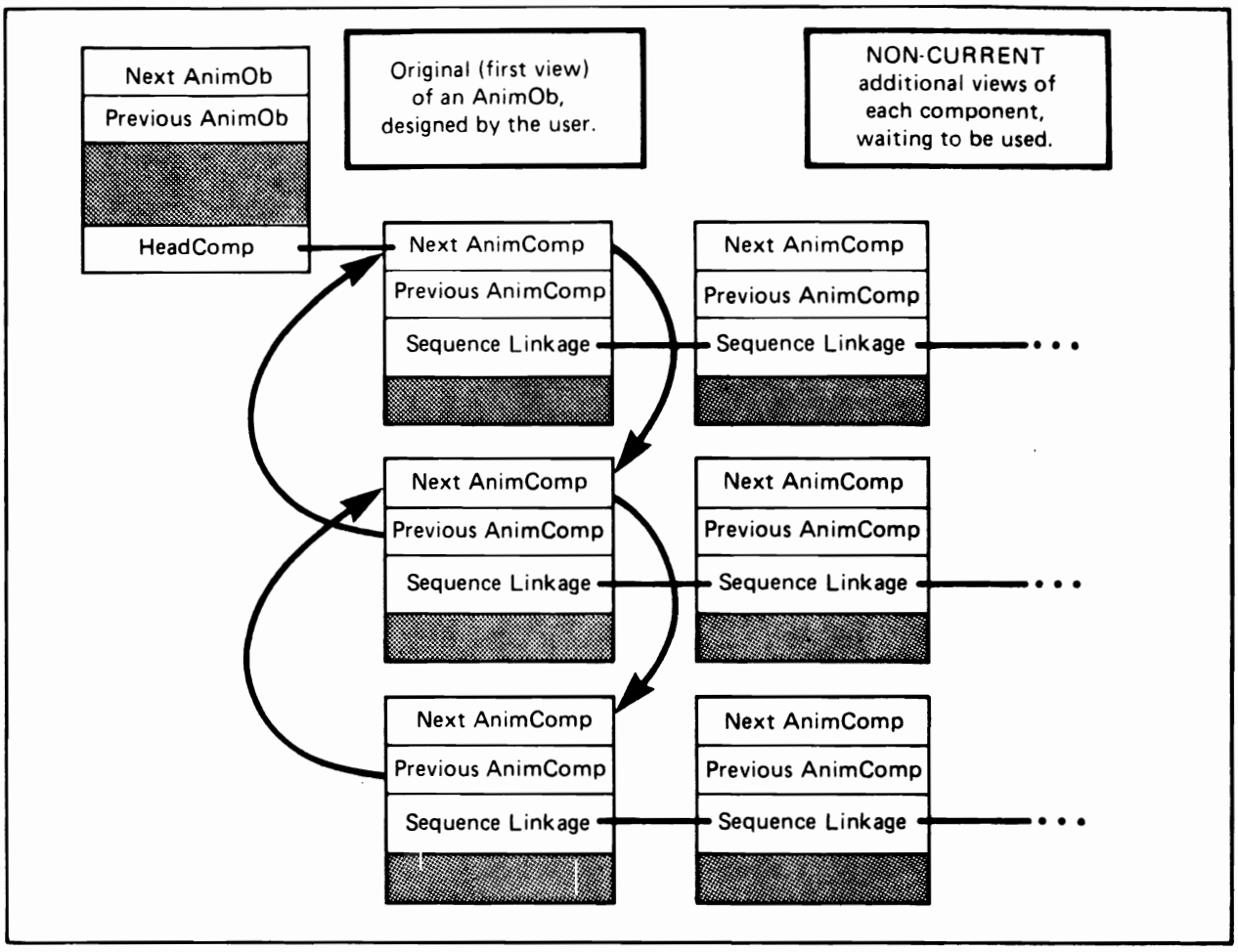


Figure 3-9: Linking AnimComps to Form an AnimOb

Now that you've explained the relationship of the components and views to the system, it's time to specify the order in which to draw them.

Drawing Precedence

The sequence in which you link the components in a list to define the object itself is immaterial. The system simply uses this list of components to define the overall object.

To specify the drawing precedence for the objects in an animation object, you use the **Before** and **After** pointers in the **Bob** structure *for the initial sequence of the animation object.*

If you refer to the description of adding **Bobs** in the section called "Adding a Bob", you will see that when you add **Bobs** to the system, the **Before** and **After** pointers control the drawing sequence and thereby the precedence of the objects. Once you've added the **Bobs** to the system with **AddBob()**, you must assign a fixed set of pointers to establish the correct drawing order.

Animation components may have several views, each of which points to a **Bob** structure. However, only one of those views is actually "active" for that component at any one time, making up part of the overall animation object. The animation system adjusts the **Before** and **After** pointers of the **Bob** structure for each of the current views to maintain the sequence of drawing for each of the components the same as that you have defined for the initial view. Adjustments take place in the sequencing any time any one of the animation components "times out" and proceeds to show a new sequence.

Therefore, if you are defining **Bobs** as part of the animation system, you only need to initialize the **Before** and **After** pointers within the **Bob** structure for the initial sequence of each of the components.

You may wish to define multiple animation objects. To assure that one complete *object* always has priority over another object, you can use the initial sequence linkage to control this as well. You use the **Bob Before** and **After** pointers to link together the last **AnimComp**'s **Bob** of one **AnimOb** to the first **AnimComp**'s **Bob** of the next **AnimOb**. The system maintains the drawing order during calls to **Animate()** from that time onward.

You may modify the drawing order during part of the animation, (such as to make one object pass in front of another during one display sequence, then pass behind it on the next sequence). You can perform this kind of activity, if you wish, during an AnimORoutine or AnimCRoutine. See the section called "Your Own Animation Routine Calls" for details.

The next step is to tell the system about the sequence of the drawings.

Animation Sequencing

To perform sequenced drawing, you must define the sequence in which you wish the drawings to be made. For each of the animation components, there is a set of pointers that allows you to define the exact sequence in which the drawings should appear.

After a period of time that you've specified, which is separately controllable for each component, the system software automatically switches from the current drawing in the sequence to the next one. For this purpose, you provide three pieces of information in the **AnimComp** structure: pointers to the previous and next drawings in the sequence that you have defined, a user flag variable called **Flags**, and a **TimeSet** variable.

After the specified time interval for each of the sequenced drawings, the system software switches to show the next drawing specified in the sequence. The next section shows how you specify the time.

Figure 3-10 illustrates how the system uses the "next sequential image" pointer to step from one image to the next at the specified time.

If you set the RINGTRIGGER bit in the **Flags** variable, the system adjusts the reference point for the sequenced drawing. See the "Sequenced Drawing" section above for details.

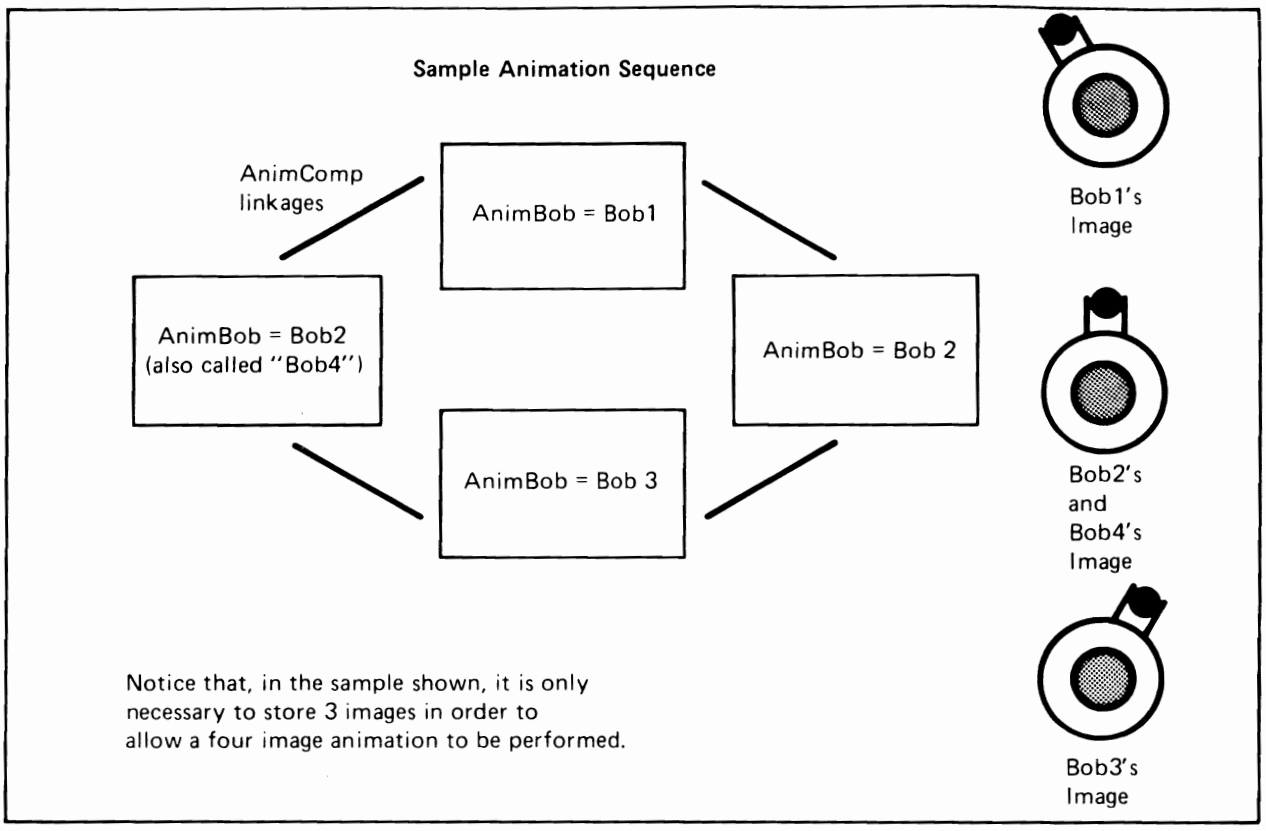


Figure 3-10: Linking AnimComps for Sequenced Drawing

Specifying Time for Each Image

When you have defined all of your animation objects and components, you call the **Animate()** routine. To manipulate the objects, you set the variable called **Timer** in the **AnimComp** structure, and you set a corresponding variable called **TimeSet** (also in the **AnimComp** structure).

When the system selects the animation component, the system copies the value currently in **TimeSet** into the variable named **Timer**. If **Timer** has a nonzero value when you call **Animate()**, then the current view of the animation component remains the active view for as many calls to **Animate()** as you specify with the value in **Timer**. When the **Timer**

value counts down to zero, the system makes the next sequential view active.

If you set the value in **TimeSet** to zero, then **Timer** remains zero. **Timer** never triggers from a non-zero state and, therefore, does *not* cause any change in the view.

When the system activates a new sequence component, it checks that component's compFlag to see if the RINGTRIGGER flag bit is set. If so, the system performs ring processing, which means that it adds the values **RingYTrans** and **RingXTrans** to **AnY** and **AnX** respectively. See the section called "Animation Types" for details.

Now let's see how this process works in an actual animation. Let's say that you are animating the figure of a man. As he walks across the screen, he swings his arm back and forth at a fixed rate. Let's say you have three drawings of the arm: swung forward, at a center position, and swung back. To animate the arm, you may follow these steps:

1. Define 4 **Bobs**: the first for the forward swing, the second for the center, the third for the back swing, and the fourth centered again.

2. Define 4 **AnimComps**, one for each of these **Bobs**. To link them together in a sequence (forward, center, back, center), use the **PrevSeq** and **NextSeq** pointers.

3. Link *one* of the **AnimComps** in this sequence to the **AnimComp** that defines the body of the man, using the **AnimComp**, **PrevComp**, and **NextComp** pointers.

4. Set the **Timer** variable for *each* sequenced **AnimComp** to a value appropriate for him to hold that pose. For example, three calls to **Animate()** for forward and back, and two calls for each of the two centered positions of his arm might be appropriate values.

5. Set the value of **XTrans** and **YTrans** for *each* **AnimComp** to position the arm properly with respect to the rest of the body for each sequence of the arm swing.

6. Continue the arm sequence by setting the RINGTRIGGER bit in the flags variable of the last sequence, thereby triggering a re-sequence to the first view again when the timer of the last view times out.

Now, each time you call **Animate()**, the animation system checks all of the **Timer** variables, as well as calling your AnimCRoutines and AnimORoutines. When each of the **Timer** variables becomes a zero, the next sequenced view of the **AnimComp** replaces the current sequence. When an **AnimComp** becomes "current", the value in its **TimeSet** variable is copied into its **Timer** variable.

This also means that you've told the system two things: first, to remove the **Bob** of the current sequence from the system **Bob** list the next time you call **DrawGList()**; and second, to use the **Bob** representing the new sequence in its place. The system automatically copies the **Bob Before** and **After** pointers from the current sequence into the new sequence **AnimComp**'s **Bob** to assure that the object is still drawn in the same order, maintaining its priority relative to other objects in the drawing area.

The next section shows some possible uses for the **AnimOb** routines.

Your Own Animation Routine Calls

The **AnimOb** and **AnimComp** structures include pointers to your own routines that you want the system to call. If you want a routine to be called, you must specify the address of the routine in this variable. If no routine is to be called, you must set this variable to zero. No values are passed to these routines, except a pointer to its **AnimOb** or **AnimComp**, respectively. However, because you set each AnimORoutine (the **AnimOb** routine) and AnimCRoutine (the **AnimComp** routine), you can use the extensions to the **AnimOb** or **Bob** or **VSprite** structures to hold the variables you need for your own routines.

Suppose you are creating the following animation:

- A man is walking a dog down a street. There is a fireplug at one side of the screen. Let's say you wish to change the appearance of the fireplug if the dog approaches too closely. You would, therefore, design an AnimORoutine to do a proximity check on the dog.

- To allow the fireplug to have different appearances, you might provide three individual views. One is normal, one is an intermediate view (comparable to the center arm-swing mentioned earlier), and the final view is a "strength pose", saying "back off dog!".

- You may set the **TimeSet** and **Timer** variables for the "normal" appearance for the fireplug at zero. This means that it should never change from this appearance no matter how many calls to **Animate()** occur, as defined above. (If it's already zero, it won't decrement; therefore, it can never go from non-zero to zero).

- You may set the **TimeSet** variable for the intermediate view to 1 (stay in the intermediate pose for only one call to **Animate()**). In addition, you may set the **TimeSet** variable for the strength pose to 10 (stay strong for 10 calls to **Animate()**).

- For each call to **Animate()**, the AnimORoutine for the fireplug checks how close the dog has approached. If it is within a certain range, the AnimORoutine changes the **Timer** variable for the normal fireplug pose to a 1.

- The next call to **Animate()** finds a value of 1 in the **Timer** variable and decrements it. This makes a value of 0, forcing a change to the next sequence (the intermediate pose). The system will remove the normal pose **Bobs** from the system **Bob** list it is to draw and the next call to **DrawGList()** will therefore draw the intermediate pose instead.

- The next call to **Animate()** finds a value of 1 in the **Timer** variable for the intermediate pose, and decrements it, causing a change to the strength pose. The fireplug remains in the strength pose for ten calls to **Animate()**, returning through the intermediate pose for one call, then to the normal pose again.

- Now that the **Timer** value has become zero again, the fireplug returns to the original state, staying in its normal pose until the dog again approaches within range.

The next section discusses **Animate()**, the routine you call to actually move the objects.

Moving the Objects

When you have defined all of the structures and have established all of the links, you can call the **Animate()** routine to move the objects. **Animate()** adjusts the positions of the objects as described above, and calls the various subroutines (AnimCRoutine, AnimORoutine) that you have specified.

After the system has completed the **Animate()** routine, as the screen objects have been moved, their order in the graphics objects list may possibly be incorrect. Therefore, as always, before ordering the system to redraw the objects, you must sort them first.

If you perform **DoCollision()** when the system has newly positioned the objects after your call to **Animate()**, your collision routines may also have an effect on the ultimate position of the objects. Therefore, you should again call **SortGList()** to assure that the system correctly orders the objects before you call **DrawGList()**, as illustrated in the following typical call sequence:

```
/* ... setup of graphics elements and objects */

Animate( key, rp );       /* "move" objects per instructions */
SortGList( rp );          /* put them in order */
DoCollision( rp );        /* software collision detect/action */
SortGList( rp );          /* put them back into right order */
DrawGList( vp, rp );        /* draw into current RastPort */
```

Chapter 4

Text

This chapter describes how to use text in Amiga displays.

4.1. INTRODUCTION

Text on the Amiga is simply another graphics primitive. Because of this, you can easily intermix text and graphics on the same screen. Typically, a 320 by 200 graphics screen can contain 40-column, 25-line text using a text font defined in an 8-by-8 matrix. The same type of font can be used to display 80-column text if the screen resolution is extended to 640 by 200. Window borders and other graphics embellishments may reduce the actual available area.

The text support routines use the **RastPort** structure to hold the variables that control the text drawing process. Therefore, any changes you make to **RastPort** variables affect both the drawing routines and the text routines.

In addition to the basic fonts provided in the ROMs, you can link your own font into the system, and ask that it be used along with the other system fonts.

This chapter shows you how to:

- print text into a drawing area
- specify the character color
- specify which font to use
- access disk-based fonts

- link in a new font
- define a new font
- define a disk-based font

4.2. PRINTING TEXT INTO A DRAWING AREA

The placement of text in the drawing area depends on several variables. Among these are:

- the current position for drawing operations,
- the font width and height, and
- the placement of the font baseline within that height.

Cursor Position

Text position and drawing position use the same variables in the **RastPort** structure—**cp_y** and **cp_x**, the current vertical and horizontal pen position. The text character begins at this point. You use the graphics call **Move(&rastPort, x, y)** to establish the cp_y and cp_x position.

Baseline Of The Text

The **cp_y** position of the drawing pen specifies the position of the baseline of the text. In other words, all text printed into a **RastPort** using a single "write string" command is positioned relative to this **cp_y** as the text baseline. Here is some sample text which includes a character which has 1 dot below the baseline, and a maximum of 7 dots above and including the baseline.

For clarity, periods (.) and asterisks (*), rather than 0's and 1's, are used for the figure.

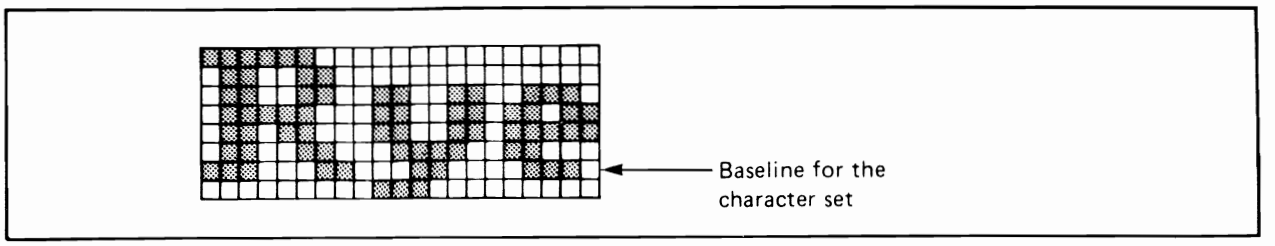


Figure 4-1: Text Baseline

The figure shows that for this font, the baseline value is 6. The baseline value is the number of lines from the top of the character to the baseline.

When the text routines output a character to a **RastPort**, the leftmost edge of the character position is specified by the **cp_x** (current horizontal position) variable.

After all characters have been written to the **RastPort**, the variable **cp_y** is unchanged. The value of **cp_x** will be changed by the number of horizontal positions that were needed to write all characters of the specified text. Both fixed-width and proportionally spaced character sets are accommodated.

The default fonts in the system are all designed to be above and below the baseline, where the baseline position is at line 6 of the character font. This means that you must specify a **cp_y** value of at least 6 when you request that text be printed to a **RastPort** in order to assure that you stay within the memory bounds of the **RastPort** itself. Location (0,0) specifies the upper left-hand corner of the memory space that is dedicated to the **RastPort**. Since all text will be written above and below the baseline, you must start at a proper position or the routines will write into non-**RastPort** memory.

You should not request that the text routines write beyond the outer bounds of the **RastPort** memory, either horizontally or vertically. Text written outside the **RastPort** bounds may be clipped if the **RastPort** supports clipping (most do). Clipping means that the system will display only that portion of the text that is written into the boundaries of the **RastPort**.

Size of the Font

Font design is covered later in this chapter. For now, simply note that the width and height of the font affect how many characters you may print on a line. The position of the baseline affects where you print a line.

Printing the Text

You may print text into a **RastPort** by using the **Text()** routine. A typical call to this routine is:

```
Text( &rastPort, string, count )
```

where:

&rastPort	is a pointer that describes where the text is to be output
string	is the address of the string output
count	is the string length

Sample Print Routine

Here is an example showing a string to be written to a **RastPort**.

The example assumes that you have alrady prepared a **RastPort** into which the text can be rendered.

```
        /* sample routine to print a single line of
        text to the screen. */
struct RastPort *rp;
test( )
{
SetAPen( rp, 1);
        /* use color number 1 to draw the text */
Move( rp, 0, 40);
        /* start down a few lines from the top */
Text( rp, "This is test text", 17 );
return();
}
```

SELECTING THE FONT

Character fonts each have a name. There are two default character fonts provided in the ROMs. One font produces either 40- or 80-column text (depending on the use of a 320 or 640 horizontal resolution respectively). The other font produces either 32- or 64-column text. The names and specifications of these default fonts are:

Table 4-1: Default Character Fonts

Font Type	Height	Name
40/80	8	topaz.font
32/64	9	topaz.font

To specify which font the system should use, you call the system routine **OpenFont()** or **OpenDiskFont()**, followed by **SetFont()**. A typical call to these routines follows.

```
font=OpenFont(textattr);
font=OpenDiskFont(textattr);
SetFont( font, rp )
```

where:

font

is a pointer to a **TextFont** data structure, returned either by **OpenFont()** or **OpenDiskFont()**.

textattr

This structure is located in the include file named *text.h*. It contains a pointer to a null-terminated string that specifies the name of the font, font height, font style bits, and font preference bits.

rp

is the address of the **RastPort** which is to use that font until told to use a different one.

The call to **OpenFont()** or **OpenDiskFont()** says "give me a font with these characteristics". The system attempts to fulfill your request by providing the font whose characteristics best match your request. The table above shows that both of the system fonts have the name "topaz.font". In the system font selections, the height of the characters distinguishes between them. If **OpenFont()** cannot be satisfied, it returns a 0.

NOTES:

1. In Chapter 1, "Graphics Primitives", you saw that the routine **InitRastPort()** initializes certain variables to default values. This routine automatically sets the default to topaz.font with the correct width according to Preferences.

The example below shows how a new font is selected.

```
/* sample routine to print 2 lines of text to
the screen; each line of text in a different font,
again assumes RastPort already set up elsewhere */

test( )
{
struct TextAttr f;
      /* provide a font structure to build on for font change */
struct TextFont *font;
f.Name = "topaz.font";
      /* set font name into font descriptor struct */
      /* initial font default is "topaz.font" */
f.YSize = 8;
      /* define font size */
f.Style = 0;
      /* define font style */
f.Preferences = 0;
      /* define font preferences */
font=OpenFont(&f);
if (font !-0) {
      SetFont( rp, font);
            /* ask system to find & set one like this */
      Move( rp, 0, 40);
      Text( rp, "topaz.font, 8 dots high", 23 );
      CloseFont(font);
{
f.Ysize=9;
font=OpenFont(&f);
if (font != 0) {
      SetFont(rp,font);
      Move( rp, 0, 48);
            /* start a few lines down from the top */
      Text( rp, "topaz.font, 9 dots high", 23);
      CloseFont(font);
{
return(0);
}
```

4.3. SELECTING THE TEXT COLOR

You can select which color to use for the text you print by using the graphics calls **SetAPen()** and **SetBPen()** and by selecting the drawing mode in your **RastPort** structure. The combination of those values determine exactly how the text will be printed.

4.4. SELECTING A DRAWING MODE

The **DrawMode** variable of a **RastPort** determines how the text will be combined with the graphics in the destination area.

NOTE: The **DrawMode** selections are *value*s, not bits. You can select from any *one* of the following DrawModes.

If **DrawMode** is JAM1, it means that the text will be drawn in the color of **FgPen** (the foreGround, or primary drawing Pen). Wherever there is a 1-bit in the text pattern, the **FgPen** color will overwrite the data present at the text position in the **RastPort**. This is called overstrike mode.

If **DrawMode** is JAM2, it means that the **FgPen** color will be used for the text, and the **BgPen** color (the background or secondary drawing color Pen) will be used as the background color for the text. The rectangle of data bits which defines the text character completely overlays the destination area in your **RastPort**. Where there is a 1-bit in the character pattern definition, the **FgPen** color is used. Where there is a 0-bit in the pattern, the **BgPen** color is used. This mode draws text with a colored background.

If DrawMode is COMPLEMENT, it means that wherever the text character is drawn a position occupied by a 1-bit causes bits in the destination **RastPort** to be changed as follows:

- If a text-character 1-bit is to be written over a destination area 0-bit, it changes the destination area to a 1-bit.
- If a text-character 1-bit is to be written over a destination area 1-bit, the result of combining the source and destination is a 0-bit. In other words, whatever is the current state of a destination area bit, a 1-bit in the source changes it to the opposite state.
- Zero-bits in the text character definition have no effect on the destination area.

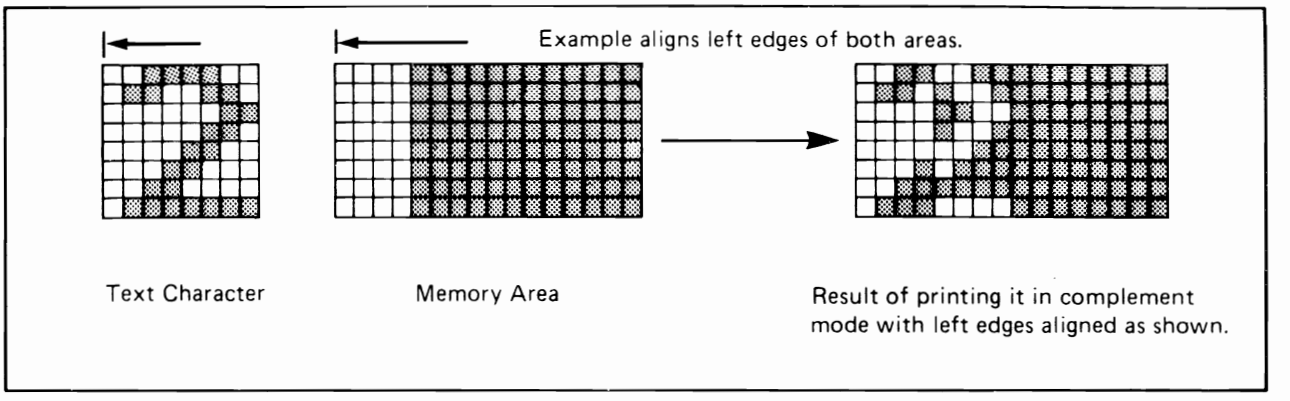


Figure 4-2: Complement Mode

If you set the INVERSVID flag to a 1, it will change all 1-bits to 0-bits and vice versa in a text or other **RastPort** writing operation before writing them into the destination area. If the drawing mode at that time is JAM2, then the pattern colors will be reversed as well. If **DrawMode** is INVERSVID, you can produce inverse video characters.

Here is an example showing each of the three modes of text that you can produce:

```
            /* sample routine to print 4 lines of text to
            the screen; each line of text in a different mode */
test( )
{
SetAPen( rp, 2);
            /* use color 2 as primary drawing color */
SetBPen( rp, 3);
            /* use color 3 as secondary drawing color */
Move( rp, 0, 6);
            /* move the drawing position near upper left */
SetDrMd( rp, JAM1 );
            /* Jam 1 color into target raster */
Text( rp, "This is JAM1 mode", 17 );
Move( rp, 0, 46);
            /* move the drawing position for next line */
SetDrMd( rp, JAM2 );
            /* Jam 2 colors into target raster */
Text( rp, "This is JAM2 mode", 17 );
Move( rp, 0, 86);
            /* move the drawing position for next line */
SetDrMd( rp, COMPLEMENT );
            /* use exclusive-or (COMPLEMENT) to write */
Text( rp, "This is COMPLEMENT mode", 23 );
Move( rp, 0, 126 );
SetDrMd(rp,JAM1+INVERSEVID);
Text( rp, "INVERSE", 7 );
return;
}
```

4.5. EFFECTS OF SPECIFYING FONT STYLE

When you call **OpenFont()**, specifying certain style characteristics, the system searches the loaded fonts to find the closest match to the font you requested. If the remainder of the characteristics match what you have requested, but the style does not match, the text routines **AskSoftStyle()** and **SetSoftStyle()** create a font styled as you have requested by modifying the existing font (that is, modifying a normal font to italic or bold by modifying its characters.) Since many fonts do not lend themselves to such modifications, it is always preferred that the font of the specific style be loaded for use. The system always tries to find the exact specified font before attempting to modify another to fit your request.

If there is a font present in the system that matches your **OpenFont()** request both in name and size, but not in style, (as determined by looking at the font style field), you may set **SetSoftStyle()** to generate the selected style algorithmically as follows:

NORMAL

The font is used exactly as defined.

UNDERLINED

An underline is generated one pixel below the baseline position.

ITALIC

The character is given a slant to the right, starting from the bottom line, and shifting subsequent upward line positions to the right one bit position for every second count up from the bottom of the character.

EXTENDED

(This attribute cannot be set with **SetSoftStyle()**.)

If you use a font that has the various style characteristics built-in, rather than generated, the internal spacing and kerning tables tell the system how to leave the proper amount of space between characters if you are simply printing them one at a time.

If you ask **Text()** to output the characters individually, **Text()** calculates character positioning and width based on the *normal* width and inter-character spacing that it finds in the font descriptor. After printing one or more characters, it automatically positions the drawing pen (cp_x) at the position it believes to be correct for the next output character. This may cause adjacent characters to overlap when printed individually.

There is a solution. If you are using generated style for a font, you must take care to build your output strings of characters before calling **Text()** to output them. **Text()** can handle character strings, correctly generating the desired style with correct inter-character spacing.

To increase inter-character spacing, you can set a field called **rp_TxSpacing** in the **RastPort**. The spacing is specified in pixels.

4.6. ADDING A NEW FONT TO THE SYSTEM

The ROM Exec code maintains a list of the text fonts that are currently linked into the system. To add another font, you must:

A. Open a disk font using the diskfont.library, or

B. Define the font

1. Reserve some memory where the font can be loaded
2. Move the font definition into that memory area
3. Link the font name and location into the system font list.

4.7. USING A DISK FONT

To use an existing disk font, you must perform the following steps:

A. Open the diskfont library

B. Open a disk font

Here are the program fragments you need to open the library. This gives you access to whatever routines the diskfont library contains:

```
struct Library *DiskfontBase;

DiskfontBase = (struct Library *)
        OpenLibrary("diskfont.library",0);
```

Before trying to use the diskfont routines, you should check that the **OpenLibrary()** call returned a value other than NULL.

Here is the program fragment you need to actually load a disk-based font. It assumes that you already know the name of the font you want to load.

```
struct TextFont *font;
struct myTextAttr = { "sapphire.font",17,0,0 };

font = OpenDiskFont(&myTextAttr);
```

4.8. FINDING OUT WHICH FONTS ARE AVAILABLE

The function **AvailFonts()** fills in a memory area designated by you to hold a list of all of the fonts available in the entire system. **AvailFonts()** searches the AmigaDOS directory path currently assigned to FONTS: and locates all available fonts. If you haven't issued a DOS ASSIGN command to change the FONTS: directory path, the system will search the sys:fonts directory.

The test program "whichfont.c" at the end of this chapter provides a list of the fonts you can use and shows you how to find the appropriate items to put into the text attribute data structure for the call to **OpenDiskFont()**.

4.9. CONTENTS OF A FONT DIRECTORY

In a font directory, you will usually find two names for each font type. A typical pair of entries in the fonts directory is:

```
sapphire.font
sapphire(dir)
```

The file named *sapphire.font* doesn't contain the actual font. It contains the description of the contents of that font family. The contents are described by a **FontContentsHeader** and one or more **FontContents** data structure entries. The **FontContentsHeader** structure is defined as:

```
struct FontContentsHeader {
    UWORD   fch_FileID;     /* FCH_ID */
    UWORD   fch_NumEntries; /* the number of FontContents elements */
        /* FontContents (1 or more) follow here */
    };
```

where:

fch_FileID
is simply a numeric identifier for this file type. The value is 0xf00.

fch_NumEntries
says how many entries of type FontContents follows this header.

The **FontContents** structure is defined as:

```
struct FontContents {
    char    fc_FileName[MAXFONTPATH];
    UWORD   fc_YSize;
    UBYTE   fc_Style;
    UBYTE   fc_Flags;
    };
```

where

fc_FileName
is the pathname which the DOS must follow to find the actual diskfont descriptive header along with the **TextFont** data structure of which this font is composed. Once DOS reaches the path named in FONTS:, it finds the filename by the path shown in this entry in **FontContents**.

fc_YSize, **fc_Style**, and **fc_Flags**
correspond to their equivalents in the **TextAttr** data structure (**ta_YSize**, **ta_Style**, and **tc_Flags**).

As an example, a typical entry in sapphire.font is:

```
"sapphire/14", a null terminated string, padded out with
            zeros for a length of MAXFONTPATH bytes,
14,         the value for fc_YSize,
00,         the value for fc_Style,
60 (hex)        the value for fc_Flags.
```

This entry indicates that the actual **DiskFontHeader** for the font to be loaded is in path FONTS:sapphire/14. This means that the sapphire subdirectory in the fonts directory must have a file named *14* in order to allow this font to be loaded.

4.10. THE DISK FONT

A disk font is constructed as a loadable, executable module. In this manner, AmigaDOS can be used to perform **LoadSegment()** and **UnloadSegment()** on it. AmigaDOS can therefore allocate memory for the font, and return the memory when the font is unloaded. The contents of the **DiskFont** are described in the include-file *diskfont.h*. The most significant item in this structure, the embedded **TextFont** structure, is described below in the topic "Defining A Font".

4.11. DEFINING A FONT

To define a font, you must specify its characteristics using the **TextFont** structure. The **TextFont** structure is specified in the include-file named *text.h*.

The following topics show the meaning of the items in a **TextFont** structure. Following the structure description is an example showing a four-character font, which is defined using this structure and can be linked into the system using **AddFont()**.

The Text Node

The first item in the **TextFont** structure is a **listNode** by which the system can link this font structure into the system **TextFonts** list. You specify the name of the font using the name pointer field of the font **listNode**.

For example:

```
struct TextFont suitFont;
        /* name chosen for sample font here */

suitFont.textNode.ln_name = "suits.font";
```

Font Height

You specify the height in the **ySize** variable. All characters of the font must be defined using this number of lines of data even if they do not require that many lines to contain all font data. Variable height fonts are not supported.

For example:

```
suitFont.ySize = 8;
        /* all characters are 8 lines high */
```

Font Style

You can specify the style of the font by specifying certain bits as 1's in the TF **style** variable. The value of **style** is determined by the sum of the style bits, defined as:

NORMAL (value = 0),	The text font is used exactly as defined.
UNDERLINED (value = 1),	The font is underlined.
BOLD (value = 2),	The font is bold.
ITALIC (value = 4),	The font is italic.
EXTENDED (value = 8),	The font is stretched out (width).

In the font structure, these bits indicate style attributes as intrinsically a part of the font; that is, the font already has them and you can never take them away.

Font Preferences

This variable provides additional information that tells the font routines how to create or access the characters. The preferences variable is composed of the sum of the preference bits, defined as follows:

FPB_ROMFONT (value = 0)

The font is located in ROM. If you are making up your own font, this variable will not be zero unless you are burning new system ROMs yourself.

FPB_REVPATH (value = 2)

The font is designed to be rendered from right to left (for example, Hebrew).

FPB_PROPORTIONAL (value = 32)

The characters in the font are not guaranteed to be **xSize** wide (see "Font Width" below.. Each character has its own width and positioning in the character space. The bit-packing of the characters is of great importance, as described below. The variables **modulo**, **charloc**, and **charspace** define how the characters are defined and bit-packed.

Font Width

The **xSize** variable specifies the nominal width of the font. For example:

```
suitFont.width = 14;
        /* specify 14 bits width */
```

Font Accessors

If you have added a font to the system list, it is possible that more than one task will be accessing a character font. A variable in the font structure keeps track of how many accessors this font currently has. Whenever you call **OpenFont()** or **OpenDiskFont()**, this variable is incremented for the font and decremented by **CloseFont()**. The font accessor value should never be reduced below zero. This accessor count should be initialized to zero *before* you first link a new font into the system, but is managed by the system after the link is performed.

If you wish to remove a font from the system to free the memory that it is currently using, you must ensure that the number of accessors is zero before ordering its removal.

Characters Represented by This Font

It is possible to create a font consisting of 0 to 255 characters. Some fonts can be exceedingly large because of their design and the size of the characters. For this reason, the text system allows the design and loading of fonts that may consist of only a few of the characters. The variables **loChar** and **hiChar** specify the numerical values for the characters represented in this font. As an example, one font could contain only the capital letters. A second font could contain the small letters; and a third could contain the punctuation marks and numerals. Depending on the size of the font itself, you may arrange to subdivide the font even further.

In the example that is being built for this chapter, a font consisting of four playing card suits is being constructed. This font might consist of only 4 items, one for each of the playing suits. For example:

```
suitFont.loChar = 160;
        /* value to use for first character chosen at whim */

suitFont.hiChar = 163;
        /* 160 to 163 range says that there are 4 characters
        represented in this font */
```

As part of the character data, in addition to defining the included character numbers, you must also define a character representation to be used as the image of a character number requested but not defined in this font. This character is placed at the end of the font definition.

For this example, any character number outside the range of 160-163 inclusive would print this "not in this font" character.

The Character Data

The font structure includes a pointer to the character set data along with descriptions of the how the data is packed into an array. The variables used are:

charData

a pointer to the memory location at which the font data begins. This is the bit-packed array of character information.

modulo

the row modulo for the font. The font is organized with the top line of the first character bit-adjacent to the top line of the second character and so on.

For example, if the bit-packed character set needs 10 words of 16-bits each to hold the top line of all of the characters in the set, then the value of the modulo will be 20 (bytes). Twenty is the number which must be added to the pointer into the character matrix to go from the first line to the second line of a specific character.

charLoc

a pointer to an array of paired values. The values are the bit-offset into the bit-packed character array for this character, and the size of the character in bits. Expressed in C language, this array of values can be expressed as:

```
struct charDef = {
    WORD charOffset;
    WORD charBitWidth;
    }
```

and in the program definition, the array to which **charLoc** points can be expressed as:

```
struct charDef suitDef[5];
      /* define an array of 4 sets of character,
      and one "not a character"
      bit-packed placement and width information */
```

For all proportional fonts, there must be one set of descriptors for each character defined in the character set.

charSpace

a pointer to an array of words of proportional spacing information. This is the width of each character rectangle, in other words, how many bits width are used to contain the edge to edge width of this character's bit definition.

For example, a narrow character may still be stored within a wide space.

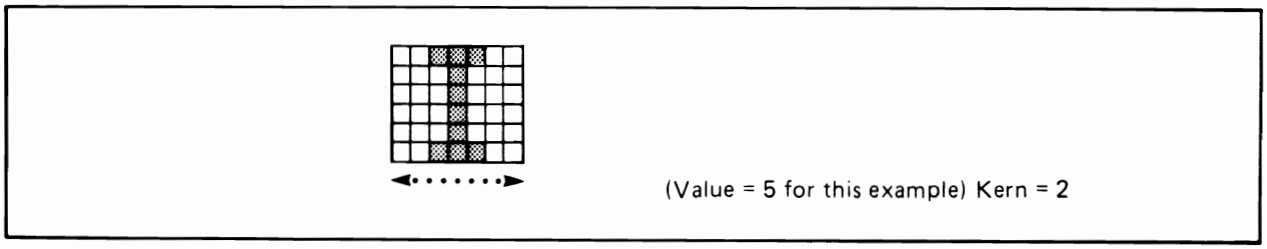


Figure 4-3: CharSpace Figure

If this pointer is null, use the nominal width for each character (**xSize**).

charKern

a pointer to an array of words of character kerning data. Kerning is the offset from the character pointer to the start of the bit data.

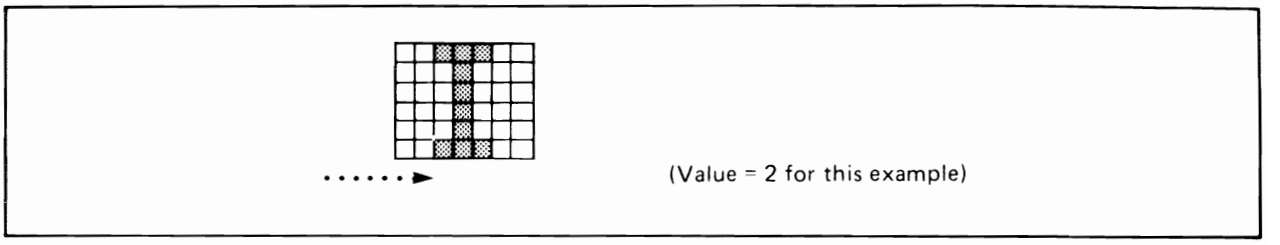


Figure 4-4: CharKern Figure

If this pointer is null, kerning is zero.

A Complete Sample Font

The sample font below pulls together all of the pieces from the above sections. It defines a font whose contents are the four suits from a set of playing cards: clubs, hearts, spades and diamonds.

The suits are defined as proportionally spaced to provide a complete example, even though each suit could as easily have been defined in a 14 wide by 8 high matrix. There is an open-centered square, which is used if you ask for a character not defined in this font.

```
* A complete sample font.  To test this font, the following must be done:
*
* 1.  In the AmigaDOS SYS:fonts directory, install a file by the name of
*     test.font, containing 264 bytes.
*
*     The first two bytes must contain the value hex 0f00, the
*     identifier for a font header.
*
*     The next word (2 bytes), should contain the value 0001,
*     which is the number of FontContents elements.  There is
*     only going to be one font in the directory that this
```

```
*      font description covers.
*
*      Follow this header material with the ascii value for
*      'test/8';  the next 250 bytes should be set to zero.
*      This represents the pathname for AmigaDOS to follow
*      from the directory SYS:fonts in order to reach this test font.
*      'test' is the directory it should go to and '8' is the font
*      file itself, as assembled and linked below.
*
*      The next two bytes (as one word) contain the font YSize, in
*      this case 0008.
*
*      The next byte contains the font Flags, in this case 00.
*
*      The last byte contains the font characteristics, in this
*      case hex 60, this says it is a disk-based font (bit 1 set)
*      and the font has been removed (bit 7 set) saying that the
*      font is not currently resident.
*
*      Summary (all in hex) of test.font file:
*
*      0f00 0001 test/8 ..... 0008 00   60
*      word word 256-bytes..... word byte byte
*
* 2.    Create a directory named 'test' in SYS:fonts.
*
*      Copy the file created by assembling and linking the test font
*      below into a file named '8' in subdirectory SYS:fonts/test.
*
*      Use the font under the Notepad program or any other.  It
*      defines ascii characters 'a' 'b' 'c' and 'd' only.  All
*      other characters print an "unknown character", a rectangle.
*
*
*------ Included Files ----------------------------------------------

        INCLUDE         "exec/types.i"
        INCLUDE         "exec/nodes.i"
        INCLUDE         "libraries/diskfont.i"

            MOVEQ   #0,D0       ;provide an easy exit in case somebody
                                ;tries to RUN this file instead of loading it.
            RTS
            DC.L    0           ; ln_Succ
            DC.L    0           ; ln_Pred
            DC.B    NT_FONT         ; ln_Type
            DC.B    0           ; ln_Pri
            DC.L    fontName        ; ln_Name
            DC.W    DFH_ID          ; FileID
            DC.W    1           ; Revision
```

```
            DC.L    0               ; Segment
fontName:
            DS.B    MAXFONTNAME     ; Name
font:
            DC.L    0               ; ln_Succ
            DC.L    0               ; ln_Pred
            DC.B    NT_FONT         ; ln_Type
            DC.B    0               ; ln_Pri
            DC.L    fontName        ; ln_Name
            DC.L    0               ; mn_ReplyPort
            DC.W    fontEnd-font    ; mn_Length
            DC.W    8               ; tf_YSize
            DC.B    0               ; tf_Style
            DC.B    FPF_DESIGNED+FPF_PROPORTIONAL   ; tf_Flags
            DC.W    14              ; tf_XSize
            DC.W    6               ; tf_Baseline

* baseline must be no greater a value than YSize-1, otherwise algorithmically
* generated style (italic particularly) can corrupt system memory.

            DC.W    1               ; tf_BoldSmear
            DC.W    0               ; tf_Accessors
            DC.B    97              ; tf_LoChar
            DC.B    100             ; tf_HiChar
            DC.L    fontData        ; tf_CharData
            DC.W    8               ; tf_Modulo, no of bytes to add to
                                    ; data pointer to go from one row of
                                    ; a character to the next row of it.
            DC.L    fontLoc         ; tf_CharLoc, bit position in the
                                    ; font data at which the character
                                    ; begins.
            DC.L    fontSpace       ; tf_CharSpace
            DC.L    fontKern        ; tf_CharKern

*************************************************************************
* These are the suits-characters that this font data defines.
* ascii lower case a,b,c,d.  The font descriptor says that there
* are 4 characters described here.  The fifth character in the
* table is the character that is to be output when there is
* no character in this character set that matches the ascii
* value requested.
```

```
*
* 97              98              99              100             256
*<              ><              ><              ><              >
*  @@@   @@@         @               @             @@@         @@@@@@@@@@@@@
* @@@@@ @@@@@      @@@@@           @@@           @@@@@         @@          @@
*  @@@@@@@@@      @@@@@@@@@       @@@@@         @@   @   @@    @@          @@
*   @@@@@@@      @@@@@@@@@@@      @@@@@@@      @@@@@@@@@@@    @@          @@
*    @@@@@      @@@ @ @@@         @@@@@       @@   @   @@     @@          @@
*     @@@           @              @@@           @            @@          @@
*      @            @@@@@           @           @@@@@         @@@@@@@@@@@@@
*
**********************************************************************
fontData:
            DC.W    $071C0,$08040,$070FF,$0F000
            DC.W    $0FBE3,$0E0E0,$0F8C0,$03000
            DC.W    $07FCF,$0F9F3,$026C0,$03000
            DC.W    $03F9F,$0FFFF,$0FFC0,$03000
            DC.W    $01F0E,$0B9F3,$026C0,$03000
            DC.W    $00E00,$080E0,$020C0,$03000
            DC.W    $00403,$0E040,$0F8FF,$0F000
            DC.W    $00000,$00000,$00000,$00000
            DC.W    $00000,$00000,$00000,$00000

* font data is bit-packed edge to edge to save space; thats what the
* fontLoc is all about.

fontLoc:
            DC.L    $00000000B,$0000B000B,$000160007,$0001D000B
            DC.L $00028000C

* Each pair of words specifies how the characters are bit-packed.  For
* example, the first character starts at bit position 0000, and is 000B
* (11) bits wide.  The second character starts at bit position 000B and
* is 000B bits wide, and so on.  Tells font handler how to unpack the
* bits from the array.

fontSpace:
            DC.W    000012,000012,000008,000012,000013

* fontSpace array:  Use a space that is this-wide to contain this character
*                   when it is printed.

fontKern:
            DC.W    000001,000001,000001,000001,000001
fontEnd:
            END
```

4.12. SAMPLE PROGRAM

The following sample program asks **AvailFonts()** to make a list of the fonts that are available, then opens a separate window and prints a description of the various attributes that can be applied to the fonts, in the font itself. Notice that not all fonts accept all attributes (garnet9 for example, won't underline). If you run this program, note also that not all fonts are as easily readable in the various bold and italicized modes. This rendering is done in a fixed manner by software and the fonts were not necessarily designed to accept it. It is always best to have a font that has been designed with a bold or italic characteristic built-in rather than trying to italicize and embolden an existing plain font.

```
/* "whichfont.c" */

#define AFTABLESIZE 2000

#include "exec/types.h"
#include "exec/io.h"
#include "exec/memory.h"

#include "graphics/gfx.h"
#include "hardware/dmabits.h"
#include "hardware/custom.h"
#include "hardware/blit.h"
#include "graphics/gfxmacros.h"
#include "graphics/copper.h"
#include "graphics/view.h"
#include "graphics/gels.h"
#include "graphics/regions.h"
#include "graphics/clip.h"
#include "exec/exec.h"
#include "graphics/text.h"
#include "graphics/gfxbase.h"
#include "devices/keymap.h"
#include "libraries/dos.h"
#include "graphics/text.h"
#include "libraries/diskfont.h"
#include "intuition/intuition.h"

struct AvailFonts *af;
struct AvailFontsHeader *afh;
extern int AvailFonts();

struct TextFont *tf;
struct TextAttr ta;

ULONG DosBase;
ULONG DiskfontBase;
ULONG IntuitionBase;
```

```
ULONG GfxBase;

struct NewWindow nw = {
      10, 10,        /* starting position (left,top) */
      620,40,        /* width, height */
      -1,-1,         /* detailpen, blockpen */
      0,            /* flags for idcmp */
      WINDOWDEPTH|WINDOWSIZING|WINDOWDRAG|SIMPLE_REFRESH|
            ACTIVATE|GIMMEZEROZERO,
                  /* window gadget flags */
      0,             /* pointer to 1st user gadget */
      NULL,            /* pointer to user check */
      "Text Font Test", /* title */
      NULL,            /* pointer to window screen */
      NULL,            /* pointer to super bitmap */
      100,45,         /* min width, height */
      640,200,        /* max width, height */
      WBENCHSCREEN};

struct Window *w;
struct RastPort *rp;

SHORT text_styles[ ] = { FS_NORMAL, FSF_UNDERLINED, FSF_ITALIC, FSF_BOLD,
                  FSF_ITALIC | FSF_BOLD, FSF_BOLD | FSF_UNDERLINED,
                  FSF_ITALIC | FSF_BOLD | FSF_UNDERLINED };

char *text[ ] = { " Normal Text", " Underlined", " Italicized", " Bold",
              " Bold Italics", " Bold Underlined",
              " Bold Italic Underlined" };
char textlength[ ] = { 12, 11, 11, 5, 13, 16, 23 };

char *pointsize[] = { " 0"," 1"," 2"," 3"," 4"," 5"," 6"," 7"," 8"," 9",
                "10","11","12","13","14","15","16","17","18","19",
                "20","21","22","23","24","25","26","27","28","29",
                "30","31"};

char fontname[40];
char dummy[100];      /* provided for string length calculation */
char outst[100];  /* build something to give to Text, see note in
                   * the program body about algorithmically
                   * generated styles
                   */

main()
{
      UBYTE fonttypes;
      int j,k,m;
      SHORT afsize;
      SHORT style;
      SHORT sEnd;   /* numerical position of end of string terminator,
```

```
            * and coincidently the length of the string. */

if( (DosBase = OpenLibrary("dos.library", 0)) == NULL) exit(-1);
if((DiskfontBase=OpenLibrary("diskfont.library",0))==NULL) exit(-4);
if((IntuitionBase=OpenLibrary("intuition.library",0))==NULL) exit(-2);
if((GfxBase=OpenLibrary("graphics.library",0))==NULL) exit(-3);

tf=NULL; /* no font currently selected */
afsize = AFTABLESIZE;    /* show how large a buffer is available */
fonttypes = 0xff;        /* show us all font types */

afh = (struct AvailFontsHeader *) AllocMem(afsize, MEMF_CLEAR);
if(afh == NULL) exit(-5);

printf("\nSearching for Fonts\n");
AvailFonts(afh, afsize, fonttypes);

af = (struct AvailFonts *) &afh[1];  /* bypass header to get to the
                                      * first of the availfonts */

for (j = 0; j < afh->afh_NumEntries; j++)
{
if((af->af_Attr.ta_Flags & FPF_REMOVED) ||
  (af->af_Attr.ta_Flags & FPF_REVPATH) ||
          ((af->af_Type&AFF_MEMORY)&&
            (af->af_Attr.ta_Flags&FPF_DISKFONT)))
            ;       /* do nothing if font is removed, or if
                     * font designed to be rendered rt->left
                     * (simple example writes left to right)
                     * or if font both on disk and in ram,
                     * don't list it twice. */

/* AvailFonts performs an AddFont to the system list;
 * if run twice, you get two entries, one of "af_Type 1" saying
 * that the font is memory resident, and the other of "af_Type 2"
 * saying the font is disk-based.  The third part of the
 * if-statement lets you tell them apart if you are scanning
 * the list for unique elements;  it says "if its in memory and
 * it is from disk, then don't list it because you'll find another
 * entry in the table that says it is not in memory, but is on disk.
 * (Another task might have been using the font as well, creating
 * the same effect).
 */

else
  {
      printf("\nFont name found was: %ls",af->af_Attr.ta_Name);
      printf("  and its point size is: %ld",af->af_Attr.ta_YSize);
      /* Style parameter is in af->af_Attr.ta_Style,
       * Flags parameter is in af->af_Attr.ta_Flags.
```

```
        */
      }
    af++;
    }
    /* now that we've listed the fonts, lets look at them */

    w = (struct Window *)OpenWindow(&nw);
    rp = w->RPort;

for(m=0; m<2; m++)   /* do normal video, then inverse video */
 {

    af = (struct AvailFonts *)&afh[1]; /* reset value of af to original */
    SetAPen(rp,1);

    if(m == 0)SetDrMd(rp,JAM1);
    else SetDrMd(rp,JAM1+INVERSVID);

    /* now print a line that says what font and what style it is */

    for (j=0; j < afh->afh_NumEntries; j++)
    {
    CStringAppend(&fontname[0],af->af_Attr.ta_Name);
                /* copy name into build-name area */
                /* already has ".font" onto end of it */

    ta.ta_Name = &fontname[0];
    ta.ta_YSize = af->af_Attr.ta_YSize;   /* ask for this size */
    ta.ta_Style = af->af_Attr.ta_Style;    /* ask for designed style */
    ta.ta_Flags = FPF_ROMFONT|FPF_DISKFONT|
          FPF_PROPORTIONAL|FPF_DESIGNED;
          /* accept it from anywhere it exists */
    style = ta.ta_Style;

    if(!((af->af_Attr.ta_Flags & FPF_REMOVED) ||
      (af->af_Attr.ta_Flags & FPF_REVPATH) ||
      ((af->af_Type&AFF_MEMORY)&&
      (af->af_Attr.ta_Flags&FPF_DISKFONT))))

      /* this is an IF-NOT, the reverse of the earlier if-test on
       * these same parameters
       */
       {
          tf = (struct TextFont *) OpenDiskFont(&ta);

          if (tf != 0)
          {
          SetFont(w->RPort, tf);
          for(k=0; k<7; k++)
              {
```

```
            style = text_styles[k];
             SetSoftStyle(w->RPort,style,255);
            SetRast(rp,0);/* erase any previous text */
            Move(rp,10,20);      /* move down a bit from the top */
            sEnd = CStringAppend(&outst[0],af->af_Attr.ta_Name);
            sEnd = sEnd + CStringAppend(&outst[sEnd],"  ");
            sEnd = sEnd + CStringAppend(&outst[sEnd],
                              pointsize[af->af_Attr.ta_YSize]);
            sEnd = sEnd + CStringAppend(&outst[sEnd]," Points, ");
            CStringAppend(&outst[sEnd],text[k]);
            Text(rp,&outst[0],CStringAppend(&dummy[0],&outst[0]));

            /* Have to build the string before sending it out to
             * text IF ALGORITHMICALLY GENERATING THE STYLE since
             * the kerning and spacing tables are based on the
             * vanilla text, and not the algorithmically generated
             * style.  If you send characters out individually,
             * it is possible that the enclosing rectangle of
             * a later character will chop off the trailing edge
             * of a preceding character
             */

            /* ************************************************
            This alternate method, when in INVERSVID, exhibits the
            problem described above.

            Text(rp,af->af_Attr.ta_Name,STRLEN(af->af_Attr.ta_Name));
            Text(rp,"  ",2);
            Text(rp,pointsize[af->af_Attr.ta_YSize],2);
            Text(rp," Points, ",9);

            Text(rp,text[k],textlength[k]);
            ************************************************ */

            Delay(40);      /* use the DOS time delay function
                     * specifies 60ths of a second */
            }
          CloseFont(tf); /* close the old one */

    /* NOTE:
    *    Even though you close a font, it doesn't get unloaded
    *    Memory unless a font with a different name is specified
    *    for loading.  In this case, any font (except the topaz
    *    set) which has been closed can have its memory area
    *    freed and it will no longer be accessible.  If you close
    *    a font to go to a different point-size, it will NOT cause
    *    a disk-access.
    *
    *  ALSO NOTE:
    *      Loading a font loads ALL of the point
```

```
        *       sizes contained in that font's directory!!!!
        */

            } /* end of if-tf-ne-0 */
          }   /* end of if-(in memory but from disk) */
        af++;
        }       /* Do next font now */
      }         /* end of for-loop, controlled by m */

        FreeMem(afh,AFTABLESIZE);
        CloseWindow(w);
        CloseLibrary(IntuitionBase);
        CloseLibrary(DosBase);
        CloseLibrary(DiskfontBase);
        CloseLibrary(GfxBase);

}

/* copy a string and return the number of characters added to
 * a string. Effectively returns the length of the string if
 * not adding anything */

int CStringAppend(dest, source)
char *dest;
char *source;
{
    int i=0;
    char *s = source;
    char *d = dest;
    while (( i <79 )&&( *d = *s )) { d++; s++; i++; }
       /* if find a NULL in source, end the copy, but the NULL itself
        * gets copied over to the destination. If no NULL, then 79
        * characters get copied, then a terminating NULL is added */
    if(i < 79) return(i);
    else {*d = 0; return(i);  }
       /* value returned is the position of the terminating NULL
        * to allow other strings to be appended simply using the
        * next append command in sequence */

}
```

Part III

Chapter 1

Audio Device

This chapter describes the functions and commands of the audio device.

1.1. INTRODUCTION

The Amiga has four hardware audio channels—two of the channels produce audio output from the left audio connector and two from the right. These channels can be used in many ways. You can combine a right and a left channel for stereo sound, use a single channel, or play a different sound through each of the four channels.

The audio software is implemented as a standard Amiga input/output device with commands that allocate audio channels and control the sound output.

Some of the audio device commands isolate the programmer from some of the idiosyncrasies of the special-chip hardware. You can also produce sound on the Amiga by directly accessing the hardware registers. For certain types of sound synthesis, this is more CPU-efficient. Some of the audio commands make most sound synthesis easier. Other commands enable your program to co-reside with other programs using the multi-tasking environment to produce sound at the same time. Programs can co-reside because the audio device handles allocation of audio channels and arbitrates among programs competing for the same resources.

Most personal computers that produce sound have hardware designed for one *specific* synthesis technique. The Amiga uses a very general method of digital sound synthesis which is quite similar to the method used in digital hi-fi components and state-of-the art keyboard and drum synthesizers, with one significant difference. The Amiga has a tightly-coupled 68000 microprocessor capable of generating and modifying the digital data while the sound is playing. How much of the CPU you can afford to use for sound synthesis depends on your application.

For programs that can afford the memory, playing sampled sounds gives you a simple and very CPU-efficient method of sound synthesis. When a sound is sampled, the amplitude of the waveform that represents a sound is measured (sampled) by an analog-to-digital

converter at a fixed interval (period) in time. This results in a table of numbers. When the sound is played back by the Amiga, the table is fed by a DMA channel into one of the four digital-to-analog converters in the custom chips. The digital-to-analog converter converts the samples into voltages that can be played through amplifiers and loudspeakers, reproducing the sound.

On the Amiga you can create sound data in many other ways. For instance, you can use trigonometric functions in your programs to create the more traditional sounds—sine waves, square waves, or triangle waves by using tables that describe their shape. Then you can combine these waves for richer sound effects by adding the tables together. Once the data is entered, you can modify it with techniques described in section 1.3, "Audio Functions and Commands".

For information about the limitations of the audio hardware and suggestions for improving system efficiency and sound quality, refer to the *Amiga Hardware Reference Manual*.

The following works are recommended for information about computer sound generation in general:

- *Musical Applications of Microprocessors* by Hal Chamberlain (Hayden, 1980)
- *Foundations of Computer Music* by Curtis Roads and John Strawn (Cambridge: MIT Press, 1985)
- *Digital Audio Signal Processing* by John Strawn (Los Altos, California: William Kaufmann, Inc., 1985)

1.2. DEFINITIONS

Some of the terms used in the following discussions may be unfamiliar. Some of the more important terms are defined below.

buffer
: An area of continuous memory, typically used for storing blocks of data.

amplitude
: The height of a waveform, corresponds to the amount of voltage or current in the electronic circuit.

amplitude modulation
: A means of producing special audio effects by using one channel to alter the amplitude of another.

cycle
: One repetition of a waveform.

channel
: One "unit" of the audio device.

frequency
: The number of times per second a waveform repeats.

frequency modulation
: A means of producing special audio effects by using one channel to affect the period of the waveform produced by another channel.

period
: The time elapsed between the output of successive sound samples, in units of system clock ticks.

precedence
: Priority of the user of a sound channel.

sample
: Byte of audio data, one of the fixed-interval points on the waveform.

volume
: The decibel level of sound coming from an audio channel.

waveform
: Graph that shows a model of how the amplitude of a sound varies over time—usually over one cycle.

1.3. AUDIO FUNCTIONS AND COMMANDS

The first part of this section gives some general information about audio functions and commands. Following the general information there is a brief description of each command. For complete specifications, see the command and function reference section and the header files, *audio.i* and *audio.h* in the appendixes to this manual.

Audio as a Device

The audio device has much in common with the other I/O devices, so general information about device I/O is not repeated here. Before reading further, you should become familiar with the general description of device I/O in the chapter in Part I called "I/O".

Audio device commands use an extended **IORequest** block instead of the standard **IORequest** block. When using an audio command, refer to the *audio.i* and *audio.h* files for the extended fields.

Scope of Commands

All audio commands (except for CMD_WRITE, ADCMD_WAITCYCLE, and CMD_READ) can operate on multiple channels. CMD_WRITE, ADCMD_WAITCYCLE, and CMD_READ operate on only one channel.

You tell the audio device driver which channels you want a command to act upon by setting the least significant four bits of the **io_unit** field of the **IORequest** block. You specify a 1 in the position of the channel you want to affect and a 0 in all other positions. For instance, you specify 5 to use channels 0 and 2:

```
0101
```

Certain of the audio device commands are actually higher level functions in that they execute more than one audio device command with a single call. For example, the **OpenDevice()** function, when used for the audio device, can perform an ADCMD_ALLOCATE command so that you can start writing data immediately. The **CloseDevice()** function can perform a ADCMD_FREE command to relinquish the channel(s) so you can exit immediately after closing the audio device.

Allocation and Arbitration

You request the use of one or more audio channels by performing the ADCMD_ALLOCATE command. If possible, ADCMD_ALLOCATE obtains the channels for you. When you request a channel, you specify a precedence number from -128 (the lowest precedence) to 127 (the highest). If a channel you want is being used and you have specified a higher precedence than the current user, ADCMD_ALLOCATE will "steal" the channel from the other user. Later on, if your precedence is lower than that of another user who is performing an allocation, the channel may be stolen from you. If, after allocating a channel with the appropriate precedence, you raise the precedence to the maximum precedence with the ADCMD_SETPREC command, then no other allocation call can steal a channel from you. When you've finished with a channel, you must relinquish it with the ADCMD_FREE command to make it available for other users.

Table 1-1 shows suggested precedence values.

Table 1-1: Suggested Precedences for Channel Allocation

Predecence	Type of Sound
127	*Unstoppable.* Sounds first allocated at lower precedence, then set to this highest level.
90 - 100	*Emergencies.* Alert, urgent situation that requires immediate action.
80 - 90	*Annunciators.* Attention, bell (CTRL-G).
75	*Speech.* Synthesized or recorded speech (narrator.device).
50 - 70	*Sonic cues.* Sounds that provide information that is not provided by graphics. Only the beginning of each sound (enough to recognize it) should be at this level; the rest should be set to sound effects level.
-50 - 50	*Music program.* Musical notes in music-oriented program. The higher levels should be used for the attack portions of each note. Notes should separately allocate channels at the start and free them at the end.
-70 - 0	*Sound effects.* Sounds used in conjunction with graphics. More important sounds should use higher levels.
-100 - -80	*Background.* Theme music and restartable background sounds.
-128	*Silence.* Lowest level (freeing the channel completely is preferred).

When you first perform a channel allocation request, the audio device provides you with an "allocation key" that is unique to the granting of your current allocation request. The allocation key is also copied in the **ioa_AllocKey** field of your I/O control block and is used by all audio commands. Later, as you queue output requests to the audio device, the device can compare the allocation key in your request block to the key currently assigned for that channel (or channels). If the channel is stolen from you by another channel user that has a higher precedence, the copy of the key maintained by the audio channel is changed. If you attempt to perform a command on a channel that has been stolen from you, an AUDIO_NOALLOCATION error is returned and the bit in the **io_unit** field corresponding to the stolen channel is cleared so you know which channel was stolen.

There is no specific separate "audio resource". Instead, the audio device, with its allocation key management, arbitrates the use of the physical audio resources.

Performing Audio Commands

To perform an audio command, sometimes you must use the system function **BeginIO()** rather than **SendIO()** or **DoIO()**. This is because the latter two functions clear the device-specific bits in the **io_Flags** field of the **IORequest** (bits 4 thru 7). Some of the audio commands use these bits to select options. If you use **SendIO()** or **DoIO()**, the flags will be set to 0 (FALSE), which may not be desirable.

Command Types

Commands and functions for audio use can be divided into three categories: system functions, allocation/arbitration commands, and hardware control commands. There are also three audio device flags.

The system functions are

- **OpenDevice()**
- **CloseDevice()**
- **BeginIO()**
- **AbortIO()**

The allocation/arbitration commands are

- ADCMD_ALLOCATE
- ADCMD_FREE
- ADCMD_SETPREC
- ADCMD_LOCK

The hardware control commands are

- CMD_WRITE
- ADCMD_FINISH
- ADCMD_PERVOL
- CMD_FLUSH
- CMD_RESET
- ADCMD_WAITCYCLE
- CMD_STOP
- CMD_START
- CMD_READ

The following paragraphs describe each function and command.

System Functions

These are standard Amiga device functions. They are used for communication with the device.

OpenDevice()

The audio device adds to the normal operation of this function. When you open the audio device with a nonzero **ioa_Length** field, then **OpenDevice()** will attempt to allocate channels based on allocation mask just as if you called the ADCMD_ALLOCATE command. This allocation is done with the ADIOF_NOWAIT flag set, so ADCMD_ALLOCATE will return immediately if it fails. If you are opening the device and are not ready to have a channel allocated to you just then, set the **ioa_Length** field to zero.

CloseDevice()

When used with the audio device, **CloseDevice()** performs an ADCMD_FREE command on any channels selected by the **io_Unit** field. If you have different allocation keys for the channels you are using, you can't use this function to close all of them at once. Instead, you will have to issue one ADCMD_FREE command for each unique allocation that you are using. After issuing the ADCMD_FREE command(s), you can call **CloseDevice()**.

BeginIO()

This function differs from normal only by taking a pointer to an **IOAudio** structure as its only argument.

AbortIO()

This function can be used to cancel requests for ADCMD_ALLOCATE, ADCMD_LOCK, CMD_WRITE, or ADCMD_WAITCYCLE. When used with the audio device, **AbortIO()** always succeeds.

Allocation/Arbitration Commands

These commands allow the audio channels to be shared among different tasks and programs. None of these commands can be called from interrupt code.

ADCMD_ALLOCATE

This command gives access to channels. You perform this command with a pointer to a data array that describes the channels you want to allocate. For example, if you want a pair of stereo channels and you have no preference about which of the left and right channels the system will choose for the allocation, you can pass the command a pointer to an array containing 3, 5, 10, and 12. Channels 0 and 3 output sound on the left side, and channels 1 and 2 on the right side. The following table shows how this array corresponds to all the possible combinations of a right and a left channel.

Table 1-2: Possible Channel Combinations

Channel 3 left	**Channel 2 right**	**Channel 1 right**	**Channel 0 left**	Decimal **Value of Allocation Mask**
0	0	1	1	3
0	1	0	1	5
1	0	1	0	10
1	1	0	0	12

How ADCMD_ALLOCATE Operates

The ADCMD_ALLOCATE command tries the first combination, 3, to see if channels 0 and 1 are not being used. If they are available, the 3 is copied into the **io_unit** field and you get an allocation key for these channels. You copy the key into other I/O blocks for the other commands you may want to perform using these channels. When finished with the channels, you perform the ADCMD_FREE command. If channels 0 and 1 are being used, ADCMD_ALLOCATE tries the other combinations in turn. If all the combinations are in use, ADCMD_ALLOCATE checks the precedence number of the users of the channels and finds the combination that requires it to steal the channel or channels of the lowest precedence. If all the combinations require stealing a channel or channels of equal or higher precedence, then the I/O request ADCMD_ALLOCATE fails. Precedence is in the **ln_Pri** field of the **io_Message** in the **IORequest** block you pass to ADCMD_ALLOCATE; it has a value from -128 to 127.

The ADIOF_NOWAIT Flag

If you need to produce a sound right now and otherwise you don't want to allocate, set the ADIOF_NOWAIT flag to 1. This will cause the command to return an IOERR_ALLOCFAILED error if it cannot allocate any of the channels. If you are producing a non-urgent sound and you can wait, set the ADIOF_NOWAIT flag to 0. Then, the **IORequest** block returns only when you gets the allocation. If ADIOF_NOWAIT is set to 0, the audio device will continue to retry the allocation request whenever channels are freed until it is successful. If the program decides to cancel the request, **AbortIO()** can be used.

ADCMD_ALLOCATE Examples

Following are some more examples of how to tell ADCMD_ALLOCATE your channel preferences. If you want any channel but want to try to get a left channel first, use an array containing 1, 8, 2, and 4:

```
0001
1000
0010
0100
```

If you want only a left channel, use 1 and 8 (channels 0 and 3)

```
0001
1000
```

For a right channel, use 2 and 4 (channels 1 and 2):

```
0010
0100
```

To produce special effects, such as hardware-controlled amplitude and frequency modulation, you may need to allocate channels that can be "attached" to each other. The following allocation map specifies the allowable combinations. (For further information about amplitude and frequency modulation, see the *Amiga Hardware Reference Manual.*)

0011	3
0110	6
1100	12

If you want all the channels:

1111	15

If you want to allocate a channel and keep it for a sound that can be interrupted and restarted, allocate it at a certain precedence. If it gets stolen, allocate it again with the ADIOF_NOWAIT flag set to 0. When the channel is relinquished, you will get it again.

The Allocation Key

If you want to perform multi-channel commands, all the channels must have the same key since the **IORequest** block has only one allocation key field. The channels must all have that same key even when they were not allocated simultaneously. If you want to use a key you already have, you can pass in that key in the allocation key field and ADCMD_ALLOCATE can allocate other channels with that existing key. The ADCMD_ALLOCATE command only returns you a new and unique key if you pass in a zero in the allocation key field.

ADCMD_FREE

ADCMD_FREE is the opposite of ADCMD_ALLOCATE. When you perform ADCMD_FREE on a channel, it does a CMD_RESET command on the hardware and "unlocks" the channel. It also checks to see if there are other pending allocation requests. You do not need to perform ADCMD_FREE on channels stolen from you.

ADCMD_SETPREC

This command changes the precedence of an allocated channel. As an example of the use of ADCMD_SETPREC, assume that you are making sound of a chime that takes a long time to decay. It is important that user hears the chime but not so important that he hears it decay all the way. You could lower precedence after the initial attack portion of the sound to let another program steal the channel. You can also set the precedence to maximum (127) if you cannot have the channel(s) stolen from you.

ADCMD_LOCK

The ADCMD_LOCK command performs the "steal verify" function. When a user is attempting to steal a channel or channels, ADCMD_LOCK gives you a chance to clean up before the channel is stolen. You perform a ADCMD_LOCK command right after the ADCMD_ALLOCATE command. ADCMD_LOCK does not return until a higher priority user attempts to steal the channel(s) or you perform an ADCMD_FREE command. If someone is attempting to steal, you must finish up and ADCMD_FREE the channel as quickly as possible.

ADCMD_LOCK is only necessary if you want to store directly to the hardware registers instead of using the device commands. If your channel is stolen, you don't get notified without the ADCMD_LOCK command, and this could cause problems for the user who has stolen the channel and is now using it. ADCMD_LOCK sets a switch that is not cleared until you perform an ADCMD_FREE command on the channel. Canceling an ADCMD_LOCK request with **AbortIO()** will not free the channel.

The following outline describes how ADCMD_LOCK works when a channel is stolen and when it is not stolen.

1. User A allocates a channel.
2. User A locks the channel.

If User B allocates the channel with a higher precedence:

3. User B's ADCMD_ALLOCATE command is suspended (regardless of the setting of the ADIOF_NOWAIT flag).

4. User A's ADCMD_LOCK command is replied with an error (ADIOF_CHANNELSTOLEN).

5. User A does whatever is needed to finish up when a channel is stolen.

6. User A frees the channel with ADCMD_FREE.

7. User B's ADCMD_ALLOCATE command is replied. Now User B has the channel.

If the channel is not allocated by another user:

3. User A finishes the sound.

4. User A performs the ADCMD_FREE command.

5. User A's ADCMD_LOCK command is replied.

Never make the freeing of a channel (if the channel is stolen) dependent on allocating another channel. This may cause a deadlock. To keep a channel and never let it be stolen, set precedence to maximum (127). Don't use a lock for this purpose.

Hardware Control Commands

These commands change hardware registers and affect the actual sound output.

CMD_WRITE

This is a single-channel command and is the main command for making sounds. You pass the following to CMD_WRITE:

- A pointer to the waveform to be played (must start on a word boundary and must be in memory accessible by the custom chips, MEMF_CHIP),
- The length of the waveform in bytes (must be an even number), and
- A count of how many times you want to play the waveform.

If the count is 0, CMD_WRITE will play the waveform from beginning to end, then repeat the waveform continuously until something aborts it.

If you want period and volume to be set at the start of the sound, you set the WRITE command's ADIOF_PERVOL flag. If you don't, the previous volume and period for that channel will be used. This is one of the flags that would be cleared by **DoIO()** and **SendIO()**. The **ioa_WriteMsg** field in the **IORequest** block is an extra message field that can be replied at the start of the CMD_WRITE. This second message is used only to tell you when the CMD_WRITE command *starts* processing, and is used only when the ADIOF_WRITEMSG flag is set to 1.

If a CMD_STOP has been performed, the CMD_WRITE requests are queued up.

The CMD_WRITE command does not make its own copy of the waveform, so any modification of the waveform before the CMD_WRITE command is finished may affect the sound. This is sometimes desirable for special effects.

To splice together two waveforms without clicks or pops, you must send a separate, second CMD_WRITE command while the first is still in progress. This technique is used in double-buffering, which is described below.

Double-Buffering

By using two waveform buffers and two CMD_WRITE requests you can compute a waveform continuously. This is called double-buffering. The following describes how you use double-buffering.

1. Compute a waveform in memory buffer A.
2. Issue CMD_WRITE command A with **io_Data** pointing to buffer A.
3. Continue the waveform in memory buffer B.
4. Issue CMD_WRITE command B with **io_Data** pointing to Buffer B.

5. Wait for CMD_WRITE command A to finish.

6. Continue the waveform in memory buffer A.

7. Issue CMD_WRITE command A with **io_Data** pointing to Buffer A.

8. Wait for CMD_WRITE command B to finish.

9. Loop back to step 3 until the waveform is finished.

10. At the end, remember to wait until both CMD_WRITE command A and CMD_WRITE command B are finished.

ADCMD_FINISH

The ADCMD_FINISH command aborts (calls **AbortIO()**) the current write request on a channel or channels. This is useful if you have something playing, such as a long buffer or some repetitions of a buffer, and you want to stop it.

ADCMD_FINISH has a flag you can set (ADIOF_SYNCCYCLE) that allows the waveform to finish the current cycle before aborting it. This is useful for splicing together sounds at zero crossings or some other place in the waveform where the amplitude at the end of one waveform matches the amplitude at the beginning of the next. Zero crossings are positions within the waveform at which the amplitude is zero. Splicing at zero crossings gives you fewer clicks and pops when the audio channel is turned off or the volume is changed.

ADCMD_PERVOL

ADCMD_PERVOL lets you change the volume and period of a CMD_WRITE that is in progress. The change can take place immediately or you can set the ADIOF_SYNCCYCLE flag to have the change occur at the end of the cycle.

This is useful to produce vibratos, glissandos, tremolos, and volume envelopes in music or to change the volume of a sound.

CMD_FLUSH

CMD_FLUSH aborts (calls **AbortIO()**) all CMD_WRITEs and all ADCMD_WAITCYCLEs that are queued up for the channel or channels. It does not abort ADCMD_LOCKs (only ADCMD_FREE clears locks).

CMD_RESET

CMD_RESET restores all the audio hardware registers. It clears the attach bits, restores the audio interrupt vectors if the programmer has changed them, and performs the CMD_FLUSH command to cancel all requests to the channels. CMD_RESET also unstops channels that have had a CMD_STOP performed on them.

CMD_RESET does not unlock channels that have been locked by ADCMD_LOCK.

ADCMD_WAITCYCLE

This is a single-channel command. ADCMD_WAITCYCLE is replied when the current cycle has completed, that is, after the current CMD_WRITE command has reached the end of the current waveform it is playing. If there is no CMD_WRITE in progress, it returns immediately.

CMD_STOP

This command stops the current write cycle immediately. If there are no CMD_WRITEs in progress, it sets a flag so any future CMD_WRITEs are queued up and don't begin processing (playing).

CMD_START

CMD_START undoes the CMD_STOP command. Any cycles that were stopped by the CMD_STOP command are actually lost because of the impossibility of determining exactly where the DMA ceased. If the CMD_WRITE command was playing two cycles and the first one was playing when CMD_STOP was issued, the first one is lost and the second one will be played.

This command is also useful when you are playing the same wave form with the same period out of multiple channels. If the channels are stopped, when the CMD_WRITE commands are issued, CMD_START exactly synchronizes them, avoiding cancellation and distortion. When channels are allocated, they are effectively started by the CMD_START command.

CMD_READ

CMD_READ is a single-channel command. Its only function is to return a pointer to the current CMD_WRITE command. It enables you to determine which request is being processed.

1.4. EXAMPLE PROGRAMS

1.5. Stereo Sound Example

This program demonstrates allocating a stereo pair of channels using the allocation/arbitration commands. For simplicity, it uses no hardware control commands and writes directly to the hardware registers. To prevent another task from stealing the channels before writing to the registers it locks the channels.

```
/************************************************************
 *
 * Stereo Sound Example
 *
 * Sam Dicker
 * 3 December 1985
 * (created:  17 October 1985)
 *
 ************************************************************/

#include "exec/types.h"
#include "exec/memory.h"
#include "hardware/custom.h"
#include "hardware/dmabits.h"
#include "libraries/dos.h"
#include "devices/audio.h"

/* audio channel assignment */
#define LEFT0B          0
#define RIGHT0B         1
#define RIGHT1B         2
#define LEFT1B          3
#define LEFT0F          1
#define RIGHT0F         2
#define RIGHT1F         4
#define LEFT1F          8

/* used by example sound */
#define WAVELENGTH    2
#define CLOCK       3579545
#define LEFTFREQ   50.0
#define RIGHTFREQ 50.1
#define MAXVOLUME     64
#define SOUNDPREC     -40
```

```
extern struct MsgPort *CreatePort();
extern struct AudChannel aud[];
extern UWORD dmacon;

/* four possible stereo pairs */
UBYTE allocationMap[] = {
    LEFT0F | RIGHT0F,
    LEFT0F | RIGHT1F,
    LEFT1F | RIGHT0F,
    LEFT1F | RIGHT1F
};

struct IOAudio *allocIOB = 0; /* used by cleanUp to determine
                               * what needs to be 'cleaned up' */
struct IOAudio *lockIOB = 0;
struct Device *device = 0;
struct MsgPort *port = 0;
BYTE *squareWaveData = 0;


main()
{
    UBYTE channels;
    struct AudChannel *leftRegs, *rightRegs;

    /* allocate I/O blocks from chip public memory and initialize to zero */

    if (((allocIOB = (struct IOAudio *)AllocMem(sizeof(struct IOAudio),
            MEMF_PUBLIC | MEMF_CLEAR)) == 0) ||
            ((lockIOB = (struct IOAudio *)AllocMem(sizeof(struct IOAudio),
            MEMF_PUBLIC | MEMF_CLEAR)) == 0))
        cleanUp("Out of memory");

    /* open the audio device */

    if (OpenDevice(AUDIONAME, 0, allocIOB, 0) != 0)
        cleanUp("Cannot open audio device");
    device = allocIOB->ioa_Request.io_Device;

    /* initialize I/O block for channel allocation */

    allocIOB->ioa_Request.io_Message.mn_Node.ln_Pri = SOUNDPREC;
    if ((port = CreatePort("sound example", 0)) == 0)
        cleanUp("Cannot create message port");
    allocIOB->ioa_Request.io_Message.mn_ReplyPort = port;
    allocIOB->ioa_Request.io_Command = ADCMD_ALLOCATE;

    /* if no channel is available immediately, abandon allocation */
    allocIOB->ioa_Request.io_Flags = ADIOF_NOWAIT;
    allocIOB->ioa_Data = allocationMap;
```

```
allocIOB->ioa_Length = sizeof(allocationMap);

/* allocate channels now.  Alternatively, ADCMD_ALLOCATE could have been
 * preformed when audio was first OpenDevice'd by setting up ioa_Data and
 * ioa_Length before OpenDevice'ing */

BeginIO(allocIOB);
if (WaitIO(allocIOB))
   cleanUp("Channel allocation failed");

/* initialize I/O block for to lock channels */

lockIOB->ioa_Request.io_Message.mn_ReplyPort = port;
lockIOB->ioa_Request.io_Device = device;

/* one lock command to lock both channels */
lockIOB->ioa_Request.io_Unit = allocIOB->ioa_Request.io_Unit;
lockIOB->ioa_Request.io_Command = ADCMD_LOCK;
lockIOB->ioa_AllocKey = allocIOB->ioa_AllocKey;

/* lock the channels */
SendIO(lockIOB);

/* if lock returned there is an error */
if (CheckIO(lockIOB))

   /* the channel must have been stolen */
   cleanUp("Channel stolen");

/* compute the hardware register addresses */

channels = (ULONG)(allocIOB->ioa_Request.io_Unit);
leftRegs = (channels & LEFT0F) ? &aud[LEFT0B] : &aud[LEFT1B];
rightRegs = (channels & RIGHT0F) ? &aud[RIGHT0B] : &aud[RIGHT1B];

/* allocate waveform memory from chip addressable ram.  AllocMem always
 * allocates memory on a word boundary which is necessary for audio
 * waveform data */

if ((squareWaveData = (BYTE *)AllocMem(WAVELENGTH, MEMF_CHIP)) == 0)
   cleanUp("Out of memory");

/* a two cycle square wave (how complex!) */

squareWaveData[0] = 127;
squareWaveData[1] = -127;

/* these registers are described in detail in the Amiga Hardware Manual */

/* write only hardware registers must be loaded separately.
```

```
    * <reg1> = <reg2> = <data>  may not work with some compilers */
    leftRegs->ac_ptr = (UWORD *)squareWaveData;
    rightRegs->ac_ptr = (UWORD *)squareWaveData;
    leftRegs->ac_len = WAVELENGTH / 2;
    rightRegs->ac_len = WAVELENGTH / 2;

    /* a slightly different frequency is used in each channel to make the
     * sound a bit more interesting */

    leftRegs->ac_per = CLOCK / LEFTFREQ / WAVELENGTH;
    rightRegs->ac_per = CLOCK / RIGHTFREQ / WAVELENGTH;

    leftRegs->ac_vol = MAXVOLUME;
    rightRegs->ac_vol = MAXVOLUME;
    dmacon = DMAF_SETCLR | channels << DMAB_AUD0;

    /* play sound until the user press CTRL-C or lock is replied*/

    puts("Press CTRL-C to stop");
    while(Wait(SIGBREAKF_CTRL_C | 1 << port->mp_SigBit) != SIGBREAKF_CTRL_C)

        /* each time the port signals, check if lock is replied
         * (a signal is not guaranteed to be valid) */

        if (CheckIO(lockIOB)) {
            puts("Channel stolen");
            break;
        }

    /* free any allocated audio channels.  In this instance explicitly
     * performing the ADCMD_FREE command is unnecessary.  CloseDevice'ing
     * with allocIOB performs it and frees the channels automaticly */

    allocIOB->ioa_Request.io_Command = ADCMD_FREE;
    DoIO(allocIOB);

    /* free up resources and exit */
    cleanUp("");
}


/* print an error message and free allocated resources */

cleanUp(message)
TEXT *message;
{
    puts(message);
    if (squareWaveData != 0)
        FreeMem(squareWaveData, WAVELENGTH);
    if (port != 0)
```

```
    DeletePort(port);
  if (device != 0)
    CloseDevice(allocIOB);
  if (lockIOB != 0)
    FreeMem(lockIOB, sizeof(struct IOAudio));
  if (allocIOB != 0)
    FreeMem(allocIOB, sizeof(struct IOAudio));
  exit();
}
```

Double-duffered Sound Synthesis Example

This program demonstrates double-buffered writing to an audio channel using the hardware control commands. This technique can be used to synthesize sound in "real-time". This program uses the mouse as a simple input device and to keep the example simple, directly reads the mouse register.

Real-time synthesis code should always be written in the fastest assembly language possible (unlike this example) and should try to pre-compute as much data as possible. In this example, a sine wave lookup table is pre-computed. Then, while the sound is being played, the table is scanned at a rate dependent on a variable (frequency) and the scanned values are copied into temporary buffers. This frequency variable is modified by mouse movement, effectively making the mouse a pitch control. In a "real" program, since pitch is the only parameter being controlled, it would be much more efficient to modify the "period" and play one fixed sine wave waveform buffer (or one waveform for each octave).

Two temporary buffers are used. One must be computed and sent to the audio device before the other one has finished playing. Otherwise, the audio device turns off the sound, making a pop. This program runs in software interrupts to insure that it gets adequate processor time to avoid this problem.

```
/*********************************************************
 *
 * Double-Buffered Sound Synthesis Example
 *
 * Sam Dicker
 * 3 December 1985 (created:  8 October 1985)
 *
 *********************************************************/

#include "exec/types.h"
#include "exec/memory.h"
#include "exec/interrupts.h"
#include "exec/errors.h"
#include "hardware/custom.h"
#include "libraries/dos.h"
#include "devices/audio.h"

#define BUFFERSIZE250
#define SINETABLEPOWER2    10
#define SINETABLESIZE   (1 << SINETABLEPOWER2)
#define SINETABLESTEP (2 * 3.141593 / SINETABLESIZE)

/* mouse register addresses */
#define XMOUSEREG       (*((BYTE *)&joy0dat + 1))
#define YMOUSEREG       (-(*(BYTE *)&joy0dat))

extern struct MsgPort *CreatePort();
extern struct Library *OpenLibrary();
extern struct Task *FindTask();
extern UWORD joy0dat;

/* channel allocation map */
UBYTE allocationMap[] = { 1, 8, 2, 4 };

struct Library *MathBase = 0;       /* used by cleanUp to determine
                                     * what needs to be 'cleaned up' */
struct MsgPort *allocPort = 0;
struct IOAudio *allocIOB = 0;
struct Device *device = 0;
struct Interrupt *interrupt = 0;
struct MsgPort *soundPort = 0;
BYTE *buffer[2] = { 0 };
struct IOAudio *soundIOB[2] = { 0 };

int newBuffer();
UBYTE sineTable[SINETABLESIZE];
ULONG angle = 0;
ULONG frequency = 0x2000000;
BYTE lastYMouse;
```

```
main()
{
    int i;
    FLOAT sine = 0.0;
    FLOAT cosine = 1.0;

    /* open the math library */

    if ((MathBase = OpenLibrary("mathffp.library", 0)) == 0)
        cleanUp("Cannot open math library");

    /* generate the sine lookup table */

    for (i = 0; i < SINETABLESIZE; ++i) {

        /* generate table values between -128 and 127 */
        sineTable[i] = 127 * sine + 0.5;

        /* compute the next point in the table.  The table could have been
         * computed by calling the 'sin' function for each point, but this
         * method is a little faster where great accuracy is not required */
        sine += SINETABLESTEP * (cosine -= SINETABLESTEP * sine);
    }

    /* read the starting mouse count */
    lastYMouse = YMOUSEREG;

    /* initialize I/O block to allocate a channel when the audio device is
     * OpenDevice'd */

    if ((allocPort = CreatePort("sound example", 0)) == 0)
        cleanUp("Cannot create reply port");
    if ((allocIOB = (struct IOAudio *)AllocMem(sizeof(struct IOAudio),
            MEMF_PUBLIC | MEMF_CLEAR)) == 0)
        cleanUp("Out of memory");

    /* allocation precedence */
    allocIOB->ioa_Request.io_Message.mn_Node.ln_Pri = -40;

    allocIOB->ioa_Request.io_Message.mn_ReplyPort = allocPort;

    /* allocate from any channel */
    allocIOB->ioa_Data = allocationMap;
    allocIOB->ioa_Length = sizeof(allocationMap);

    /* open the audio device with channel allocation and check for errors */

    switch (OpenDevice(AUDIONAME, 0, allocIOB, 0)) {
    case IOERR_OPENFAIL:
        cleanUp("Cannot open audio device");
```

```
case ADIOERR_ALLOCFAILED:
   cleanUp("Cannot allocate audio channel");
}
device = allocIOB->ioa_Request.io_Device;

/* initialize the software interrupt structure */

if ((interrupt = (struct Interrupt *)AllocMem(sizeof(struct Interrupt),
      MEMF_CLEAR | MEMF_PUBLIC)) == 0)
   cleanUp("Out of memory");
interrupt->is_Code = (VOID (*)())newBuffer;

/* initialize the reply port for CMD_WRITE's to generate software
   interrupts */

if ((soundPort = (struct MsgPort *)AllocMem(sizeof(struct MsgPort),
      MEMF_CLEAR | MEMF_PUBLIC)) == 0)
   cleanUp("Out of memory");
soundPort->mp_Flags = PA_SOFTINT;
soundPort->mp_SigTask = (struct Task *)interrupt;
soundPort->mp_Node.ln_Type = NT_MSGPORT;
NewList(&soundPort->mp_MsgList);

/* initialize both I/O blocks for the CMD_WRITES */

for (i = 0; i < 2; ++i) {

   /* allocate waveform memory from chip addressable ram.  AllocMem
    * always allocates memory on a word boundary which is necessary
    * for audio waveform data */
   if ((buffer[i] = (BYTE *)AllocMem(BUFFERSIZE, MEMF_CHIP))
         == 0)
      cleanUp("Out of memory");

   if ((soundIOB[i] = (struct IOAudio *)AllocMem(sizeof(struct IOAudio),
         MEMF_PUBLIC | MEMF_CLEAR)) == 0)
      cleanUp("Out of memory");
   soundIOB[i]->ioa_Request.io_Message.mn_ReplyPort = soundPort;
   soundIOB[i]->ioa_Request.io_Device = device;
   soundIOB[i]->ioa_Request.io_Unit = allocIOB->ioa_Request.io_Unit;
   soundIOB[i]->ioa_Request.io_Command = CMD_WRITE;

   /* load the volume and period registers */
   soundIOB[i]->ioa_Request.io_Flags = ADIOF_PERVOL;

   soundIOB[i]->ioa_AllocKey = allocIOB->ioa_AllocKey;
   soundIOB[i]->ioa_Data = buffer[i];
   soundIOB[i]->ioa_Length = BUFFERSIZE;

   /* some arbitrary period and volume */
```

```
        soundIOB[i]->ioa_Period = 200;
        soundIOB[i]->ioa_Volume = 64;

        /* play one cycle of each buffer, then reply */
        soundIOB[i]->ioa_Cycles = 1;

        /* this really "primes the pump" by causing the reply port
         * to generate a software interrupt and write the first buffers */
        ReplyMsg(soundIOB[i]);
    }

    /* wait for CTRL-C to stop the program */

    puts("Press CTRL-C to stop");
    Wait(SIGBREAKF_CTRL_C);

    /* free up resources and exit */
    cleanUp("");
}

/* print an error message and free allocated resources */

cleanUp(message)
TEXT *message;
{
    int i;

    puts(message);
    if (device != 0)

        /* CloseDevice'ing with 'allocIOB' preforms an ADCMD_FREE on any
         * channel allocated with 'allocIOB's ioa_AllocKey.  ADCMD_FREE
         * performs a CMD_RESET, which performs a CMD_FLUSH, which AbortIO's
         * any CMD_WRITES to those channels */
        CloseDevice(allocIOB);

    for (i = 0; i < 2; ++i) {
        if (soundIOB[i])
            FreeMem(soundIOB[i], sizeof(struct IOAudio));
        if (buffer[i])
            FreeMem(buffer[i], BUFFERSIZE);
    }
    if (soundPort)
        FreeMem(soundPort, sizeof(struct MsgPort));
    if (interrupt)
        FreeMem(interrupt, sizeof(struct Interrupt));
    if (allocIOB)
        FreeMem(allocIOB, sizeof(struct IOAudio));
    if (allocPort)
        DeletePort(allocPort, sizeof(struct MsgPort));
```

```
    if (MathBase)
        CloseLibrary(MathBase);
    exit();
}

/* software interrupt server code */

newBuffer()
{
    int i;
    struct IOAudio *ioa;
    BYTE *buffer;
    BYTE mouseChange, curYMouse;
    ULONG newFreq;

    /* get I/O block from reply port */
    ioa = (struct IOAudio *)GetMsg(soundPort);

    /* check if there really was an I/O blocks on the port and if there
     * are no errors. An error would indicate either the channel was
     * aborted from being stolen (IOERR_ABORTED), it stolen before the
     * write was performed and had the wrong allocation key
     * (ADIOF_NOALLOCATION), or it was aborted by being CloseDevice'd
     * In any case if there is an error do not send the next write. The
     * program will just wait around silently */
    if (ioa && ioa->ioa_Request.io_Error == 0) {

        /* determine how far the mouse has moved */

        curYMouse = YMOUSEREG;
        mouseChange = curYMouse - lastYMouse;
        lastYMouse = curYMouse;

        /* modify the frequency proportionally */
        newFreq = frequency + mouseChange * (frequency >> 6);

        /* limit the frequency range */
        if (newFreq > 0x800000 && newFreq < 0x40000000)
            frequency = newFreq;

        /* scan the table and copy each new sample into the audio waveform buffer */

        for (i = 0, buffer = ioa->ioa_Data; i < BUFFERSIZE; ++i)
            *buffer++ = sineTable[(angle += frequency) >>
                (32 - SINETABLEPOWER2)];

        /* send the write I/O block */
        BeginIO(ioa);
    }
}
```

Chapter 2

Timer Device

This chapter describes the Amiga timer device, which provides a general time delay capability.

2.1. INTRODUCTION

The timer device can signal you when *at least* a certain amount of time has passed. Because the Amiga is a multi-tasking system, the timer device cannot guarantee that exactly the specified amount of time has elapsed.

To use a timer device you open up a channel of communication to the device and send the device a message saying how much time should elapse. At the end of that time, the device returns a message to you stating that the time has elapsed.

2.2. TIMER DEVICE UNITS

There are two units in the timer device. One uses the vertical blank interrupt for its "tick" and is called UNIT_VBLANK. The other uses a programmable timer in the 8520 CIA chip and is called UNIT_MICROHZ. These are the names you use when calling **OpenDevice()**. The examples at the end of the chapter demonstrate how you call **OpenDevice()**.

The VBLANK timer unit is very stable and has a precision comparable to the vertical blanking time, that is, +/- 16.67 milliseconds. When you make a timing request, such as "signal me in 21 seconds," the reply will come in 21 +/- .017 seconds. This timer has very low overhead, and should be used for all long duration requests.

The MICROHZ timer unit uses the built-in precision hardware timers to create the timing interval you request. It accepts the same type of command—"signal me in so many seconds and microseconds." The microhertz timer has the advantage of greater resolution than the

vertical blank timer, but it has less accuracy over comparable periods of time. The microhertz timer also has much more system overhead. It is primarily useful for short burst timing where critical accuracy is not required.

2.3. SPECIFYING THE TIME REQUEST

Both timer units have identical external interfaces. Time is specified via a **timeval** structure.

```
struct timeval {
    ULONG seconds;
    ULONG micro;
};
```

The time specified is measured from the time the request is posted. For example, you must post a timer request for 30 minutes, rather than for 10:30 pm. The **micro** field is the number of microseconds in the request. Logically, seconds and microseconds are concatenated by the driver. The number of microseconds must be "normalized"; it should be a value less than one million.

The primary means of specifying a requested time is via a **timeRequest** structure. A time request consists of an **IORequest** structure followed by a **timeval** structure as shown below.

```
struct timeRequest {
   struct IORequest tr_node;
   struct timeval tr_time;
};
```

Note that the timer driver does not use a "standard extension" **IORequest** block. It only uses the base **IORequest** structure. When the specified amount of time has elapsed, the driver will send the **IORequest** back via **ReplyMsg()** (the same as all other drivers). This means that you must fill in the **ReplyPort** pointer of the **IORequest** structure if you wish to be signaled.

When you submit a timer request, the driver destroys the values you have provided in the **timeval** structure. This means that you must reinitialize the time specification before reposting the **IORequest**.

Multiple requests may be posted to the timer driver. For example, you can make 3 time requests in a row to the timer, specifying:

Signal me in 20 seconds (request 1)
Signal me in 30 seconds (request 2)
Signal me in 10 seconds (request 3)

As the timer queues these requests, it changes the time values and sorts the timer requests to service each request at the requested interval, resulting effectively in the following order:

(request 3) in now+10 seconds
(request 1) 10 seconds after request 3 is satisfied
(request 2) 10 seconds after request 1 is satisfied

A sample timer program is given at the end of this chapter.

2.4. OPENING A TIMER DEVICE

To gain access to a timer unit, you must first open that unit. This is done by using the system command **OpenDevice()**. A typical C-language call is shown below:

```
struct timereq timer_request_block
error = OpenDevice(TIMERNAME,UNIT_VBLANK,timer_request_block,0);
```

The parameters shown above are:

TIMERNAME
: a define for the null-terminated string, currently "timer.device"

unit_number
: which timer unit you wish to use. See below for definition.

timer_request_block
: the address of an **IORequest** data structure which will later be used to communicate with the device. The **OpenDevice()** command will fill in the unit and device fields of this data structure.

2.5. ADDING A TIME REQUEST

You add a timer request to the device by passing a acorrectly initialized I/O request to the timer. The code fragment below demonstrates a sample request:

```
set_timer(seconds,microseconds)
ULONG seconds, microseconds;
{
        timermsg->io_Command = TR_ADDREQUEST;
        timermsg->SECONDS = seconds;
        timermsg->MICROSECONDS = microseconds;
        DoIO(timermsg);
}
```

NOTE: Using **DoIO()** here puts your task to sleep until the time request has been satisfied (see the sample program at the end of the chapter).

If you wish to send out multiple time requests, you have to create multiple request blocks (referenced here as "timermsgs" and then use **SendIO()** to transmit each to the timer. The program named "timer2.c" demonstrates this alternate technique.

2.6. CLOSING A TIMER

After you have finished using a timer device, you should close it:

```
CloseDevice(timermsg);
```

2.7. ADDITIONAL TIMER FUNCTIONS AND COMMANDS

There are two additional timer commands (accessed as standard device commands, using an **IORequest** block as shown above) and three additional functions (accessed as though they were library functions).

The additional timer commands are:

- TR_GETSYSTIME — get the system time
- TR_SETSYSTIME — set the system time

The additional timer library-like functions are:

- **SubTime(Dest, Source)** — subtract one time request from another
- **AddTime(Dest, Source)** — add one time request to another
- result = **CmpTime(Dest, Source)** - compare the time in two time requests

System Time

The "system timer" is unrelated to the system time as it appears in the **DateStamp** command of AmigaDOS. It is provided simply for the convenience of the developer and is utilized by Intuition.

The command TR_SETSYSTIME sets the system's idea of what time it is. The system starts out at time "zero" so it is safe to set it forward to the "real" time. However, care should be taken when setting the time backwards. System time is specified as being monotonically increasing.

The time is incremented by a special power supply signal that occurs at the external line frequency. This signal is very stable over time, but it can vary by several percent over short periods of time. System time is stable to within a few seconds a day.

In addition, system time is changed every time someone asks what time it is using TR_GETSYSTIME. This way the return value of the system time is unique and unrepeating. This allows system time to be used as a unique identifier.

NOTE: The timer device set system time to zero at boot time. The DOS will set the system time when it reads in the boot disk if it has not already been set by someone else (more exactly, if the time is less than 86 400 seconds [one day]). The DOS sets the time to the last modification time of the boot disk. The time device does not interpret system time to any physical value. The DOS treats system time relative to midnight, 1 January 1978.

Here is a program which can be used to inquire the system time. Instead of using the Exec support function **CreateStdIO()** for the request block, the block is initialized "correctly" for use as a **timeval** request block. The command is executed by the timer device and, on return, the caller can find the data in his request block.

```
/* TIMER PROGRAM TO INQUIRE WHAT IS THE CURRENT "SYSTEM TIME" */

#include <exec/types.h>
#include <exec/lists.h>
#include <exec/nodes.h>
#include <exec/ports.h>
#include <exec/io.h>
#include <exec/devices.h>
#include <devices/timer.h>
#define msgblock tr.tr_node.io_Message
struct timerequest tr;

main()
{
        msgblock.mn_Node.ln_Type = NT_MESSAGE;
        msgblock.ln_Pri = 0;
        msgblock.ln_Name = NULL;
        msgblock.mn_ReplyPort = NULL;

        msgblock.io_Command = TR_GETSYSTIME;
        DoIO(&tr);

        printf("\\nSystem Time is: %ld Seconds, %ld Microseconds",
                tr.tr_time.tv_secs, tr.tr_time.tv_micro);

}
```

Using the Time Arithmetic Routines

As indicated above, the time arithmetic routines are accessed in the timer device structure as though it was a routine library. To use them, you create an **IORequest** block and open the timer. In the **IORequest** block is a pointer to the device's base address. This address is needed to access each routine as an offset—for example, **_LVOAddTime**, **_LVOSubTime**, **_LVOCmpTime**—from that base address. (See the function appendixes for these commands.)

There are C-language interface routines in amiga.lib which perform this interface task for you. They are accessed through a variable called **TimerBase**. You prepare this variable by the following method (not a complete example, only a partial example):

```
struct timeval time1, time2, time3;
SHORT result;

struct Device *TimerBase;   /* declare the interface variable */

/* see "timedelay" example above for init of "timermsg" pointer */

TimerBase = timermsg->Device;

/* now that TimerBase is initialized, it is permissable to call
 * the time comparison or time arithmetic routines */

time1.tv_secs = 3;     time1.tv_micro = 0;     /* 3.0 seconds */
time2.tv_secs = 2;     time2.tv_micro = 500000; /* 2.5 seconds */
time3.tv_secs = 1;     time2.tv_micro = 900000; /* 1.9 seconds */

/* result of this example is +1 ... first parameter has
 * greater time value than second parameter
 */
result = CmpTime( &time1, &time2 );

/* add to time1 the values in time2 */
AddTime( &time1, &time2);
/* subtract values in time3 from the value currently in time1 */
SubTime( &time1, &time3);
```

Why Use Time Arithmetic

As mentioned earlier in this section, because of the multi-tasking capability of the Amiga, the timer device can provide timings that are at least a slong as the specified amount of time. If you need more precision than this, using the system timer along with the time arithmetic routines can at least, in the long run, let you synchronize your software with this precision timer after a selected period of time.

Say, for example, that you select timer intervals so that you get 161 signals within each 3-minute span. Therefore, the **timeval** you would have selected would be 180/161 which comes out to 1 second and 118012 microseconds per interval. Considering the time it takes to set up a call to **set_timer** and delays due to task-switching (especially if the system is very busy) it is possible that after 161 timing intervals, you may be somewhat beyond the 3 minutes time. Here is a method you can use to keep in sync with system time:

0. Begin.

1. Read system time; save it.

2. Perform your loop however many times in your selected interval.

3. Read system time again, compare it to the old value you saved. (for this example, it will be more or less than 3 minutes as a total time elapsed).

4. Calculate a new value for the time interval; that is, one that (if precise) would put you exactly in sync with system time the next time around. Timeval will be a lower value if the loops took too long, and a higher value if the loops didn't take long enough.

5. Repeat the cycle.

Over the long run, then, your average number of operations within a specified period of time can become precisely what you have designed.

2.8. SAMPLE TIMER PROGRAM

Here is an example program showing how to use a timer device.

```
/* SIMPLE TIMER EXAMPLE PROGRAM:
 *
 * Includes dynamic allocation of data structures needed to communicate
 * with the timer device as well as the actual device IO
 */

#include "exec/types.h"
#include "exec/nodes.h"
#include "exec/lists.h"
#include "exec/memory.h"
#include "exec/interrupts.h"
#include "exec/ports.h"
#include "exec/libraries.h"
#include "exec/tasks.h"
#include "exec/io.h"
#include "exec/devices.h"
#include "devices/timer.h"

APTR TimerBase;          /* to get at the time comparison functions */

/* manifest constants -- "never will change" */
#define    SECSPERMIN   (60)
```

```
#define    SECSPERHOUR      (60*60)
#define    SECSPERDAY  (60*60*24)

main()
{
      LONG seconds;

      /* save what system thinks is the time.... we'll advance it
       * temporarily
       */
      struct timeval oldtimeval;
      struct timeval mytimeval;
      struct timeval currentval;

      printf("\ntimer test\n");

      /* sleep for two seconds */
      currentval.tv_secs = 2;
      currentval.tv_micro = 0;
      TimeDelay( &currentval, UNIT_VBLANK );
      printf( "After 2 seconds delay\n" );

      /* sleep for four seconds */
      currentval.tv_secs = 4;
      currentval.tv_micro = 0;
      TimeDelay( &currentval, UNIT_VBLANK );
      printf( "After 4 seconds delay\n" );

      /* sleep for 500,000 micro-seconds = 1/2 second */
      currentval.tv_secs = 0;
      currentval.tv_micro = 500000;
      TimeDelay( &currentval, UNIT_MICROHZ );
      printf( "After 1/2 second delay\n" );

      printf( "\n" );

      (void) Execute( "date", 0, 0 );

      printf( "\n" );

      GetSysTime( &oldtimeval );
      printf( "Current system time is %ld current seconds\n",
            oldtimeval.tv_secs );

      printf("Setting a new system time\n");

      seconds = 1000 * SECSPERDAY + oldtimeval.tv_secs;

      SetNewTime( seconds );
      /* (if user executes the AmigaDOS DATE command now, he will
```

```
        * see that the time has advanced something over 1000 days */

       printf( "\n" );
       (void) Execute( "date", 0, 0 );

       printf( "\n" );

       /* added the microseconds part to show that time keeps
        * increasing even though you ask many times in a row */
       GetSysTime( &mytimeval );
       printf( "Original system time is %ld.%06ld\n",
            mytimeval.tv_secs, mytimeval.tv_micro );

       GetSysTime( &mytimeval );
       printf( "First system time is %ld.%06ld\n",
            mytimeval.tv_secs, mytimeval.tv_micro );

       GetSysTime( &mytimeval );
       printf( "Second system time is %ld.%06ld\n",
            mytimeval.tv_secs, mytimeval.tv_micro );

       printf( "Resetting to former time\n" );
       SetNewTime( oldtimeval.tv_secs );

       GetSysTime( &mytimeval );
       printf( "Current system time is %ld.%06ld\n",
            mytimeval.tv_secs, mytimeval.tv_micro );

       /* just shows how to set up for using
        * the timer functions, does not demonstrate
        * the functions themselves.  (TimerBase must
        * have a legal value before AddTime, SubTime or CmpTime
        * are performed.
        */
       tr = CreateTimer( UNIT_MICROHZ );
       TimerBase = (APTR)tr->tr_node.io_Device;

       /* and how to clean up afterwards */
       TimerBase = -1;
       DelteTimer( tr );

}

extern struct MsgPort *CreatePort();
extern struct IORequest *CreateExtIO();

struct timerequest *
CreateTimer( unit )
ULONG unit;
{
```

```
    /* return a pointer to a time request.  If any problem, return NULL */

    int error;

    struct MsgPort *timerport;
    struct timerequest *timermsg;

    timermsg = (struct timerequest *)
       CreateExtIO( timerport, sizeof( struct timerequest ) );
    if( timermsg == NULL ) {
          return( NULL );
    }

    timerport = CreatePort( 0, 0 );
    if( timerport == NULL ) {
          DeleteTimer( timermsg );
          return( NULL );
    }


    error = OpenDevice( TIMERNAME, unit, timermsg, 0 );
    if( error != 0 )
    {
          DeleteTimer( timermsg );
          return( NULL );
    }

    return( timermsg );
}

/* more precise timer than AmigaDOS Delay() */
TimeDelay( tv, unit )
struct timeval *tv;
int unit;
{
    int precise;
    struct timerequest *tr;
    if( seconds < 0 || unit == UNIT_MICROHZ ) {
          /* do delay in terms of microseconds */
          /* yes, use the precision timer.     */
          unit = UNIT_MICROHZ;
    } else {
          /* use the more efficient vertical blank timer */
          unit = UNIT_VBLANK;
    }

    /* get a pointer to an initialized timer request block */
    tr = CreateTimer( unit );

    /* any nonzero return says timedelay routine didn't work. */
```

```
    if( tr == NULL ) return( -1 );

    WaitForTimer( tr, tv );

    /* deallocate temporary structures */
    DeleteTimer( tr );
    return( 0 );
}

void
WaitForTimer( tr, tv )
ULONG seconds,microseconds;
struct timerequest *tr;
struct timeval *tv;
{
    tr->tr_node.io_Command = TR_ADDREQUEST; /* add a new timer request */

    /* structure assignment */
    tr->tv_time = *tv;

    /* post request to the timer -- will go to sleep till done */
    DoIO( tr );
}

int
SetNewTime( secs )
LONG secs;      /* seconds since 1 Jan 78 */
{
    struct timerequest *tr;

    tr = CreateTimer( UNIT_MICROHZ );

    /* non zero return says error */
    if( tr == 0 ) return( -1 );

    tr->tr_node.io_Command = TR_SETSYSTIME;
    tr->tr_time.tv_secs = secs;
    tr->tr_time.tv_micro = 0;
    DoIO( tr );

    DeleteTimer(tr);
    return(0);
}

int
GetSysTime(tv)
struct timeval *tv;
{
    struct timerequest *tr;
```

```
    tr = CreateTimer( UNIT_MICROHZ );

    /* non zero return says error */
    if( tr == 0 ) return( -1 );

    tr->tr_node.io_Command = TR_GETSYSTIME;
    DoIO( tr );

    /* structure assignment */
    *tv = tr->tr_time;

    DeleteTimer( tr );
    return( 0 );

}

void
DeleteTimer( tr )
struct timerequest *tr;
{
    struct MsgPort *tp;

    if( tr != 0 )
    {
        tp = tr->tr_node.io_Message.mn_ReplyPort;
        if(tp != 0) {
            DeletePort(tp);
        }

        CloseDevice( tr );
        DeleteExtIO( tr, sizeof(struct timerequest) );
    }
}
```

Chapter 3

Trackdisk Device

This chapter covers the Amiga trackdisk device, which directly drives the disk, controls the disk motors, reads raw data from the tracks, and writes raw data to the tracks.

3.1. INTRODUCTION

Normally, you use the AmigaDOS functions to write or read data from the disk. The trackdisk driver is the lowest level software access to the disk data and is used by AmigaDOS to get its job done.

The trackdisk device supports the usual commands such as CMD_WRITE and CMD_READ. In addition, it supports an extended form of these commands to allow additional control over the disk driver.

The trackdisk device can queue up command sequences so that your task can do something else while it is waiting for a particular disk activity to occur. If several sequenced write commands are queued to a disk, a task assumes that all such writes are going to the same disk. The trackdisk driver itself can stop a command sequence if it senses that the disk has been changed, returning all subsequent **IORequest** blocks to the caller with an error ("disk changed").

When the trackdisk device is requested to provide status information for commands such as TD_REMOVE or TD_CHANGENUM, the value is returned in the **io_Actual** field of the **IORequest**.

3.2. THE AMIGA FLOPPY DISK

The Amiga floppy disk consists of NUMHEADS (2) heads, NUMCYLS (80) cylinders, and NUMSECS (11) sectors per cylinder. Each sector has TD_SECTOR (512) usable data bytes plus TD_LABELSIZE (16) of sector label area. This gives useful space of 880K bytes plus 28K bytes of label area per floppy disk.

Although the disk is logically divided up into sectors, all I/O to the disk is implemented as an entire track. This allows access to the drive with no interleaving and increases the useful storage capacity by about 20 percent. Normally, a read of a sector will only have to copy the data from the track buffer. If the track buffer contains another track's data, then the buffer will first be written back to the disk (if it is "dirty") and the new track will be read in. All track boundaries are transparent to the user. The driver ensures that the correct track is brought into memory.

The performance of the disk is greatly enhanced if you make effective use of the track buffer. The performance of sequential reads will be up to an order of magnitude greater than reads scattered across the disk.

The disk driver uses the blitter to encode and decode the data to and from the track buffer. Because the blitter can only access chip memory (memory that is accessible to the special purpose chips and within the lowest 512K bytes of the system, known as MEMF_CHIP to the memory allocator **AllocMem()**) all buffers submitted to the disk must be in chip memory. In addition, only full sector writes on sector boundaries are supported. Note also that the user's buffer must be word aligned.

The disk driver is based upon a standard driver structure. It has the following restrictions:

- All reads and writes must use an **io_Length** that is an integer multiple of TD_SECTOR bytes (the sector size in bytes).
- The offset field must be an integer multiple of TD_SECTOR.
- The data pointer must be word-aligned.
- The data pointer must be in MEMF_CHIP memory. This is because the disk driver uses the blitter to fill the data buffer.
- Only the 3 1/2-inch disk format is supported by the trackdisk driver. The 5 1/4 inch-format is supported by the IBM PC emulation software.

3.3. TRACKDISK DRIVER COMMANDS

The trackdisk driver allows the following system interface functions and commands. In addition to the usual device commands, the trackdisk driver has a set of extended commands.

The system interface functions are

OpenDevice()	Obtain exclusive use of a particular disk unit.
CloseDevice()	Release the unit to another task.
Expunge()	Remove the device from the device list.
BeginIO()	Dispatch a device command; queue commands.
AbortIO()	Abort a device command.

The device-specific commands are

CMD_READ	Read one or more sectors.
CMD_WRITE	Write one or more sectors.
CMD_UPDATE	Write out a track buffer.
CMD_CLEAR	Mark a track buffer as invalid.
TD_MOTOR	Turn the motor on or off.
TD_SEEK	Move the head to a specific track.
TD_FORMAT	Initialize one or more tracks.
TD_REMOVE	Establish a software interrupt procedure for disk removal.
TD_CHANGENUM	Discover the current disk-change number.
TD_CHANGESTATE	See if there is a disk present in a drive.
TD_PROTSTATUS	See if a disk is write-protected.

In addition to the device-specific commands listed above, the trackdisk driver has the following extended commands. These commands are similar to their normal counterparts but have the following additional features:

- They allow you to control whether a command will be executed if the disk has been changed.
- They allow you to read or write to the sector label portion of a sector.

Extended commands take a slightly larger I/O request block, which contains information that is needed only by the extended command and is ignored by the standard form of that command. The extra information takes the form of two extra longwords at the end of the data structure. These commands are performed only if the change count is less than or equal to the one in the **iotd_Count** field of the command's I/O request block.

ETD_READ	Read one or more sectors.
ETD_WRITE	Write one or more sectors.
ETD_MOTOR	Turn the motor on or off.
ETD_UPDATE	Write out a track buffer.
ETD_CLEAR	Mark a track buffer as invalid.
ETD_SEEK	Move the head to a specific track.

3.4. CREATING AN I/O REQUEST

The trackdisk device, like other devices, requires that you create an I/O request message that you pass to the device for processing. The message contains the command and several other items of control information.

Here is a program fragment that can be used to create the message block that you use for trackdisk communications. In the fragment, the routine **CreateStdIO()** is called to return a pointer to a message block. This is acceptable for the standard form of the commands. If you wish to use the extended form of the command, you will need an extended form of the request block. In place of **CreateStdIO()**, you can use the routine **CreateExtIO()**, a listing of which appears in the appendix under the heading "Exec Support Functions".

```
struct IOStdReq *diskreq;   /* I/O request block pointer for
                               non-extended commands */
struct IOExtTD *diskextreq; /* I/O request block pointer for
                               extended commands */
struct Port *diskreqPort;   /* a port at which to receive replies */

diskreqPort = CreatePort("diskreq.port",0);
if(diskreqPort == 0) exit(100); /* error in createport */
diskreq =  CreateStdIO(diskreqPort);
if(diskreq == 0) { DeletePort(diskreqPort); exit(200); }
                 /* error in createstdio */
diskextreq = CreateExtIO(diskreqPort,sizeof(struct IOExtTd));
if(diskextreq == 0) { DeletePort(diskreqPort); exit(300) };
```

The routine **CreatePort()** is part of amiga.lib. It returns a pointer to a **Port** structure that can be used to receive replies from the trackdisk driver.

The routine **CreateStdIO()** is also in amiga.lib. It returns a pointer to an **IOStdReq** block that becomes the message you pass to the trackdisk driver to tell it the command to perform. **CreateExtIO()** is provided in source form in the appendixes to this manual.

The data structure **IOExtTD** takes the form:

```
struct IOExtTD {
        struct  IOStdReq iotd_Req;
        ULONG   iotd_Count;
        ULONG   iotd_SecLabel;
};
```

where

IOStdReq
is a standard **IORequest** block that contains fields used to transmit the standard commands (explained below).

iotd_Count
helps keep old I/O requests from being performed when the diskette has been changed. All extended commands treat as an error any case where the disk change counter is greater than **iotd_Count**. Any I/O request found with an **iotd_Count** less than the current change counter value will be returned with a characteristic error (TDERR_DiskChange) in the **io_Error** field of the I/O request block. This allows stale I/O requests to be returned to the user after a disk has been changed. The current disk-change counter value can be obtained by TD_CHANGENUM.

If the user wants extended disk I/O but does not care about disk removal, then **iotd_Count** may be set to the maximum unsigned long integer value (0xFFFFFFFF).

iotd_Seclabel
allows access to the sector identification section of the sector header.

Each sector has 16 bytes of descriptive data space available to it; the disk driver does not interpret this data. If **iotd_Seclabel** is null, then this descriptive data is ignored. If it is not null, then **iotd_Seclabel** should point to a series of 16-byte chunks (one for each sector that is to be read or written). These chunks will be written out to the sector's label region on a write or filled with the sectors's label area on a read. If a CMD_WRITE (the standard write call) is done, then the sector label area is left unchanged.

3.5. OPENING A TRACKDISK DEVICE

To gain access to a disk unit, you must first open the unit by using the system command **OpenDevice()**. A typical C-language call is shown below:

```
error = OpenDevice(TD_NAME,unit_number,disk_request_block,flags);
```

where:

TD_NAME
is a define for the null-terminated string, currently "trackdisk.device".

unit_number
is the disk unit you wish to use (defined below).

disk_request_block
is the address of an **IORequest** data structure that will later be used to communicate with the device. The **OpenDevice()** command will fill in the unit and device fields of this data structure.

flags
tell how the I/O is to be accomplished. For an **OpenDevice()** command, this field is normally set to zero.

The **unit_number** can be any value from 0 to 3. Unit 0 is the built-in 3 1/2-inch disk. Units 1 through 3 represent additional 3 1/2-inch disks which may be daisy-chained from the external disk unit connector on the back of the Amiga. The first unit (plugged directly into the Amiga) is unit 1. The second unit (plugged into unit 1), is designated as unit 2. The end-unit, farthest electrically from the Amiga, is unit 3.

Following are some common errors that may be returned from an **OpenDevice()** call.

Device in use
Some other task has already been granted exclusive use of this device.

Bad unit number
Either you have specified a unit number outside the range of 0-3 or you don't have a unit connected in the specified position.

Bad device type
You may be trying to use a 5 1/4-inch drive with the trackdisk driver. This is not supported.

3.6. SENDING A COMMAND TO THE DEVICE

You send a command to this device by initializing the appropriate fields of your **IOStdReq** or **IOExtTD** and then using **SendIO()**, **DoIO()**, or **BeginIO()** to transmit the command to the device. Here is an example:

```
MotorOn()
{
        diskreq->io_Length = 1;          /* 1 says turn it on */
        diskreq->io_Command = TD_MOTOR;
        DoIO(diskreq);                   /* task sleep till command done */
        return(0);
}
```

3.7. TERMINATING ACCESS TO THE DEVICE

As with all exclusive-access devices, you *must* close the trackdisk device when you have finished using it. Otherwise, the system will be unable to allocate the device to any other task until the system is rebooted.

3.8. DEVICE-SPECIFIC COMMANDS

The following device-specific commands are supported.

ETD_READ and CMD_READ

ETD_READ obeys all of the trackdisk driver restrictions noted above. ETD_READ transfers data from the track buffer to the user's buffer, if and only if the disk has not been changed. If the desired sector is already in the track buffer, no disk activity is initiated. If the desired sector is not in the buffer, the track containing that sector is automatically read in. If the data in the current track buffer has been modified, it is written out to the disk before the new track is read. CMD_READ does not check if the disk has been changed before executing this command.

ETD_WRITE and CMD_WRITE

ETD_WRITE obeys all of the trackdisk driver restrictions noted above. ETD_WRITE transfers data from the user's buffer to track buffer if and only if the disk has not been changed. If the track that contains this sector is already in the track buffer, no disk activity is initiated. If the desired sector is not in the buffer, the track containing that sector is automatically read in. If the data in the current track buffer has been modified, it is written out to the disk before the new track is read in for modification. CMD_WRITE does not check for disk change before performing the command.

ETD_UPDATE and CMD_UPDATE

ETD_UPDATE obeys all of the trackdisk driver restrictions noted above. The Amiga trackdisk driver doesn't write data sectors unless it is necessary (you request that a different track be used) or until the user requests that an update be performed. This improves system speed by caching disk operations. Therefore, ETD_UPDATE writes out the track buffer if and only if the data in that buffer is known to have been modified since it was read in and checks that the disk hasn't been changed. ETD_UPDATE ensures that any buffered data is flushed out to the disk. CMD_UPDATE, doesn't check for a disk change before performing the update.

ETD_CLEAR and CMD_CLEAR

ETD_CLEAR marks the track buffer as invalid, forcing a re-read of the disk on the next operation.

ETD_UPDATE or CMD_UPDATE would be used to force data out to the disk before turning the motor off. ETD_CLEAR or CMD_CLEAR is usually used after the disk has been removed, to prevent caching of data to the new diskette. ETD_CLEAR or CMD_CLEAR will not do an update, nor will an update command do a clear. CMD_CLEAR doesn't check for disk change.

ETD_MOTOR and TD_MOTOR

TD_MOTOR is called with a standard **IORequest** block. The **io_Length** field contains the requested state of the motor. A 1 will turn the motor on; a 0 will turn it off. The old state of the motor is returned in **io_Actual**. If **io_Actual** is zero, then the motor was off. Any other value implies that the motor was on. If the motor is just being turned on, the driver will delay the proper amount of time to allow the drive to come up to speed. Normally, turning the drive on is not necessary—the driver does this automatically if it receives a request when the motor is off. However, turning the motor off is the user's responsibility. In addition, the standard instructions to the user are that it is safe to remove a diskette if and only if the motor is off (that is, the disk light is off).

TD_FORMAT

TD_FORMAT is used to write data to a track that either has not yet been formatted or has had a hard error on a standard write command. TD_FORMAT completely ignores all data currently on a track and doesn't check for disk change before performing the command. TD_FORMAT is called with a standard **IORequest**. The **io_Data** field must point to at least one track worth of data. The **io_Offset** field must be track aligned, and the **io_Length** field must be in units of track length (that is, NUMSECS*TD_SECTOR). The driver will format the requested tracks, filling each sector with the contents of the **io_Data** field. You should do a read pass to verify the data. The command TD_FORMAT does not check whether the disk has been changed before the command is performed.

If you have a hard write error during a normal write, you may find it necessary to use the TD_FORMAT command to reformat the track as part of your error recovery process.

TD_REMOVE

TD_REMOVE is called with a standard **IORequest**. The **APTR io_Data** field points to a software interrupt structure. The driver will post this software interrupt whenever a disk is inserted or removed. To find out the current state of the disk, TD_CHANGENUM and

TD_CHANGESTATE should be used. If TD_REMOVE is called with a null **io_Data** argument, then disk removal interrupts are suspended.

3.9. STATUS COMMANDS

The commands that return status on the current disk in the unit are TD_CHANGENUM, TD_CHANGESTATE, and TD_PROTSTATUS.

TD_CHANGENUM

TD_CHANGENUM returns the current value of the disk-change counter (as used by the extended commands—see below). The disk change counter is incremented each time the disk is inserted or removed.

TD_CHANGESTATE

TD_CHANGESTATE returns zero if a disk is currently in the drive, and nonzero if the drive has no disk.

TD_PROTSTATUS

TD_PROTSTATUS returns nonzero if the current diskette is write-protected. All these routines return their values in **io_Actual**. These routines are safe to call from an interrupt routine (such as the software interrupt specified in TD_REMOVE). However, care should be taken when calling these routines from an interrupt. You should never **Wait()** for them to complete while in interrupt processing—it is never legal to go to sleep on the interrupt stack.

3.10. COMMANDS FOR DIAGNOSTICS AND REPAIR

There is currently only one command, TD_SEEK, provided for internal diagnostics and for disk repair.

TD_SEEK is called with a standard **IORequest**. The **io_Offset** field should be set to the (byte) offset to which the seek is to occur. TD_SEEK will not verify its position until the next read. That is, TD_SEEK only moves the heads; it does not actually read any data and it does not check to see if the disk has been changed.

3.11. TRACKDISK DRIVER ERRORS

Following is a list of error codes that can be returned by the trackdisk driver. When an error occurs, these error numbers will be returned in the **io_Error** field of your **IORequest** block.

Table 3-1: Trackdisk Driver Error Codes

Error Name	Error Number	Meaning
TDERR_NotSpecified	20	Error couldn't be determined
TDERR_NoSecHdr	21	Couldn't find sector header
TDERR_BadSecPreamble	22	Error in sector preamble
TDERR_BadSecID	23	Error in sector identifier
TDERR_BadHdrSum	24	Header field has bad checksum
TDERR_BadSecSum	25	Sector data field has bad checksum
TDERR_TooFewSecs	26	Incorrect number of sectors on track
TDERR_BadSecHdr	27	Unable to read sector header
TDERR_WriteProt	28	Disk is write-protected
TDERR_DiskChanged	29	Disk has been changed or is not currently present
TDERR_SeekError	30	While verifying seek position, found seek error
TDERR_NoMem	31	Not enough memory to do this operation
TDERR_BadUnitNum	32	Bad unit number (unit # not attached)
TDERR_BadDriveType	33	Bad drive type (not an Amiga 3 1/2 inch disk)
TDERR_DriveInUse	34	Drive already in use (only one task exclusive)

3.12. EXAMPLE PROGRAM

The following sample program exercises a few of the trackdisk driver commands.

```
#include "exec/types.h"
#include "exec/nodes.h"
#include "exec/lists.h"
#include "exec/memory.h"
#include "exec/interrupts.h"
#include "exec/ports.h"
#include "exec/libraries.h"
#include "exec/io.h"
#include "exec/tasks.h"
#include "exec/execbase.h"
#include "exec/devices.h"
#include "devices/trackdisk.h"

#define TD_READ CMD_READ
#define BLOCKSIZE TD_SECTOR

SHORT error;
struct MsgPort *diskport;
struct IOExtTD *diskreq;
BYTE diskbuffer[BLOCKSIZE];
BYTE *diskdata;
SHORT testval;

extern struct MsgPort *CreatePort();
extern struct IORequest *CreateExtIO();

ULONG diskChangeCount;

ReadCylSec(cyl, sec, hd)
SHORT cyl, sec, hd;
{
        LONG offset;

        diskreq->iotd_Req.io_Length = BLOCKSIZE;
        diskreq->iotd_Req.io_Data = (APTR)diskbuffer;
                /* show where to put the data when read */
        diskreq->iotd_Req.io_Command = ETD_READ;
                /* check that disk not changed before reading */
        diskreq->iotd_Count = diskChangeCount;

        /* convert from cylinder, head, sector to byte-offset value to get
         * right one (as dos and everyone else sees it)...*/

        /* driver reads one CYLINDER at a time (head does not move for
         * 22 sequential sector reads, or better-put, head doesnt move for
         * 2 sequential full track reads.)
         */
```

```
        offset = TD_SECTOR * (sec + NUMSECS * hd + NUMSECS * NUMHEADS * cyl);
        diskreq->iotd_Req.io_Offset = offset;
        DoIO(diskreq);
        return(0);
}


MotorOn()
{
        /* TURN ON DISK MOTOR ... old motor state is returned in io_Actual */
        diskreq->iotd_Req.io_Length = 1;
        /* this says motor is to be turned on */
        diskreq->iotd_Req.io_Command = TD_MOTOR;
        /* do something with the motor */
        DoIO(diskreq);
        printf("\nOld motor state was: %ld",diskreq->iotd_Req.io_Actual);
        printf("\nio_Error value was: %ld",diskreq->iotd_Req.io_Error);
        return(0);
}

MotorOff()
{
        printf("\n\nNow turn it off");
        diskreq->iotd_Req.io_Length = 0;
        /* says that motor is to be turned on */
        diskreq->iotd_Req.io_Command = TD_MOTOR;
        /* do something with the motor */
        DoIO(diskreq);
        printf("\nOld motor state was: %ld",diskreq->iotd_Req.io_Actual);
        printf("\nio_Error value was: %ld",diskreq->iotd_Req.io_Error);
        return(0);
}

SeekFullRange(howmany)
SHORT howmany;
{
int i;
for(i=0; i<howmany; i++)
        {
                diskreq->iotd_Req.io_Offset =
                        ((NUMCYLS -1)*NUMSECS*NUMHEADS -1 ) * 512;
                /* seek to cylinder 79, head 1 */
                diskreq->iotd_Req.io_Command = TD_SEEK;
                DoIO(diskreq);
                if(diskreq->iotd_Req.io_Error != 0)
                        printf("\nSeek Cycle Number %ld, Error = %ld",
                                        i, diskreq->iotd_Req.io_Error);
                diskreq->iotd_Req.io_Offset = 0;
                        /* seek to cylinder 0, head 0 */
                diskreq->iotd_Req.io_Command = TD_SEEK;
                DoIO(diskreq);
                if(diskreq->iotd_Req.io_Error != 0)
                        printf("\nSeek Cycle Number %ld, Error = %ld",
                                        i, diskreq->iotd_Req.io_Error);
```

```
            printf("\nCompleted a seek");
        }
        return(0);
}


main()
{
        SHORT cylinder,head,sector;

        diskdata = &diskbuffer[0];
              /* point to first location in disk buffer */
        diskport = CreatePort(0,0);
        if(diskport == 0) exit(100);    /* error in createport */
        diskreq = (struct IOExtTD *)CreateExtIO(diskport,
                                        sizeof(struct IOExtTD));
              /* make an io request block for communicating with the disk */
        if(diskreq == 0) { DeletePort(diskport); exit(200); }

           error = OpenDevice(TD_NAME,0,diskreq,0);
                 /* open the device for access, unit 0 is builtin drive */
           printf("\nError value returned by OpenDevice was: %lx", error);

           /* now get the disk change value */
           diskreq->iotd_Req.io_Command = TD_CHANGENUM;
           DoIO(diskreq);
           diskChangeCount = diskreq->iotd_Req.io_Actual;
           printf("\nChange number for disk is currently %ld",diskChangeCount);

           MotorOn();
           SeekFullRange(10);
           for(cylinder=0; cylinder<80; cylinder++)      /* tracks to test */
             {
             for(head=0; head<2; head++) /* number of heads to test */
           for(sector=0; sector<11; sector++)      /* sectors to test */
                 {
                 ReadCylSec(cylinder, sector, head);
                 if(diskreq->iotd_Req.io_Error != 0)
                   printf("\nError At Cyl=%ld, Sc=%ld, Hd=%ld, Error=%ld",
                              cylinder,sector,head,
                              diskreq->iotd_Req.io_Error);
                 }
              printf("\nCompleted reading Cylinder=%ld",cylinder);
              }
           MotorOff();
           CloseDevice(diskreq);

        DeleteExtIO(diskreq, sizeof(struct IOExtTD));
        DeletePort(diskport);
}       /* end of main */
```

Chapter 4

Console Device

This chapter describes how you do console (keyboard and screen) input and output on the Amiga. The console device acts like an enhanced ASCII terminal. It obeys many of the standard ANSI sequences as well as additional special sequences unique to the Amiga.

4.1. INTRODUCTION

Console I/O is tied closely to the Amiga Intuition interface; a console must be tied to a window that is already opened. From the **Window** data structure, the console device determines how many characters it can display on a line and how many lines of text it can display in a window without clipping at any edge.

You can open the console device many times, if you wish. The result of each open call is a new console unit. AmigaDOS and Intuition see to it that only one window is the currently active one, and its console, if any, is the only one (with a few exceptions) that receives notification of input events, such as keystrokes. Later in this chapter you'll see that other Intuition events can be sensed by the console device as well.

NOTE: For this entire chapter the characters "<CSI>" represent the *control sequence introducer*. For output you may either use the two-character sequence "<Esc>[" or the one-byte value $9B (hex). For input you will receive $9B's.

4.2. SYSTEM FUNCTIONS

The various system functions, such as **DoIO()**, **SendIO()**, **AbortIO()**, **CheckIO()** and so on operate normally. The only caveats are that CMD_WRITE may cause the caller to wait internally, even with **SendIO()**, and a task waiting on response from a console is at the user's whim. If a user never reselects that window, and the console response provides the only wakeup-call, that task may well sleep indefinitely.

4.3. CONSOLE I/O

The console device may be thought of as a kind of a terminal. You send a character stream to the console device. You also receive a character stream from the console device. These streams may be characters or special sequences.

General Console Screen Output

Console character screen output (as compared to console command sequence transmission) outputs all standard printable characters (character values hex 20 thru 7E and A0 thru FF) normally. Many control characters such as BACKSPACE and RETURN are translated into their exact ANSI equivalent actions. The line-feed character is a bit different, in that it can be translated into a new-line character. The net effect is that the cursor moves to the first column of the next line whenever a <LF> is displayed. This is set via the mode control sequences discussed under "Control Sequences for Screen Output".

Console Keyboard Input

If you read from the console device, the keyboard inputs are preprocessed for you. You will get the ASCII characters such as "B". Most normal text gathering programs will read from the console device in this manner. Special programs like word processors and music keyboard programs will use raw input. Keys are converted via the keymap associated with the unit.

The sections below deal with the following topics:

- Setting up for console I/O (creating an I/O request structure)
- Writing to the console to control its behavior
- Reading from the console
- Closing down a console device

This section shows you how to set up for console I/O.

4.4. CREATING AN I/O REQUEST

Console I/O, like other devices, requires that you create an I/O request message that you pass to the console device for processing. The message contains the command, as well as a data area. In the data area, for a write, there will be a pointer to the stream of information you wish to write to the console. For a read, this data pointer shows where the console is to copy the data it has for you. There is also a length field that says how many characters (maximum) are to be copied either from or to the console device.

Here is a program fragment that can be used to create the message block that you use for console communications.

For *writing* to the console:

```
struct IOStdReq *consoleWriteMsg; /* I/O request block pointer */
struct Port *consoleWritePort;      /* a port at which to receive replies*/

consoleWritePort = CreatePort("mycon.write",0);
if(consoleWritePort == 0) exit(100);      /* error in createport */
consoleWriteMsg =  CreateStdIO(consoleWritePort);
if(consoleWriteMsg == 0) exit(200);       /* error in createstdio */
```

For *reading* from the console:

```
struct IOStdReq *consoleReadMsg;      /* I/O request block pointer */
struct Port *consoleReadPort;         /* a port at which to receive replies */

consoleReadPort = CreatePort("mycon.read",0);
if(consoleReadPort == 0) exit(300);     /* error in createport */
consoleReadMsg =  CreateStdIO(consoleReadPort);
if(consoleReadMsg == 0) exit(400);      /* error in createstdio */
```

These fragments show two messages and ports being set up. You would use this if you want to have a read command continuously queued up while using a separate message with its associated port to send control command sequences to the console.

In addition, if you want to queue up multiple commands to the console, you may wish to create multiple messages (but probably just one port for receiving replied messages from the device).

4.5. OPENING A CONSOLE DEVICE

For other devices, you normally use **OpenDevice()** to pass an uninitialized **IORequest** block to the device. For a console device, this is slightly different. You must have initialized two fields in the request block; namely, the data pointer and the length field. Here is a subroutine that can be used to open a console device (attach it to an existing window). It assumes that intuition.library is already open, a window has also been opened, and this new console is to be attached to the open window.

```
/* this function returns a value of 0 if the console
 * device opened correctly and a nonzero value (the error
 * returned from OpenDevice) if there was an error.
 */
OpenConsole(writerequest,readrequest,window)
      struct IOStdReq *writerequest;
      struct IOStdReq *readrequest;
      struct Window *window;
      {
            int error;
            writerequest->io_Data = (APTR) window;
            writerequest->io_Length = sizeof(*window);
            error = OpenDevice("console.device", 0, writerequest, 0);
            readrequest->io_Device = writerequest->io_Device;
            readrequest->io_Unit   = writerequest->io_Unit;
                  /* clone required parts of the request */
            return(error);
      }
```

Notice that this routine opens the console using one I/O request (write), then copies the write request values into the read request. This assures that both input and output go to the same console device.

Sending a Character Stream to the Console Device

To perform console I/O, you fill in fields of the console I/O standard request, and pass this block to the console device using one of the normal I/O functions. When the console device has completed the action, the device returns the message block to the port you have designated within the message itself. The function **CreateStdIO()** initializes the message to contain the address of the **ReplyPort**.

The following subroutines use the **IOStdReq** created above. Note that the **IOStdReq** itself contains a pointer to the unit with which it is communicating. Thus, a single function can be used to communicate with multiple consoles.

```
/* output a single character to a specified console */

ConPutChar(request,character)
struct IOStdReq *request;
char character;
{
        request->io_Command = CMD_WRITE;
        request->io_Data = &character;
        request->io_Length = 1;
        DoIO(request);
        return;
}

/* output a stream of known length to a console */

ConWrite(request,string,length)
struct IOStdReq *request;
char *string;
int length;
{
        request->io_Command = CMD_WRITE;
        request->io_Data = string;
        request->io_Length = length;
        DoIO(request);
        return;
}

/* output a NULL-terminated string of characters to a console */

ConPutStr(request,string)
struct IOStdReq *request;
char *string;
{
request->io_Command = CMD_WRITE;
request->io_Data = string;
request->io_Length = -1;  /* tells console to end when it
                           * sees a terminating zero on
                           * the string. */
DoIO(request);
        return;
}
```

4.6. CONTROL SEQUENCES FOR SCREEN OUTPUT

Table 4-1 lists the functions that the console device supports, along with the character stream that you must send to the console to produce the effect. Where the function table indicates multiple characters, it is more efficient to use the **ConWrite()** function rather than **ConPutChar()** since it avoids the overhead of transfering the message block multiple times. The table below uses the second form of <CSI>, that is, the hex value 9B, to minimize the number of characters to be transmitted to produce a function.

In table 4-1, if an item is enclosed in square brackets, it is an optional item and may be omitted. For example, for INSERT [N] CHARACTERS the value for N or M is shown as optional. The console device responds to such optional items by treating the value of N as if it is not specified. The value of N or M is always a decimal number, having one or more ASCII digits to express its value.

Table 4-1: Console Control Sequences

Command	**Sequence of Characters (in hexadecimal)**
BACKSPACE (move left one column)	08
LINE FEED (move down one text line as specified by the mode function below)	0A
VERTICAL TAB (move up one text line)	0B
FORM FEED (clear the console's screen)	0C
CARRIAGE RETURN (move to first column)	0D
SHIFT IN (undo SHIFT OUT)	OE
SHIFT OUT (set MSB of each character before displaying)	0F
ESC (escape; can be part of the control sequence introducer)	1B
CSI (control sequence introducer)	
RESET TO INITIAL STATE	9B 63
INSERT [N] CHARACTERS (Inserts one or more spaces, shifting the remainder of the line to the right.)	9B [N] 40
CURSOR UP [N] CHARACTER POSITIONS (default = 1)	9B [N] 41
CURSOR DOWN [N] CHARACTER POSITIONS (default = 1)	9B [N] 42
CURSOR FORWARD [N] CHARACTER POSITIONS (default = 1)	9B [N] 43

CURSOR BACKWARD [N] CHARACTER POSITIONS (default = 1)	9B [N] 44
CURSOR NEXT LINE [N] (to column 1)	9B [N] 45
CURSOR PRECEDING LINE [N] (to column 1)	9B [N] 46
MOVE CURSOR TO ROW; COLUMN where N is row, M is column, and semicolon (hex 3B) must be present as a separator, or if row is left out, so the console device can tell that the number after the semicolon actually represents the column number.	9B [N] [3B N] 48
ERASE TO END OF DISPLAY	9B 4A
ERASE TO END OF LINE	9B 4B
INSERT LINE (above the line containing the cursor)	9B 4C
DELETE LINE (remove current line, move all lines up one position to fill gap, blank bottom line)	9B 4D
DELETE CHARACTER [N] (that cursor is sitting on and to the right if [N] is specified)	9B [N] 50
SCROLL UP [N] LINES (Remove line(s) from top of screen, move all other lines down, blanks [N] bottom lines).	9B [N] 53
SCROLL DOWN [N] LINES (Remove line(s) from bottom of screen, move all other lines up, blanks [N] top lines).	9B [N] 54
SET MODE (cause LINEFEED to respond as RETURN-LINEFEED)	9B 32 30 68
RESET MODE (cause LINEFEED to respond only as LINEFEED)	9B 32 30 6C
DEVICE STATUS REPORT (cause console to insert into your read-stream a CURSOR POSITION REPORT; see "Reading from the Console" for more info).	9B 6E
SELECT GRAPHIC RENDITION <style>;<fg>;<bg> (select text style foreground color, background color) (See the note below.)	See note below.

NOTE: For SELECT GRAPHIC RENDITION any number of parameters, in any order, are valid. They are separated by semicolons. The parameters follow.

<style> =
- 0 plain text
- 1 bold-face
- 3 italic
- 4 underscore
- 7 inverse-video

<fg> =
- 30 - 37 selecting system colors 0-7 for foreground
  transmitted as two ASCII characters

<bg> =
- 40 - 47 selecting system colors 0-7 for background
  transmitted as two ASCII characters

For example, to select bold-face, with color 3 as foreground and color 0 as background, send the sequence:

9B 31 3B 33 33 3B 34 30 6D

representing the ASCII sequence:

"<CSI>1;33;40m"

where <CSI> is the control sequence introducer, here used as the single-character value 9B hex.

The following are not ANSI standard sequences; they are private Amiga sequences.

In the four command descriptions that follow, length, width, and offset are comprised of one or more ASCII digits, defining a decimal value.

Command	**Sequence of Characters (in hexadecimal)**
SET PAGE LENGTH (in character raster lines, causes console to recalculate, using current font, how many text lines will fit on the page.	9B <length> 74
SET LINE LENGTH (in character positions, using current font, how many characters should be placed on each line).	9B <width> 75
SET LEFT OFFSET (in raster columns, how far from the left of the window	

should the text begin).	9B <offset> 78
SET TOP OFFSET (in raster lines, how far from the top of the window's **RastPort** should the topmost line of the character begin).	9B <offset> 79

NOTE: The console normally handles the above four functions automatically. To allow it to do so again after setting your own values, you can send the function without a parameter.

Command	**Sequence of Characters (in hexadecimal)**
SET RAW EVENTS—see the separate topic "Selecting Raw Input Events" below for more details.	
RESET RAW EVENTS—see "Selecting Raw Input Events" below.	
SET CURSOR RENDITION - make the cursor visible or invisible:	
invisible:	9B 30 20 70
visible:	9B 20 70
WINDOW STATUS REQUEST - ask the console device to tell you the current bounds of the window, in upper and lower row and column character positions. (User may have resized or repositioned it). See "Window Bounds Report" below.	9B 71

Examples

Move cursor right by 1:

Character string equivalents: <CSI>C or <CSI>1C
Numeric (hex) equivalents: 9B 43 9B 31 43

Move cursor right by 20:

Character string equivalent: <CSI>20C
Numeric (hex) equivalent: 9B 32 30 43

Move cursor to upper left corner (home):

Character string equivalents:
<CSI>H or
<CSI>1;1H or
<CSI>;1H or
<CSI>1;H

Numeric (hex) equivalents:
9B 48
9B 31 3B 31 48
9B 3B 31 48
9B 31 3B 48

Move cursor to the fourth column of the first line of the window:

Character string equivalents:
<CSI>1;4H or
<CSI>;4H

Numeric (hex) equivalents:
9B 31 3B 34 48
9B 3B 34 48

Clear the screen:

Character string equivalents:
<FF> or CTRL-L {clear screen character} or
<CSI>H<CSI>J {home and clear to end of screen} or

Numeric (hex) equivalents:
0C
9B 48 9B 4A

Reading from the Console

Reading input from the console device returns an ANSI 3.64 standard byte stream. This stream may contain normal characters and/or RAW input event information. You may also request other RAW input events using the SET RAW EVENTS and RESET RAW EVENTS control sequences discussed below. See "Selection of Raw Input Events".

The following subroutines are useful for setting up for console reads. Only a single-character-at-a-time version is shown here.

NOTE: This example does not illustrate the fact that a request for more than one character can be satisfied by only one, thus requiring you to look at **io_Actual**.

```
/* queue up a read request to a console, show where to
 * put the character when ready to be returned. Most
 * efficient if this is called right after console is
 * opened */

QueueRead(request,whereto)
struct IOStdReq *request;
char *whereto;
{
        request->io_Command = CMD_READ;
        request->io_Data = whereto;
        request->io_Length = 1;
        SendIO(request);
        return;
}

/* see if there is a character to read.  If none, don't wait,
 * come back with a value of -1 */
int
ConMayGetChar(consolePort,request,whereto)
struct Port *consolePort
struct IOStdReq *request;
char *whereto;
{
        register temp;

        if ( GetMsg(consolePort) == NULL ) return(-1);
        temp = *whereto;
        QueueRead(request,whereto);
        return(temp);
}
```

```
/* go and get a character; put the task to sleep if
 * there isn't one present */

UBYTE
ConGetChar(consolePort,request,whereto)
struct IOStdReq *request;
struct Port *consolePort;
char *whereto;
{
        register temp;
        while((GetMsg(consolePort) == NULL)) WaitPort(consolePort);
        temp = *whereto;      /* get the character */
        QueueRead(request,whereto);
        return(temp);
}
```

Information About the Read-Stream

For the most part, keys whose keycaps are labeled with ANSI standard characters will ordinarily be translated into their ASCII equivalent character by the console device through the use of its keymap. A separate section in this chapter has been dedicated to the method used to establish a keymap and the internal organization of the keymap.

For keys other than those with normal ASCII equivalents, an escape sequence is generated and inserted into your input stream. For example, in the default state (no raw input events selected) the function and arrow keys will cause the following sequences to be inserted in the input stream:

Table 4-2: Special Key Report Sequences

Key	Unshifted Sends	Shifted Sends
F1	<CSI>0~	<CSI>10~
F2	<CSI>1~	<CSI>11~
F3	<CSI>2~	<CSI>12~
F4	<CSI>3~	<CSI>13~
F5	<CSI>4~	<CSI>14~
F6	<CSI>5~	<CSI>15~
F7	<CSI>6~	<CSI>16~
F8	<CSI>7~	<CSI>17~
F9	<CSI>8~	<CSI>18~
F10	<CSI>9~	<CSI>19~
HELP	<CSI>?~	<CSI>?~ (same)
Arrow keys:		
Up	<CSI>A	<CSI>T~
Down	<CSI>B	<CSI>S~
Left	<CSI>C	<CSI> A~ (notice the space)
Right	<CSI>D	<CSI> @~ (notice the space)

Cursor Position Report

If you have sent the DEVICE STATUS REPORT command sequence, the console device returns a cursor position report into your input stream. It takes the form:

<CSI><row>;<column>R

For example, if the cursor is at column 40, and row 12, here are the ASCII values you receive in a stream:

9B 34 30 3B 31 32 52

Window Bounds Report

A user may have either moved or resized the window to which your console is bound. By issuing a WINDOW STATUS REPORT to the console, you can read the current position and size in the input stream. This window bounds report takes the following form:

<CSI>1;1;<bottom margin>;<right margin>r

Note that the top and left margins are always 11 for the Amiga. The bottom and right margins give you the window row and column dimensions as well. For a window that holds 20 lines with 60 characters per line, you will receive the following in the input stream:

9B 31 3B 31 3B 32 30 3B 36 30 73

Selecting Raw Input Events

If the keyboard information, including "cooked" keystrokes does not give you enough information about input events you can request additional information from the console driver.

The command to SET RAW EVENTS is formatted as:

"<CSI>[event-types-separated-by-semicolons]{"

If, for example, you need to know when each key is pressed and released you would request "RAW keyboard input". This is done by writing "<CSI>1{" to the console. In a single SET RAW EVENTS request, you can ask the console to setup for multiple event types at one time. You must send multiple numeric parameters, separating them by semicolons (;). Example: ask for gadget pressed, gadget released and close gadget events - write: "<CSI>7;8;11{" (all as ASCII characters, without the quotes).

You can reset, that is, delete from reporting, one or more of the raw input event types by using the RESET RAW EVENTS command, in the same manner as the SET RAW EVENTS was used to establish them in the first place. This command stream is formatted as:

<CSI>[event-types-separated-by-semicolons]}

So, for example, you could reset all of the events set in the above example by transmitting the command sequence: "<CSI>7;8;11}" Here is a list of the valid raw input event types:

Table 4-3: Raw Input Event Types

Request Number	Description	
0	nop	Used internally.
1	RAW keyboard input	Intuition swallows all except the select button.
2	RAW mouse input	
3	Event	Sent whenever your window is made active.
4	Pointer position	
5	(unused)	
6	Timer	
7	Gadget pressed	
8	Gadget released	
9	Requester activity	
10	Menu numbers	
11	Close Gadget	
12	Window resized	
13	Window refreshed	
14	Preferences changed	
15	Disk removed	
16	Disk inserted	

4.7. COMPLEX INPUT EVENT REPORTS

If you select any of these events you will start to get information about the events in the following form:

<CSI><class>;<subclass>;<keycode>;<qualifiers>;<x>;<y>;
<seconds>;<microseconds>|

where:

<CSI>
is a one-byte field. It is the "control sequence introducer", 9B in hex.

<class>
is the RAW input event type, from the above table.

<subclass>
is usually 0. If the mouse is moved to the right controller, this would be 1.

<keycode>
indicates which key number was pressed (see figure 4-1 and table 4-5). This field can also be used for mouse information.

<qualifiers>
indicates the state of the keyboard and system. The qualifiers are defined as follows:

Table 4-4: Input Event Qualifiers

Bit	Mask	Key	
0	0001	left shift	
1	0002	right shift	
2	0004	capslock	* Associated keycode is special; see below.
3	0008	control	
4	0010	left ALT	
5	0020	right ALT	
6	0040	left Amiga key pressed	
7	0080	right Amiga key pressed	
8	0100	numeric pad	
9	0200	repeat	
10	0400	interrupt	Not currently used.
11	0800	multi broadcast	This window (active one) or all windows.
12	1000	left mouse button	
13	2000	right mouse button	
14	4000	middle mouse button	(Not available on standard mouse.)
15	8000	relative mouse	Indicates mouse coordinates are relative, not absolute.

The CAPS LOCK key is handled in a special manner. It only generates a keycode when it is pressed, not when it is released. However the up/down bit (80 hex) is still used and reported. If pressing the caps lock key causes the LED to light then key code 62 (caps lock pressed) is sent. If pressing the caps lock key extinguishes the LED then key code 190 (caps lock released) is sent. In effect, the keyboard reports this key as held down until it is struck again.

The <x> and <y> fields are filled by some classes with an Intuition address: x<<16+y.

The <seconds> and <microseconds> fields contain the system time stamp taken at the time the event occurred. These values are stored as long-words by the system.

With RAW keyboard input selected, keys will no longer return a simple one-character "A" to "Z" but will rather return raw keycode reports of the form:

<CSI>1;0;<keycode>;<qualifiers>;0;0;<seconds>;<microseconds>|

For example, if the user pressed and released the "B" key with the left shift and right Amiga keys also pressed you might receive the following data:

<CSI>1;0;35;129;0;0;23987;99|
<CSI>1;0;163;129;0;0;24003;18|

The <keycode> field is an ASCII decimal value representing the key pressed or released. Adding 128 to the pressed key code will result in the released keycode. Figure 4-1 lets you convert quickly from a key to its keycode. The tables let you convert quickly from a keycode to a key.

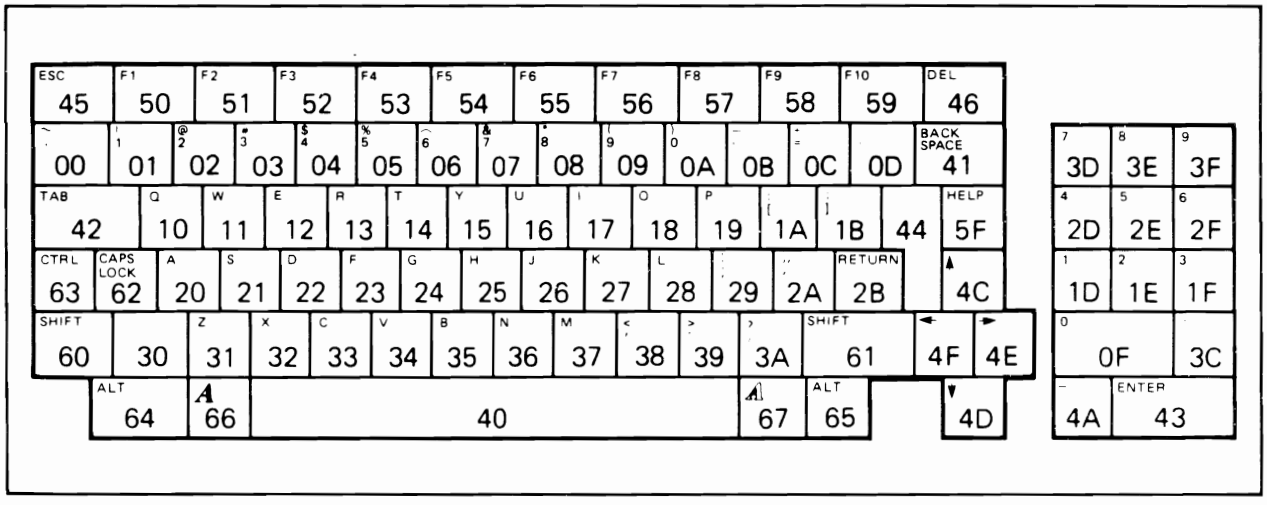


Figure 4-1: The Amiga Keyboard, Showing Keycodes in Hex

The default values given correspond to:

1) The values the console device will return when these keys are pressed, and

2) The Keycaps as shipped with the standard American keyboard.

Table 4-5: System Default Console Key Mapping

Raw Key Number	Keycap Legend	Unshifted Default Value	Shifted Default Value
00	` ~	` (Accent grave)	~ (tilde)
01	1 !	1	!
02	2 @	2	@
03	3 #	3	#
04	4 $	4	$
05	5 %	5	%
06	6 ^	6	^
07	7 &	7	&
08	8 *	8	*
09	9 (	9	(
0A	0)	0	)
0B	- _	- (Hyphen)	_ (Underscore)
0C	= +	=	+
0D	\ |	\	|
0E		(undefined)	
0F	0	0	0 (Numeric pad)
10	Q	q	Q
11	W	w	W
12	E	e	E
13	R	r	R
14	T	t	T
15	Y	y	Y
16	U	u	U
17	I	i	I
18	O	o	O
19	P	p	P
1A	[{	[	{
1B	] }	]	}
1C		(undefined)	
1D	1	1	1 (Numeric pad)
1E	2	2	2 (Numeric pad)
1F	3	3	3 (Numeric pad)
20	A	a	A
21	S	s	S
22	D	d	D
23	F	f	F
24	G	g	G
25	H	h	H
26	J	j	J
27	K	k	K
28	L	l	L
29	; :	;	:

2A	' "	' (single quote)	"
2B		(RESERVED)	(RESERVED)
2C		(undefined)	
2D	4	4	4 (Numeric pad)
2E	5	5	5 (Numeric pad)
2F	6	6	6 (Numeric pad)
30		(RESERVED)	(RESERVED)
31	Z	z	Z
32	X	x	X
33	C	c	C
34	V	v	V
35	B	b	B
36	N	n	N
37	M	m	M
38	, <	, (comma)	<
39	. >	. (period)	>
3A	/ ?	/	?
3B		(undefined)	
3C	.	.	. (Numeric pad)
3D	7	7	7 (Numeric pad)
3E	8	8	8 (Numeric pad)
3F	9	9	9 (Numeric pad)
40	(Space bar)	20	20
41	BACK SPACE	08	08
42	TAB	09	09
43	ENTER	0D	0D (Numeric pad)
44	RETURN	0D	0D
45	ESC	1B	1B
46	DEL	7F	7F
47		(undefined)	
48		(undefined)	
49		(undefined)	
4A	-	-	- (Numeric Pad)
4B		(undefined)	
4C	Up Arrow	<CSI>A~	<CSI>T~
4D	Down Arrow	<CSI>B~	<CSI>S~
4E	Forward Arrow	<CSI>C~	<CSI> A~ [2]
4F	Backward Arrow	<CSI>D~	<CSI> @~

[2] In shifted Forward Arrow and Backward arrow, note blank space after <CSI>.

Raw Key Number	Keycap Legend	Unshifted Default Value	Shifted Default Value
50	F1	<CSI>0~	<CSI>10~
51	F2	<CSI>1~	<CSI>11~
52	F3	<CSI>2~	<CSI>12~
53	F4	<CSI>3~	<CSI>13~
54	F5	<CSI>4~	<CSI>14~
55	F6	<CSI>5~	<CSI>15~
56	F7	<CSI>6~	<CSI>16~
57	F8	<CSI>7~	<CSI>17~
58	F9	<CSI>8~	<CSI>18~
59	F10	<CSI>9~	<CSI>19~
5A		(undefined)	
5B		(undefined)	
5C		(undefined)	
5D		(undefined)	
5E		(undefined)	
5F	HELP	<CSI>?~	<CSI>?~

Raw Key Number	Function or Keycap Legend	
60	SHIFT (left of space bar)	
61	SHIFT (right of space bar)	
62	CAPS LOCK	
63	CTRL	
64	(Left) ALT	
65	(Right) ALT	
66	Amiga (left of space bar)	Close Amiga
67	Amiga (right of space bar)	Open Amiga
68	Left mouse button (not converted)	Inputs are only for the mouse connected to Intuition, currently "gameport" one.
69	Right mouse button (not converted)	
6A	Middle mouse button (not converted)	
6B	(undefined)	
6C	(undefined)	
6D	(undefined)	
6E	(undefined)	
6F	(undefined)	

Raw Key Number	Function
70-7F	(undefined)
80-F8	Up transition (release or unpress key of one of the above keys). 80 for 00, F8 for 7F.
F9	Last keycode was bad (was sent in order to resynchronize).
FA	Keyboard buffer overflow.
FB	(undefined, reserved for keyboard processor catastrophe)
FC	Keyboard selftest failed.
FD	Power-up key stream start. Keys pressed or stuck at power-up will be sent between FD and FE.
FE	Power-up key stream end.
FF	(undefined, reserved)
FF	Mouse event, movement only, no button change. (not converted)

Notes about the preceding table:

1) "(undefined)" indicates that the current keyboard design should not generate this number. If you are using **SetKeyMap()** to change the key map the entries for these numbers must still be included.

2) "(not converted)" refers to mouse button events. You must use the sequence "<CSI>2{" to inform the console driver that you wish to receive mouse events; otherwise these will not be transmitted.

3) "(RESERVED)" Indicates that these keycodes have been reserved for non-US keyboards. The "2B" code key will be between the double-quote(") and RETURN keys. The "30" code key will be between the SHIFT and "Z" keys.

4.8. KEYMAPPING

The Amiga has the capability of mapping the keyboard in any manner that you wish. In other computers, this capability is normally provided through the use of "keyboard enhancers". In the Amiga, however, the capability is already present and the vectors that control the remapping are user accessible.

The functions called **GetKeyMap()** and **SetKeyMap()** each deal with a set of 8 longword pointers, known as the **KeyMap** data structure. The **KeyMap** data structure is shown below.

```
struct KeyMap {
    APTR km_LoKeyMapTypes;
    APTR km_LoKeyMap;
    APTR km_LoCapsable;
    APTR km_LoRepeatable;
    APTR km_HiKeyMapTypes;
    APTR km_HiKeyMap;
    APTR km_HiCapsable;
    APTR km_HiRepeatable;
    };
```

GetKeyMap() returns a pointer to this table of pointers, showing where in memory each of the tables representing the keymapping may be found.

As a prelude to the following material, note that the Amiga keyboard transmits raw key information to the computer in the form of a key position and a transition. Figure 4-1 shows a physical layout of the keys and the hexadecimal number that is transmitted to the system when a key is pressed. When the key is released, its value, plus hexadecimal 80, is transmitted to the computer. The key mapping described herein refers to the translation from this raw key transmission into console device output to the user.

The low key map provides translation of the key values from hex 00-3F; the high key map provides translation of key value from hex 40-67.

Raw output from the keyboard for the LoKeyMap does not include the space bar, TAB, ALT, CTRL, arrow keys and several other keys (shown in HiKeyMap, below).

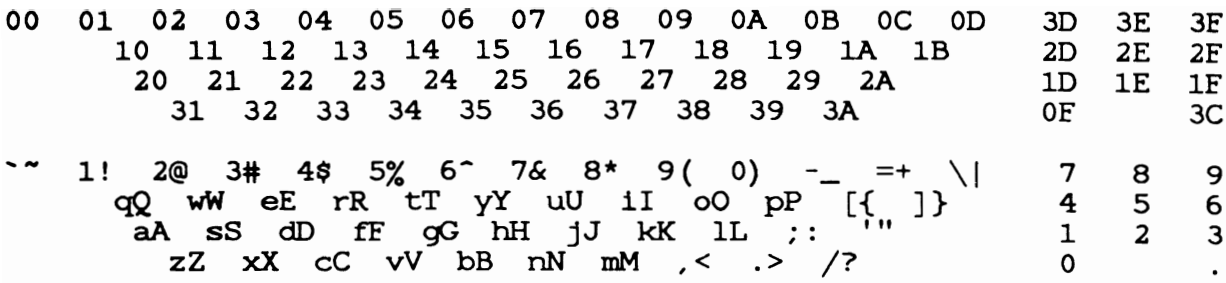


Figure 4-2: Low Key Map Translation Table

Table 4-6: High Key Map Hex Values

40	Space
41	Backspace
42	Tab
43	Enter
44	Return
45	Escape
46	Delete
4A	Numeric Pad -
4C	Cursor Up
4D	Cursor Down
4E	Cursor Forward
4F	Cursor Backward
50-59	Function keys F1-F10
5F	Help
60	Left Shift
61	Right Shift
62	Caps Lock
63	Control
64	Left Alt
65	Right Alt
66	Left Amiga
67	Right Amiga

The keymap table for the low and high keymaps consists of 4-byte entries, one per hex key-code. These entries are interpreted in one of two possible ways:

a. as 4 separate bytes, specifying how the key is to be interpreted when pressed:

   - o alone
   - o with one qualifier
   - o with another qualifier
   - o with both qualifiers

   where a qualifier is one of three possible keys:

   - o CTRL (control)
   - o ALT
   - o SHIFT

b. as a longword containing the address of a string descriptor, where a string of hex digits is to be output when this key is pressed. If a string is to be output, any combination of qualifiers may affect the string that may be transmitted.

NOTE: The keymap table *must* begin aligned on a word boundary. Each entry is 4-bytes long, thereby maintaining word alignment throughout the table. This is necessary since some of the entries may be longword addresses and *must* be aligned properly for the 68000.

About Qualifiers

As you may have noticed, there are three possible qualifiers, but only a 4-byte space in the table for each key. This does not allow space to describe what the computer should output for all possible combinations of qualifiers. This problem is solved by only allowing all three qualifiers to affect the output at the same time in string mode. Here is how that works.

For "vanilla" keys, such as the alphabetic keys, use the 4 bytes to represent the data output for the key alone, shifted key, ALT'ed key, and shifted-and-ALT'ed key. Then for the CTRL-key-plus-vanilla-key, use the code for the key alone with bits 6 and 5 set to 0.

For other keys, such as the return key or escape key, the qualifiers specified in the keytypes table (up to two) are the qualifiers used to establish the response to the key. This is done as follows. In the keytypes table, the values listed for the key types are those listed for the qualifiers in *keymap.h* and *keymap.i*. Specifically, these qualifier equates are:

KC_NOQUAL	0x00
KCF_SHIFT	0x01
KCF_ALT	0x02
KCF_CONTROL	0x04
KC_VANILLA	0x07
KCF_DOWNUP	0x08
KCF_STRING	0x40

As shown above, the qualifiers for the various types of keys occupy specific bit positions in the key types control byte.

A keymap table entry looks like this, in assembly code:

```
SOME_KEY:
   DC.B   VALUE_1, VALUE_2, VALUE_3, VALUE_4
```

Here is how you interpret the keymap for various combinations of the qualifier bits:

Table 4-7: Keymap Qualifier Bits

If Keytype is:	Then value in this position in the keytable is output when the key is pressed along with:			
KC_NOQUAL	-	-	-	alone
KCF_SHIFT	-	-	SHIFT	alone
KCF_ALT	-	-	ALT	alone
KCF_CONTROL	-	-	CTRL	alone
KCF_ALT+KCF_SHIFT	SHIFT+ALT	ALT	SHIFT	alone
KCF_CONTROL+KCF_ALT	CTRL+ALT	CTRL	ALT	alone
KCF_CONTROL+KCF_SHIFT	CTRL+SHIFT	CTRL	SHIFT	alone
KC_VANILLA	SHIFT+ALT	ALT	SHIFT	alone*

* special case—CTRL key, when pressed with one of the alphabet keys and certain others, is to output key-alone value with the bits 6 and 5 set to zero.

Keytype Table Entries

The vectors named km_LoKeyTypes and km_HiKeyTypes contain one byte per raw key code. This byte defines the entry type that is made in the key table by a set of bit positions.

Possible key types are:

- any of the qualifier groupings noted above, or

- KCF_STRING + any combination of KCF_SHIFT, KCF_ALT, KCF_CONTROL (or KC_NOQUAL) if the result of pressing the key is to be a stream of bytes (and key-with-one-or-more-qualifiers is to be one or more alternate streams of bytes).

  Any key can be made to output up to 8 unique byte streams if KCF_STRING is set in its keytype. The only limitation is that the total length of all of the strings assigned to a key be within the "jump range" of a single byte increment. See the "String-Output Keys" section below for more information.

The low keytype table covers the raw keycodes from hex 00-3F, and contains one byte per keycode. Therefore this table contains 64 (decimal) bytes. The high keytype table covers the raw keycodes from hex 40-67, and contains 38 (decimal) bytes.

String-Output Keys

When a key is to output a string, the keymap table contains the address of a string descriptor in place of a 4-byte mapping of a key as shown above. Here is a partial table for a new high key map table which contains only three entries thus far. The first two are for the space bar and the backspace key; the third is for the tab key, which is to output a string that says "[TAB]". An alternate string, "[SHIFTED-TAB]", is also to be output when a shifted TAB key is pressed.

Table 4-8: Composing an Alternate Key Map

```
newHiMapTypes:
    DC.B        KCF_ALT,KC_NOQUAL,
    DC.B        KCF_STRING+KCF_SHIFT,
                ...                         ;(more)
```

```
newHiMap:
   DC.B        0,0,$A0,$20          ;space bar, and ALT-space bar
   DC.B        0,0,0,$08            ;BACKSPACE key only
   DC.L        newkey42             ;new definition for string to
                                    ;output for TAB key
               ...                  ;(more)
newkey42:
   DC.B        new42ue - new42us    ;length of the
                                    ;unshifted string
   DC.B        new42us - newkey42
                                    ;number of bytes from start of
                                    ;string descriptor to start of
                                    ;this string
   DC.B        new42se - new42ss
                                    ;length of the shifted string
   DC.B        new42ss - newkey42
                                    ;number of bytes from start of
                                    ;string descriptor to start of
                                    ;this string
new42us:
   DC.B        '[TAB]'
new42ue:
new42ss:
   DC.B        '[SHIFTED-TAB]'
new42se:
```

The new high map table points to the string descriptor at address newkey42. The new high map types table says that there is one qualifier, which means that there are two strings in the key string descriptor.

Each string in the descriptor takes two bytes in this part of the table; the first byte is the length of the string, the second byte is the distance from the start of the descriptor to the start of the string. Therefore, a single string (KCF_STRING + KC_NOQUAL) takes 2 bytes of string descriptor. If there is one qualifier, 4 bytes of descriptor are used. If there are two qualifiers, 8 bytes of descriptor are used. If there are 3 qualifiers, 16 bytes of descriptor are used. All strings start immediately following the string descriptor in that they are accessed as single byte offsets from the start of the descriptor itself. Therefore, the distance from the start of the descriptor to the last string in the set (the one that uses the entire set of specified qualifiers) must start within 255 bytes of the descriptor address.

Since the length of the string is contained in a single byte, the length of any single string must be 255 bytes or less while also meeting the "reach" requirement. However, the console input buffer size limits the string output from any individual key to 32 bytes maximum.

The length of a keymap containing string descriptors and strings is variable and depends on the number and size of the strings that you provide.

Capsable Bit Table

The vectors called km_LoCapsable and km_HiCapsable point to the first byte in an 8-byte table that contains more information about the keytable entries.

Specifically, if the CAPS LOCK key has been pressed the CAPS LOCK LED is on), and if there is a bit *on* in that position in the capsable map, then this key will be treated as though the shift-key is now currently pressed. For example, in the default key mapping, the alphabetic keys are "capsable" but the punctuation keys are not. This allows you to set the shift-lock, just as on a normal typewriter, and get all capital letters. However, unlike a normal typewriter, you need not go out of shift-lock to correctly type the punctuation symbols or numeric keys.

In the table, the bits that control this feature are numbered from the lowest bit in the byte, and from the lowest memory byte address to the highest. For example, the bit representing capsable status for the key that transmits raw code 00 is bit 0 in byte 0; for the key that transmits raw code 08 it's bit 0 in byte 1, and so on.

There are 64 bits (8-bytes) in each of the two capsable tables.

Repeatable Bit Table

For both the low and high key maps there is an 8-byte table that provides one bit per possible raw key code. This bit indicates whether or not the specified key should repeat at the rate set by the Preferences program. The bit positions correspond to those specified in the capsable bit table.

If there is a 1 in a specific position, the key can repeat. The vectors that point to these tables are called km_LoRepeatable and km_HiRepeatable.

Default Low Key Map

In the default low key map, all of the keys are treated in the same manner. That is:

- When pressed alone, they transmit the ASCII equivalent of the unshifted key.
- When shifted, they translate the ASCII equivalent of the shifted value when printed on the keycap.
- When "ALT'ed" (pressed along with an ALT key), they transmit the alone-value with the high bit of a byte set (value plus hex 80).
- When shifted and ALT'ed, they transmit the shifted-value plus hex 80.

In this table, the bytes that describe the data to be transmitted are positioned as the example for the "A" key shown here:

```
key_A   DC.B   ('A')+$80    ;shifted and ALT'ed
        DC.B   ('a')+$80    ;ALT'ed only
        DC.B   ('A')        ;shifted only
        DC.B   ('a')        ;not shifted or ALT'ed.
```

In addition to the response to the key alone, shifted, ALT'ed and shifted-and-ALT'ed, the default low keymap also responds to the key combination of "CTRL + key" by stripping off bits 6 and 5 of the generated data byte. For example, CTRL + A generates the translated keycode 01 (61 with bits 6 and 5 set to 0).

All keys in the low key map are mapped to their ASCII equivalents as noted in the low key map key table shown above.

Since the low key table contains 4 bytes per key, and describes the keys (raw codes) from hex 00-3F, there are 64 times 4 or 256 bytes in this table.

Default High Key Map

Most of the keys in the high key map generate strings rather than single character mapping. The following keys map characters with no qualifier, along with their byte mapping:

Key	**Generates Value:**
BACKSP	$08
ENTER	$0D
DEL	$7F

The following keys map characters and use a single qualifier:

Key	*Generates value:*	*If used with qualifier, Generates value:*
SPACE	$20	$A0 (qualifier = ALT)
RETURN	$0D	$0A (qualifier = CONTROL)
ESC	$1B	$9B (qualifier = ALT)
numeric pad '-'	$2D	$FF (qualifier = ALT)

The following keys generate strings:

Key	Generates value:	If used with <SHIFT>, generates value:
TAB	$09	$9B, followed by 'Z'
cursor:		
UP	$9B, followed by 'A'	$9B, followed by 'T'
DOWN	$9B, followed by 'B'	$9B, followed by 'S'
FWD	$9B, followed by 'C'	$9B, followed by ' ', followed by '@'
BACKWD	$9B, followed by 'D'	$9B, followed by ' ', followed by 'A'
function keys:		
F1	$9B, followed by '0~ '	$9B, followed by '10~ '
F2	$9B, followed by '1~ '	$9B, followed by '11~ '
F3	$9B, followed by '2~ '	$9B, followed by '12~ '
F4	$9B, followed by '3~ '	$9B, followed by '13~ '
F5	$9B, followed by '4~ '	$9B, followed by '14~ '
F6	$9B, followed by '5~ '	$9B, followed by '15~ '
F7	$9B, followed by '6~ '	$9B, followed by '16~ '
F8	$9B, followed by '7~ '	$9B, followed by '17~ '
F9	$9B, followed by '8~ '	$9B, followed by '18~ '
F10	$9B, followed by '9~ '	$9B, followed by '19~ '
HELP	$9B, followed by '?~ '	(no qualifier used)

4.9. CLOSING A CONSOLE DEVICE

When you have finished using a console, it must be closed so that the memory areas it utilized may be returned to the system memory manager. Here is a sequence that you can use to close a console device:

```
CloseDevice(requestBlock);
```

Note that you should also delete the messages and ports associated with this console after the console has been closed:

```
DeleteStdIO(consoleWriteMsg);
DeleteStdIO(consoleReadMsg);
DeletePort(consoleWritePort);
DeletePort(consoleReadPort);
```

If you have finished with the window used for the console device, you can now close it.

4.10. EXAMPLE PROGRAM

The following is a console device demonstration program with supporting macro routines..

```
/* cons.c */

/* This program is supported by the Amiga C compiler, version 1.1 and beyond.
 * (v1.0 compiler has difficulties if string variables do not have their
 * initial character aligned on a longword boundary.  Compiles acceptably
 * but won't run correctly.)
 */

#include "exec/types.h"
#include "exec/io.h"
#include "exec/memory.h"

#include "graphics/gfx.h"
#include "hardware/dmabits.h"
#include "hardware/custom.h"
#include "hardware/blit.h"
#include "graphics/gfxmacros.h"
#include "graphics/copper.h"
#include "graphics/view.h"
#include "graphics/gels.h"
#include "graphics/regions.h"
#include "graphics/clip.h"
#include "exec/exec.h"
#include "graphics/text.h"
#include "graphics/gfxbase.h"

#include "devices/console.h"
#include "devices/keymap.h"

#include "libraries/dos.h"
#include "graphics/text.h"
#include "libraries/diskfont.h"
#include "intuition/intuition.h"
```

```
UBYTE escdata[] = { 0x9b, '@',  /* insert character */
                    0x9b, 'A',  /* cursor up */
                    0x9b, 'B',  /* cursor down */
                    0x9b, 'C',  /* cursor left */
                    0x9b, 'D',  /* cursor right */
                    0x9b, 'E',  /* cursor next line */
                    0x9b, 'F',  /* cursor prev line */
                    0x9b, 'J',  /* erase to end of display */
                    0x9b, 'K',  /* erase to end of line */
                    0x9b, 'L',  /* insert line */
                    0x9b, 'M',  /* delete line */
                    0x9b, 'P',  /* delete character */
                    0x9b, 'S',  /* scroll up */
                    0x9b, 'T',  /* scroll down */
                    0x1b, 'c',  /* reset to initial state */
                    0x9b, 'q',  /* window status request */
                    0x9b, 'n',  /* device status report */
                    0x9b, ' ', 'p',  /* cursor on */
                    0x9b, '0', ' ', 'p',  /* cursor off */
                    0x9b, '2', '0', 'h',  /* set mode */
                    0x9b, '2', '0', 'l',  /* reset mode */
                   };

    /* COVER A SELECTED SUBSET OF THE CONSOLE AVAILABLE FUNCTIONS */

#define INSERTCHARSTRING      &escdata[0]
#define CURSUPSTRING          &escdata[0+2]
#define CURSDOWNSTRING        &escdata[0+4]
#define CURSFWDSTRING         &escdata[0+6]
#define CURSBAKSTRING         &escdata[0+8]
#define CURSNEXTLINE          &escdata[0+10]
#define CURSPREVLINE          &escdata[0+12]
#define ERASEEODSTRING        &escdata[0+14]
#define ERASEEOLSTRING        &escdata[0+16]
#define INSERTLINESTRING      &escdata[0+18]
#define DELETELINESTRING      &escdata[0+20]
#define DELCHARSTRING         &escdata[0+22]
#define SCROLLUPSTRING        &escdata[0+24]
#define SCROLLDOWNSTRING      &escdata[0+26]
#define RESETINITSTRING       &escdata[0+28]
#define WINDOWSTATSTRING      &escdata[0+30]
#define DEVSTATSTRING         &escdata[0+32]
#define CURSONSTRING          &escdata[0+34]
#define CURSOFFSTRING         &escdata[0+37]
#define SETMODESTRING         &escdata[0+41]
#define RESETMODESTRING       &escdata[0+45]

#define BACKSPACE(r)          ConPutChar(r,0x08)
#define TAB(r)                ConPutChar(r,0x09)
#define LINEFEED(r)           ConPutChar(r,0x0a)
```

```
#define VERTICALTAB(r)          ConPutChar(r,0x0b)
#define FORMFEED(r)             ConPutChar(r,0x0c)
#define CR(r)                   ConPutChar(r,0x0d)
#define SHIFTOUT(r)             ConPutChar(r,0x0e)
#define SHIFTIN(r)              ConPutChar(r,0x0f)
#define CLEARSCREEN(r)          ConPutChar(r,0x0c)

#define RESET(r)                ConWrite(r,RESETINITSTRING,2)
#define INSERT(r)               ConWrite(r,INSERTCHARSTRING,2)
#define CURSUP(r)               ConWrite(r,CURSUPSTRING,2)
#define CURSDOWN(r)             ConWrite(r,CURSDOWNSTRING,2)
#define CURSFWD(r)              ConWrite(r,CURSFWDSTRING,2)
#define CURSBAK(r)              ConWrite(r,CURSBAKSTRING,2)
#define CURSNEXTLN(r)           ConWrite(r,CURSNEXTLINE,2)
#define CURSPREVLN(r)           ConWrite(r,CURSPREVLINE,2)
#define ERASEEOD(r)             ConWrite(r,ERASEEODSTRING,2)
#define ERASEEOL(r)             ConWrite(r,ERASEEOLSTRING,2)
#define INSERTLINE(r)           ConWrite(r,INSERTLINESTRING,2)
#define DELETELINE(r)           ConWrite(r,DELETELINESTRING,2)
#define SCROLLUP(r)             ConWrite(r,SCROLLUPSTRING,2)
#define SCROLLDOWN(r)           ConWrite(r,SCROLLDOWNSTRING,2)
#define DEVICESTATUS(r)         ConWrite(r,DEVSTATSTRING,2)
#define WINDOWSTATUS(r)         ConWrite(r,WINDOWSTATSTRING,2)
#define DELCHAR(r)              ConWrite(r,DELCHARSTRING,2)
#define CURSORON(r)             ConWrite(r,CURSONSTRING,3)
#define CURSOROFF(r)            ConWrite(r,CURSOFFSTRING,4)
#define SETMODE(r)              ConWrite(r,SETMODESTRING,4)
#define RESETMODE(r)            ConWrite(r,RESETMODESTRING,4)

#define CloseConsole(r)         CloseDevice(r)

ULONG DosBase;
ULONG DiskfontBase;
ULONG IntuitionBase;
ULONG GfxBase;


struct NewWindow nw = {
    10, 10,     /* starting position (left,top) */
    620,90,     /* width, height */
    -1,-1,      /* detailpen, blockpen */
    0,          /* flags for idcmp */
    WINDOWDEPTH|WINDOWSIZING|WINDOWDRAG|SIMPLE_REFRESH
        |ACTIVATE|GIMMEZEROZERO,
                /* window gadget flags */
    0,          /* pointer to 1st user gadget */
    NULL,       /* pointer to user check */
    "Console Test", /* title */
    NULL,       /* pointer to window screen */
    NULL,       /* pointer to super bitmap */
```

```
    100,45,         /* min width, height */
    640,200,        /* max width, height */
    WBENCHSCREEN};

struct Window *w;
struct RastPort *rp;

struct IOStdReq *consoleWriteMsg;   /* I/O request block pointer */
struct MsgPort *consoleWritePort;     /* a port at which to receive */
struct IOStdReq *consoleReadMsg;   /* I/O request block pointer */
struct MsgPort *consoleReadPort;      /* a port at which to receive */

extern struct MsgPort *CreatePort();
extern struct IOStdReq *CreateStdIO();

char readstring[200];   /* provides a buffer even though using only one char */

main()
{

    SHORT i;
    SHORT status;
    SHORT problem;
    SHORT error;
    problem = 0;

    if((DosBase = OpenLibrary("dos.library", 0)) == NULL)
        { problem = 1; goto cleanup1; }
    if((DiskfontBase=OpenLibrary("diskfont.library",0))==NULL)
        { problem = 2; goto cleanup2; }
    if((IntuitionBase=OpenLibrary("intuition.library",0))==NULL)
        { problem = 3; goto cleanup3; }
    if((GfxBase=OpenLibrary("graphics.library",0))==NULL)
        { problem = 4; goto cleanup4; }

    consoleWritePort = CreatePort("my.con.write",0);
    if(consoleWritePort == 0)
        { problem = 5; goto cleanup5; }
    consoleWriteMsg =  CreateStdIO(consoleWritePort);
    if(consoleWritePort == 0)
        { problem = 6; goto cleanup6; }

    consoleReadPort = CreatePort("my.con.read",0);
    if(consoleReadPort == 0)
        { problem = 7; goto cleanup7; }
    consoleReadMsg =  CreateStdIO(consoleReadPort);
    if(consoleReadPort == 0)
        { problem = 8; goto cleanup8; }

    w = (struct Window *)OpenWindow(&nw);  /* create a window */
```

```
    if(w == NULL)
        { problem = 9; goto cleanup9; }

    rp = w->RPort;              /* establish its rastport for later */

/* ************************************************************************ */
/* NOW, Begin using the actual console macros defined above.                */
/* ************************************************************************ */

    error = OpenConsole(consoleWriteMsg,consoleReadMsg,w);
    if(error != 0)
        { problem = 10; goto cleanup10; }
        /* attach a console to this window, initialize
         * for both write and read */

    QueueRead(consoleReadMsg,&readstring[0]); /* tell console where to
                                               * put a character that
                                               * it wants to give me
                                               * queue up first read */
    ConWrite(consoleWriteMsg,"Hello, World\r\n",14);

    ConPutStr(consoleWriteMsg,"testing BACKSPACE");
    for(i=0; i<10; i++)
        { BACKSPACE(consoleWriteMsg); Delay(30); }

    ConPutStr(consoleWriteMsg,"\r\n");

    ConPutStr(consoleWriteMsg,"testing TAB\r");
    for(i=0; i<6; i++)
        { TAB(consoleWriteMsg); Delay(30); }

    ConPutStr(consoleWriteMsg,"\r\n");

    ConPutStr(consoleWriteMsg,"testing LINEFEED\r");
    for(i=0; i<4; i++)
        { LINEFEED(consoleWriteMsg); Delay(30); }

    ConPutStr(consoleWriteMsg,"\r\n");

    ConPutStr(consoleWriteMsg,"testing VERTICALTAB\r");
    for(i=0; i<4; i++)
        { VERTICALTAB(consoleWriteMsg); Delay(30); }

    ConPutStr(consoleWriteMsg,"\r\n");

    ConPutStr(consoleWriteMsg,"testing FORMFEED\r");
    Delay(30);
    for(i=0; i<2; i++)
        { FORMFEED(consoleWriteMsg); Delay(30); }
```

```
ConPutStr(consoleWriteMsg,"\r\n");

ConPutStr(consoleWriteMsg,"testing CR");
Delay(30);
CR(consoleWriteMsg);
Delay(60);
ConPutStr(consoleWriteMsg,"\r\n");

ConPutStr(consoleWriteMsg,"testing INSERT\r");
for(i=0; i<4; i++)
      { INSERT(consoleWriteMsg); Delay(30); }

ConPutStr(consoleWriteMsg,"\r\n");

ConPutStr(consoleWriteMsg,"    testing DELCHAR\r");
CR(consoleWriteMsg);
for(i=0; i<4; i++)
      { DELCHAR(consoleWriteMsg); Delay(30); }

ConPutStr(consoleWriteMsg,"\r\n");

ConPutStr(consoleWriteMsg,"testing INSERTLINE\r");
CR(consoleWriteMsg);
for(i=0; i<3; i++)
      { INSERTLINE(consoleWriteMsg); Delay(30); }
ConPutStr(consoleWriteMsg,"\r\n");

ConPutStr(consoleWriteMsg,"testing DELETELINE\r");
CR(consoleWriteMsg);
LINEFEED(consoleWriteMsg);
Delay(60);
for(i=0; i<4; i++)
      { DELETELINE(consoleWriteMsg); Delay(30); }
ConPutStr(consoleWriteMsg,"\r\n");

ConPutStr(consoleWriteMsg,"testing CURSUP\r");
for(i=0; i<4; i++)
      { CURSUP(consoleWriteMsg); Delay(30); }

ConPutStr(consoleWriteMsg,"\r\n");

ConPutStr(consoleWriteMsg,"testing CURSDOWN\r");
for(i=0; i<4; i++)
      { CURSDOWN(consoleWriteMsg); Delay(30); }

ConPutStr(consoleWriteMsg,"\r\n");

ConPutStr(consoleWriteMsg,"testing CURSFWD\r");
for(i=0; i<4; i++)
      { CURSFWD(consoleWriteMsg); Delay(30); }
```

```
ConPutStr(consoleWriteMsg,"\r\n");

ConPutStr(consoleWriteMsg,"testing CURSBAK");
for(i=0; i<4; i++)
      { CURSBAK(consoleWriteMsg); Delay(30); }

ConPutStr(consoleWriteMsg,"\r\n");

ConPutStr(consoleWriteMsg,"testing CURSPREVLN");
for(i=0; i<4; i++)
      { CURSPREVLN(consoleWriteMsg); Delay(30); }

ConPutStr(consoleWriteMsg,"\r\n");

ConPutStr(consoleWriteMsg,"testing CURSNEXTLN");
for(i=0; i<4; i++)
      { CURSNEXTLN(consoleWriteMsg); Delay(30); }

ConPutStr(consoleWriteMsg,"\r\n");

ConPutStr(consoleWriteMsg,"testing ERASEEOD");
CURSPREVLN(consoleWriteMsg);
CURSPREVLN(consoleWriteMsg);
CURSPREVLN(consoleWriteMsg);
Delay(60);
for(i=0; i<4; i++)
      { ERASEEOD(consoleWriteMsg); Delay(30); }

ConPutStr(consoleWriteMsg,"\r\n");

ConPutStr(consoleWriteMsg,"testing ERASEEOL.junk");
CURSBAK(consoleWriteMsg);
CURSBAK(consoleWriteMsg);
CURSBAK(consoleWriteMsg);
CURSBAK(consoleWriteMsg);
CURSBAK(consoleWriteMsg);
Delay(60);
ERASEEOL(consoleWriteMsg);
Delay(30);
ConPutStr(consoleWriteMsg,"\r\n");

ConPutStr(consoleWriteMsg,"testing SCROLLUP");
for(i=0; i<4; i++)
      { SCROLLUP(consoleWriteMsg); Delay(30); }

ConPutStr(consoleWriteMsg,"\r\n");
ConPutStr(consoleWriteMsg,"testing SCROLLDOWN");
ConPutStr(consoleWriteMsg,"\n\n\n");
for(i=0; i<4; i++)
      { SCROLLDOWN(consoleWriteMsg); Delay(30); }
```

```
    ConPutStr(consoleWriteMsg,"\r\n");

    ConPutStr(consoleWriteMsg,"testing CURSOROFF");
    CURSOROFF(consoleWriteMsg);
    ConPutStr(consoleWriteMsg, "printed.with.cursor.off");
    Delay(60);
    ConPutStr(consoleWriteMsg,"\r\n");

    CURSORON(consoleWriteMsg); Delay(30);
    ConPutStr(consoleWriteMsg,"testing CURSORON");


/* ************************************************************************** */
    Delay(120);/* wait 2 seconds (120/60 ticks) */

    status = CheckIO(consoleReadMsg);    /* see if console read
                                          * anything, abort if not */
    if(status == FALSE) AbortIO(consoleReadMsg);
    WaitPort(consoleReadPort); /* wait for abort to complete */
    GetMsg(consoleReadPort);  /* and strip message from port */

    CloseConsole(consoleWriteMsg);
cleanup10:
cleanup9:
    CloseWindow(w);
cleanup8:
    DeleteStdIO(consoleReadMsg);
cleanup7:
    DeletePort(consoleReadPort);
cleanup6:
    DeleteStdIO(consoleWriteMsg);
cleanup5:
    DeletePort(consoleWritePort);
cleanup4:
     CloseLibrary(GfxBase);
cleanup3:
    CloseLibrary(IntuitionBase);
cleanup2:
     CloseLibrary(DiskfontBase);
cleanup1:
     CloseLibrary(DosBase);
    if(problem > 0) exit(problem+1000);
    else
          return(0);

}    /* end of main() */


/* Open a console device */
```

```
    /* this function returns a value of 0 if the console
     * device opened correctly and a nonzero value (the error
     * returned from OpenDevice) if there was an error.
     */

    int
OpenConsole(writerequest,readrequest,window)
    struct IOStdReq *writerequest;
    struct IOStdReq *readrequest;
    struct Window *window;
    {
        int error;
        writerequest->io_Data = (APTR) window;
        writerequest->io_Length = sizeof(*window);
        error = OpenDevice("console.device", 0, writerequest, 0);
        readrequest->io_Device = writerequest->io_Device;
        readrequest->io_Unit   = writerequest->io_Unit;
            /* clone required parts of the request */
        return(error);
    }

/* Output a single character to a specified console */

    int
ConPutChar(request,character)
    struct IOStdReq *request;
    char character;
    {
        request->io_Command = CMD_WRITE;
        request->io_Data = (APTR)&character;
        request->io_Length = 1;
        DoIO(request);
        /* command works because DoIO blocks until command is
         * done (otherwise pointer to the character could become
         * invalid in the meantime).
         */
        return(0);
    }

/* Output a stream of known length to a console */

    int
ConWrite(request,string,length)
    struct IOStdReq *request;
    char *string;
    int length;
    {
        request->io_Command = CMD_WRITE;
        request->io_Data = (APTR)string;
        request->io_Length = length;
```

```
        DoIO(request);
        /* command works because DoIO blocks until command is
         * done (otherwise pointer to string could become
         * invalid in the meantime).
         */
        return(0);
    }

/* Output a NULL-terminated string of characters to a console */

    int
ConPutStr(request,string)
    struct IOStdReq *request;
    char *string;
    {
        request->io_Command = CMD_WRITE;
        request->io_Data = (APTR)string;
        request->io_Length = -1;  /* tells console to end when it
                                   * sees a terminating zero on
                                   * the string. */
        DoIO(request);
        return(0);
    }

    /* queue up a read request to a console, show where to
     * put the character when ready to be returned.  Most
     * efficient if this is called right after console is
     * opened */

    int
QueueRead(request,whereto)
    struct IOStdReq *request;
    char *whereto;
    {
        request->io_Command = CMD_READ;
        request->io_Data = (APTR)whereto;
        request->io_Length = 1;
        SendIO(request);
        return(0);
    }

    /* see if there is a character to read.  If none, don't wait,
     * come back with a value of -1 */

    int
ConMayGetChar(request,requestPort, whereto)
    struct IOStdReq *request;
    char *whereto;
    {
        register temp;
```

```
        if ( GetMsg(requestPort) == NULL ) return(-1);
        temp = *whereto;
        QueueRead(request,whereto);
        return(temp);
    }

    /* go and get a character; put the task to sleep if
     * there isn't one present */

    UBYTE
ConGetChar(consolePort,request,whereto)
    struct IOStdReq *request;
    struct MsgPort *consolePort;
    char *whereto;
    {
        register UBYTE temp;
        while((GetMsg(consolePort) == NULL)) WaitPort(consolePort);
        temp = *whereto;      /* get the character */
        QueueRead(request,whereto);
        return(temp);
    }
```

Chapter 5

Input Device

This chapter describes the Amiga input device, which is a combination of three other devices: keyboard device, gameport device, and timer device. The input device merges separate input event streams from the keyboard, mouse, and timer into a single stream. This single stream can then be interpreted by the prioritized linked list of input handlers that are watching the input stream.

Note that two additional messages can appear in the input stream: "disk inserted" and "disk removed". These messages come from AmigaDOS and are sent to the input device for further propgation.

5.1. INTRODUCTION

The input device is automatically opened by AmigaDOS by any call to open the console device. When the input device is opened, a task, appropriately named "input.device", is started. The input device task communicates directly with the keyboard device to obtain raw key inputs. It also communicates with the gameport device to obtain mouse button and mouse movement events and with the timer device to obtain time events. In addition to these event streams, you can also directly input an event to the input device, to be fed to the handler chain. This topic is also covered below.

The keyboard device is also accessible directly. However, while the input device task is operating, it attempts to retrieve all incoming keyboard events and add them to the input stream.

The gameport device has two units available to it. As you view the Amiga looking at the gameport connectors, connector "1" is assigned as the primary mouse input for Intuition and contributes gameport input events to the input event stream. Connector "2" is handled by the other gameport unit and is currently unassigned. Each unit of the gameport device is an exclusive access object, in that you can specify what type of controller is attached. It is then assumed that only one task is sending requests for input from that unit. While the input device task is running, that task expects to read the input from connector 1. Direct use of

the gameport device is covered in a separate chapter of this manual.

The timer device provides time events for the input device. It also provides time interval reports for controlling key repeat rate and key repeat threshold. The timer device is a shared-access device and is described in its own separate section.

5.2. INPUT DEVICE COMMANDS

The input device allows the following system functions:

Command	Operation
OpenDevice()	Obtain shared use of the input device.
CloseDevice()	Relinquish use of the input device.
DoIO()	Initiate a command, and wait for it to complete.
SendIO()	Initiate a command, and return immediately.
AbortIO()	Abort a command already in the queue.

Only the Start, Stop, Invalid, and Flush commands have been implemented for this device. All other commands are no-operations.

The input device also supports the following device-specific commands:

Table 5-1: Input Device Commands

I/O Command	Operation
IND_WRITEEVENT	Propagate an input event stream to all devices
IND_ADDHANDLER	Add an input-stream handler into the handler chain.
IND_REMHANDLER	Remove an input-stream handler from the handler chain.
IND_SETTHRESH	Set the repeating key hold-down time before repeat starts.
IND_SETPERIOD	Set the period at which a repeating key repeats.
IND_SETMPORT	Set the gameport port to which the mouse is connected.
IND_SETMTRIG	Read conditions that must be met by a mouse before a pending read request will be satisfied.
IND_SETMTYPE	Set the type of device at the mouse port.

The device-specific commands outlined above are described in the following paragraphs. A description of the contents of an input event is given first because the input device deals in input events. An input event is a data structure that describes:

- the class of the event—often describes the device that generated the event
- the subclass of the event—space for more information if needed
- the code—keycode if keyboard, button information if mouse, others
- a qualifier such as "ALT key also down", "key repeat active"
- a position field which contains a data address or a mouse position count.
- a time stamp, showing the sequence in which events have occurred
- a link-field by which input events are linked together

The various types of input events are listed in the include-file *devices/inputevent.h*. That information is not repeated here. You can find more information about input events in the chapters titled "Gameport Device" and "Console Device".

There is a difference between simply receiving an input event from a device (gameport, keyboard, or console) and actually becoming a handler of an input event stream. A handler is a routine that is passed an input event, and it is up to the handler to decide if it can process the input event. If the handler does not recognize the event, it passes the address of the event as a return value.

Because of the input event field called **ie_NextEvent**, it is possible for the input event to be a pointer to the first event in a linked list of events to be handled. Thus the handler should be designed to handle multiple events if such a link is used. Note that handlers can, themselves, generate new linked lists of events which can be passed down to lower priority handlers.

IND_ADDHANDLER Command

You add a handler to the chain by the command IND_ADDHANDLER. Assuming that you have a properly initialized an **IOStdReq** block as a result of a call to **OpenDevice()** (for the input device), here is a typical C-language call to the IND_ADDHANDLER function:

```
struct Interrupt handlerStuff;
handlerStuff.is_Data = &hsData;
            /* address of its data area */
handlerStuff.is_Code = myhandler;
            /* address of entry point to handler */
handlerStuff.is_Node.ln_Pri = 51;
            /* set the priority one step higher than
             * Intution, so that our handler enters
             * the chain ahead of Intuition.
             */
inputRequestBlock.io_Command = IND_ADDHANDLER;
inputRequestBlock.io_Data = &handlerStuff;

DoIO(&inputRequestBlock);
```

Notice from the above that Intuition is one of the input device handlers and normally distributes all of the input events. Intuition inserts itself at priority position 50. You can choose the position in the chain at which your handler will be inserted by setting the priority field in the list-node part of the interrupt data structure you are feeding to this routine.

Note also that *any* processing time expended by a handler subtracts from the time available before the next event happens. Therefore, handlers for the input stream must be fast.

Rules for Input Device Handlers

The following rules should be followed when you are designing an input handler:

1. If an input handler is capable of processing a specific kind of an input event and that event has no links (**ie_NextEvent** = 0), the handler can end the handler chain by returning a NULL (0) value.

2. If there are multiple events linked together, the handler can feel free to delink an event from the input event chain, thereby passing a shorter list of events to subsequent handlers. The starting address of the modified list is the return value.

3. If a handler wishes to add new events to the chain that it passes to a lower priority handler, it may initialize memory to contain the new event or event chain. The handler, when it again gets control on the next round of event handling, should assume nothing about the current contents of the memory blocks it attached to the event chain. Lower priority handlers may have modified the memory as they handled their part of the event. The handler that allocates the memory for this purpose

should keep track of the starting address and the size of this memory chunk so that the memory can be returned to the free memory list when it is no longer needed.

Your routine should be structured so that it can be called as though from the following C-language statement:

```
newEventChain = yourHandlerCode(oldEventChain, yourHandlerData);
```

where:

- **yourHandlerCode** is the entry point to your routine,
- **oldEventChain** is the starting address for the current chain of input events, and
- **newEventChain** is the starting address of an event chain which you are passing to the next handler, if any.

A NULL (0) value terminates the handling.

Memory that you use to describe a new input event that you've added to the event chain is available for re-use or deallocation when the handler is called again or after the IND_REMHANDLER command for the handler is complete.

Because IND_ADDHANDLER installs a handler in any position in the handler chain, it can, for example, ignore specific types of input events as well as act upon and modify existing streams of input. It can even create new input events for Intuition or other programs to interpret.

IND_REMHANDLER Command

You remove a handler from the handler chain with the command IND_REMHANDLER. Assuming that you have a properly initialized **IOStdReq** block as a result of a call to **OpenDevice()** (for the input device) and you have already added the handler using IND_ADDHANDLER, here is a typical C-language call to the IND_REMHANDLER function:

```
inputRequestBlock.io_Command = IND_REMHANDLER;
inputRequestBlock.io_Data = &handlerStuff;
      /* tell it which one to remove */
DoIO(&inputRequestBlock);
```

IND_WRITEEVENT Command

As noted in the overview of this chapter, input events are normally generated by the timer device, keyboard device or gameport device. A user can also generate an input event and send it to the input device. It will then be treated as any other event and passed through to the input handler chain. You can create your own stream of events, then send them to the input device using the IND_WRITEEVENT command. Here is an example, assuming a correctly initialized **input_request_block**. The example sends in a single event, which is a phony mouse-movement:

```
struct InputEvent phony;

input_request_block.io_Command = IND_WRITEEVENT;
input_request_block.io_Flags = 0;
input_request_block.io_Length = sizeof(struct InputEvent);
input_request_block.io_Data = &phony;

phony.ie_NextEvent = NULL;    /* only one */
phony.ie_Class = IECLASS_RAWMOUSE;
phony.ie_TimeStamp.tv_secs = 0;
phony.ie_TimeStamp.tv_micro = 0;
phony.ie_Code = IECODE_NOBUTTON;
phony.ie_Qualifier = IEQUALIFIER_RELATIVEMOUSE;
phony.ie_X = 10;
phony.ie_Y = 5;
            /* mouse didn't move, but program made
             * system think that it did.
             */
DoIO(&input_request_block);
```

NOTE: This command adds the input event to the end of the current event stream. The system links other events onto the end of this event, thus modifying the contents of the data structure you constructed in the first place.

IND_SETTHRESH Command

This command sets the timing in seconds and microseconds for the input device to indicate how long a user must hold down a key before it begins to repeat. This command is normally performed by the Preferences tool or by Intuition when it notices that the Preferences have been changed. If you wish, you can call this function. The following typical sequence assumes that you have already correctly initialized the request block by opening the input device. Only the fields shown here need be initialized.

```
struct InputEvent thresh_event;

input_request_block.io_Command = IND_SETTHRESH;
input_request_block.io_Flags = 0;
input_request_block.io_Data = &thresh_event;

thresh_event.ie_NextEvent = 0;
thresh_event.ie_TimeStamp.tv_secs = 1;      /* one second */
thresh_event.ie_TimeStamp.tv_micro = 500000;
            /* 500,000 microseconds = 1/2 second */
DoIO(&input_request_block);
```

IND_SETPERIOD Command

This command sets the time period between key repeat events once the initial period threshhold has elapsed. Again, it is a command normally issued by Intuition and preset by the Preferences tool. A typical calling sequence is as shown above; change the command number and the timing period values to suit your application.

5.3. INPUT DEVICE AND INTUITION

There are several ways to receive information from the various devices that are part of the input device. The first way is to communicate directly with the device. This way is, as specified above, occasionally undesirable (while the input device task is running). The second way is to become a handler for the stream of events which the input device produces. That method is also shown above.

The third method of getting input from the input device is to retrieve the data from the console device or from the IDCMP (Intuition Direct Communications Message Port).

If you choose this third method, you should be aware of what happens to input events if your task chooses not to respond to them. If there is no active window and no active console, then input events (key strokes or left button mouse clicks usually) will simply be ignored. If, however, there is an active window (yours), and you choose to simply let the messages pile up without responding to them as quickly as possible, here is what happens:

- Another event occurs. If the system has no empty message that it can fill in to report this new event, then memory is dynamically allocated to hold this new information and the new message is transmitted to the message port for the task.

- When the task finally responds to the message, the allocated memory isn't returned to the system until the window is closed. Therefore, a task that chooses not to respond to its incoming messages for a long period of time can potentially remove a great deal of memory from the system free memory list, making that memory space unavailable to this or other tasks until this task is completed.

Thus it is always a good idea to respond to input messages as quickly as possible to maximize the amount of free memory in the system while your task is running.

5.4. SAMPLE PROGRAM

This sample program adds an input handler to the input stream. Note that this program also uses the PrepareTimer() and SetTimer() and DeleteTimer() routines described in the example program of the "Timer Device" chapter. Note also that compiling this program native on the Amiga requires a separate compile for this program, a separate assembly for the "handler.interface.asm", and a separate alink phase. Alink will be used to tie together the object files produced by the separate language phases.

```
#include <exec/types.h>
#include <exec/ports.h>
#include <exec/memory.h>
#include <exec/io.h>
#include <exec/tasks.h>
#include <exec/interrupts.h>
#include <devices/input.h>
#include <exec/devices.h>
#include <devices/inputevent.h>

#define F1KEYUP 0xD0
struct InputEvent copyevent;      /* local copy of the event */
                        /* assumes never has a next.event attached */
```

```
struct MsgPort *inputDevPort;
struct IOStdReq *inputRequestBlock;
struct Interrupt handlerStuff;

struct InputEvent dummyEvent;

extern struct MsgPort *CreatePort();
extern struct IOStdReq *CreateStdIO();

struct MemEntry me[10];

      /* If we want the input handler itself to add anything to the
       * input stream, we will have to keep track of any dynamically
       * allocated memory so that we can later return it to the system.
       * Other handlers can break any internal links the handler puts
       * in before it passes the input events.
       */

struct InputEvent
*myhandler(ev, mydata)
      struct InputEvent *ev;       /* and a pointer to a list of events */
      struct MemEntry *mydata[]; /* system will pass me a pointer to my
                                  * own data space.
                                 */
{
      /* demo version of program simply reports input events as
       * its sees them; passes them on unchanged.  Also, if there
       * is a linked chain of input events, reports only the lead
       * one in the chain, for simplicity.
       */
      if(ev->ie_Class == IECLASS_TIMER)
      {
            return(ev);
      }
      /* don't try to print timer events!!! they come every 1/10th sec. */
      else
      {
            Forbid();  /* don't allow a mix of events to be reported */
            copyevent.ie_Class = ev->ie_Class;
            copyevent.ie_SubClass = ev->ie_SubClass;
            copyevent.ie_Code =  ev->ie_Code;
            copyevent.ie_Qualifier = ev->ie_Qualifier;
            copyevent.ie_X = ev->ie_X;
            copyevent.ie_Y = ev->ie_Y;
            copyevent.ie_TimeStamp.tv_secs = ev->ie_TimeStamp.tv_secs;
            copyevent.ie_TimeStamp.tv_micro = ev->ie_TimeStamp.tv_micro;
            Permit();
      }

      /* There will be lots and lots of events coming through here;
```

```
        * rather than make the system slow down because something
        * is busy printing the previous event, lets just print what
        * we find is current, and if we miss a few, so be it.
        *
        * Normally this loop would "handle" the event or perhaps
        * add a new one to the stream.  (At this level, the only
        * events you should really be adding are mouse, rawkey or timer,
        * because you are ahead of the intuition interpreter.)
        * No printing is done in this loop (lets main() do it) because
        * printf can't be done by anything less than a 'process'
        */
       return(ev);
       /* pass on the pointer to the event (most handlers would
        * pass on a pointer to a changed or an unchanged stream)
        * (we are simply reporting what is seen, not trying to
        * modify it in any way) */
}


/* NOTICE:  THIS PROGRAM LINKS ITSELF INTO THE INPUT STREAM AHEAD OF
 * INTUITION.  THEREFORE THE ONLY INPUT EVENTS THAT IT WILL SEE AT
 * ALL ARE TIMER, KEYBOARD and GAMEPORT.  AS NOTED IN THE PROGRAM,
 * THE TIMER EVENTS ARE IGNORED DELIBERATELY */


extern struct Task *FindTask();
struct Task *mytask;
LONG mysignal;
extern HandlerInterface();

struct timerequest *mytimerRequest;

extern struct timerequest *PrepareTimer();
extern int WaitTimer();
extern int DeleteTimer();

main()
{
       SHORT error;
       ULONG oldseconds, oldmicro, oldclass;

       /* init dummy event, this is what we will feed to other handlers
        * while this handler is active */

       dummyEvent.ie_Class = IECLASS_NULL; /* no event happened */
       dummyEvent.ie_NextEvent = NULL;  /* only this one in the chain */

       inputDevPort = CreatePort(0,0);        /* for input device */
       if(inputDevPort == NULL) exit(-1);   /* error during createport */
```

```
inputRequestBlock = CreateStdIO(inputDevPort);
if(inputRequestBlock == 0) { DeletePort(inputDevPort); exit(-2); }
                        /* error during createstdio */

mytimerRequest = PrepareTimer();
if(mytimerRequest == NULL) exit(-3);

handlerStuff.is_Data = (APTR)&me[0];
            /* address of its data area */
handlerStuff.is_Code = (VOID)HandlerInterface;
            /* address of entry point to handler */
handlerStuff.is_Node.ln_Pri = 51;
            /* set the priority one step higher than
             * Intution, so that our handler enters
             * the chain ahead of Intuition.
             */
error = OpenDevice("input.device",0,inputRequestBlock,0);
if(error == 0) printf("\nOpened the input device");

inputRequestBlock->io_Command = IND_ADDHANDLER;
inputRequestBlock->io_Data = (APTR)&handlerStuff;

DoIO(inputRequestBlock);
copyevent.ie_TimeStamp.tv_secs = 0;
copyevent.ie_TimeStamp.tv_micro = 0;
copyevent.ie_Class = 0;
oldseconds = 0;
oldmicro = 0;
oldclass =0;

for(;;)             /* FOREVER */
{
WaitForTimer(mytimerRequest, 0, 100000);
            /* TRUE = wait; time = 1/10th second */

      /* note: while this task is asleep, it is very very likely that
       * one or more events will indeed pass through the input handler.
       * This task will only print a few of them, but won't intermix
       * the pieces of the input event itself because of the Forbid()
       * and Permit() (not allow task swapping when a data structure
       * isn't internally consistent)
       */
if(copyevent.ie_Class == IECLASS_RAWKEY &
      & copyevent.ie_Code == F1KEYUP) break;
      /* exit from forever */
else
{
      Forbid();
      if(copyevent.ie_TimeStamp.tv_secs != oldseconds ||
            copyevent.ie_TimeStamp.tv_micro != oldmicro ||
```

```
            copyevent.ie_Class != oldclass )
        {
                oldseconds = copyevent.ie_TimeStamp.tv_secs;
                oldmicro   = copyevent.ie_TimeStamp.tv_micro;
                oldclass   = copyevent.ie_Class;
                showEvents(&copyevent);
        }
        Permit();
    }
    }
    /* Although this task sleeps (main loop), the handler is independently
     * called by the input device.
     */

    /* For keystrokes that might be recognized by AmigaDOS, such as
     * alphabetic or numeric keys, you will notice that after the
     * first such keystroke, AmigaDOS appears to lock out your task
     * and accepts all legal keystrokes until you finally hit return.
     * This is absolutely true.... when both you and AmigaDOS try to
     * write into the same window, as is true if you run this program
     * from the CLI, the first keystroke recognized by AmigaDOS locks
     * the layer into which it is writing.  Any other task trying
     * to write into this same layer is put to sleep.  This allows
     * AmigaDOS to edit the input line and prevents other output to
     * that same window from upsetting the input line appearance.
     * In the same manner, while your task is sending a line of output,
     * AmigaDOS can be put to sleep it too must output at that time.
     *
     * You can avoid this problem if you wish by opening up a separate
     * window and a console device attached to that window, and output
     * strings to that console.  If you click the selection button on
     * this new window, then AmigaDOS won't see the input and your
     * task will get to see all of the keystrokes.  The other alternative
     * you can use, for demonstration sake, is to:
     *
     *    1. Make the AmigaDOS window slightly smaller in the
     *          vertical direction.
     *    2. Then click in the Workbench screen area outside
     *          of any window.
     *
     * Now there is no console device (particularly not AmigaDOS's
     * console) receiving the raw key stream and your task will report
     * as many keystrokes as it can catch (while not sleeping, that
     * is).
     */

    /* remove the handler from the chain */
    inputRequestBlock->io_Command = IND_REMHANDLER;
    inputRequestBlock->io_Data = (APTR)&handlerStuff;
    DoIO(inputRequestBlock);
```

```
    /* close the input device */
    CloseDevice(inputRequestBlock);

    /* delete the IO request */
    DeleteStdIO(inputRequestBlock);

    /* free other system stuff */
    DeletePort(inputDevPort);
    DeleteTimer(mytimerRequest);
}                           /* end of main */

int
showEvents(e)
struct InputEvent *e;
{
    printf("\n\nNew Input Event");
    printf("\nie_Class = %lx",e->ie_Class);
    printf("\nie_SubClass = %lx",e->ie_SubClass);
    printf("\nie_Code = %lx", e->ie_Code);
    printf("\nie_Qualifier = %lx",e->ie_Qualifier);
    printf("\nie_X = %ld", e->ie_X);
    printf("\nie_Y = %ld", e->ie_Y);
    printf("\nie_TimeStamp(seconds) = %lx", e->ie_TimeStamp.tv_secs);
    return(0);
}


*****************************************************************************
*  HandlerInterface()
*
*  This code is needed to convert the calling sequence performed by
*  the input.task for the input stream management into something
*  that a C program can understand.
*
*  This routine expects a pointer to an InputEvent in A0, a pointer
*  to a data area in A1.  These values are transferred to the stack
*  in the order that a C program would need to find them.  Since the
*  actual handler is written in C, this works out fine.

   XREF   _myhandler
   XDEF   _HandlerInterface

_HandlerInterface:
   MOVEM.L    A0/A1,-(A7)
   JSR        _myhandler
   ADDQ.L     #8,A7
   RTS

   END
```

Chapter 6

Keyboard Device

This chapter describes the keyboard device, which gives system access to the Amiga keyboard.

6.1. INTRODUCTION

When you send this device the command to read one or more keystrokes from the keyboard, for each keystroke (whether key-up or key-down) the keyboard device creates a data structure called an input event, to describe what happened. A keyboard input event includes:

- the key code (including up or down transition status)
- information about the current state of the left and right shift key
- whether the key came from the numeric keypad area

Thus the keyboard device provides more information than simply the "raw"key input that might be obtained by directly reading the hardware registers. In addition, the keyboard device can buffer keystrokes for you. If your task takes more time to process prior keystrokes, the keyboard device senses additional keystrokes and saves several keystrokes as a type-ahead feature. If your task takes an exceptionally long time to read this information from the keyboard, any keystrokes queued up beyond the number the system can handle will be ignored. Normally, the input device task processes these keyboard events, turning them into input device events to that no keystrokes are lost. You can find more information about keyboard event queueing in the chapter titled "Input Device" in the topic titled "Input Device and Intuition".

6.2. KEYBOARD DEVICE COMMANDS

The keyboard device allows the following system functions. The system functions operate normally.

Command	Operation
OpenDevice()	Obtain shared use of the keyboard device.
CloseDevice()	Relinquish use of the keyboard device.
DoIO()	Initiate a command, and wait for it to complete.
SendIO()	Initiate a command, and return immediately.
AbortIO()	Abort a command already in the queue.

The keyboard device also responds to the following commands:

I/O Command	Operation
KBD_ADDRESETHANDLER	add a reset handler to the device.
KBD_REMRESETHANDLER	remove a reset handler from the device.
KBD_RESETHANDLERDONE	indicate that a handler has completed its job and reset could possible occur now.
KBD_READMATRIX	read the state of every key in the keyboard.
KBD_READEVENT	read one (or more) key event from the keyboard device.

KBD_ADDRESETHANDLER

This command adds a routine to a chain of reset-handlers. When a user presses the key sequence CTRL-left AMIGA-right AMIGA (the reset sequence), the keyboard device senses this and calls a prioritized chain of reset-handlers. These might be thought of as cleanup routines that "must" be performed before reset is allowed to occur. For example, if a disk write is in process, the system should finish that before resetting the hardware so as not to corrupt the contents of the disk. There are probably few reasons why a program may wish to add its own reset handler as well. Note that if you add your own handler to this chain, you *must* ensure that your handler allows the rest of reset processing to occur. Reset *must* continue to function.

You add a handler to the chain by the command KBD_ADDRESETHANDLER. Assuming that you have a properly initialized IOStdReq block as a result of a call to **OpenDevice()** (for the input device), here is a typical C-language call to the IND_ADDRESETHANDLER function:

```
struct Interrupt resetHandlerStuff;
resetHandlerStuff.is_Data = &resetHandlerData;
        /* address of its data area */
resetHandlerStuff.is_Code = myResetHandler;
        /* address of entry point to handler */
resetHandlerStuff.is_Node.ln_Pri = myPriority;
keyboardRequestBlock.io_Command = KBD_ADDRESETHANDLER;
keyboardRequestBlock.io_Data = &resetHandlerStuff;

DoIO(&keboardRequestBlock);
```

The priority field in the list node structure establishes the sequence in which reset handlers are processed by the system. Your routine should be structured so that it can be called as though from the following C-language sequence:

```
myResetHandler(resetHandlerData);
```

Any return value from this routine is ignored. All keyboard reset handlers are activated if time permits.

The final command in your handler routine should be KBD_RESETHANDLERDONE, as described below.

NOTE: Due to the time-critical nature of handlers, handlers are usually written in assembly code. However, keyboard reset processing can take a little longer and is therefore less critical if written in a language such as C.

KBD_REMRESETHANDLER

This command is used to remove a keyboard reset handler from the system. The only differences from the calling sequence shown in KBD_ADDRESETHANDLER above is a change in the command number to KBD_REMRESETHANDLER, and there is no need to specify the priority of the handler.

KBD_RESETHANDLERDONE

This command tells the system that this handler is finished with its essential activities. If this is the last handler in the chain, it completes the reset sequence. If not, then the next handler in the chain gets its chance to function.

Here is a typical statement sequence used to end a keyboard reset handler, again assuming a properly initialized **inputRequestBlock**:

```
keyboardRequestBlock.io_Command = KBD_RESETHANDLERDONE;
keyboardRequestBlock.io_Data = &resetHandlerStuff;
SendIO(&keyboardRequestBlock);
return;          /* return so that other handlers can
          * also do their jobs
          */
```

Note that **SendIO()** is used instead of **DoIO()**. This routine is being executed within a software interrupt, and it is illegal to allow a **Wait()** within such routines.

KBD_READMATRIX

This command lets you discover the current state (UP = 0, DOWN = 1) of every key in the key matrix. You provide a data area which is at least large enough to hold one bit per key, approximately 16 bytes. The keyboard layout is shown in the figure below, indicating the numeric value each transmits (raw) when it is pressed. This value is the numeric position that this key occupies in the key-matrix read by this command.

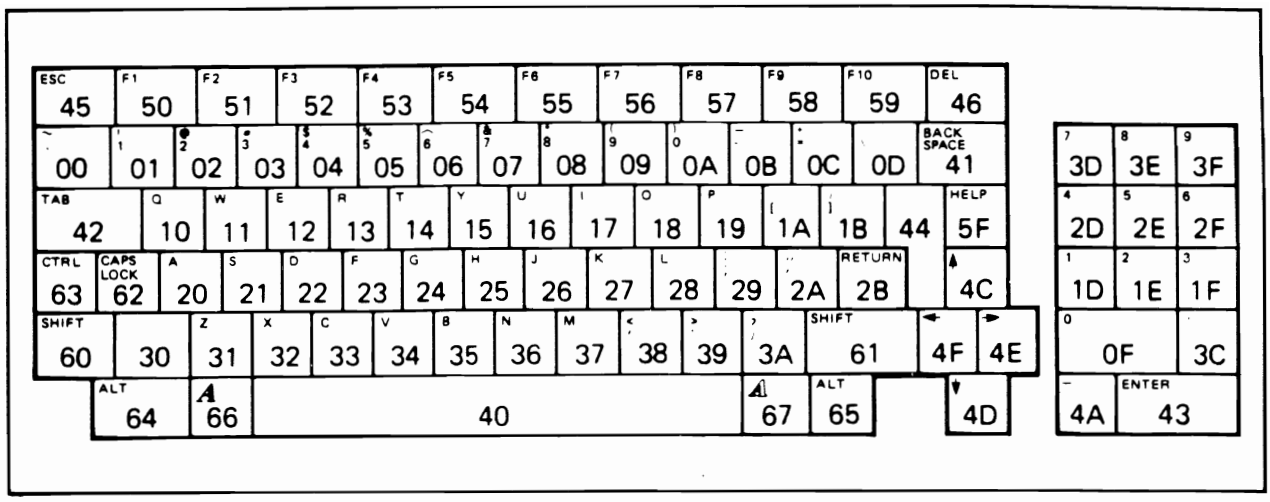


Figure 6-1: Raw Key Matrix

Assuming that you have already initialized an IOStdReq block for communication with the keyboard device, here is a typical calling sequence for sending the read-matrix command:

```
UBYTE keyMatrix[16];
keyboardRequestBlock.io_Command = KBD_READMATRIX;
keyboardRequestBlock.io_Data = &keyMatrix[0];
    /* where to put the key matrix */
DoIO(&keyboardRequestBlock);
```

Now to find the status of a particular key (for example, if the F2 key is down), you find the bit that specifies the current state by dividing the key matrix value (hex 51 = decimal 81) by 8. This indicates that the bit is in byte number 10 of the matrix. Then take the same number (decimal 81) modulo 8 to determine which bit position within that byte represents the state of the key. This yields a value of 1. So, by reading bit position 1 of byte number 10, you determine the status of the function key F2.

Reading keyboard events is normally done at a different level than by direct access to the keyboard device. See the documentation for the input device for the intimate linkage between that device and the keyboard device. This section is provided primarily to show you the parts of which a keyboard input event is composed.

The figure above shows the code value which each key places into the **ie_Code** field of the input event for a key down event. For a key up event, a value of hexadecimal 80 is or'ed with the value shown above. Additionally, if either shift key is down, or if the key is one of those in the numeric keypad, the qualifier field of the keyboard input event will be filled in accordingly.

NOTE: The keyboard device can queue up several keystrokes without a task requesting a report of keyboard events. However, when the keyboard event buffer has been filled with no task interaction, additional keystrokes will be discarded.

6.3. EXAMPLE KEYBOARD READ-EVENT PROGRAM

NOTE: This sample program will run properly only if the DOS and input device are not active.

```
/* sample program to demonstrate direct communications with the keyboard,
 * won't work unless input device is disabled, so that keyboard can
 * be accessed individually. (It will compile and it will run, but
 * this program will get some of the keyboard's inputs, and the input
 * device will steal the rest... no guarantee that F1 Key can break it out.
 *
 * To try the program, if run under the AmigaDOS CLI, strike any key, then
 * hit return. (You won't see any responses until each return key... DOS
 * is sitting on the input stream with its input editor as well as the
 * input device.) By rapidly hitting F1 then Return several times,
 * eventually you can generate a hex 50 that exits the program. This
 * program is provided for those who are taking over the machine. It
 * is not intended as a general purpose keyboard interface under DOS.
 */

#include <exec/types.h>
#include <exec/io.h>
#include <exec/devices.h>
```

```
#include <devices/keyboard.h>
#include <devices/inputevent.h>

#define F1KEY 0x50

extern struct MsgPort *CreatePort();
extern struct IOStdReq *CreateStdIO();

SHORT error;

struct IOStdReq *keyreq;
struct MsgPort *keyport;
struct InputEvent *keydata;      /* pointer into the returned data area
                                    where an input event has been sent */
BYTE keybuffer[sizeof( struct InputEvent )];

main()
{
      keyport = CreatePort(0,0);
      if(keyport == 0) { printf("\nError during CreatePort");
                         exit(-1);
                       }
      keyreq = CreateStdIO(keyport);
            /* make an io request block for
             * communicating with the keyboard */
      if(keyreq == 0) { printf("\nError during CreateStdIO");
                        DeletePort(keyport);
                        exit(-2);
                      }
      error = OpenDevice("keyboard.device",0,keyreq,0);
                  /* open the device for access */

      if (error != 0) { printf("\nCan't open keyboard!");
                        ReturnMemoryToSystem();
                        exit(-100);
                      }
      keyreq->io_Length = sizeof(struct InputEvent);
      /* read one event each time we go back to the keyboard */

      keyreq->io_Data = (APTR)keybuffer;
      /* show where to put the data when read */

      keydata = (struct InputEvent *)keybuffer;

      keyreq->io_Command = KBD_READEVENT;    /* get an event!! */


      for(;;)       /* FOREVER */
      {
      printf("\n Ready to retrieve another key0);
```

```
        DoIO( keyreq );
        if(keydata->ie_Code == F1KEY) break;
        printf("\n Raw key found this time was %lx",keydata->ie_Code);
        }
        printf("\nFINALLY found an F1 key!!!  Exiting...");
        ReturnMemoryToSystem(); /* can't get here because of FOREVER,
                                 * but if user provides an exit..... */
}

ReturnMemoryToSystem()
{
        DeleteStdIO(keyreq);
        DeletePort(keyport);
        return(0);
}
```

Chapter 7

Gameport Device

This chapter shows you how to use the gameport device, which is the means of access to the Amiga gameports.

7.1. INTRODUCTION

There are two units in the gameport device. Unit 0 controls the front gameport connector (connector 1). Unit 1 controls the rear gameport connector (connector 2).

You must tell the system the type of device connected to the gameport connector and how the device is to respond. That is, should the device return status immediately each time you ask for information, or should it only return status once certain conditions have been met?

When the input device is operating, the left gameport connector is usually dedicated to that device. Therefore, this chapter's examples concentrate on the right connector, which is not dedicated to the input device. Note that if the input device is not started, the left connector, as gameport unit 0, can perform the same functions as shown below for the right connector.

When a gameport unit finally reponds to a request for input, it formulates an input event. The contents of the input event vary based on the type of device you have told the unit is connected and the trigger conditions it must look for.

7.2. GAMEPORT DEVICE COMMANDS

The gameport device allows the following system functions.

Command	Operation
OpenDevice()	Obtain exclusive use of one unit of the gameport device. Returns an error value of -1 if another task already has control of the unit you have requested.
CloseDevice()	Relinquish use of the gameport device.
DoIO()	Initiate a command and wait for it to complete.
SendIO()	Initiate a command and return immediately.
AbortIO()	Abort a command already in the queue.

The gameport device also responds to the following commands:

I/O Command	Operation
GPD_SETCTYPE	set the type of the controller to be monitored.
GPD_ASKCTYPE	ask the type of the controller being monitored.
GPD_SETTRIGGER	preset the conditions that will trigger a gameport event.
GPD_ASKTRIGGER	inquire the conditions that have been preset for triggering.
GPD_READEVENT	read one (or more) gameport events from an initialized unit.

GPD_SETCTYPE

This command establishes the type of controller that is to be connected to the specific gameport device. You must have already successfully opened that specific unit before you will be able to tell it what type of controller is connected. As of this writing, there are three different legal controller types: mouse, absolute-joystick, relative-joystick, and "no-controller".

A mouse controller can report input events for one, two or three buttons, and for positive or negative (x,y) movements. A trackball controller or driving controller for various games is generally of the same type, and can be declared as a mouse controller.

An absolute joystick is one that reports one single event for each change in its current location. If, for example, the joystick is centered and a user pushes the stick forward, a forward-switch event will be generated. A relative joystick, on the other hand, is comparable to an absolute joystick with "autorepeat" installed. As long as the user holds the stick in a position other than centered, the gameport device continues to generate position reports.

As of this writing, there is no direct system software support for proportional joysticks or proportional controllers.

You specify the controller type by the following code or its equivalent:

```
struct IOStdReq *gameIOMsg;

setControllerType(type)
UBYTE *type;
{
gameIOMsg->io_Command = GPD_SETCTYPE;
      /* set type of controller */
gameIOMsg->io_Data = &type;
      /* show where data can be found */
DoIO(gameIOMsg);
return(0);
}
```

GPD_GETCTYPE

You use this command to find out what kind of controller has been specified for a particular unit. This command puts the controller type into the data area that you specify with the command. Here is a sample call:

```
SHORT getControllerType(type);
UBYTE *type;
{
gameIOMsg->io_Command = GPD_GETCTYPE;
      /* get type of controller */
gameIOMsg->io_Data = &type;
      /* show where data should be placed */
DoIO(gameIOMsg);
return (gamebuffer[0]);
}
```

The value that is returned corresponds to one of the four controller types noted in GPD_SETCTYPE above. Controller type definitions can be found in the include-file named *gameport.h.*

GPD_SETTRIGGER

You use this command to specify the conditions that can trigger a gameport event. The device won't reply to your read request until the trigger conditions have been satisfied.

For a mouse device, you can trigger on a certain minimum-sized move in either the x or y direction, on up or down transitions of the mouse buttons, on a timed basis, or any combination of these conditions. Here is an example that shows why you might want to use both time and movement. Suppose you normally signal mouse events if the mouse moves at least 10 counts in either the x or y directions. If you are moving the cursor to keep up with mouse movements and the user moves the mouse less than 10 counts, after a period of time you will want to update the position of the cursor to exactly match the mouse position. Thus the timed report with current mouse counts will be desirable.

For a joystick device, you can select timed reports as well as button-up and button-down report trigger conditions.

The information needed for gameport trigger setting is placed into a **GameTrigger** data structure:

```
struct GamePortTrigger {
    UWORD   gpt_Keys;       /* key transition triggers */
    UWORD   gpt_Timeout;    /* time trigger (vertical blank units) */
    UWORD   gpt_XDelta;     /* X distance trigger */
    UWORD   gpt_YDelta;     /* Y distance trigger */
    };
```

The field **gpt_Keys** can be set to a value of GPTF_UPKEYS to report up-transitions or GPTF_DOWNKEYS to report down-transitions.

The field **gpt_Timeout** is set to count how many vertical blank units should occur (1/60th of a second each) between reports in the absence of another trigger condition. Thus, this specifies the maximum report interval.

NOTE: If a task sets trigger conditions and does not ask for the position reports (by sending an IORequest to be filled-in with available reports), the gameport device will queue up several additional reports. If the trigger conditions again occur and there as many events as the system can handle are already queued, the additional triggers will be ignored until the buffer of one or more of the existing triggers is read by a device read request.

```
struct GamePortTrigger mousetrigger = {
    GPTF_UPKEYS + GPTF_DOWNKEYS,
    1800,
    XMOVE,
    YMOVE };
    /* trigger on all mouse key transitions,
    every 30 seconds, (1800 = 30 times 60 per sec)
    for any 10 in an x or y direction */
```

You set the trigger by using the following code or the equivalent:

```
gameIOMsg->io_Command = GPD_SETTRIGGER;
      /* command to set the trigger conditions */
gameIOMsg->io_Data = &mousetrigger;
      /* show where to find the trigger condition info */
DoIO(gameIOMsg);
```

7.3. EXAMPLE PROGRAMS

Mouse Program

Here is a complete sample program that lets you open the right gameport device unit and define it as a mouse device. You are directed to unplug the mouse and plug it into the right connector. Mouse moves and button clicks are reported to the console device that started the program. If you don't move the mouse for 30 seconds, a report is generated automatically. If you don't move it for 2 minutes, the program exits.

```
/* ********************************************************************** */
/* mouse test, for right game port on the Amiga.

   Notes: The right port is used for this test because the input.device
   task is busy continuously with the lefthand port, feeding input events
   to intuition or console devices. If Intuition is not activated
   (applications which take over the whole machine may decide not to
   activate Intuition), and if no console.device is activated either,
   the input.device will never activate... allowing the application
   free reign to use either the left OR the right hand joystick/mouse
```

```
  port.  If either intuition or the console device are activated,
  the lefthand port will yield, at best, every alternate input
  event to an external application such as this test program.

  This will undoubtedly mess up either of the two applications
  and should therefore be avoided.  It was ok to use the right
  port in this case, since the system has no particular interest
  in monitoring it.

  Using a function called SetMPort, you can reconfigure so that the
  mouse is expected in the other port, but that isnt demonstrated here.
************************************************************************** */

#include <exec/types.h>
#include <exec/devices.h>
#include <graphics/gfx.h>
#include <devices/gameport.h>
#include <devices/inputevent.h>

LONG GfxBase=0;

#define XMOVE 10
#define YMOVE 10
#define MAX(m,n) (m > n ? m : n)

/* trigger on all mouse key transitions, and every
 * 30 seconds, and for any 10 in an x or y direction */

struct GamePortTrigger mousetrigger = {
      GPTF_UPKEYS + GPTF_DOWNKEYS,
      1800,
      XMOVE,
      YMOVE };

struct InputEvent *game_data;    /* pointer into the returned data area
                        * where input event has been sent */
SHORT          error;

struct IOStdReq *game_io_msg;

BYTE           gamebuffer[sizeof( struct InputEvent )];
BYTE           *gamedata;

SHORT          testval;

struct MsgPort  *game_msg_port;

SHORT movesize;
extern struct MsgPort *CreatePort();
extern struct IOStdReq *CreateStdIO();
```

```
SHORT codeval, timeouts;

#define IF_NOT_IDLE_TWO_MINUTES while(timeouts < 4)

main()
{

        GfxBase = OpenLibrary("graphics.library", 0);
        if (GfxBase == NULL)
        {
                printf("Unable to open graphics library\n");
                exit(1000);
        }
        printf("Mouseport Demo\n");
        printf("\nMove Mouse from Left Port to Right Port\n");
        printf("\nThen move the mouse and click its buttons");


        timeouts = 0;

        gamedata = &gamebuffer[0];
        /* point to first location in game buffer */

        game_msg_port = CreatePort(0,0);
        /* provide a port for the IO response */
        if(game_msg_port == 0)
        {
                printf("\nError While Performing CreatePort");
                exit(-1);
        }

        game_io_msg = CreateStdIO(game_msg_port);
        /* make an io request block for communicating with
                                the keyboard */

        if(game_io_msg == 0)
        {
                printf("\nError While Performing CreateStdIO");
                DeletePort(game_msg_port);
                exit(-2);
        }

        error = OpenDevice("gameport.device",1,game_io_msg,0);
        /* open the device for access, unit 1 is right port */

        if(error != 0)
        {
                printf("\nError while opening the device, exiting");
                DeleteStdIO(game_io_msg);
                DeletePort(game_msg_port);
```

```
        exit(-3);
    }

    game_io_msg->io_Length = sizeof(struct InputEvent);
    /* read one event each time we go back to the gameport */

    game_io_msg->io_Data = (APTR)gamebuffer;
    /* show where to put the data when read */

    game_data = (struct InputEvent *)gamebuffer;

    /* test the mouse in this loop */

    set_controller_type(GPCT_MOUSE);

    /* specify the trigger conditions */
    game_io_msg->io_Command = GPD_SETTRIGGER;
    /* show where to find the trigger condition info */
    game_io_msg->io_Data = (APTR)&mousetrigger;

    /* this command doesn't wait... returns immediately */
    SendIO(game_io_msg);
    WaitPort(game_msg_port);
    GetMsg(game_msg_port);

    printf("\nI will report:");
    printf("\n     Mouse X or Y moves if either is over 10 counts");
    printf("\n     Button presses (along with mouse moves if any)");
    printf("\n     Or every 30 seconds (along with mouse moves if any)");
    printf("\n          if neither move or click happens\n");
    printf("\nIf no activity for 2 minutes, the program exits\n");

    /* from now on, just read input events
     * into the input buffer, one at a time.
     * read-event waits for the preset conditions */

    game_io_msg->io_Command = GPD_READEVENT;
    game_io_msg->io_Data = (APTR)gamebuffer;

    IF_NOT_IDLE_TWO_MINUTES
    {
        game_io_msg->io_Length = sizeof(struct InputEvent);
        /* read one event each time we go back to the gameport */

        printf("\n Waiting For Mouse Report\n");

        SendIO(game_io_msg);

        WaitPort(game_msg_port);
        /* this is NOT a busy wait... it is a task-sleep */
```

```
        GetMsg(game_msg_port);

        codeval = game_data->ie_Code;
        switch(codeval)
        {
        case IECODE_LBUTTON:
              printf("\nMouse Left Button Pressed");
              maybe_mouse_moved();
              break;

        case IECODE_RBUTTON:
              printf("\nMouse Right Button Pressed");
              maybe_mouse_moved();
              break;

        case (IECODE_LBUTTON + IECODE_UP_PREFIX):
              printf("\nMouse Left Button Released");
              maybe_mouse_moved();
              break;

        case (IECODE_RBUTTON + IECODE_UP_PREFIX):
              printf("\nMouse Right Button Released");
              maybe_mouse_moved();
              break;

        case IECODE_NOBUTTON:
              timeouts++;     /* after 2 minutes, dump program
                               * if user loses interest
                               */
              movesize = maybe_mouse_moved();
              if(movesize == 0)
              {
                 printf("\n30 seconds passed, no trigger events");
              }
              else if(movesize < XMOVE && movesize < YMOVE )
              {
                 printf("\n(Even though less than trigger count,");
                 printf("\n reporting mouse move at the selected");
                 printf("\n timing interval for user info)");
              }
              break;
         default:
              break;
        }
}

set_controller_type(GPCT_NOCONTROLLER);

CloseDevice(game_io_msg);
DeleteStdIO(game_io_msg);
```

```
    DeletePort(game_msg_port);

    printf("\nExiting program... 2 minutes with no activity sensed\n1> ");
    return(0);
}

    /* if mouse didnt move far enough to trigger a report, then caller
     * will also report that 30 seconds (1800 vblanks) has elapsed
     */

int maybe_mouse_moved()
{
    int xmove, ymove;
    xmove = game_data->ie_X;
    ymove = game_data->ie_Y;

    if(xmove != 0 || ymove != 0)
    {
        printf("\nMouse Moved by X-value %ld, Y-value %ld",
        xmove, ymove);
        timeouts = 0;
    }
    if(xmove < 0) xmove = -xmove;
    if(ymove < 0) ymove = -ymove;

    return(MAX(xmove,ymove));
}

int set_controller_type(type)
SHORT type;
{
    /* set type of controller to mouse */
    game_io_msg->io_Command = GPD_SETCTYPE;

    *gamedata = type;

    /* set it up */
    /* this command doesn't wait... returns immediately */
    SendIO(game_io_msg);

    WaitPort(game_msg_port);
    GetMsg(game_msg_port);
    return(0);
}
```

Joystick Program

```
/* ************************************************************************ */
/* joystick test, for right game port on the Amiga.

   Notes: The right port is used for this test because the input.device
   task is busy continuously with the lefthand port, feeding input events
   to intuition or console devices. If Intuition is not activated
   (applications which take over the whole machine may decide not to
   activate Intuition), and if no console.device is activated either,
   the input.device will never activate... allowing the application
   free reign to use either the left OR the right hand joystick/mouse
   port. If either intuition or the console device are activated,
   the lefthand port will yield, at best, every alternate input
   event to an external application such as this test program.
   This will undoubtedly mess up either of the two applications
   and should therefore be avoided. It was ok to use the right
   port in this case, since the system has no particular interest
   in monitoring it.

************************************************************************** */


#include <exec/types.h>
#include <exec/devices.h>
#include <graphics/gfx.h>
#include <devices/gameport.h>
#include <devices/inputevent.h>

LONG GfxBase=0;

#define XMOVE 10
#define YMOVE 10
#define MAX(m,n) (m > n ? m : n)
#define FOREVER for(;;)
struct InputEvent *game_data;   /* pointer into the returned data area
                                 * where input event has been sent */
SHORT           error;

struct IOStdReq *game_io_msg;

BYTE            gamebuffer[sizeof( struct InputEvent )];
BYTE            *gamebuff;

SHORT           testval;
SHORT           codevalue;
```

```
struct MsgPort  *game_msg_port;

SHORT movesize;
extern struct MsgPort *CreatePort();
extern struct IOStdReq *CreateStdIO();

SHORT codeval, timeouts;

#define IF_NOT_IDLE_TWO_MINUTES while(timeouts < 4)

main()
{
      printf("Joystick Demo\n");
      printf("\nPlug a Joystick Into Right Port\n");
      printf("\nThen move the stick and click its buttons");


      /* point to first location in game buffer */
      gamebuff = &gamebuffer[0];

      /* SYSTEM DEVICE COMMUNICATIONS SUPPORT SETUP ROUTINES ****** */

      /* provide a port for the IO response */
      game_msg_port = CreatePort(0,0);
      if(game_msg_port == 0)
      {
            printf("\nError While Performing CreatePort");
            exit(-1);
      }

      /* make an io request block for communicating with the gameport */
      game_io_msg = CreateStdIO(game_msg_port);

      if(game_io_msg == 0)
      {
            printf("\nError While Performing CreateStdIO");
            DeletePort(game_msg_port);
            exit(-2);
      }
      /* ********************************************************** */
      /* OPEN THE DEVICE */

      /* open the device for access, unit 1 is right port */
      error = OpenDevice("gameport.device",1,game_io_msg,0);

      if(error != 0)
      {
            printf("\nError while opening the device, exiting");
            DeleteStdIO(game_io_msg);
            DeletePort(game_msg_port);
```

```
        exit(-3);
}
/* ********************************************************** */
/* SET THE DEVICE TYPE */

game_data == (struct InputEvent *)gamebuffer;

/* test the joystick in this loop */

if (set_controller_type(GPCT_ABSJOYSTICK) != 0)
{
        printf("\nError while trying to set GPCT_ABSJOYSTICK");
        DeleteStdIO(game_io_msg);
        DeletePort(game_msg_port);
        exit(-4);
}
/* ********************************************************** */
/* SET THE DEVICE TRIGGER */
if (set_controller_trigger() != 0)
{
        printf("\nError while trying to set controller trigger");
        DeleteStdIO(game_io_msg);
        DeletePort(game_msg_port);
        exit(-4);
}
/* ********************************************************** */
/* TELL USER WHAT YOU WILL BE DOING */

printf("\nI will report: \n");
printf("\n     Stick X or Y moves");
printf("\n     Button presses (along with stick moves if any)");

/* ********************************************************** */
/* SETUP THE IO MESSAGE BLOCK FOR THE ACTUAL DATA READ */

/* from now on, just read input events into the
 * input buffer, one at a time; read-event waits
 * for the preset conditions */

game_io_msg->io_Command = GPD_READEVENT;
game_io_msg->io_Data = (APTR)gamebuffer;

/* read one event each time we go back to the gameport */
game_io_msg->io_Length = sizeof(struct InputEvent);

/* dont use quick io */
game_io_msg->io_Flags = 0;

/* ********************************************************** */
/* LOOP FOREVER */
```

```
    FOREVER
    {
        /* read one event each time we go back to the gameport */
        game_io_msg->io_Length = sizeof(struct InputEvent);

        printf("\n Waiting For Joystick Report\n");
        SendIO(game_io_msg);
        WaitPort(game_msg_port);
        /* this is NOT a busy wait... it is a task-sleep */
        GetMsg(game_msg_port);

        codevalue = game_data->ie_Code;

        if(codevalue == IECODE_LBUTTON)
            printf("\nFire Button pressed");
        if(codevalue == (IECODE_LBUTTON + IECODE_UP_PREFIX))
            printf("\nFire Button released");

        which_direction();
        showbugs();
    }

    /* PROGRAM EXIT ..... temporarily no way to get here from FOREVER */
    set_controller_type(GPCT_NOCONTROLLER);

    CloseDevice(game_io_msg);
    DeleteStdIO(game_io_msg);
    DeletePort(game_msg_port);

    printf("\nExiting program... 2 minutes with no activity sensed\n1> ");
    return(0);
}

int which_direction()
{
    SHORT xmove, ymove;
    xmove = game_data->ie_X;
    ymove = game_data->ie_Y;

    switch(ymove)
    {
        case (-1):
        printf("\nForward");
        break;
        case (1):
        printf("\nBack");
        break;
    default:
        break;
    }
```

```
        switch(xmove)
        {
                case (-1):
                printf("\nLeft");
                break;
                case (1):
                printf("\nRight");
                break;
        default:
                break;
        }
        return(0);
}

int set_controller_type(type)
SHORT type;
{
        game_io_msg->io_Command = GPD_SETCTYPE;
        /* set type of controller to mouse */
        game_io_msg->io_Length = 1;
        game_io_msg->io_Data = (APTR)gamebuff;
        *gamebuff = type;

        SendIO(game_io_msg);
        /* set it up */
        /* this command doesn't wait... returns immediately */
        WaitPort(game_msg_port);
        GetMsg(game_msg_port);
        return((int)game_io_msg->io_Error);
}

int set_controller_trigger()
{
        struct GamePortTrigger gpt;

        game_io_msg->io_Command = GPD_SETTRIGGER;
        game_io_msg->io_Length = sizeof(gpt);
        game_io_msg->io_Data = (APTR)&gpt;
        gpt.gpt_Keys = GPTF_UPKEYS+GPTF_DOWNKEYS;
        gpt.gpt_Timeout = 0;
        gpt.gpt_XDelta = 1;
        gpt.gpt_YDelta = 1;

        return(DoIO(game_io_msg));
}

showbugs()
{
        struct InputEvent *e;
```

```
    e = (struct InputEvent *)&gamebuffer[0];
    /* where the input event gets placed */
    printf("\nie_Class = %lx",e->ie_Class);
    printf("\nie_SubClass = %lx",e->ie_SubClass);
    printf("\nie_Code = %lx", e->ie_Code);
    printf("\nie_Qualifier = %lx",e->ie_Qualifier);
    printf("\nie_X = %ld", e->ie_X);
    printf("\nie_Y = %ld", e->ie_Y);
    printf("\nie_TimeStamp(seconds) = %lx", e->ie_TimeStamp.tv_secs);
    return(0);
}
```

Chapter 8

Narrator Device

This chapter provides routines for accessing both the narrator device and the translator library and shows how some of the parameters passed to the device can affect the output.

In addition, this chapter contains a non-technical explanation of how to effectively utilize the speech device. A more technical explanation is also provided for those who may be interested in how the speech is actually produced.

8.1. INTRODUCTION

Two different subsystems comprise the speech system on the Amiga. They are

- The *narrator device*, which communicates with the audio device to actually produce human-like speech.
- The *translator library*, which contains a routine that translates english text into phonemes suitable for the narrator device.

8.2. THE TRANSLATOR LIBRARY

The translator library provides a single routine, named **Translate()**, that converts an English language string into a phonetic string. To use this function, you must first open the library.

Setting a global variable, **TranslatorBase**, to the value returned from the call to **OpenLibrary()** enables the Amiga linker to correctly locate the translator library:

```
struct Library *TranslatorBase;
    ...
TranslatorBase = OpenLibrary("translator.library",REVISION);
if(TranslatorBase == NULL) exit (CANT_OPEN_TRANSLATOR);
```

Note that to allow the **OpenLibrary()** call to succeed, the directory currently assigned by AmigaDOS as "LIBS:" must contain *translator.library*.

Using the Translate Function

Once the library is open, you can call the translate function:

```
UBYTE *sampleinput;       /* pointer to sample input string */
UBYTE outputstring[500];  /* place to put the translation */
SHORT rtnCode;                  /* return code from function */

sampleinput = "this is a test"; /* a test string of 14 characters */
rtnCode = Translate(sampleinput,14,outputstring,500);
```

The input string will be translated into its phoneme equivalent, and can be used to feed the narrator device.

If you receive a nonzero return code, you haven't provided enough output buffer space to hold the entire translation. In this case, the **Translate()** function breaks the translation at the end of a word in the input stream and returns the position in the input stream at which the translation ended. You can use the output buffer, then call the **Translate()** function again, starting at this original ending position, to continue the translation where you left off.

Note, however, that the value returned is *negative*. Therefore you must use **-rtnCode** as the starting point for a new translation.

Additional Notes About Translate

The English language has many words that don't sound the same as they are spelled. The translator library has an exception table that it consults as the translation progresses. Words that are not in the exception table are translated literally. Therefore, it is possible that certain words will not translate well. You can improve the quality of the translation, by handling those words on your own, using the tutorial information included at the end of this chapter.

As with all other libraries of routines, if you have opened the translator library for use, be sure to close it before your program exits. If the system needs memory resources, it can then expell closed libraries to gain additional space.

8.3. THE NARRATOR DEVICE

The narrator device on the Amiga provides two basic functions:

- You can write to the device and ask it to speak a phoneme-encoded string in a specific manner—pitch, male/female, various speaking rates, and so on.
- You can read from the device. As it speaks, the device can generate mouth shapes for you and you can use the shapes to perform a graphics rendering of a face and mouth.

Opening the Narrator Device

To use the narrator device, you must first open the device. The narrator device is disk-resident. For the **OpenDevice()** call to succeed, the narrator device must be present in the directory currently assigned by AmigaDOS to the "DEVS:" directory.

As with all devices, you must pass to **OpenDevice()** an **IORequest** block for communicating with the device. The block used by the narrator device for a write is a special format called a **narrator_rb**. The block used for a read is also a special format, called a **mouth_rb**. Both blocks are described in the sections that follow. A sample **OpenDevice()** sequence for the narrator device follows. Notice that two request blocks are created, one for

writing to the device, and one for reading from it. For brevity, the error checking is left out of this short example. It is, however, utilized in the sample program later on.

```
struct narrator_rb *writeNarrator;
struct narrator_rb *readNarrator;
writeport = CreatePort(0,0);
readport = CreatePort(0,0);
writeNarrator = (struct narrator_rb *)CreateExtIO(writeport,
        sizeof(struct narrator_rb);
readNarrator = (struct mouth_rb *)CreateExtIO(readport,
        sizeof(struct mouth_rb);
```

The routine **CreateExtIO()** is in the Exec support routines appendix of this manual. **CreatePort()** is contained in *amiga.lib* and can be accessed by linking your program to *amiga.lib*.

Contents of the Write Request Block

You can control several characteristics of the speech, as indicated in the narrator request block structure shown below.

```
struct narrator_rb {
    struct IOStdReq  message;  /* Standard IORB */
    UWORD   rate;              /* Speaking rate (words/minute) */
    UWORD   pitch;             /* Baseline pitch in Hertz */
    UWORD   mode;              /* Pitch mode */
    UWORD   sex;               /* Sex of voice */
    UBYTE   *ch_masks;         /* Pointer to audio alloc maps */
    UWORD   nm_masks;          /* Number of audio alloc maps */
    UWORD   volume;            /* Volume. 0 (off) thru 64 */
    UWORD   sampfreq;          /* Audio sampling freq */
    UBYTE   mouths;            /* If non-zero, generate mouths */
    UBYTE   chanmask;          /* Which ch mask used (internal)*/
    UBYTE   numchan;           /* Num ch masks used (internal) */
    UBYTE   pad;               /* For alignment */
};
```

where:

rate
is the speed in words per minute that you wish it to speak.

pitch
is the baseline pitch. If you are using an expressive voice rather than a monotone, the pitch will vary above and below this baseline pitch.

mode
determines whether you have a monotone or expressive voice.

sex
determines if the voice is male or female.

ch_masks, nm_masks, volume, sampfreq
are described in the chapter called "Audio Device."

mouths
is set to nonzero before starting a write if you want to read mouths using the read command while the system is speaking.

chanmask, numchan, pad
are for system use only.

The system default values are shown in the files *narrator.h* and *narrator.i*. When you call **OpenDevice()**, the system initializes the request block to the default values. If you want other than the defaults, you must change them *after* the device is open.

Contents of the Read Request

The **mouth_rb** data structure follows. Notice that it is an extended form of the **narrator_rb** structure.

```
struct mouth_rb {
      struct  narrator_rb voice;     /* Speech IORB                */
      UBYTE   width;                 /* Width (returned value)     */
      UBYTE   height;                /* Height (returned value)    */
      UBYTE   shape;                 /* Internal use, do not modify */
      UBYTE   pad;                   /* For alignment              */
      };
```

The fields **width** and **height** will, on completion of a read-request, contain an integer value proportional to the mouth width and height that are appropriate to the phoneme currently being spoken. When you send a read request, the system doesn't return any response until

one of two things happens. Either a different mouth size is available (this prevents you from drawing and redrawing the same shape or having to check whether or not it is the same) or the speaking has completed. You must check the error return field when the read request block is returned to determine if the request block contains a new mouth shape or simply is returning status of **ND_NoWrite** (no write in progress, all speech ended for this request).

Opening the Narrator Device

This section demonstrates opening the device as well as synchronizing a read request so that it responds only to the write request for which the device is opened. You can only read the mouth shapes if the write request contains the same unit number and a write is currently in progress; the system returns an error if the numbers don't match or if the write has completed. Note again that error checking is deferred to the example program at the end of the chapter.

```
SHORT openError;

openError = OpenDevice("narrator.device",0,writeNarrator,0);
      /* after error checking, synchronize the read and write requests */
readNarrator->narrator_rb.message.io_Device =
      writeNarrator->message.io_Device;     /* copy device info */
readNarrator->narrator_rb.message.io_Unit =
      writeNarrator->message.io_Unit; /* copy unit info */
```

At this point, it is acceptable to change the default values before issuing a write.

More details about what **OpenDevice()** performs are contained in the narrator device summary pages.

Performing a Write and a Read

You normally perform a write command by using the functions **BeginIO()** or **SendIO()** to transmit the request block to the narrator device. This allows the narrator's task to begin the I/O, while your task is free to do something else. The something else may be issuing a series of read commands to the device to determine mouth shapes and drawing them on-screen. The following sample set of function calls implements both the write and read commands in a single loop. Again, error checking is deferred to the sample program.

```
SHORT readError;

writeNarrator->message.io_Length = strlen(outputstring);
      /* tell it how many characters the translate function returned */
writeNarrator->message.io_Data = outputstring;
      /* tell it where to find the string to speak */
SendIO(writeNarrator);
      /* return immediately, run tasks concurrently */

readNarrator->voice.message.io_Error = 0;
while((readError = readNarrator->voice.message.io_Error) !=
      ND_NoWrite)
{
      DoIO(readNarrator);
            /* put task to sleep waiting for a different
             * mouth shape or return of the message block
             * with the error field showing no write in
             * process
             */
      DrawMouth(readNarrator->width,readNarrator->height);
            /* user's own unique routine, not provided here */
}
GetMsg(writeport);     /* remove the write message from the
                        * writeport so that it can be reused */
```

The loop continues to send read requests to the narrator device until the speech output has ended. **DoIO()** automatically removes the read request block from the readport for reuse. **SendIO()** is used to transmit the write request. When it completes, the write request will be appended to the writeport, and must be removed before it can be reused.

8.4. SAMPLE PROGRAM

The following sample program uses the system default values returned from the **OpenDevice()** call. It translates and speaks a single phrase.

```
#include "exec/types.h"
#include "exec/exec.h"

#include "exec/nodes.h"
#include "exec/lists.h"
#include "exec/memory.h"
#include "exec/interrupts.h"
#include "exec/ports.h"
#include "exec/libraries.h"
```

```
#include "exec/io.h"
#include "exec/tasks.h"
#include "exec/execbase.h"

#include "devices/narrator.h"
#include "libraries/translator.h"

struct MsgPort *readport=0;
struct MsgPort *writeport=0;

extern struct MsgPort *CreatePort();
extern struct IORequest *CreateExtIO();

struct narrator_rb *writeNarrator=0;
struct mouth_rb *readNarrator=0;
struct Library *TranslatorBase=0;
UBYTE *sampleinput;       /* pointer to sample input string */
UBYTE outputstring[500];  /* place to put the translation */
SHORT rtnCode;                  /* return code from function */
SHORT readError;
SHORT writeError;
SHORT error;
BYTE  audChanMasks[4] = { 3,5,10,12 }; /* which channels to use */

#define CANT_OPEN_TRANSLATOR -100
#define CANT_OPEN_NARRATOR -200
#define CREATE_PORT_PROBLEMS -300
#define CREATE_IO_PROBLEMS -400
#define CANT_PERFORM_WRITE -500
#define REVISION 1

extern struct Library *OpenLibrary();

main()
{
      TranslatorBase = OpenLibrary("translator.library",REVISION);
      if(TranslatorBase == NULL) exit (CANT_OPEN_TRANSLATOR);
      sampleinput = "this is a test"; /* a test string of 14 characters */
      rtnCode = Translate(sampleinput,14,outputstring,500);
      error = rtnCode + 100;
      if(rtnCode != 0) goto cleanup0;

      writeport = CreatePort(0,0);
      if(writeport == NULL) { error=CREATE_PORT_PROBLEMS; goto cleanup1; }
      readport = CreatePort(0,0);
      if(readport == NULL) { error=CREATE_PORT_PROBLEMS; goto cleanup2; }
      writeNarrator = (struct narrator_rb *)CreateExtIO(writeport,
                              sizeof(struct narrator_rb));
      if(writeNarrator == NULL) { error=CREATE_IO_PROBLEMS; goto cleanup3; }
      readNarrator = (struct mouth_rb *)CreateExtIO(readport,
```

```
                    sizeof(struct mouth_rb));

    if(readNarrator == NULL) { error=CREATE_IO_PROBLEMS; goto cleanup4; }
/* SET UP PARAMETERS FOR WRITE-MESSAGE TO THE NARRATOR DEVICE */

    /* show where to find the channel masks */
    writeNarrator->ch_masks = (audChanMasks);

    /* and tell it how many of them there are */
    writeNarrator->nm_masks = sizeof(audChanMasks);

    /* tell it where to find the string to speak */
    writeNarrator->message.io_Data = (APTR)outputstring;

    /* tell it how many characters the translate function returned */
    writeNarrator->message.io_Length = strlen(outputstring);

    /* if nonzero, asks that mouths be calculated during speech */
    writeNarrator->mouths = 1;

    /* tell it this is a write-command */
    writeNarrator->message.io_Command = CMD_WRITE;

/* Open the device  */

    error = OpenDevice("narrator.device", 0, writeNarrator, 0);
    if(error != 0) goto cleanup4;

/* SET UP PARAMETERS FOR READ-MESSAGE TO THE NARRATOR DEVICE */

    /* tell narrator for whose speech a mouth is to be generated */
    readNarrator->voice.message.io_Device =
        writeNarrator->message.io_Device;
    readNarrator->voice.message.io_Unit =
        writeNarrator->message.io_Unit;

    readNarrator->width = 0;
    readNarrator->height = 0; /* initial mouth parameters */

    readNarrator->voice.message.io_Command = CMD_READ;
        /* initial error value */
    readNarrator->voice.message.io_Error = 0;

/* Send an asynchronous write request to the device */

    writeError = SendIO(writeNarrator);
    if(writeError != NULL) { error=CANT_PERFORM_WRITE; goto cleanup5; }
    /* return immediately, run tasks concurrently */

/* keep sending reads until it comes back saying "no write in progress" */
```

```
while((readError = readNarrator->voice.message.io_Error) !=
      ND_NoWrite)
{
      DoIO(readNarrator);
      /* put task to sleep waiting for a different
       * mouth shape or return of the message block
       * with the error field showing no write in
       * process
       */
      DrawMouth(readNarrator->width,readNarrator->height);
      /* user's own unique routine, not provided here */
}

Delay(30);

rtnCode = Translate("No it is not",13,outputstring,500);
writeNarrator->sex = FEMALE;
writeNarrator->pitch = MAXPITCH;  /* raise pitch from default value */
writeNarrator->message.io_Data = (APTR)outputstring;
writeNarrator->message.io_Length = strlen(outputstring);
DoIO(writeNarrator);

Delay(30);

rtnCode = Translate("Please! I am speaking now!",26,outputstring,500);
writeNarrator->sex = MALE;
writeNarrator->pitch = DEFPITCH;
writeNarrator->message.io_Data = (APTR)outputstring;
writeNarrator->message.io_Length = strlen(outputstring);
DoIO(writeNarrator);

Delay(30);

rtnCode = Translate(
      "Well, you are not very interesting, so I am going home!",
      55,outputstring,500);
writeNarrator->sex = FEMALE;
writeNarrator->pitch = MAXPITCH;
writeNarrator->message.io_Data = (APTR)outputstring;
writeNarrator->message.io_Length = strlen(outputstring);
DoIO(writeNarrator);

Delay(30);

rtnCode = Translate("Bye Bye",7,outputstring,500);
writeNarrator->sex = MALE;
writeNarrator->pitch = DEFPITCH;
writeNarrator->rate = 7;   /* slow him down */
writeNarrator->message.io_Data = (APTR)outputstring;
writeNarrator->message.io_Length = strlen(outputstring);
```

```
    DoIO(writeNarrator);

  cleanup5:
    if(writeNarrator != 0)
        CloseDevice(writeNarrator);
                /* terminate access to the device */

    /* now return system memory to the memory allocator */

  cleanup4:
    if(readNarrator != 0)
        DeleteExtIO(readNarrator,sizeof(struct mouth_rb));
  cleanup3:
    if(writeNarrator != 0)
        DeleteExtIO(writeNarrator,sizeof(struct narrator_rb));
  cleanup2:
    if(readport != 0)
        DeletePort(readport);
  cleanup1:
    if(writeport != 0)
        DeletePort(writeport);
  cleanup0:
    if(TranslatorBase != 0)
        CloseLibrary(TranslatorBase);
                /* terminate access to the library */

    if(error != 0) exit(error);
} /* end of test */

DrawMouth(w,h)
SHORT w,h;
{    return(0);  /* dummy routine */ }

int strlen(string)
char *string;
{
     int i,length;
     length = -1;
     for(i=0; i<256; i++)  /* 256 characters max length at this time */
          {
          if(*string++ == ' ') { length = i+1; break; };
          }
     return(length);
}
```

The loop continues to send read requests to the narrator device until the write request has completed. Then the program cleans up and exits.

You can experiment with the narrator device by using other than the default values, changing them before the write command is sent to the device.

8.5. HOW TO WRITE PHONETICALLY FOR NARRATOR

This section describes in detail the procedure used to specify phonetic strings to the *Narrator* speech synthesizer. No previous experience with phonetics is required. The only thing you may need is a good pronouncing dictionary for those times when you doubt your own ears. You do not have to learn a foreign language or computer language. You are just going to learn how to write down the English that comes out of your own mouth. In writing phonetically you do not have to know how a word is spelled, just how it is said.

Narrator works on utterances at the sentence level. Even if you want to say only one word, Narrator will treat it as a complete sentence. Therefore, Narrator wants one of two punctuation marks to appear at the end of every sentence—a period (.) or a question mark (?). If no punctuation appears at the end of a string, Narrator will append a period to it. The period is used for almost all utterances and will cause a final fall in pitch to occur at the end of a sentence. The question mark is used at the end of yes/no questions only, and results in a final rise in pitch. For example, the question, "Do you enjoy using your Amiga?", would take a question mark at the end because the answer to the question is either yes or no. The question, "What is your favorite color?" would not take a question mark and should be followed by a period. Narrator recognizes other punctuation marks as well, but these are be left for later discussion.

Phonetic Spelling

Utterances are usually written phonetically using an alphabet of symbols known as I.P.A. (for "International Phonetic Alphabet"). This alphabet is found at the front of most good dictionaries. The symbols can be hard to learn and are not available on computer keyboards, so the Advanced Research Projects Agency (ARPA) came up with *Arpabet*, a way of representing each symbol using one or two upper-case letters. Narrator uses an expanded version of Arpabet to specify phonetic sounds.

A phonetic sound, or *phoneme*, is a basic speech sound, almost a speech atom. Working backwards, sentences can be broken into words, words into syllables, and syllables into phonemes. The word "cat" has three letters and (coincidentally) three phonemes. Looking at the table of phonemes we find the three sounds that make up the word "cat". They are K, AE, and T, written as KAET. The word "cent" translates as S, EH, N and T, or SEHNT. Notice that both words begin with a "c" but because the "c" says "k" in "cat" we use the phoneme K. In "cent" the "c" says "s" so we use the phoneme S. You may also have noticed that there is no C phoneme.

The above example illustrates that a word rarely sounds like it looks in English spelling. These examples introduce you to a very important concept: spell it like it sounds, not like it looks.

Choosing the Right Vowel

Phonemes, like letters, are divided into the two categories of vowels and consonants. Loosely defined, a vowel is a continuous sound made with the vocal cords vibrating and air exiting the mouth (as opposed to the nose). All vowels use a two-letter code. A consonant is any other sound, such as those made by rushing air (like S or TH), or by interruptions in air flow by the lips or tongue (like B or T). Consonants use a one- or two-letter code.

In English we write with only five vowels: a, e, i, o and u. It would be easy if we only *said* five vowels. Unfortunately, we say more than 15 vowels. Narrator provides for most of them. You choose the proper vowel by listening. Say the word out loud, perhaps extending the vowel sound you want to hear. Compare the sound you are making to the sounds made by the vowels in the example words to the right of the phoneme list. For example, the "a" in "apple" sounds the same as the "a" in "cat," not like the "a"s in "Amiga", "talk," or "made." Notice also that some of the example words in the list do not even use any of the same letters contained in the phoneme code, like AA as in "hot".

Vowels are divided into two groups: those that maintain the same sound throughout their durations and those that change their sound. The ones that change are called "diphthongs". Some of us were taught the terms "long" and "short" to describe vowel sounds. Diphthongs fall into the long category, but these two terms are inadequate to fully differentiate between vowels and should be avoided. The diphthongs are the last six vowels listed in the table. Say the word "made" out loud very slowly. Notice how the "a" starts out like the "e" in "bet" but ends up like the "e" in "beet." The "a" therefore is a diphthong in this word and we would use EY to represent it. Some speech synthesis systems require you to specify the changing sounds in diphthongs as separate elements, but Narrator takes care of the assembly of diphthongal sounds for you.

Choosing the Right Consonant

Consonants are divided into many catagories by phoneticians, but we need not concern ourselves with most of them. Picking the correct consonant is very easy if you pay attention to just two catagories: voiced and unvoiced. A voiced consonant is made with the vocal cords vibrating, and an unvoiced one is made when the vocal cords are silent. Sometimes English uses the same letter combinations to represent both. Compare the "th" in "thin" and in "then." Notice that the first is made with air rushing between the tongue and upper teeth.

In the second, the vocal cords are vibrating also. The voiced "th" phoneme is DH, the unvoiced is TH. Therefore "thin" is spelled TH, IH, N or THIHN and "then" is spelled DH, EH, N or DHEHN. A sound that is particularly subject to mistakes is voiced and unvoiced "s" spelled Z or S. To put it clearly, "bats" ends in S, "suds" ends in Z. What kind of "s" does "closet" have? How about "close"? Say all of these words out loud to find out. Actually "close" changes its meaning when the "s" is voiced or unvoiced: "I love to be close to you." versus "What time do you close?"

Another sound that causes some confusion is the "r" sound. There are two different r-like phonemes in the Narrator alphabet, R under the consonants and ER under the vowels. Which one do you use? Use ER if the "r" sound is the vowel sound in the syllable. Words that take ER are "absurd", "computer" and "flirt". Use R if the "r" sound precedes or follows another vowel sound in that syllable, such as in "car," "write," or "craft". "Rooster" uses both kinds of "r". Can you tell which is which?

Contractions and Special Symbols

There are several phoneme combinations that appear very often in English words. Some of these are caused by our laziness in pronunciation. Take the word "connector" for example. The "o" in the first syllable is almost swallowed out of existence. You would not use the AA phoneme; you would use the AX instead. It is because of this "relaxation" of vowels that we find ourselves using AX and IX very often. Since this relaxation frequently occurs before l, m and n, Narrator has a shortcut for typing these combinations. Instead of "personal" being spelled PERSIXNAXL, we can spell it PERSINUL, making it a little more readable. "Anomaly" goes from AXNAAMAXLIY to UNAAMULIY, and KAAMBIXNEYSHIXN becomes KAAMBINEYSHIN for "combination". It may be hard to decide whether to use the AX or IX brand of relaxed vowel. The only way to find out is to try both and see which sounds best.

Other special symbols are used internally by Narrator. Sometimes they are inserted into or substituted for part of your input sentence. You can type them in directly if you wish. The most useful is probably the Q or glottal stop; an interruption of air flow in the glottis. The word "Atlantic" has one between the "t" and the "l". Narrator knows there should be a glottal stop there and saves you the trouble of typing it. But Narrator is only close to perfect, so sometimes a word or word pair might slip by that would have sounded better with a Q stuck in someplace.

Stress and Intonation

It isn't enough to tell Narrator what you want said. For the best results you must also tell Narrator how you want it said. In this way you can alter a sentence's meaning, stress important words, and specify the proper accents in polysyllabic words. These things improve the naturalness and thus the intelligibility of Narrator's spoken output.

Stress and intonation are specified by the single digits 1-9 following a vowel phoneme code. Stress and intonation are two different things but are specified by a single number. Stress is, among other things, the elongation of a syllable. Because a syllable is either stressed or not, the presence of a number after the vowel in a syllable indicates stress on that syllable. The value of the number indicates the intonation. From this point onward, these numbers will be referred to as "stress marks". Intonation here means the pitch pattern or contour of an utterance. The higher the stress mark, the higher the potential for an accent in pitch (a rise and fall). A sentence's basic contour is comprised of a quickly rising pitch gesture up to the first stressed syllable in the sentence, followed by a slowly declining tone throughout the sentence, and finally a quick fall to a low pitch on the last syllable. The presence of additional stressed syllables causes the pitch to break its slow, declining pattern with rises and falls around each stressed syllable. Narrator uses a very sophisticated procedure to generate natural pitch contours based on how you mark the stressed syllables.

How and Where To Put the Stress Marks

The stress marks go immediately to the right of vowel phoneme codes. The word "cat" has its stress marked after the AE so we get KAE5T or KAE9T. You generally have no choice about the location of a number; there is definitely a right and wrong. Either a number should go after a vowel or it shouldn't. Narrator won't flag an error if you forget to put a stress mark in or if you place one on the wrong vowel. It will only tell you if a stress mark is in the wrong place, such as after a consonant.

The rules for placing stress marks are:

1) Always place a stress mark in a *content* word. A content word is one that contains some meaning. Nouns, verbs, and adjectives are all content words. "Boat", "huge", "tonsils" and "hypertensive" are all content words; they tell the listener what you're talking about. Words like "but", "the", "if" and "is" are not content words. They don't convey any real world meaning at all but are required to make the sentence function. Thus, they are given the name *function* words.

2) Always place a stress mark on the accented syllable(s) of polysyllabic words, whether they are content or function words. A polysyllabic word is any word of more than one syllable. "Commodore" has its stress (or accent as it is often called) on the first syllable and would be spelled KAA5MAXDOHR. "Computer" is stressed on the second syllable, giving KUMPYUW5TER.

   If you are in doubt about which syllable gets the stress, look the word up in a dictionary and you will find an accent mark over the stressed syllable. If more than one syllable in a word receives stress, they usually are not of equal value. These are referred to as primary and secondary stresses. The word "understand" has its first and last syllables stressed, with "stand" getting primary stress and "un" secondary, giving AH1NDERSTAE4ND. Syllables with secondary stress should be marked with a value of only 1 or 2.

   Compound words (words with more than one root) such as "base/ball", "soft/ware", "lunch/wagon" and "house/boat" can be written as one word but should be thought of as separate words when marking stress. Thus, "lunchwagon" would be spelled LAH5NCHWAE2GIN. Notice that "lunch" got a higher stress mark than "wagon". This is common in compound words; the first word usually receives the primary stress.

What Stress Value Do I Use?

If you get the spelling and stress mark positions correct, you are 95 percent of the way to a good sounding sentence. The next thing to do is decide on the stress mark values. They can be roughly related to parts of speech, and you can use the following table as a guide to assign values.

Table 8-1: Recommended Stress Values

Part of Speech	Stress Value	
Nouns	5	
Pronouns	3	
Verbs	4	
Adjectives	5	
Adverbs	7	
Quantifiers	7	
Exclamations	9	
Articles	0	(no stress)
Prepositions	0	
Conjunctions	0	
Secondary stress	1	(sometimes 2)

The above values merely suggest a range. If you want attention directed to a certain word, raise its value. If you want to downplay a word, lower it. Sometimes even a function word can be the focus of a sentence. It is quite conceivable that the word "to" in the sentence "Please deliver this to Mr. Smith." could receive a stress mark of 9. This would add focus to the word "to" indicating that the item should be delivered to Mr. Smith no less than in person.

Punctuation

In addition to the period or question mark that is required at the end of a sentence, Narrator recognizes several other punctuation marks. These are the dash, comma, and parentheses. The comma goes where you would normally put a comma in an English sentence. It causes Narrator to pause with a slightly rising pitch, indicating that there is more to come. The use of additional commas, that is, more than would be required for written English is often helpful. They serve to set clauses off from one another. There is a tendency for a listener to lose track of the meaning of a sentence if the words run together. Read your sentence aloud while pretending to be a newscaster. The locations for additional commas should leap out at you.

The dash serves almost the same purpose as the comma, except that the dash does not cause the pitch to rise so severely. A rule of thumb is: Use dashes to divide phrases, commas to divide clauses. For a definition of these terms, consult a high school English book.

Parentheses provide additional information to Narrator's intonation routine. They should be put around noun phrases of two or more content words. This means that the noun phrase, "a giant yacht" should be surrounded with parentheses because it contains two content words, "giant" and "yacht". The phrase "my friend" should not have parentheses around it

because it contains only one content word. Noun phrases can get pretty big like, "the silliest guy I ever saw" or "a big basket of fruit and nuts". The parentheses really are most effective around these large phrases; the smaller ones can sometimes go without. The effect of parentheses is a subtle one and in some sentences you might not even notice their presence. In sentences of great length, however, they help provide for a very natural contour.

Hints for Intelligibility

There are a few tricks you can use to improve the intelligibility of a sentence. Often, a polysyllabic word is more recognizable than a monosyllabic word. For instance, instead of saying "huge", say "enormous". The longer version contains information in every syllable, thus giving the listener three times the chance to hear it correctly. This can be taken to extremes, so try not to do things like "This program has a plethora of insects in it."

Another good practice is to keep sentences to an optimal length. Writing for reading and writing for speaking are two different things. Try not to write a sentence that cannot be easily spoken in one breath. Such a sentence tends to give the impression that the speaker has an infinite lung capacity. Try to keep sentences confined to one main idea. A run-on sentence tends to lose its meaning after a while.

New terms should be highly stressed the first time they are heard. If you are doing a tutorial or something similar, stress a new term at its first occurrence. All subsequent occurrences of that term need not be stressed as highly because it is now "old news."

The above techniques are but a few ways to enhance the performance of Narrator. You will probably find some of your own. Have fun.

Example of English and Phonetic Texts

Cardiomyopathy. I had never heard of it before, but there it was listed as the form of heart disease that felled not one or two but all three of the artificial heart recipients. A little research produced some interesting results. According to an article in the Nov. 8, 1984, New England Journal of Medicine, cigarette smoking causes this lethal disease that weakens the heart's pumping power. While the exact mechanism is not clear, Dr. Arthur J. Hartz speculated that nicotine or carbon monoxide in the smoke somehow poisons the heart and leads to heart failure.

KAA1RDIYOWMAYAA5PAXTHIY. AY /HAED NEH1VER HER4D AXV IHT BIXFOH5R, BAHT DHEH5R IHT WAHZ - LIH4STIXD AEZ (DHAX FOH5RM AXV /HAA5RT DIHZIY5Z) DHAET FEH4LD (NAAT WAH5N OHR TUW5) - BAHT (AO7L THRIY5 AXV

DHAX AA5RTAXFIHSHUL /HAA5RT RIXSIH5PIYINTS). (AH LIH5TUL RIXSER5CH) PROHDUW5ST (SAHM IH5NTRIHSTIHNX RIXZAH5LTS). AHKOH5RDIHNX TUW (AEN AA5RTIHKUL IHN DHAX NOWVEH5MBER EY2TH NAY5NTIYNEYTIYFOH1R NUW IY5NXGLIND JER5NUL AXV MEH5DIXSIN), (SIH5GEREHT SMOW5KIHNX) KAO4ZIHZ (DHIHS LIY5THUL DIHZIY5Z) DHAET WIY4KINZ (DHAX /HAA5RTS PAH4MPIHNX PAW2ER). WAYL (DHIY IHGZAE5KT MEH5KINIXZUM) IHZ NAAT KLIY5R, DAA5KTER AA5RTHER JEY2 /HAA5RTS SPEH5KYULEYTIHD DHAET NIH5KAXTIYN OHR KAA5RBIN MUNAA5KSAYD IHN DHAX SMOW5K - SAH5M/HAW1 POY4ZINZ DHAX /HAA5RT - AEND LIY4DZ TUW (/HAA5RT FEY5LYER).

Concluding Remarks

This guide should get you off to a good start in phonetic writing for Narrator. The only way to get really proficient is to practice. Many people become good at it in as little as one day. Others make continual mistakes because they find it hard to let go of the rules of English spelling, so trust your ears.

8.6. THE MORE TECHNICAL EXPLANATION

The *SoftVoice* speech synthesis system is a computer model of the human speech production process. It attempts to produce accurately spoken utterances of any English sentence, given only a phonetic representation as input. Another program in the system, *Translator*, derives the required phonetic spelling from English text. Timing and pitch contour are produced automatically by the synthesizer software.

In humans, the physical act of producing speech sounds begins in the lungs. To create a voiced sound, the lungs force air through the vocal folds (sometimes called the vocal cords), which are held under tension and periodically interrupt the flow of air, thus creating a buzz-like sound. This buzz, which has a spectrum rich in harmonics then passes through the vocal tract and out the lips which alters its spectrum drastically. This is because the vocal tract acts as a frequency filter, selectively reinforcing some harmonics and surpressing others.

It is this filtering that gives a speech sound its identity. The amplitude versus frequency graph of the filtering action is called the "vocal tract transfer function". Changing the shape of the throat, tongue and mouth retunes the filter system to accent different frequencies.

The sound travels as a pressure wave through the air, and if we're not talking to ourselves, it causes the listener's eardrum to vibrate. The ear and brain of the listener decodes the incoming frequency pattern. From this the listener can subconsciously make a judgement about what physical actions were performed by the speaker to make the sound. Thus the speech chain is completed, the speaker having encoded his physical actions on a buzz via

selective filtering and the listener having turned the sound into guesses about physical actions by frequency decoding.

Now that we know how we do it, how does a machine do it? It turns out that the vocal tract is not random, but tends to accentuate energy in narrow regions called *formants*. The formant positions move smoothly as we speak, and it is the formant frequencies to which our ears are sensitive. So, luckily, we do not have to model throat, tongue, teeth and lips with our computer, we can imitate formant action.

A good representation of speech requires up to five formants, but only the lowest three are required for intelligibility. We begin with an oscillator that produces a waveform similar to that which is produced by the vocal folds, and pass it through a series of resonators each tuned to a different formant frequency By controlling the volume and pitch of the oscillator and the frequencies of the resonators, we can produce highly intelligible and natural sounding speech. Of course the better the model, the better the speech; but more importantly, experience has shown that the better the control of the model's parameters, the better the speech.

Oscillators, volume controls and resonators can all be simulated mathematically in software, and it is by this method that the SoftVoice system operates. The input phonetic string is converted into a series of target values for the various parameters illustrated. A system of rules then operates on the string to determine things like the duration of each phoneme and the pitch contour. Transitions between target values are created and smoothed to produce natural continuous changes from one sound to the next.

New values are computed for each parameter for every 8 ms. of speech. That's about 120 acoustic changes per second. These values drive a mathematical model of the speech synthesizer. The accuracy of this simulation is quite good. Human speech has more formants than the SoftVoice model, but they are low in energy content.

The human speech production mechanism is an extremely complex and wonderful thing. The more we learn about it, the better we can make our computer simulations. Meanwhile, we can use synthetic speech as yet another computer output device to enhance the man/machine dialogue.

8.7. TABLE OF PHONEMES

Vowels

Phoneme	Example	Phoneme	Example
IY	beet	IH	bit
EH	bet	AE	bat
AA	hot	AH	under
AO	talk	UH	look
ER	bird	OH	border
AX*	about	IX*	solid

*AX and IX should never be used in stressed syllables.

Diphthongs

Phoneme	Example	Phoneme	Example
EY	made	AY	hide
OY	boil	AW	power
OW	low	UW	crew

Consonants

Phoneme	Example	Phoneme	Example
R	red	L	yellow
W	away	Y	yellow
M	men	N	men
NX	sing	SH	rush
S	sail	TH	thin
F	fed	ZH	pleasure
Z	has	DH	then
V	very	J	judge
CH	check	/C	loch
/H	hole	P	put
B	but	T	toy
D	dog	G	guest
K	Commodore		

Special Symbols

Phoneme	Example
DX	pity
(tongue flap)	
Q	kitt_en
(glottal stop)	
QX	pause
(silent vowel)	
RX	car
LX	call
(postvocalic R and L)	

Contractions

(see text)

UL	=	AXL
IL	=	IXL
UM	=	AXM
IM	=	IXM
UN	=	AXN
IN	=	IXN

Digits and Punctuation

Digits 1-9	Syllabic stress, ranging from secondary through emphatic
.	Period—sentence final character
?	Question mark—sentence final character
-	Dash—phrase delimiter
,	Comma—clause delimiter
( )	Parentheses—noun phrase delimiters (see text)

Chapter 9

Serial Device

This chapter describes software access to the serial port. The serial device is accessed via the standard system device access routines and provides some additional functions specifically appropriate to use of this device.

9.1. INTRODUCTION

The serial device can be opened in either exclusive access mode or shared mode. The serial device can be set to transmit and receive many different baud rates (send and receive baud rates are identical). It can support a seven-wire handshaking as well as a three-wire interconnect to a serial hardware device.

Handshaking and access mode must be specified before the serial device is opened. Other serial parameters can be specified using the **SDCMD_SETPARAMS** command after the device has been opened.

9.2. OPENING THE SERIAL DEVICE

Typically, you open the serial device by using the following function calls:

```
LONG error;
struct Port *mySerPort;
struct IOExtSer *mySerReq;

/* create a reply port to which serial
 * device can return the request */
mySerPort = CreatePort("mySerial",0);
if(mySerPort == NULL) exit(100); /* cant create port? */

/* create a request block appropriate to serial */
mySerReq = (struct IOExtSer *)CreateExtIO(mySerPort,
                sizeof(struct IOExtSer));
if(mySerReq == NULL) goto cleanup1; /* error during CreateExtIO? */

mySerReq->io_SerFlags = 0;
/* accept the default, i.e. Exclusive Access and
 * XON/XOFF protocol is enabled.
 * Remaining flags all zero, see devices/serial.h
 * for bit-positions.  Definitions included in this
 * chapter. */

error = OpenDevice("serial.device",0,mySerReq,0);
if(error != 0) goto cleanup2; /* device not available? */

...
cleanup2:
  DeleteExtIO(mySerReq,sizeof(struct IOExtSer));
cleanup1:
  DeletePort(mySerPort);
```

The routines **CreatePort()** and **DeletePort()** are part of amiga.lib. Information about the routines **CreateExtIO()** and **DeleteExtIO()** can be found in the appendixes of this manual.

During open, the only flags that the serial device pays any attention to are the shared/exclusive-access flag and the seven-wire flag (the seven-wire flag enables RS-232-C DTR/DSR,RTS/CTS handshaking protocol). All other bits in **io_SerFlags** are ignored. However, for consistency, the other flag bits should be set to zero when the device is opened.

When the serial device is opened, it opens the timer device and then allocates an input buffer of the size last used (default and minimum = 512 bytes). As with any of the other serial port parameters, you can later change the value used for the read buffer size with the **SDCMD_SETPARMS** command. The **OpenDevice()** routine will fill the latest parameter settings into the **io_Request** block.

Once the serial device is opened, all characters received will be saved, even if there is no current request for them. Note that a parameter change cannot be performed while actually processing an I/O request, because it would invalidate request handling already in progress.

Therefore you must use **SDCMD_SETPARAMS** only when you have no serial I/O requests pending.

9.3. READING FROM THE SERIAL DEVICE

You read from the serial device by sending your **IORequest** (**IOExtSer**) to the device, with a read command. You specify how many bytes are to be transferred, and where the data is to be placed. Depending on how you have set your parameters, the request may read the requested number of characters, or it may terminate early.

Here is a sample read command:

```
char myDataArea[100];
mySerReq->IOSer.io_Data = &myDataArea[0]; /* where to put the data */
mySerReq->IOSer.io_Length = 100;    /* read 100 characters */
mySerReq->IOSer.io_Command = CMD_READ; /* say it is a read */
DoIO(mySerReq);                          /* synchronous request */
```

If you use this example, your task will be put to sleep waiting until the serial device reads 100 bytes (or terminates early) and copies them into your read-buffer. Early termination can be caused by error conditions or by the serial device sensing an end of file condition.

Note that the **io_Length** value, if set to -1, tells the serial device that you want to read a null terminated string. The device will read all incoming characters up to and including a byte value of 0x00 in the input stream, then report to you an **io_Actual** value that is the actual length of the string, excluding the 0 value. Be aware that you must encounter a 0 value in the input stream before the system fills up the buffer you have specified. The **io_Length** is, for all practical purposes, indefinite. Therefore, you could potentially overwrite system memory if you never encountered the null termination (zero value byte) in the input stream.

First Alternative Mode for Reading

As an alternative to **DoIO()** you can use **SendIO()** to transmit the command to the device. In this case, your task can go on to do other things while the serial device is collecting the bytes for you. You can occasionally do a **CheckIO(mySerReq)** to see if the I/O is completed.

```
struct Message *myIO;

/* same code as in above example, except:  */
SendIO(mySerReq);

      /* do something */
      /* (user code)  */

myIO = CheckIO(mySerReq);
if(myIO != FALSE) goto ioDone; /* this IO is done */

      /* do something else */
      /* (user code) */
WaitIO(mySerReq);
myIO = mySerReq;    /* if had to wait,
                     * need a value for myIO */
}
ioDone:
      Remove(mySerPort->mp_MsgList,myIO);
/* use the Remove function rather than the GetMsg function */

/* now check for errors, and so on. */
```

The **Remove()** function is used instead of the **GetMsg()** function to demonstrate that you might have established only one port at which all of your I/O requests will be returned, and you may be checking each request, in turn, with **CheckIO()** to see if it has completed (maybe a disk request, a serial request and a parallel request, all simultaneously outstanding, all using **SendIO()** to transmit their commands to the respective devices).

It is possible that while you are doing other things and checking for completion of I/O, one device may complete its operations and append its message block to your reply port while you are about to check the status of a later-arriving block. If you find that this later one has completed and you call **GetMsg()**, you will remove whichever message is at the head of the list. This message may not necessarily be the one you expect to be removing from the port. **CheckIO()** returns the address of the **IORequest** if the I/O is complete, and you can use this address for the **Remove()** function to remove the correct request block for processing and reuse.

Second Alternative Mode for Reading

Instead of transmitting the read command with either **DoIO()** or **SendIO()**, you might elect to use **BeginIO()**, (the lowest level interface to a device) with the "quick I/O" bit set in the **io_Flags** field.

```
/* same code as in read example, except: */
mySerReq->IOSer.io_Flags = IOF_QUICK; /* use QUICKIO */

BeginIO(mySerReq);
```

The serial device may support quick I/O for certain read requests. As documented in the "I/O" chapter in this manual, this command may be synchronous or asynchronous. Any write request always clears the quick I/O bit. Various read commands may or may not clear it, depending on whether or not quick I/O occurs.

After executing the code shown above, your program needs to know if the I/O happened synchronously and it must also test to see if the I/O took place.

```
if((mySerReq->IOSer.io_Flags & IOF_QUICK) == 0)
        {
        /* QUICKIO couldn't happen for some reason, so it did
         * it normally... queued the request, cleared the QUICKIO
         * bit, and used the equivalent of SendIO. Might want to
         * have the task doing something else while awaiting the
         * completion * of the I/O. After knowing it is done, must
         * remove the message from the reply port for possible reuse.
         */
        WaitIO(mySerReq);
              /* assumes single threaded I/O, as compared to
               * the SendIO example in the previous section */
        }
else
        {
        /* If flag is still set, IO was synchronous,
         * means that the IORequest was NOT appended
         * to the reply port and there is no need to
         * remove the message from the reply port;
         * continue on with something else.
         */
        ;
        }
```

The way you read from the device depends on your need for processing speed. Generally the **BeginIO()** route provides the lowest system overhead when quick I/O is possible. However, if quick I/O didn't work, it still requires some overhead for handling of the **IORequest** block.

Termination of the Read

Reading from the serial device can terminate early if an error occurs or if an end-of-file is sensed. You can specify a set of possible end-of-file characters that the serial device is to look for in the input stream. These are contained in an **io_TermArray** that you provide, using the **SDCMD_SETPARAMS** command. *NOTE*: **io_TermArray** is used only when EOF mode is selected.

If EOF mode is selected, each input data character read into the user's data block, is compared against those in **io_TermArray**. If a match is found, the **IORequest** is terminated as complete, and the count of characters read (including the **TermChar**) is stored in **io_Actual**. To keep this search overhead as efficient as possible, the serial device requires that the array of characters be in descending order (an example is shown in the summary page in the appendixes for **SDCMD_SETPARAMS**). The array has eight bytes and all must be valid (that is, don't pad with zeros unless zero is a valid EOF character).

Fill to the end of the array with the least value **TermChar**. When making an arbitrary choice of EOF character(s), it's advisable to use the lowest value(s) available.

9.4. WRITING TO THE SERIAL DEVICE

You can write to the serial device as well as read from it. It may be wise to have a separate block for reading and writing to allow simultaneous operation of both reading and writing. The sample code below creates a separate reply port and request for writing to the serial device. Note that it assumes that the **OpenDevice()** function worked properly for the read. It copies the initialized read request block to initialize the write request block. Error checking has been deliberately left out of this code fragment for brevity but should, of course, be provided in a functional program.

```
LONG i;
BYTE *b,*c;

struct Port *mySerWritePort;
struct IOExtSer *mySerWriteReq;

mySerWritePort = CreatePort("mySerialWrite",0);

mySerWriteReq = (struct IOExtSer *)CreateExtIO(mySerWritePort,
                sizeof(struct IOExtSer));

b = (BYTE *)mySerReq;    /* start of read request block */
c = (BYTE *)mySerWriteReq; /* start of write request block */

for(i=0; i< sizeof(struct IOExtSer); i++)
     *c++ = *b++;
/* clones the request block on a byte by byte basis */
/* NOTE:  it might simply be easier, here, to have opened the
 *      serial device twice.  This would reflect the fact that
 *      there are two "software entities" that are currently
 *      using the device.  However, if you are using exclusive
 *      access mode, this is not possible and the request block
 *      must be copied anyway.
 */
```

Note that this code would require the following cleanup at the termination of the program:

```
cleanupWriteIO:
     DeleteExtIO(mySerWriteReq);
cleanupWritePort:
     DeletePort(mySerWritePort);
```

Now, to perform a write:

```
char dataToWrite[100];
mySerReq->IOSer.io_Data = &dataToWrite[0]; /* where to get the data */
mySerReq->IOSer.io_Length = n;             /* write n characters */
mySerReq->IOSer.io_Command = CMD_WRITE; /* say it is a write */
DoIO(mySerReq);                    /* synchronous request */
```

You can use the **SendIO()** or **BeginIO()** functions as well as **DoIO()**. The same warnings apply as shown above in the discussions about alternative modes of reading.

Note that if **io_Length** is set to -1, the serial device will output your serial buffer until it encounters a value of 0x00 in the data. It transmits this 0 value in addition to the data to match the technique used for serial read shown above. (You can also read data zero-terminated).

9.5. SETTING SERIAL PARAMETERS

You can control the following serial parameters. The parameter name within the serial data structure is shown below. All of the fields described in this section are filled in when you call **OpenDevice()** to reflect the current settings of the serial device. Thus, you needn't worry about any parameter that you don't need to change.

Table 9-1: Serial Parameters

Parameter Name	Characteristic It Controls
io_CtlChar	Control characters to use for XON, XOFF, INQ, ACK respectively. Positioned within an unsigned long word in the sequence from low address to high as listed. INQ and ACK handshaking is not currently supported.
io_RBufLen	Size of the buffer that the serial device should allocate for incoming data. Minimum size is 512 bytes. It won't accept a smaller value. This buffer is dynamically allocated by the serial device. If, as you do an **SDCMD_SETPARAMS** command, it senses a difference between its current value and the value of buffer size you request, it deallocates the old buffer and allocates a new one. Note that it discards all characters that may already be in that old buffer that you may not have yet had a chance to read. Thus it is wise to assure that you don't attempt buffer size changes (or any change to the serial device, for that matter) while any I/O is actually taking place.
io_ExtFlags	Reserved for future use.
io_Baud	The real baud rate you wish to use. A long value from 110 to 292000. When a value of 110 is requested, it defaults to 112 (the lowest value the hardware can support). Although baud rates above 19200 are supported by the hardware, software overhead may limit your ability to "catch" every single character that should be received. Output data rate, however, is not software dependent.

io_BrkTime If you issue a break command, this variable specifies how long, in microseconds, the break condition lasts. This value controls the break time for all future break commands until modified by another **SDCMD_SETPARAMS**.

io_TermArray A byte-array of eight termination characters, must be in descending order. If EOFMODE is set in the serial flags, this array specifies 8 possible choices of character to use as an end of file mark. See the section above titled "Termination of the Read" and the **SDCMD_SETPARAMS** summary page in the function appendix for more information.

io_ReadLen How many bits per read character. Typically a value of 7 or 8.

io_WriteLen How many bits per write character. Typically a value of 7 or 8.

io_StopBits How many stop bits are to be expected when reading a character and to be produced when writing a character. Typically 1. A value of 2 is allowed if **io_WriteLen** = 7.

io_SerFlags Explained below; see "Serial Flags".

Bit	Active	Function
0	low	(reserved)
1	low	(reserved)
2	low	(reserved)
3	low	Data set ready
4	low	Clear to send
5	low	Carrier detect
6	low	Ready to send
7	low	Data terminal ready
8	high	Read overrun
9	high	Break sent
10	high	Break received
11	high	Transmit x-OFFed
12	high	Receive x-OFFed
13-15	(not)	(reserved)

Serial Flags

The following flags can be set to affect the operation of the serial device. Note that the default state of all of these flags is zero.

Table 9-2: Serial Flags

Flag Name	Effect on Device Operation
SERB_XDISABLED	Disable XON-XOFF feature.
SERB_EOFMODE	Set this bit if you want the serial device to check I/O characters against **io_TermArray** and terminate the **IORequest** immediately if an end-of-file character has been encountered. *NOTE*: This bit can be set and reset directly in the user's **IORequest** (**IOExtSer**) block without a call to **SDCMD_SETPARAMS**.
SERB_SHARED	Set this bit if you want to allow other tasks to simultaneously access the serial port. The default is exclusive-access. If someone already has the port, whether for exclusive or shared, and you ask for exclusive-access, your **OpenDevice()** call will fail (should be modified only at **OpenDevice()**).
SERB_RAD_BOOGIE	If set, this bit activates high-speed mode. Certain peripheral devices, (MIDI, for example) may require high serial throughput. Setting this bit high causes the serial device to skip certain of its internal checking code so as to speed throughput. In particular, it: - disables parity checking - bypasses XON/XOFF handling - uses only 8-bit character length - won't test for a break signal - automatically sets SERB_XDISABLED bit Note that the Amiga is a multi-tasking system and has immediate processing of software interrupts. If there are other tasks running, it is possible that the serial driver may be unable to keep up with high data transfer rates, even with this bit set.

SERB_QUEUEDBRK	If set, every break command that you transmit will be enqueued. This means that the current serial output commands will be executed in sequence. Then the break command will be executed, all on a FIFO (first in, first out) basis. If this bit is cleared (the default), a break command takes immediate precedence over any serial output already enqueued. When the break command has finished, the interrupted request will continue (if it's not aborted by the user).
SERB_7WIRE	If set (should be established only at **OpenDevice**()), the serial device is to use a seven-wire handshaking for RS-232-C communications. Default is three-wire (pins 2, 3, and 7).
SERB_PARTY_ODD	If set, selects odd parity. If clear, selects even parity.
SERB_PARTY_ON	If set, parity usage and checking is enabled.

Setting the Parameters

You set the serial parameters by setting the flags and parameters as you desire, then transmitting the command **SDCMD_SETPARAMS** to the device. Here is an example:

```
mySerReq->IOSer.io_SerFlags &= ~ SERF_PARTY_ODD;  /* 'and' with inverse */
mySerReq->IOSer.io_SerFlags |= SERF_QUEUEDBRK | SERF_PARTY_ON;
mySerReq->io_BrkTime = 500000;     /* 500k microseconds = 1/2 second */
mySerReq->IOSer.io_Command = SDCMD_SETPARAMS;
DoIO(mySerReq);                    /* synchronous request */
```

The above command would set the bits for queued break and even parity while leaving the other flags unchanged. Notice the difference between the flag names and the flags that you actually set using C. "SERB..." is the name applied to the bit-position within the flag word. "SERF..." is the name of a 1 bit in a mask at that bit position.

9.6. ERRORS FROM THE SERIAL DEVICE

The possible error returns from the serial device are listed below. The abbreviated naming shown for the error numbers is self-explanatory.

Table 9-3: Serial Device Errors

```
#define SerErr_DevBusy          1
#define SerErr_BaudMismatch     2
#define SerErr_InvBaud          3
#define SerErr_BufErr           4
#define SerErr_InvParam         5
#define SerErr_LineErr          6
#define SerErr_NotOpen          7
#define SerErr_PortReset        8
#define SerErr_ParityErr        9
#define SerErr_InitErr          10
#define SerErr_TimerErr         11
#define SerErr_BufOverflow      12
#define SerErr_NoDSR            13
#define SerErr_NoCTS            14
#define SerErr_DetectedBreak    15
```

9.7. CLOSING THE SERIAL DEVICE

When the (final, if shared access) **CloseDevice()** is performed, the input buffer is deallocated, the timer device is closed, and the latest parameter settings are saved for the next open.

Typically, you close the serial device with the following function call:

```
CloseDevice(mySerReq);
```

This assumes that the serial device has completed all activities you have requested and has returned all I/O requests to you.

When you have finished with the serial device, it is up to you to deallocate any memory and dependencies you might have used for the serial device communications. If you have used the techniques shown earlier in this chapter to establish the communications in the first place, your cleanup typically will consist of the following code:

```
cleanup2:
  DeleteExtIO(mySerReq,sizeof(struct IOExtSer));
cleanup1:
  DeletePort(mySerPort);
cleanupWriteIO:
  DeleteExtIO(mySerWriteReq);
cleanupWritePort:
  DeletePort(mySerWritePort);
```

9.8. EXAMPLE PROGRAM

Here is an example program that uses static rather than dynamic allocation of the **IOExtSer** request block. It assumes that you have connected a serial terminal device to the Amiga serial port, and it uses the baud rate you have established in Preferences. The program outputs the following status lines to the CLI window:

```
Serial Device opened and accepted parameters
Testing character exact-count output thru SendWaitWrite
Test string length of -1 (make system find end of string)
Type 16 characters to send to amiga...
If no external terminal is attached, waits forever!
```

and outputs the following lines to the external terminal:

```
Device opened ok
User counts characters in string to send
or if null terminated string, say '-1'
Type 16 characters to send to Amiga.
```

At this point, you must type 16 characters on your external terminal. This sample program does not echo characters that you type, so you will not see anything more until all 16 have been typed. Finally the program will respond (to the external terminal) with:

```
You typed these printable characters:
<here it lists the 16 characters>
End of test
54321.....exit
```

Then the program exits, printing "Test completed!" to the CLI window.

```
#include      "exec/types.h"
#include      "exec/nodes.h"
#include      "exec/lists.h"
#include      "exec/ports.h"
#include      "exec/libraries.h"
#include      "exec/devices.h"
#include      "exec/io.h"
#include      "devices/serial.h"

struct IOExtSer *IORser;
struct MsgPort *port;
char   buffer[200];
extern struct MsgPort *CreatePort();
extern struct IORequest *CreateExtIO();

/* note:  to run this program, you must have an external terminal, set
 * at 9600 baud, attached to the Amiga serial port.  Additionally the
 * serial.device file must be located in the directory currently
 * assigned to DEVS:  (to check this, in AmigaDOS, type:  ASSIGN
 * then check the directory (usually the boot CLI disk volume, devs directory.)
 */

main()
{
      int     error;
      int     actual;
      unsigned long rbl;
      unsigned long brk;
      unsigned long baud;
      unsigned char rwl;
      unsigned char wwl;
      unsigned char sf;
      unsigned long t0;
      unsigned long t1;

      /* SET UP the message port in the I/O request */
      port = CreatePort (SERIALNAME,0);
      if (port == NULL) {
            printf("\nProblems during CreatePort");
            exit(100);
      }

      /* Create the request block for passing info
       * to and from the serial device. */

      IORser = (struct IOExtSer *)CreateExtIO(port,sizeof(struct IOExtSer));
      if (IORser == NULL)
      {
```

```
        printf("\nProblems during CreateExtIO");
        goto cleanup1;
    }

open:
    /* OPEN the serial.device */
    if ((error = OpenDevice (SERIALNAME, 0, IORser, 0)) != 0) {
        printf ("Serial device did not open, error = %ld",error);
        goto cleanup1;
    }

    /*   SET PARAMS for the serial.device */
    rbl = 4096;
    rwl = 0x08;
    wwl = 0x08;
    brk = 750000;
    baud= 9600;
    sf  = 0x00;
    t0  = 0x51040303;
    t1  = 0x03030303;

    if ((error = SetParams (IORser,rbl,rwl,wwl,brk,baud,sf,t0,t1)) != 0) {
        printf ("Set parameters command returned an error: %ld",error);
        goto cleanup2;
    }

    printf("\nSerial Device opened and accepted parameters");
    WriteSer (IORser,"\n\015Device opened ok\n\015", -1);

    printf("\nTesting character exact-count output thru SendWaitWrite");
    SendWaitWrite (IORser,
        "User counts characters in string to send\n\015", 42);

    printf("\nTest string length of -1 (make system find end of string)");
    SendWaitWrite (IORser,
        "or if null terminated string, say '-1'\n\015", -1);

    printf("\nType 16 characters to send to amiga...");
    printf("\nIf no external terminal is attached, waits forever!!");
    WriteSer (IORser,
        "\n\015Type 16 characters to send to amiga\n\015", -1);
    actual = ReadSer (IORser,buffer,16);
    WriteSer (IORser,
        "\n\015You typed these printable characters:\n\015", -1);
    WriteSer (IORser,buffer, actual);
    WriteSer (IORser,"\n\015End of test\n\015", -1);
    WriteSer (IORser,"54321.....exit\n\015", 16);
    printf("\nTest completed!\n");

    /* CLOSE the serial.device */
```

```
cleanup2:
        CloseDevice (IORser);
cleanup1:
        DeletePort (port);
        exit (0);
}

/* SERIAL I/O functions */

SetParams(io,rbuf_len,rlen,wlen,brk,baud,sf,ta0,ta1)

struct IOExtSer *io;
unsigned long rbuf_len;
unsigned char rlen;
unsigned char wlen;
unsigned long brk;
unsigned long baud;
unsigned char sf;
unsigned long ta0;
unsigned long ta1;

{
        int error;

        io->io_ReadLen      = rlen;
        io->io_BrkTime      = brk;
        io->io_Baud         = baud;
        io->io_WriteLen     = wlen;
        io->io_StopBits     = 0x01;
        io->io_RBufLen      = rbuf_len;
        io->io_SerFlags     = sf;
        io->IOSer.io_Command = SDCMD_SETPARAMS;
        io->io_TermArray.TermArray0 = ta0;
        io->io_TermArray.TermArray1 = ta1;

        if ((error = DoIO (io)) != 0) {
                printf ("serial.device setparams error %ld \n", error);
        }
        return (error);
}

ReadSer(io,data,length)
struct IOExtSer *io;
char *data;
ULONG length;
{
        int error;

        io->IOSer.io_Data = data;
        io->IOSer.io_Length = length;
```

```
        io->IOSer.io_Command = CMD_READ;

        if ((error = DoIO (io)) != 0) {
                printf ("serial.device read error %ld \n", error);
        }
        return (io->IOSer.io_Actual);
}


WriteSer(io,data,length)
struct IOExtSer *io;
char *data;
int length;
{
        int     error;

        io->IOSer.io_Data = data;
        io->IOSer.io_Length = length;
        io->IOSer.io_Command = CMD_WRITE;

        if ((error = DoIO (io)) != 0) {
                printf ("serial.device write error %ld \n", error);
        }
        return (error);
}

SendWaitWrite(io,data,length)
struct IOExtSer *io;
char *data;
int length;
{
        int     error;

        io->IOSer.io_Data = data;
        io->IOSer.io_Length = length;
        io->IOSer.io_Command = CMD_WRITE;

        SendIO (io);

        if ((error = WaitIO (io)) != 0) {
                printf ("serial.device waitio error %ld \n", error);
        }
        return (io->IOSer.io_Actual);
}
```

Chapter 10

Parallel Device

This chapter describes software access to the parallel port. The parallel device is accessed via the standard system device access routines and provides some additional functions specifically appropriate to use of this device.

10.1. INTRODUCTION

The parallel device can be opened either in exclusive access or shared mode. Other parallel device parameters can be specified using the PDCMD_SETPARAMS command after the device has been opened.

10.2. OPENING THE PARALLEL DEVICE

Typically, you open the parallel device by using the following function calls:

```
LONG error;
struct Port *myParPort;
struct IOExtPar *myParReq;

/* create a reply port to which parallel
 * device can return the request */
myParPort = CreatePort("myParallel",0);
if(myParPort == NULL) exit(100);       /* cant create port? */

/* create a request block appropriate to parallel */
myParReq = (struct IOExtPar *)CreateExtIO(myParPort,
                sizeof(struct IOExtPar));
if(myParReq == NULL) goto cleanup1; /* error during CreateExtIO? */

myParReq->io_ParFlags = 0;
/* accept the default, i.e. Exclusive Access
 * Remaining flags all zero, see devices/parallel.h
 * for bit-positions.  Definitions included in this
 * chapter. */

error = OpenDevice("parallel.device",0,myParReq,0);
if(error != 0) goto cleanup2; /* device not available? */

...
cleanup2:
  DeleteExtIO(myParReq,sizeof(struct IOExtPar));
cleanup1:
  DeletePort(myParPort);
```

The routines **CreatePort()** and **DeletePort()** are part of amiga.lib. Information about the routines **CreateExtIO()** and **DeleteExtIO()** can be found in the appendixes of this manual.

The parallel device is disk-resident. If it hasn't yet been loaded from disk, it will be read from "DEVS:parallel.device" on the boot AmigaDOS disk. Its parameters will be set up from default values.

During open, the only flag that the parallel device uses is the shared/exclusive-access flag. For consistency, however, the other flag bits should be set to zero when the device is opened.

When the parallel device is opened, it opens the timer device, and fills the latest parameter settings into the **io_Request** block. The **OpenDevice()** routine will fill the latest parameter settings into the **io_Request** block. Note that a parameter change cannot be performed while actually processing an I/O request, because it would invalidate request handling already in progress. Therefore you must use **PDCMD_SETPARAMS** only when you have no parallel I/O requests pending.

10.3. READING FROM THE PARALLEL DEVICE

You read from the parallel device by sending your **IORequest** (**IOExtPar**) to the device with a read command. You specify how many bytes are to be transferred and where the data is to be placed. Depending on how you have set your parameters, the request may read the requested number of characters, or it may terminate early.

Here is a sample read command:

```
char myDataArea[100];
myParReq->IOPar.io_Data = &myDataArea[0]; /* where to put the data */
myParReq->IOPar.io_Length = 100;  /* read 100 characters */
myParReq->IOPar.io_Command = CMD_READ; /* say it is a read */
DoIO(myParReq);                        /* synchronous request */
```

If you use this example, your task will be put to sleep waiting until the parallel device reads 100 bytes (or terminates early) and copies them into your read-buffer. Early termination can be caused by error conditions or by the parallel device sensing an end of file condition.

Note that the **io_Length** value, if set to -1, tells the parallel device that you want to read a null terminated string. The device will read all incoming characters up to and including a byte value of 0x00 in the input stream, then report to you an **io_Actual** value that is the actual length of the string, excluding the 0 value. Be aware that you must encounter a 0 value in the input stream before the system fills up the buffer you have specified. The **io_Length** is, for all practical purposes, indefinite. Therefore, you could potentially overwrite system memory if you never encountered the null termination (zero value byte) in the input stream.

Alternative Mode for Reading

As an alternative to **DoIO()** you can use **SendIO()** to transmit the command to the device. In this case, your task can go on to do other things while the parallel device is collecting the bytes for you. You can occasionally do a **CheckIO(myParReq)** to see if the I/O is completed.

```
struct Message *myIO;

/* same code as in above example, except:  */
SendIO(myParReq);

        /* do something */
        /* (user code)  */

myIO = CheckIO(myParReq);
if(myIO != FALSE) goto ioDone; /* this IO is done */

        /* do something else */
        /* (user code) */
WaitIO(myParReq);
myIO = myParReq;   /* if had to wait,
                    * need a value for myIO */
}
ioDone:
        Remove(myParPort->mp_MsgList,myIO);
/* use the Remove function rather than the GetMsg function */

/* now check for errors, and so on. */
```

The **Remove()** function is used instead of the **GetMsg()** function to demonstrate that you might have established only one port at which all of your I/O requests will be returned, and you may be checking each request, in turn, with **CheckIO()** to see if it has completed These requests could be, for example, a disk request, a parallel request and a serial request, all simultaneously outstanding and all using **SendIO()** to transmit their commands to the respective devices.

It is possible that while you are doing other things and checking for completion of I/O, one device may complete its operations and append its message block to your reply port while you are about to check the status of a later-arriving block. If you find that this later one has completed and you call **GetMsg()**, you will remove the message at the head of the list. This message may not necessarily be the one you expect to be removing from the port. **CheckIO()** returns the address of the **IORequest** if the I/O is complete, and you can use this address for the **Remove()** function to remove the correct request block for processing and reuse.

Termination of the Read

Reading from the parallel device can terminate early if an error occurs or if an end-of-file is sensed. You can specify a set of possible end-of-file characters that the parallel device is to look for in the input stream. These are contained in an **io_TermArray** that you provide, using the **PDCMD_SETPARAMS** command. *NOTE*: **io_TermArray** is used only when EOF mode is selected.

If EOF mode is selected, each input data character read into the user's data block, is compared against those in **io_TermArray**. If a match is found, the **IORequest** is terminated as complete, and the count of characters read (including the **TermChar**) is stored in **io_Actual**. To keep this search overhead as efficient as possible, the parallel device requires that the array of characters be in descending order (an example is shown in the summary page in the appendixes for **PDCMD_SETPARAMS**). The array has eight bytes and all must be valid (that is, don't pad with zeros unless zero is a valid EOF character).

Fill to the end of the array with the least value **TermChar**. When making an arbitrary choice of EOF character(s), it's advisable to use the lowest value(s) available.

10.4. WRITING TO THE PARALLEL DEVICE

You can write to the parallel device as well as read from it. It may be wise to have a separate **IORequest** block for reading and writing to allow simultaneous operation of both reading and writing. The sample code below creates a separate reply port and request for writing to the parallel device. Note that it assumes that the **OpenDevice()** function worked properly for the read. It copies the initialized read request block to initialize the write request block. Error checking has been deliberately left out of this code fragment for brevity but should, of course, be provided in a functional program.

```
LONG i;
BYTE *b,*c;

struct Port *myParWritePort;
struct IOExtPar *myParWriteReq;

myParWritePort = CreatePort("myParallelWrite",0);

myParWriteReq = (struct IOExtPar *)CreateExtIO(myParWritePort,
                sizeof(struct IOExtPar));

b = (BYTE *)myParReq;   /* start of read request block */
c = (BYTE *)myParWriteReq; /* start of write request block */

for(i=0; i< sizeof(struct IOExtPar); i++)
      *c++ = *b++;
/* clones the request block on a byte by byte basis */
/* NOTE:  it might simply be easier, here, to have opened the
 *       parallel device twice.  This would reflect the fact that
 *       there are two "software entities" that are currently
 *       using the device.  However, if you are using exclusive
 *       access mode, this is not possible and the request block
 *       must be copied anyway.
 */
```

Note that this code would require the following cleanup at the termination of the program:

```
cleanupWriteIO:
      DeleteExtIO(myParWriteReq);
cleanupWritePort:
      DeletePort(myParWritePort);
```

Now, to perform a write:

```
char dataToWrite[100];
myParReq->IOPar.io_Data = &dataToWrite[0]; /* where to get the data */
myParReq->IOPar.io_Length = n;          /* write n characters */
myParReq->IOPar.io_Command = CMD_WRITE; /* say it is a write */
DoIO(myParReq);                         /* synchronous request */
```

You can use the **SendIO()** or **BeginIO()** functions as well as **DoIO()**. The same warnings apply as shown above in the discussions about alternative modes of reading.

Note that if **io_Length** is set to -1, the parallel device will output your parallel buffer until it encounters a value of 0x00 in the data. It transmits this 0 value in addition to the data to match the technique used for parallel read shown above. (You can also read data zero-terminated).

10.5. SETTING PARALLEL PARAMETERS

You can control the following parallel parameters. The parameter name within the parallel data structure is shown below. All of the fields described in this section are filled in when you call **OpenDevice()** to reflect the current settings of the parallel device. Thus, you needn't worry about any parameter that you don't need to change.

Table 10-1: Parallel Parameters

Parameter Name	Characteristic It Controls
io_PExtFlags	Reserved for future use.
io_TermArray	A byte-array of eight termination characters, must be in descending order. If EOFMODE is set in the parallel flags, this array specifies 8 possible choices of character to use as an end of file mark. See the section above titled "Termination of the Read" and the **PDCMD_SETPARAMS** summary page in the function appendix for more information.
io_ParFlags	Explained below; see "Parallel Flags".

Parallel Flags

The following flags can be set to affect the operation of the parallel device. Note that the default state of all of these flags is zero.

Table 10-2: Parallel Flags

Flag Name	Effect on Device Operation
PARB_EOFMODE	Set this bit if you want the parallel device to check I/O characters against **io_TermArray** and terminate the **IORequest** immediately if an end-of-file character has been encountered. *NOTE*: This bit can be set and reset directly in the user's **IORequest** (**IOExtPar**) block without a call to **PDCMD_SETPARAMS**.
PARB_SHARED	Set this bit if you want to allow other tasks to simultaneously access the parallel port. The default is exclusive-access. If someone already has the port, whether for exclusive or shared, and you ask for exclusive-access, your **OpenDevice()** call will fail (should be modified only at **OpenDevice()**).

Setting the Parameters

You set the parallel parameters by setting the flags and parameters as you desire, then transmitting the command **PDCMD_SETPARAMS** to the device. Here is an example:

```
myParReq->IOPar.io_ParFlags &= ~ PARF_EOFMODE;    /* 'and' with inverse */
myParReq->IOPar.io_Command = PDCMD_SETPARAMS;
DoIO(myParReq);                    /* synchronous request */
```

The above command would cancel EOFMODE (use of the **io_TermArray**) leaving the other flags unchanged. Notice the difference between the flag names and the flags that you actually set using C. "PARB..." is the name applied to the bit-position within the flag word. "PARF..." is the name of a 1 bit in a mask at that bit position.

10.6. ERRORS FROM THE PARALLEL DEVICE

The possible error returns from the parallel device are listed below. The abbreviated naming shown for the error numbers is self-explanatory.

Table 10-3: Parallel Device Errors

```
#define ParErr_DevBusy     1
#define ParErr_InvParam    3
#define ParErr_LineErr     4
#define ParErr_NotOpen     5
#define ParErr_PortReset   6
#define ParErr_InitErr     7
```

10.7. CLOSING THE PARALLEL DEVICE

When the (final, if shared access) **CloseDevice()** is performed, the timer device is closed, and the latest parameter settings are saved for the next open.

Typically, you close the parallel device with the following function call:

```
CloseDevice(myParReq);
```

This assumes that the parallel device has completed all activities you have requested and has returned all I/O requests to you.

When you have finished with the parallel device, it is up to you to deallocate any memory and dependencies you might have used for the parallel device communications. If you have used the techniques shown earlier in this chapter to establish the communications in the first place, your cleanup typically will consist of the following code:

```
cleanup2:
  DeleteExtIO(myParReq,sizeof(struct IOExtPar));
cleanup1:
  DeletePort(myParPort);
cleanupWriteIO:
  DeleteExtIO(myParWriteReq);
cleanupWritePort:
  DeletePort(myParWritePort);
```

10.8. EXAMPLE PROGRAM

Here is an example program that uses static rather than dynamic allocation of the **IOExtPar** request block. It assumes that you have connected a parallel I/O device to the Amiga parallel port.

```
#include      "exec/types.h"
#include      "exec/nodes.h"
#include      "exec/lists.h"
#include      "exec/ports.h"
#include      "exec/libraries.h"
#include      "exec/devices.h"
#include      "exec/io.h"
#include      "devices/parallel.h"

struct IOExtPar IORpar;
struct MsgPort *port;
char   buffer[64000];
extern struct MsgPort *CreatePort();

main()
{
   int    error;
   int    actual;
   unsigned char pflags;
   unsigned long pt0;
   unsigned long pt1;

open:
 /* OPEN the parallel.device */
   if ((error = OpenDevice (PARALLELNAME, 0, &IORpar, 0)) != 0) {
      printf ("bad news %ld on Open \n", error);
      exit (error);
   }

 /* SET UP the message port in the I/O request */
   port = CreatePort (PARALLELNAME,0);
   IORpar.IOPar.io_Message.mn_ReplyPort = port;

/*    SET PARAMS for the parallel.device */
   pflags = PARF_EOFMODE;
   pt0  = 0x51040303;
   pt1  = 0x03030303;

  if ((error = setparams (pflags,pt0,pt1)) != 0) {
      printf ("bad news %ld on setup \n", error);
      DeletePort();
      exit (error);
```

```
    }

    actual = readPar (buffer,60000);

 /* CLOSE the parallel.device */
    CloseDevice (&IORpar);
    DeletePort (port);
    exit (0);
}

 /* PARALLEL I/O functions */

setparams(pf,ta0,ta1)

    unsigned char pf;
    unsigned long ta0;
    unsigned long ta1;

{
    int error;

    IORpar.io_ParFlags      = pf;
    IORpar.IOPar.io_Command = PDCMD_SETPARAMS;
    IORpar.io_PTermArray.PTermArray0 = ta0;
    IORpar.io_PTermArray.PTermArray1 = ta1;

    if ((error = DoIO (&IORpar)) != 0) {
        printf ("parallel.device setparams error %ld \n", error);
    }
    return (error);
}

readPar(data,length)
    char *data;
    ULONG length;
{
    int error;

    IORpar.IOPar.io_Data = data;
    IORpar.IOPar.io_Length = length;
    IORpar.IOPar.io_Command = CMD_READ;

    if ((error = DoIO (&IORpar)) != 0) {
        printf ("parallel.device read error %ld \n", error);
    }
    return (IORpar.IOPar.io_Actual);
}

writePar(data,length)
    char *data;
```

```
    int length;
{
    int     error;

    IORpar.IOPar.io_Data = data;
    IORpar.IOPar.io_Length = length;
    IORpar.IOPar.io_Command = CMD_WRITE;

    if ((error = DoIO (&IORpar)) != 0) {
        printf ("parallel.device write error %ld \n", error);
    }
    return (error);
}
```

Chapter 11

Printer Device

This chapter deals with the following basic topics:

- Using the AmigaDOS pathways to the printer device (opening the printer as an output file).
- Setting up for Exec printer I/O (creating an I/O request structure).
- Writing to the Exec printer to control its behavior.
- Writing characters or causing a graphics dump to a graphics capable printer.
- Closing the printer device.
- Creating your own printer device driver.

11.1. INTRODUCTION

There are four basic ways of doing output to a printer on the Amiga computer and three basic kinds of output you can send. You can send your output to:

- **PRT:** the DOS printer device
- **SER:** the DOS serial device
- **PAR:** the DOS parallel device

- **printer.device** (directly access the printer device itself)

Your output can take the form of:

- A character stream, consisting of commands and data (if sent through DOS or directly to the printer device)
- A command (if sent directly to the printer device)
- A graphics dump (also sent directly to the printer device)

The following section explains the various possible access pathways to the printer itself, along with advantages and disadvantages of each.

PRT: the DOS Printer Device

PRT: is the AmigaDOS printer device. By using the Workbench Preferences tool, you can direct the output to either a serial or parallel printer, which is the generic printer configured on the system. You may print (output) escape sequences to **PRT:** to specify the options you want. The escape sequences you send are interpreted by the printer driver and (usually different) escape sequences are forwarded to the printer. This is by far the easiest method for most applications. **PRT:** may be opened just like any other AmigaDOS file.

SER: the DOS Serial Device

SER: is the AmigaDOS serial device. If you "know" that the printer is connected to the serial port (you shouldn't) and you "know" what kind of printer it is (again, you shouldn't) then you could use AmigaDOS to open **SER:** and output characters to it, causing it to print. *This practice is strongly discouraged!* Characters you send are not examined or converted.

PAR: the DOS Parallel Device

PAR: is the AmigaDOS parallel device. The warnings given in the paragraph above apply here as well.

The Printer Device

By opening the Exec printer device directly, you have full control over the printer. You can either send escape sequences as shown in the command definitions table below for printer control, or call the **RawWrite()** routine to send raw characters directly to your printer with no processing at all. Using this technique would be similar to sending raw characters to **SER:** or **PAR:** from AmigaDOS, (but you don't need to know which one has the printer connected). Also note that all "commands" to the printer transmitted through the DOS printer access path must take the form of a character stream. Direct access to the printer device allows you to transmit other commands, such as reset or flush or, for graphics dumps, **DumpRPort()** (dump a raster to a graphics-capable printer).

11.2. PRINTER DEVICE OUTPUT

The printer device can be thought of as kind of a filter, in that some printers respond in one way to a command output and some respond in another. The printer device, as a standard printer interface, recognizes command sequences. Depending on the printer-dependent configuration that is currently loaded (by the Preferences tool), the printer device either ignores the command sequences or perhaps translates them into an entirely different sequence that this printer can actually understand and obey.

11.3. OPENING THE DOS PRINTER DEVICE

You can open the DOS printer device just as though it were a normal DOS output file. Here is an example program segment that accomplishes this:

```
struct File *file;

file = Open( "PRT:", MODE_NEWFILE );
if (file == 0) exit(PRINTER_WONT_OPEN);
```

Then, to print use code like this:

```
actual_length = Write(file, dataLocation, length);
```

where:

file
is a file handle (see the *AmigaDOS Developers Manual*).

dataLocation
is a pointer to the first character in the output stream you wish to write.

length
is the length of the output stream.

actual_length
is the actual length of the write, which, for the printer device, if there are no errors, is likely to always be the same as the length of write requested. The only exception is if you specify a value of -1 for length. In this case, -1 for length means that a null (0) terminated stream is being written to the printer device. The device returns the count of characters written prior to encountering the null. If it returns a value of -1 as **actual_length**, there has been an error.

Note that the **Open()** function could be called with **SER:** or **PAR:** if you don't want to have any character translation performed during the printer I/O.

When the printer I/O is complete, and your program is ready to exit, you should, as with all DOS devices (or Exec devices) close the device. Here is a sample function call that you could use:

```
Close(file);
```

Note that printer I/O through the DOS versions of the printer device must be done by a process, not by a task. DOS utilizes information in the process control block and would become confused if a simple task attempted to perform these activities. Printer I/O using the printer device directly, however, *can* be performed by a task.

11.4. DATA STRUCTURES USED DURING PRINTER I/O

This section shows you how to set up for Exec printer I/O. There are three distinct kinds of data structures required by the printer I/O routines. Some of the printer commands, such as start, stop, and flush, require only an **IOStdReq**. Others, such as write, require a larger data structure called an **IODRPReq** (for "dump a RastPort") or **IOPrtCmdReq** (for "printer command request"). For convenience, the printer device has defined a single data structure, called **printerIO**, that can be utilized to represent any of the three different kinds of printer communications request blocks.

The data structure type **printerIO** used in the following examples is a C-language union defined as:

```
union {
struct IOStdReq ios;
struct IODRPReq iodrp;
struct IOPrtCmdReq iopc;
} printerIO;
```

This means that one memory area can be used to represent three distinct forms of memory layout for the three different types of data structures that must be used to pass commands to the printer device. Some of the commands are simple and can use an **IOStdReq**. Some of the commands require many more parameters and extend the basic I/O request block accordingly. If you use the function **CreateExtIO()**, you can automatically allocate enough memory to hold the largest structure in the union statement.

11.5. CREATING AN I/O REQUEST

Printer I/O, like the I/O of other devices, requires that you create an I/O request message that you pass to the printer device for processing. The message contains the command as well as a data area. For a write, there will be a pointer in the data area to the stream of information you wish to write to the printer.

The following program fragment can be used to create the message block that you use for printer communications.

```
union printerIO *printerMsg;   /* I/O request block pointer */
struct Port *printerPort;      /* a port at which to receive */

printerPort = CreatePort("my.print.port",0);
printerMsg = (union printerIO *)CreateExtIO(printerPort,
        sizeof(union printerIO));
```

Error handling is not shown here. It is deferred to the example at the end of the chapter.

The routine **CreatePort()** is part of *amiga.lib*. The routine **CreateExtIO()** is in the Exec support functions printed in the appendix of this manual.

Note that there are two additional kinds of I/O request blocks that, for some commands, must be prepared for sending to the printer. They are called **IODRPReq** and **IOPrtCmdReq**. Both are outlined in the include file *devices/printer.h*. The function call to **CreateExtIO()** returns a pointer to a memory block the size of the largest form of printer **IORequest**.

11.6. OPENING A PRINTER DEVICE

You open a path to the printer device using code like the following.

```
int
OpenPrinter(request)
union printerIO *request;
{
        return(OpenDevice("printer.device",0,request,0));
}
```

This routine returns a value of zero if the printer device was opened successfully and a value other than zero if it did not open.

11.7. WRITING TO THE PRINTER

There are three forms of writing to the printer. The first uses a character stream that you create, possibly containing escape sequences to be processed by the printer driver ("PrintString" example) or containing just about anything else that is to be passed directly to the printer ("PrintRaw" example). The second form of write passes a command to the printer("PrintCmd" example). The third form asks for a graphics dump of a drawing area ("PrinterDump" example).

To write to the printer, you pass to the printer device the system standard command **CMD_WRITE**. Here are routines that can be used to send this command.

```
/* Send a NULL terminated string to the printer */

/* Assumes printer device is open and printerMsg
 * is correctly initialized.  Watches for embedded
 * "escape-sequences" and handles them as defined.
 */

int
PrintString(request,string)
char *string;
printerIO *request;
{
        request->ios.io_Command = CMD_WRITE;
        request->ios.io_Data = string;
        request->ios.io_Length = -1;
                /* if -1, the printer assumes it has been given
                 * a null terminated string.
                 */
        return(DoIO(request));
}

/* Send RAW character stream to the printer directly,
 * avoid "escape-sequence" parsing by the device.
 */

int
PrintRaw(request,buffer,count)
char *buffer;           /* where is the output stream of characters */
printerIO *request;     /* a properly initialized request block */
int count;          /* how many characters to output */
{
        /* queue a printer raw write */
        request->ios.io_Command = PRD_RAWWRITE;
        request->ios.io_Data = buffer;
        request->ios.io_Length = count;
        return(DoIO(request));
}
```

Printer Command Definitions

The following table describes the supported printer functions. You can use the escape sequences with **PRT:** and the printer device.

To transmit a command to the printer device, you can either:

- formulate a character stream containing the material shown in the "Escape Sequence" column of the table below, or
- send an **IORequest** to the printer device specifying which of these commands you wish to have performed. A sample routine for transmitting commands is shown immediately following the command table.

Again, recall that **SER:** and **PAR:** will ignore all of these and pass them directly on to the attached device.

Table 11-1: Printer Device Command Functions

Name	Cmd No.	Escape Sequence	Function	Defined by:
aRIS	0	ESCc	reset	ISO
aRIN	1	ESC#1	initialize	+++
aIND	2	ESCD	lf	ISO
aNEL	3	ESCE	return,lf	ISO
aRI	4	ESCM	reverse lf	ISO
aSGR0	5	ESC[0m	normal char set	ISO
aSGR3	6	ESC[3m	italics on	ISO
aSGR23	7	ESC[23m	italics off	ISO
aSGR4	8	ESC[4m	underline on	ISO
aSGR24	9	ESC[24m	underline off	ISO
aSGR1	10	ESC[1m	boldface on	ISO
aSGR22	11	ESC[22m	boldface off	ISO
aSFC	12	ESC[nm	set foreground color where n stands for a pair of ASCII digits, 3 followed by any number 0-9	ISO
aSBC	13	ESC[nm	set background color where n stands for a pair of ASCII digits, 4 followed by any number 0-9	ISO
aSHORP0	14	ESC[0w	normal pitch	DEC
aSHORP2	15	ESC[2w	elite on	DEC
aSHORP1	16	ESC[1w	elite off	DEC
aSHORP4	17	ESC[4w	condensed fine on	DEC
aSHORP3	18	ESC[3w	condensed off	DEC
aSHORP6	19	ESC[6w	enlarged on	DEC
aSHORP5	20	ESC[5w	enlarged off	DEC
aDEN6	21	ESC[6"z	shadow print on	DEC (sort of)
aDEN5	22	ESC[5"z	shadow print off	DEC

aDEN4	23	ESC[4"z	doublestrike on	DEC
aDEN3	24	ESC[3"z	doublestrike off	DEC
aDEN2	25	ESC[2"z	NLQ on	DEC
aDEN1	26	ESC[1"z	NLQ off	DEC
aSUS2	27	ESC[2v	superscript on	+++
aSUS1	28	ESC[1v	superscript off	+++
aSUS4	29	ESC[4v	subscript on	+++
aSUS3	30	ESC[3v	subscript off	+++
aSUS0	31	ESC[0v	normalize the line	+++
aPLU	32	ESCL	partial line up	ISO
aPLD	33	ESCK	partial line down	ISO
aFNT0	34	ESC(B	US char set	DEC
aFNT1	35	ESC(R	French char set	DEC
aFNT2	36	ESC(K	German char set	DEC
aFNT3	37	ESC(A	UK char set	DEC
aFNT4	38	ESC(E	Danish I char set	DEC
aFNT5	39	ESC(H	Swedish char set	DEC
aFNT6	40	ESC(Y	Italian char set	DEC
aFNT7	41	ESC(Z	Spanish char set	DEC
aFNT8	42	ESC(J	Japanese char set	+++
aFNT9	43	ESC(6	Norwegian char set	DEC
aFNT10	44	ESC(C	Danish II char set	+++
aPROP2	45	ESC[2p	proportional on	+++
aPROP1	46	ESC[1p	proportional off	+++
aPROP0	47	ESC[0p	proportional clear	+++
aTSS	48	ESC[n E	set proportional offset	ISO
aJFY5	49	ESC[5 F	auto left justify	ISO
aJFY7	50	ESC[7 F	auto right justify	ISO
aJFY6	51	ESC[6 F	auto full justify	ISO
aJFY0	52	ESC[0 F	auto justify off	ISO
aJFY3	53	ESC[3 F	letter space (justify)	ISO (special)
aJFY1	54	ESC[1 F	word fill(auto center)	ISO (special)
aVERP0	55	ESC[0z	1/8" line spacing	+++
aVERP1	56	ESC[1z	1/6" line spacing	+++
aSLPP	57	ESC[nt	set form length n	DEC
aPERF	58	ESC[nq	perf skip n (n>0)	+++
aPERF0	59	ESC[0q	perf skip off	+++
aLMS	60	ESC#9	Left margin set	+++
aRMS	61	ESC#0	Right margin set	+++
aTMS	62	ESC#8	Top margin set	+++
aBMS	63	ESC#2	Bottom margin set	+++
aSTBM	64	ESC[n;nr	T&B margins	DEC
aSLRM	65	ESC[n;ns	L&R margin	DEC
aCAM	66	ESC#3	Clear margins	+++
aHTS	67	ESCH	Set horiz tab	ISO
aVTS	68	ESCJ	Set vertical tabs	ISO
aTBC0	69	ESC[0g	Clr horiz tab	ISO
aTBC3	70	ESC[3g	Clear all h tab	ISO

aTBC1	71	ESC[1g	Clr vertical tabs	ISO
aTBC4	72	ESC[4g	Clr all v tabs	ISO
aTBCALL	73	ESC#4	Clr all h & v tabs	+++
aTBSALL	74	ESC#5	Set default tabs	+++
aEXTEND	75	ESC[n"x	Extended commands	+++

Legend:

ISO indicates that the sequence has been defined by the International Standards Organization. This is also very similar to ANSI x3.64.

DEC indicates a control sequence defined by Digital Equipment Corporation.

+++ indicates a sequence unique to Amiga.

n stands for a decimal number expressed as a set of ascii digits, for example 12.

11.8. TRANSMITTING A COMMAND TO THE PRINTER DEVICE

As noted above, to transmit a command to the printer device, you can either formulate an escape sequence and send it via the **CMD_WRITE** command, or you can utilize the command names and pass parameters and the command to the device. Here is a sample routine that uses the system command **PRD_PRTCOMMAND** to transmit a command to the device:

```
int
PrintCommand(request,command, p0, p1, p2, p3)
union printerIO *request;
int command, p0, p1, p2, p3;      /* command and its parameters */
{
      /* queue a printer command */
      request->iopc.io_Command = PRD_PRTCOMMAND;
      request->iopc.io_PrtCommand = command;
      request->iopc.io_Parm0 = p0;
      request->iopc.io_Parm1 = p1;
      request->iopc.io_Parm2 = p2;
      request->iopc.io_Parm3 = p3;
      return(DoIO(request));
}
```

As an example, suppose you wanted to set the left and right margins on your printer to columns 1 and 79 respectively. Here is a sample call to the **PrintCommand()** function for this purpose:

```
PrintCommand(aSLRM, 1, 79, 0, 0);
```

Consult the function table. Wherever there is a value of "n" to be substituted, it will be utilized from the next available parameter for this command. Most of the commands in the table need no parameters; some need one; some need two. Few if any require more than two parameters; however, this function provides room for expansion.

11.9. DUMPING A RASTPORT TO THE PRINTER

You can dump a **RastPort** (drawing area) to the printer by sending the command **PRD_DUMPRPORT** to the printer, along with several parameters that define how the dump is to be accomplished. The parameters shown in the sample dump function below are completely described in the summary section titled "printer.doc" in appendix B of this manual under the heading "printer.device/DumpRPort".

```
int
DumpRPort(request,rastPort, colorMap, modes, sx,sy, sw,sh, dc,dr, s)
        union printerIO *request;
        struct RastPort *rastPort;
        struct ColorMap *colorMap;
        ULONG modes;
        UWORD sx, sy, sw, sh;
        LONG dc, dr;
        UWORD s;
{
        request->iodrp.io_Command = PRD_DUMPRPORT;
        request->iodrp.io_RastPort = rastPort;
        request->iodrp.io_ColorMap = colorMap;
        request->iodrp.io_Modes = modes;
        request->iodrp.io_SrcX = sx;
        request->iodrp.io_SrcY = sy;
        request->iodrp.io_SrcWidth = sw;
        request->iodrp.io_SrcHeight = sh;
        request->iodrp.io_DestCols = dc;
        request->iodrp.io_DestRows = dr;
        request->iodrp.io_Special = s;
        return(DoIO(request));
}
```

As an example of this function, suppose you wanted to dump the current contents of the Workbench screen to the printer. The typical program code shown below would accomplish it. Note that during the dump no other tasks should be writing to the screen, nor should you use the mouse to move windows or otherwise modify the screen appearance.

```
#include "exec/types.h"
#include "intuition/intuition.h"
#include "devices/printer.h"
#define INTUITION_WONT_OPEN 1000

extern union printerIO *request;
extern int DumpRPort();
extern struct IORequest *CreateExtIO();

struct IntuitionBase *IntuitionBase;

struct NewWindow nw = {
      0, 0, 100, 40, 0, 1, 0, 0, NULL, NULL, NULL, NULL, NULL,
      0, 0, 0, 0, WBENCHSCREEN
      };

main()
{
      struct Window *w;
      struct Screen *screen;
      struct RastPort *rp;
      struct ViewPort *vp;
      struct ColorMap *cm;
      int modes,width,height;
      struct Port *printerPort;      /* at which to receive reply */
      IntuitionBase = OpenLibrary("intuition.library", 0);
      if (IntuitionBase === NULL) exit(INTUITION_WONT_OPEN);

      w = OpenWindow(&nw);
      if(w === NULL) goto cleanup1;

      screen = w->WScreen;      /* get screen address from window */
      CloseWindow(w);           /* once have screen address, no
                                 * more need for window, close it.
                                 */
      vp = &screen->ViewPort;  /* get screen's ViewPort, from
                                 * which the colormap will be gotten */
      rp = &screen->RastPort; /* get screen's RastPort, which
                                 * is what gets dumped to printer */

      cm = vp->ColorMap;/* retrieve pointer to colormap for
                                 * the printer dump */
      modes = vp->Modes;/* retrieve the modes variable */
      width = vp->DWidth;       /* retrieve width to print */
      height = vp->DHeight;     /* retrieve height to print */
```

```
        printerPort = CreatePort("my.print.port",0);
        request =  (union printerIO *)CreateExtIO(printerPort,
                        sizeof(union printerIO));

        error = OpenPrinter(request);
        if(error != 0) goto cleanup2;

        error = DumpRPort(
              request,/* pointer to initialized request */
              rp,   /* rastport pointer */
              cm,   /* color map pointer */
              modes,      /* low, high res, etc (display modes)*/
              0, 0,  /* x and y offsets into rastport */
              width,height, /* source size */
              width,height, /* dest rows, columns */
              0      /* io Special value, says print
                      * as pixels only, direct copy */
              };

        ClosePrinter(request);
cleanup2:
        DeleteExtIO(request, sizeof(union printerIO));
              DeletePort(printerPort);
        cleanup1:
              CloseLibrary(IntuitionBase);

        }           /* end of demo screen dump */
```

Additional Notes About Graphics Dumps

The print command accepts a 'use the largest area you've got' specification, that looks at the preferences active print width and active print height to bound the size of the print. These values are specified as a character count and a character size specification. Thus the width of the print is bounded by the number of inches specified by the following equation: (RIGHT_MARGIN - LEFT_MARGIN + 1) / CHARACTERS_PER_INCH. The height is specified by the equation: LENGTH / LINES_PER_INCH. Therefore, if characters are narrower, a constant number of them are also narrower.

NumRows in the printer tag refers to the number of dots in the graphics print element, and can be used by graphics render code to determine how much buffer space is needed to compose a line of graphics output. It has not been used in practice; the number has instead been hard coded into the render function specific to the printer.

If the printer you are developing for can be set to uni-directional mode under software control, we recommend that you put this in the initialization code for the printer (case 0 Master Initialization). This produces better looking printouts and (believe it or not) under most conditions, a faster printout.

11.10. CREATING A PRINTER DRIVER

Creating a printer-dependent code fragment for the printer device involves writing the data structures and code, compiling and assembling it, and linking it to produce an Amiga object binary file. The first piece in that file is the **PrinterSegment** structure described in *devices/prtbase.h* and *devices/prtbase.i* (which is pointed to by the BPTR returned by the **LoadSeg()** of the object file).

You specify the printer-dependent object file to load by specifying "custom printer" in Preferences and filling in the custom printer name with the name of the object file (relative to the directory *DEVS:printers/*).

The printer-dependent code **PrinterSegment** contains the **PrinterExtendedData** (PED) structure (also described in *devices/prtbase.h* and *devices/prtbase.i* at the beginning of the object). The PED structure contains data describing the capabilities of the printer, as well as pointers to code and other data. Here is the assembly code for a sample **PrinterSegment**, which would be linked to the beginning of the sequence of files describing the printer dependent code fragment.

```
*****************************************************************************
*
*     printer device dependent code tag
*
*****************************************************************************
                ; named sections are easier to exactly place in the linked file
                SECTION     custom

*------ Included Files ----------------------------------------

                INCLUDE     "exec/types.i"
                INCLUDE     "exec/nodes.i"

                INCLUDE     "revision.i"        ; contains VERSION & REVISION

                INCLUDE     "devices/prtbase.i"

*------ Imported Names ----------------------------------------

                XREF        _Init
                XREF        _Expunge
                XREF        _Open
```

```
                XREF        _Close
                XREF        _CommandTable
                XREF        _DoSpecial
                XREF        _Render

*------ Exported Names -----------------------------------------

                XDEF        _PEDData

*****************************************************************************

                ; in case anyone tries to execute this
                MOVEQ       #0,D0
                RTS

                DC.W        VERSION
                DC.W        REVISION
_PEDData:
                DC.L        printerName
                DC.L        _Init
                DC.L        _Expunge
                DC.L        _Open
                DC.L        _Close
                DC.B        PPC_BWGFX           ; PrinterClass
                DC.B        PCC_BW              ; ColorClass
                DC.B        80                  ; MaxColumns
                DC.B        1                   ; NumCharSets
                DC.W        8                   ; NumRows
                DC.L        960                 ; MaxXDots
                DC.L        0                   ; MaxYDots
                DC.W        120                 ; XDotsInch
                DC.W        82                  ; YDotsInch
                DC.L        _CommandTable       ; Command Strings
                DC.L        _DoSpecial          ; Command Code
                DC.L        _Render             ; Graphics Render
                DC.L        30                  ; Timeout

printerName:
                DC.B        'Custom Printer Name'
                DC.B        0
                EVEN
```

The printer name should be the brand name of the printer that is available for use by programs wishing to be specific about the printer name in any diagnostic or instruction messages. The four functions at the top of the structure are used to initialize this printer dependent code:

(*(PED->ped_Init))(PD);
is called when the printer-dependent code is loaded, and provides a pointer to the printer device for use by the printer-dependent code. It can also be used to open up any libraries or devices needed by the printer-dependent code.

(*(PED->ped_Expunge))();
is called immediately before the printer-dependent code is unloaded, to allow it to close any resources obtained at initialization time.

(*(PED->ped_Open))(ior);
is called in the process of an **OpenDevice()** call, after the Preferences are read and the correct primitive I/O device (parallel or serial) is opened. It must return zero if the open is successful, or nonzero to terminate the open and return an error to the user.

(*(PED->ped_Close))(ior);
is called in the process of a **CloseDevice()** call to allow the printer-dependent code to close any resources obtained at open time.

The **pd_** variable provided as a parameter to the initialization call is a pointer to the **PrinterData** structure described in *devices/prtbase.h* and *devices/prtbase.i*. This is also the same as the **io_Device** entry in printer I/O requests.

pd_SegmentData
points back to the **PrinterSegment**, which contains the PED.

pd_PrintBuf
is available for use by the printer-dependent code—it is not otherwise used by the printer device.

(*pd_PWrite)(data, length);
is the interface routine to the primitive I/O device. This routine uses two I/O requests to the primitive device, so writes are double-buffered. The data parameter points to the byte data to send, and the length is the number of bytes.

(*pd_PBothReady)();
waits for both primitive I/O requests to complete. This is useful if your code does not want to use double buffering. If you want to use the same data buffer for successive **pd_PWrites**, you must separate them with a call to this routine.

pd_Preferences
is the copy of Preferences in use by the printer device, obtained when the printer was opened.

The timeout field is the number of seconds that an I/O request from the printer device will remain posted and unsatisfied to the primitive I/O device (parallel or serial) before the timeout requester is presented to the user. This value should be large enough to avoid the

requester during normal printing.

Sample Code Provided

To help you in developing custom printer drivers for the Amiga, four sets of source files have been included as a part of this document. The files include "init.asm", "printertag.asm", "data.c", "render.c", and "dospecial.c".

Four sets of files for four different types of printers are provided:

diablo_c - an example of a ymcb color printer
epson - an example of a b/w printer
okimate20 - an example of a ymc_bw printer (has two render.c functions)
hpplus - an example of a single sheet multiple density printer

The source files for the hpplus includes one additional C-language source, named "density.c".

In addition, you will also need certain files that are common to all printer drivers. These are called *macros.i* and are printer assembly code macros that "init.asm" uses. All of these files are in appendix H of this manual.

Writing a Custom Graphics Printer Driver

Designing the graphics portion of a custom printer driver consists of two steps: writing a printer specific "render.c" function, and replacing the printer-specific values in "printertag.asm".

Note that a printer that does *not* support graphics has a very simple form of **Render()**; it returns an error. Here is sample code for **Render()** for a non-graphics printer (typically, an alphacom or diablo_630):

```
#include "exec/types.h"
#include "devices/printer.h"
int
Render()
{
        return(PDERR_NOTGRAPHICS);
}
```

The following section describes the contents of a typical driver for a printer that actually supports graphics. The example code for the Epson printer, contained in appendix H, shows a typical **Render()** function based on this description.

Render.c

This function is the main printer-specific code module and consists of six parts:

- Master initialization
- Pixel rendering
- Dumping a pixel buffer to the printer
- Clearing and initializing the pixel buffer
- Closing down
- Density selection

Master Initialization (case 0)

When this call is made, you are passed the width (in pixels) in x and the height (in pixels) in y of the picture as it should appear on the printer. Note that the printer non-specific code (using the printer specific values in printertag.asm (that will be discussed later), has already verified that these values are within range for the printer. It is recommended that you use

these values to allocate enough memory for a temporary buffer in which to build a command buffer for the printer. The buffer size needed is dependent on the specific printer, (usually) the width, and (sometimes) the height. In general, the buffer represents the commands and data required for one pass of the print head and usually takes the form of:

<start gfx cmd> <data> <end gfx cmd>

where:

<start
is the command required to define the graphic dump for each line.

<data>
is the binary data.

<end
is a terminator telling the printer to print the data (usually a carriage return).

For color printers, (usually) enough buffer space must be allocated for each different color ribbon, ink, and so on that the printer has (the okimate-20 and diablo_c-150 are provided as examples of this). Please refer to the sample drivers.

The example "render.c" functions use double buffering to reduce the dump time, that is why the **AllocMem()** call is for

(BUFSIZE times two)

where BUFSIZE represents the amount of memory for one entire print cycle (usually one pass of the print head).

Printers that would do more than one pass of the print head on a dump call are those that have to do a pass for each different main color that they want to lay down on the paper (like the okidata-20 with three colors and the epson_jx-80 with four colors). A printer such as the diablo_c-150 that can lay down all the colors in a single pass only needs to do one pass.

The number of passes the printer has to do is irrelevent to you. This topic was specified mainly to illustrate the true meaning of the term "one print cycle." You want to send the printer an entire print cycle to allow the main non-printer-specific driver to continue onward, computing the values for the next print cycle while the printer is printing the previous dots. This is why you will find double buffering used in the example driver code.

Any other initialization that the printer requires should also be done at this time. It is advisable that you also do a reset command so that you know what state the printer is in before you try to send it any further commands.

In addition, after performing a reset command it is advisable to send no other commands for at least one second to allow the printer to "calm down". Waiting after a reset is *strongly* recommended. The function **PWait(seconds,microseconds)** has been provided in the *wait.asm* file (see Appendix H) for this purpose. The *wait.asm* file must be assembled and linked into your custom printer device code.

Render Pixel (case 1)

When this call is made, your routine will be passed the x,y position of a single pixel and its color type. Note that the x,y value is an absolute value and you will have to do some modulus math (usually an AND) to compute the relative pixel position in your buffer. The absolute values will range from 0 to width-1 for x and 0 to height-1 for y. The color types are 0-black, 1-yellow, 2-magenta, and 3-cyan. Currently there is no provision for an RGB (red-green-blue) printer.

Dump Buffer to Printer (case 2)

When this call is made, you must send the buffer to the printer. As it now stands, there should be no need for you to change this routine. It should be common to all printers. It simply sends the buffer that you have been filling (via Case 1) to the printer.

You would want to change this routine only if you need to do some post-processing on the buffer before it is sent to the printer. For example, if your printer uses the hexadecimal number $03 as a command and requires that you send $03 $03 to send $03 as data, you would probably want to scan the buffer and expand $03's to $03 $03. Of course, you'll need to allocate space somewhere in order to expand the buffer.

Since the printer driver does not send you the blank pixels, you must initialize the buffer to values for blank pixels (usually 0). Clearing the buffer should be the same for all printers. Initializing the buffer is printer-specific and it includes placing the printer-specific control codes in the buffer ahead and behind of where the data will go.

Closing Down (case 4)

When this call is made you must wait for the print buffers to clear and then de-allocate the memory. This routine should be common to all printers. It simply waits for both buffers to empty, then deallocates the memory that they used. There should be no need for you to change this routine. If you do change it, however, make sure that the amount of memory allocated for Case 0 is deallocated by this routine.

Pre-Master Initialization (case 5)

Currently this option is implemented only on the HPLaserJet and HPLaserJet PLUS printers, although the call is made to each printer specific driver. Ignoring it causes no problems as the call is made simply to give you a chance to select a different density from the default one. You should note that this call is made *before* the master initialization call (case 0) and gives you a chance to alter any variables that the master initialization may use to program the printer. Refer to the HPLaserJet PLUS printer driver for an example of density selection.

Printertag.asm

The printer specific values that need to be filled in here are:

MaxXDots
the maximum number of dots the printer can print across the page.

MaxYDots
the maximum number of dots the printer can print down the page. Generally, if the printer supports roll or form feed paper this value should be 0 indicating that there is no limit. If the printer has a definte y dots maximum (as the HPLaserJet) this number should be entered here.

XDotsInch
the dot density in x (ie. 120 dpi).

YDotsInch
the dot density in y (ie. 144 dpi).

PrinterClass
the printer class the printer falls into. Current choices are:

PPC_BWALPHA - alphanumeric, no graphics.
PPC_BWGFX - black&white (only) graphics.
PPC_COLORGFX - color (and maybe b/w) graphics.

ColorClass
the color class the printer falls into. Current choices are:

PCC_BW - Black&White only (for example, EPSON).
PCC_YMC - Yellow Magenta Cyan only.
PCC_YMC_BW - Yellow or Black&White but not both (for example, Okimate 20).
PCC_YMCB - YellowMagentaCyanBlack (for example, Diablo_c-150).

NumRows
the number of pixel rows printed by one pass of the print head. This number is used by the non-printer-specific code to determine when to make a case 2 (see above) call to you. You have to keep this number in mind when determining how big a buffer you'll need to store one print cycle's worth of data.

Writing a Custom Alphanumeric Printer Driver

This alphanumeric section is meant to be read with the alpha listing for the EpsonX80 and Diablo Adv 25 close at hand.

The alphanumeric portion of the printer driver is designed to convert ANSI x3.64 style commands into the specific escape codes required by each individual printer. For example, the ANSI code for italics on is ESC[3m. The Epson FX80 printer would like a ESC%G to begin italic output mode. By using the printer driver all printers may be handled in a similar manner.

There are two parts to the alphanumeric portion of the printer driver: the **CommandTable** data table and the **DoSpecial()** routine.

Command Table

The **CommandTable** is used to convert all escape codes which can be handled by simple substitution. It has one entry per ANSI command supported by the printer driver. When you are creating a custom **CommandTable**, you must maintain the order of the commands in the same sequence as that shown in *printer.h* and *printer.i*. By placing the specific codes for your printer in the proper position, the conversion takes place automatically.

NOTE: If the code for your printer requires a decimal 0 (an ASCII NULL character), you enter this NULL into the **CommandTable** as octal 376 (decimal 254).

Placing an octal value of 377 (255 decimal) in a position in the command table indicates to the printer device that no simple conversion is available on this printer for this ANSI command. For example, if a printer does not support one of the functions (for example, if a daisy wheel printer does not have a foreign character set), 377 octal (255 decimal) is placed in that position. However, 377 in a position can also mean that the ANSI function is to be handled by code located in the **DoSpecial()** function.

DoSpecial() Function

The **DoSpecial()** function is meant to implement all the ANSI functions that can't be done by simple substitution, but can be handled by a more complex sequence of control characters sent to the printer. These are functions that need parameter conversion, read values from Preferences, and so on.

The **DoSpecial()** function is set up as follows:

```
#include       "exec/types.h"
#include "../devices/printer.h"
#include "../devices/prtbase.h"

extern struct PrinterData *PD;

DoSpecial(command,outputBuffer,vline,currentVMI,crlfFlag,Parms)
      char outputBuffer[];
      UWORD *command;
      BYTE *vline;
      BYTE *currentVMI;
      BYTE *crlfFlag;
      UBYTE Parms[];
{           /* code begins here... */
```

where:

command
: points to the command number. The *printer.h* contains the definitions for the routines to use (aRIN is initialize, and so on).

vline
points to the value for the current line position.

currentVMI
points to the value for the current line spacing.

crlfFlag
points to the setting of the "add line feed after carriage return" flag.

Parms
contain whatever parameters were given with the ANSI command.

outputBuffer
points to the memory buffer into which the converted command is returned.

Almost every printer will require an aRIN (initialize) command in **DoSpecial()**. This command reads the printer settings from Preferences and creates the proper control sequence for the specific printer. Also, it returns the character set to normal (not italicized, not bold, and so on). Other functions depend on the printer.

Certain functions are implemented both in the **CommandTable** and the **DoSpecial()** routine. These are functions like superscript, subscript, PLU (partial line up) and PLD (partial line down) that can often be handled by a simple conversion. However, certain of these functions must also adjust the printer device's line position variable.

Part IV

Chapter 1

Math Functions

This chapter describes the structure and calling sequences required to access the Motorola Fast Floating Point and IEEE Double Precision math libraries via the Amiga-supplied interfaces.

1.1. INTRODUCTION

In its present state, the FFP library consists of three separate entities: the basic math library, the transcendental math library, and C and ASM interfaces to the basic math library plus FFP conversion functions

The IEEE Double Precision library presently consists of one (1) entity: the basic math library

1.2. FFP FLOATING POINT DATA FORMAT

FFP Floating point variables are defined within C by the float or FLOAT directive. In assembler they are simply defined by a DC.L/DS.L statement. All FFP floating point variables are defined as 32-bit entities (long words) with the following format:

Table 1-1: FFP Floating Point Variable Format

MMMMMMMM	MMMMMMMM	MMMMMMMM	SEEEEEEE
31	23	15	7

where:

M = 24-bit mantissa

S = Sign of FFP number

E = Exponent in excess-64 notation

The mantissa is considered to be a binary fixed-point fraction; except for 0, it is always normalized (has a 1 bit in its highest position). Thus, it represents a value of less than 1 but greater than or equal to 1/2.

The sign bit is reset (0) for a positive value and set (1) for a negative value.

The exponent is the power of two needed to correctly position the mantissa to reflect the number's true arithmetic value. It is held in excess-64 notation which means that the twos-complement values are adjusted upward by 64, thus changing $40 (-64) through $3F (+63) to $00 through $7F. This facilitates comparisons among floating point values.

The value of 0 is defined as all 32 bits being 0s. The sign, exponent and mantissa are entirely cleared. Thus, 0s are always treated as positive.

The range allowed by this format is as follows:

DECIMAL:

```
 9.22337177 x 10**18  >  +VALUE  >  5.42101070  x 10**-20

-9.22337177 x 10**18  <  -VALUE  <  -2.71050535 x 10**-20
```

BINARY (HEXADECIMAL):

```
 .FFFFFF x 2**63  >  +VALUE  >  .800000  x 2**-63

-.FFFFFF x 2**63  <  -VALUE  <  -.800000 x 2**-64
```

Remember that you cannot perform *any* arithmetic on these variables without using the fast floating point libraries. The formats of the variables are *incompatible* with the arithmetic format of C-generated code, hence all floating point operations are performed through function calls.

1.3. FFP BASIC MATHEMATICS LIBRARY

The FFP basic math library resides in ROM and is opened by making a call to the **OpenLibrary()** function with **mathffp.library** as the argument. In C, this might be implemented as shown below.

```
int MathBase;

main()
{
        char lib_name[] = "mathffp.library";

        if ((MathBase = OpenLibrary(lib_name, 0)) < 1 ) {
                printf("Can't open %s: vector = %08x\n", lib_name,
                        MathBase);
                        exit(); }
                                .
                                .
                                .
}
```

The global variable **MathBase** is used internally for all future library references.

This library contains entries for the basic mathematics functions such as add, subtract, and so on. The C-called entry points are accessed via code generated by the C compiler when standard numerical operators are given within the source code. Note that to use either the C or assembly language interfaces to the basic math library all user code must be linked with the library *mathlink.lib*. The C entry points defined for the basic math functions are

ffixi — Convert FFP variable to integer
Usage: i1 = (int) f1;

fflti — Convert integer variable to FFP
Usage: f1 = (FLOAT) i1;

fcmpi — Compare two FFP variables
Usage: if (f1 <> f2) {};

ftsti — Test an FFP variable against zero
Usage: if (!f1) {};

fabsi — Take absolute value of FFP variable
Usage: f1 = abs(f2);

fnegi — Take 2's complement of FFP variable

Usage: f1 = -f2;

faddi — Add two FFP variables

Usage: f1 = f2 + f3;

fsubi — Subtract two FFP variables

Usage: f1 = f2 - f3;

fmuli — Multiply two FFP variables

Usage: f1 = f2 * f3;

fdivi — Divide two FFP variables

Usage: f1 = f2 / f3;

Be sure to include proper data type definitions as shown in the example below.

```
#include <mathffp.h>
int MathBase;

main()
{
      FLOAT f1, f2, f3;
      int   i1, i2, i3;
      char lib_name[] = "mathffp.library";

      if((MathBase = OpenLibrary(lib_name, 0)) < 1 ) {
            printf("Can't open %s: vector = %08x\n", lib_name,
                  MathBase);
            exit();  }

      i1 = (int) f1;          /* Call ffixi entry */
      fi = (FLOAT) i1;        /* Call fflti entry */

      if (f1 < f2) {};        /* Call fcmpi entry */
      if (!f1) {};            /* Call ftsti entry */

      f1 = abs(f2);           /* Call fabsi entry */
      f1 = -f2;               /* Call fnegi entry */
      f1 = f2 + f3;           /* Call faddi entry */
      f1 = f2 - f3;           /* Call fsubi entry */
      f1 = f2 * f3;           /* Call fmuli entry */
      f1 = f2 / f3;           /* Call fdivi entry */
}
```

The Amiga assembly language interface to the Motorola Fast Floating Point basic math routines is shown below, including some details about how the system flags are affected by each operation. This interface resides in the library file *mathlink.lib* and must be linked with the user code. Note that the access mechanism from assembly language is:

```
LEA    _LVOSPFix,A6
JSR    _MAthBase(A6)
```

_LVOSPFix - Convert FFP to integer

Inputs:	D0 = FFP argument
Outputs:	D0 = Integer (2's complement) result
Condition codes:	N = 1 if result is negative
	Z = 1 if result is zero
	V = 1 if overflow occurred
	C = undefined
	X = undefined

_LVOSPFlt - Convert integer to FFP

Inputs:	D0 = Integer (2's complement) argument
Outputs:	D0 = FFP result
Condition codes:	N = 1 if result is negative
	Z = 1 if result is zero
	V = 0
	C = undefined
	X = undefined

_LVOSPCmp - Compare

Inputs:	D1 = FFP argument 1
	D0 = FFP argument 2
Outputs:	D0 = +1 if arg1 < arg2
	D0 = -1 if arg1 > arg2
	D0 = 0 if arg1 = arg2
Condition codes:	N = 0
	Z = 1 if result is zero
	V = 0
	C = undefined
	X = undefined
	GT = arg2 > arg1
	GE = arg2 >= arg1
	EQ = arg2 = arg1
	NE = arg2 <> arg1
	LT = arg2 < arg1
	LE = arg2 <= arg1

_LVOSPTst - Test

Inputs:	D1 = FFP argument
Outputs:	D0 = +1 if arg > 0.0
	D0 = -1 if arg < 0.0
	D0 = 0 if arg = 0.0
Condition codes:	N = 1 if result is negative
	Z = 1 if result is zero
	V = 0
	C = undefined
	X = undefined
	EQ = arg = 0.0
	NE = arg <> 0.0
	PL = arg >= 0.0
	MI = arg < 0.0

_LVOSPAbs - Absolute value

Inputs:	D0 = FFP argument
Outputs:	D0 = FFP absolute value result
Condition codes:	N = 0
	Z = 1 if result is zero
	V = 0
	C = undefined
	X = undefined

_LVOSPNeg - Negate

Inputs:	D0 = FFP argument
Outputs:	D0 = FFP negated result
Condition codes:	N = 1 if result is negative
	Z = 1 if result is zero
	V = 0
	C = undefined
	X = undefined

_LVOSPAdd - Addition

Inputs:	D1 = FFP argument 1
	D0 = FFP argument 2
Outputs:	D0 = FFP addition of arg1+arg2 result
Condition codes:	N = 1 if result is negative
	Z = 1 if result is zero
	V = 1 if result overflowed
	C = undefined
	Z = undefined

_LVOSPSub - Subtraction

Inputs:	D1 = FFP argument 1 D0 = FFP argument 2
Outputs:	D0 = FFP subtraction of arg1-arg2 result
Condition codes:	N = 1 if result is negative Z = 1 if result is zero V = 1 if result overflowed C = undefined Z = undefined

_LVOSPMul - Multiply

Inputs:	D1 = FFP argument 1 D2 = FFP argument 2
Outputs:	D0 = FFP multiplication of arg1*arg2 result
Condition codes:	N = 1 if result is negative Z = 1 if result is zero V = 1 if result overflowed C = undefined Z = undefined

_LVOSPDiv - Divide

Inputs:	D1 = FFP argument 1 D0 = FFP argument 2
Outputs:	D0 = FFP division of arg1/arg2 result
Condition codes:	N = 1 if result is negative Z = 1 if result is zero V = 1 if result overflowed C = undefined Z = undefined

1.4. FFP TRANSCENDENTAL MATHEMATICS LIBRARY

The FFP transcendental math library resides on disk and must be accessed in the same way as the basic math library after it is loaded into system RAM. The name to be included in the **OpenLibrary()** call is *mathtrans.library*. In C, this might be implemented as follows.

```
int MathBase;
int MathTransBase;

main()
{
      char bmath_name[] = "mathffp.library";
      char tmath_name[] = "mathtrans.library";

      if((MathBase = OpenLibrary(bmath_name, 0)) < 1 ) {
            printf("Can't open %s: vector = %08x\n", bmath_name,
                  MathBase);
            exit(); }

      if((MathTransBase = OpenLibrary(tmath_name, 0)) < 1 ) {
            printf("Can't open %s: vector = %08x\n", tmath_name,
            MathTransBase);
      exit(); }
            .
            .
            .
}
```

The global variables **MathBase** and **MathTransBase** are used internally for all future library references. Note that the transcendental math library is dependent upon the basic math library and, therefore, is opened after the basic math library has been opened.

This library contains entries for the transcendental math functions sine, cosine, and so on. The C-called entry points are accessed via code generated by the C compiler when the actual function names are given within the source code. The C entry points defined for the transcendental math functions are

SPAsin — Return arcsine of FFP variable.

Usage: f1 = SPAsin(f2);

SPAcos — Return arccosine of FFP variable.

Usage: f1 = SPAcos(f2);

SPAtan — Return arctangent of FFP variable.

Usage: f1 = SPAtan(f2);

SPSin — Return sine of FFP variable. This function accepts an FFP radian argument and returns the trigonometric sine value. For extremely large arguments where little or no precision would result, the computation is aborted and the "V" condition code set. A direct return to the caller is made.

Usage: f1 = SPSin(f2);

SPCos Return cosine of FFP variable. This function accepts an FFP radian argument and returns the trigonometric cosine value. For extremely large arguments where little or no precision would result, the computation is aborted and the "V" condition code set. A direct return to the caller is made.

Usage: f1 = SPCos(f2);

SPTan Return tangent of FFP variable. This function accepts an FFP radian argument and returns the trigonometric tangent value. For extremely large arguments where little or no precision would result, the computation is aborted and the "V" condition code set. A direct return to the caller is made.

Usage: f1 = SPTan(f2);

SPSincos Return sine and cosine of FFP variable. This function accepts an FFP radian argument and returns both the trigonometric sine and cosine values. If both the sine and cosine are required for a single radian value of interest, this function will result in almost twice the execution speed of calling the sin and cos functions independently. For extremely large arguments where little or no precision would result, the computation is aborted and the "V" condition code set. A direct return to the caller is made.

Usage: f1 = SPSincos(&f3, f2);

SPSinh Return hyperbolic sine of FFP variable.

Usage: f1 = SPSinh(f2);

SPCosh Return hyperbolic cosine of FFP variable.

Usage: f1 = SPCosh(f2);

SPTanh Return hyperbolic tangent of FFP variable.

Usage: f1 = SPTanh(f2);

SPExp Return *e* to the FFP variable power. This function accepts an FFP argument and returns the result representing the value of *e* (2.71828...) raised to that power.

Usage: f1 = SPExp(f2);

SPLog Return natural log (base *e*) of FFP variable.

Usage: f1 = SPLog(f2);

SPLog10 Return naparian log (base 10) of FFP variable.

Usage: f1 = SPLog10(f2);

SPPow Return FFP arg2 to FFP arg1.

Usage: f1 = SPPow(f3, f2);

SPSqrt Return square root of FFP variable.

Usage: f1 = SPSqrt(f2);

SPTieee Convert FFP variable to IEEE format
Usage: i1 = SPTieee(f1);

SPFieee Convert IEEE variable to FFP format.
Usage: f1 = SPFieee(i1);

Be sure to include proper data type definitions as shown in the example below.

```
#include <mathffp.h>

int MathBase;
int MathTransBase;

main()
{
      FLOAT f1, f2, f3;
      int   i1, i2, i3;
      char bmath_name[] = "mathffp.library";
      char tmath_name[] = "mathtrans.library";

      if((MathBase = OpenLibrary(bmath_name, 0)) < 1 ) {
            printf("Can't open %s: vector = %08x\n", bmath_name, MathBase);
            exit();  }

      if((MathTransBase = OpenLibrary(tmath_name, 0)) < 1 ) {
            printf("Can't open %s: vector = %08x\n", tmath_name, MathTransBase);
            exit(); . }


        f1 = SPAsin(f2);          /* Call SPAsin entry */
        f1 = SPAcos(f2);          /* Call SPAcos entry */
        f1 = SPAtan(f2);          /* Call SPAtan entry */

        f1 = SPSin(f2);           /* Call SPSin entry */
        f1 = SPCos(f2);           /* Call SPCos entry */
        f1 = SPTan(f2);           /* Call SPTan entry */
        f1 = SPSincos(&f3, f2);   /* Call SPSincos entry */

        f1 = SPSinh(f2);          /* Call SPSinh entry */
        f1 = SPCosh(f2);          /* Call SPCosh entry */
        f1 = SPTanh(f2);          /* Call SPTanh entry */

        f1 = SPExp(f2);           /* Call SPExp entry */
        f1 = SPLog(f2);           /* Call SPLog entry */
        f1 = SPLog10(f2);         /* Call SPLog10 entry */
        f1 = SPPow(f2);           /* Call SPPow entry */
        f1 = SPSqrt(f2);          /* Call SPSqrt entry */

        i1 = SPTieee(f2);         /* Call SPTieee entry */
```

```
f1 = SPFieee(i1);          /* Call SPFieee entry */
}
```

The section below describes the Amiga assembly language interface to the Motorola Fast Floating Point transcendental math routines and includes some details about how the system flags are affected by the operation. Again, this interface resides in the library file *mathlink.lib* and must be linked with the user code. Note that the access mechanism from assembly language is:

```
LEA    _LVOSPAsin,A6
JSR    _MathTransBase(A6)
```

_LVOSPAsin - Arcsine

Inputs:	D0 = FFP argument
Outputs:	D0 = FFP arctangent radian result
Condition codes:	N = 0
	Z = 1 if result is zero
	V = 0
	C = undefined
	X = undefined

_LVOSPAcos - Arccosine

Inputs:	D0 = FFP argument
Outputs:	D0 = FFP arctangent radian result
Condition codes:	N = 0
	Z = 1 if result is zero
	V = 0
	C = undefined
	X = undefined

_LVOSPAtan - Arctangent

Inputs:	D0 = FFP argument
Outputs:	D0 = FFP arctangent radian result
Condition codes:	N = 0
	Z = 1 if result is zero
	V = 0
	C = undefined
	X = undefined

_LVOSPSin - Sine

Inputs:	D0 = FFP argument in radians
Outputs:	D0 = FFP sine result
Condition codes:	N = 1 if result is negative
	Z = 1 if result is zero
	V = 1 if result is meaningless
	(that is, input magnitude too large)
	C = undefined
	X = undefined

_LVOSPCos - Cosine

Inputs:	D0 = FFP argument in radian
Outputs:	D0 = FFP cosine result
Condition codes:	N = 1 if result is negative
	Z = 1 if result is zero
	V = 1 if result is meaningless
	(that is, input magnitude too large)
	C = undefined
	X = undefined

_LVOSPTan - Tangent

Inputs:	D0 = FFP argument in radians
Outputs:	D0 = FFP tangent result
Condition codes:	N = 1 if result is negative
	Z = 1 if result is zero
	V = 1 if result is meaningless
	(that is, input magnitude too large)
	C = undefined
	X = undefined

_LVOSPSincos - Sine and cosine

Inputs:	D0 = FFP argument in radians
	D1 = Address to store cosine result
Outputs:	D0 = FFP sine result
	(D1) = FFP cosine result
Condition codes:	N = 1 if result is negative
	Z = 1 if result is zero
	V = 1 if result is meaningless
	(that is, input magnitude too large)
	C = undefined
	X = undefined

_LVOSPSinh - Hyperbolic sine

Inputs:	D0 = FFP argument in radians
Outputs:	D0 = FFP hyperbolic sine result
Condition codes:	N = 1 if result is negative
	Z = 1 if result is zero
	V = 1 if overflow occurred
	C = undefined
	X = undefined

_LVOSPCosh - Hyperbolic cosine

Inputs:	D0 = FFP argument in radians
Outputs:	D0 = FFP hyperbolic cosine result
Condition codes:	N = 1 if result is negative
	Z = 1 if result is zero
	V = 1 if overflow occurred
	C = undefined
	X = undefined

_LVOSPTanh - Hyperbolic tangent

Inputs:	D0 = FFP argument in radians
Outputs:	D0 = FFP hyperbolic tangent result
Condition codes:	N = 1 if result is negative
	Z = 1 if result is zero
	V = 1 if overflow occurred
	C = undefined
	X = undefined

_LVOSPExp - Exponential

Inputs:	D0 = FFP argument
Outputs:	D0 = FFP exponential result
Condition codes:	N = 0
	Z = 1 if result is zero
	V = 1 if overflow occurred
	C = undefined
	Z = undefined

_LVOSPLog - Natural logarithm

Inputs:	D0 = FFP argument
Outputs:	D0 = FFP natural logarithm result
Condition codes:	N = 1 if result is negative
	Z = 1 if result is zero
	V = 1 if argument negative or zero
	C = undefined
	Z = undefined

_LVOSPLog10 - Naparian (base 10) logarithm

Inputs:	D0 = FFP argument
Outputs:	D0 = FFP natural logarithm result
Condition codes:	N = 1 if result is negative
	Z = 1 if result is zero
	V = 1 if argument negative or zero
	C = undefined
	Z = undefined

_LVOSPPow - Power

Inputs:	D1 = FFP argument value
	D0 = FFP exponent value
Outputs:	D0 = FFP result of arg taken to exp power
Condition codes:	N = 0
	Z = 1 if result is zero
	V = 1 if result overflowed or arg < 0
	C = undefined
	Z = undefined

_LVOSPSqrt - Square root

Inputs:	D0 = FFP argument
Outputs:	D0 = FFP square root result
Condition codes:	N = 0
	Z = 1 if result is zero
	V = 1 if argument was negative
	C = undefined
	Z = undefined

_LVOSPTieee - Convert to IEEE format

Inputs:	D0 = FFP format argument
Outputs:	D0 = IEEE floating point format result
Condition codes:	N = 1 if result is negative
	Z = 1 if result is zero
	V = undefined
	C = undefined
	Z = undefined

_LVOSPFieee - Convert from IEEE format

Inputs:	D0 = IEEE floating point format argument
Outputs:	D0 = FFP format result
Condition codes:	N = undefined
	Z = 1 if result is zero
	V = 1 if result overflowed FFP format
	C = undefined
	Z = undefined

1.5. FFP MATHEMATICS CONVERSION LIBRARY

The FFP mathematics conversion library is accessed by linking code into the executable file being created. The name of the file to include in the library description of the link command line is *mathlink_lib.lib*. With this inclusion, direct calls are made to the conversion functions. Only a C interface exists for the conversion functions; there is no assembly language interface. The basic math library is required in order to access these functions and might be opened as shown below.

```
int MathBase;

main()
{
      char bmath_name[] = "mathffp.library";

      if ((MathBase = OpenLibrary(bmath_name, 0)) < 1 ) {
            printf("Can't open %s: vector = %08x\n", bmath_name,
                  MathBase);
            exit(); }

                  .
                  .
                  .
}
```

The global variable **MathBase** is used internally for all future basic math library references.

This library contains entries for the conversion functions associated with math library usage. The C-called entry points are accessed via code generated by the C compiler when the actual function names are given within the source code. The C entry points defined for the math conversion functions are:

afp — Convert ASCII string into FFP equivalent.
Usage: fnum = afp(&string[0]);

fpa — Convert FFP variable into ASCII equivalent.
Usage: exp = fpa(fnum, &string[0]);

arnd — Round ASCII representation of FFP number.
Usage: arnd(place, exp, &string[0]);

dbf — Convert FFP dual-binary number to FFP equivalent.
Usage: fnum = dbf(exp, mant);

fpbcd Convert FFP variable to BCD equivalent.

Usage: fpbcd(fnum, &string[0]);

Be sure to include proper data type definitions as shown in the example below. Print statements have been included to help clarify the format of the math conversion function calls.

```
#include <mathffp.h>

char st1[80] = "3.1415926535897";
char st2[80] = "2.718281828459045";
char st3[80], st4[80];

int MathBase;

main()
{
      FLOAT num1, num2, num3, num4, num5, num6, num7, num8, num9;
      FLOAT n1, n2, n3, n4, n5, n6, n7, n8, n9;
      int   i1, i2, i3, i4, i5, i6, i7, i8, i9;
      int exp1, exp2, exp3, exp4, mant1, mant2,
            mant3, mant4, place1, place2;

if ((MathBase=OpenLibrary("mathffp.library",0)) < 1 ) {
            printf("Can't open mathffp.library:vector =%08x\n",
                  MathBase);
            exit();
      }

      n1 = afp(st1);                /* Call afp entry */
      n2 = afp(st2);                /* Call afp entry */
      printf("\n\nASCII %s converts to floating point %f",
            st1, n1);
      printf("\nASCII %s converts to floating point %f",
            st2, n2);

      num1 = 3.1415926535897;
      num2 = 2.718281828459045;

      exp1 = fpa(num1, st3);        /* Call fpa entry */
      exp2 = fpa(num2, st4);        /* Call fpa entry */
      printf("\n\nfloating point %f converts to ASCII %s",
            num1, st3);
      printf("\nfloating point %f converts to ASCII %s",
            num2, st4);

      place1 = -2;
      place2 = -1;
      arnd(place1, exp1, st3);      /* Call arnd entry */
```

```
    arnd(place2, exp2, st4);        /* Call arnd entry */
    printf("\nASCII round of %f to %d places yields %s",
        num1, place1, st3);
    printf("\nASCII round of %f to %d places yields %s",
        num2, place2, st4);

    exp1 = -3;  exp2 = 3;   exp3 = -3;  exp4 = 3;
    mant1 = 12345;  mant2 = -54321;  mant3 = -12345;
    t4 = 54321;
    n1 = dbf(exp1, mant1);          /* Call dbf entry */
    n2 = dbf(exp2, mant2);          /* Call dbf entry */
    n3 = dbf(exp3, mant3);          /* Call dbf entry */
    n4 = dbf(exp4, mant4);          /* Call dbf entry */
    printf("\n\ndbf of exp = %d and mant = %d yields FFP number
        of %f",     exp1, mant1, n1);
    printf("\ndbf of exp = %d and mant = %d yields FFP number
        of %f",     exp2, mant2, n2);
    printf("\ndbf of exp = %d and mant = %d yields FFP number
        of %f",     exp3, mant3, n3);
    printf("\ndbf of exp = %d and mant = %d yields FFP number
        of %f",     exp4, mant4, n4);

    num1 = -num1;
    fpbcd(num1, st3);               /* Call fpbcd entry */
    st3[8] = '\0';
    strcpy(&i2, &st3[4]);
    st3[4] = '\0';
    strcpy(&i1, st3);
    printf("\n\nfloating point %f converts to BCD
        %08x%08x", num1, i1, i2);
    num2 = -num2;
    fpbcd(num2, st4);               /* Call fpbcd entry */
    st4[8] = '\0';
    strcpy(&i4, &st4[4]);
    st4[4] = '\0';
    strcpy(&i3, st4);
    printf("\nfloating point %f converts to BCD
        %08x%08x", num2, i3, i4);
}
```

1.6. IEEE DOUBLE-PRECISION BASIC MATH LIBRARY

The IEEE double precision basic math library resides on disk and is opened by making a call to the **OpenLibrary()** function with **mathieeedoubbas.library** as the argument. In C, this might be implemented as shown below.

```
int MathIeeeDoubBasBase;

main()
{
      char lib_name[] = "mathieeedoubbas.library";

      if ((MathIeeeDoubBasBase = OpenLibrary(lib_name, 0)) < 1 ) {
            printf("Can't open %s: vector = %08x\n", lib_name,
                  MathIeeeDoubBasBase);
                  exit(); }
                        .
                        .
                        .
}
```

The global variable **MathIeeeDoubBasBase** is used internally for all future library references.

This library contains entries for the basic mathematics functions such as add, subtract, and so on. The C-called entry points are accessed via code generated by the C compiler when the actual function names are given within the source code. The C entry points defined for the IEEE double precision basic math functions are

IEEEDPFix
: Convert IEEE double precision variable to integer
  Usage: i1 = IEEEDPFix(f1);

IEEEDPFlt
: Convert integer variable to IEEE double precision
  Usage: f1 = IEEEDPFlt(i1);

IEEEDPCmp
: Compare two IEEE double precision variables
  Usage: switch (IEEEDPCmp(f1, f2)) {};

IEEEDPTst
: Test an IEEE double precision variable against zero
  Usage: switch (IEEEDPTst(f1)) {};

IEEEDPAbs
: Take absolute value of IEEE double precision variable
  Usage: f1 = IEEEDPAbs(f2);

IEEEDPNeg

Take 2's complement of IEEE double precision variable

Usage: f1 = IEEEDPNeg(f2);

IEEEDPAdd

Add two IEEE double precision variables

Usage: f1 = IEEEDPAdd(f2, f3);

IEEEDPSub

Subtract two IEEEDPSub variables

Usage: f1 = IEEEDPSub(f2, f3);

IEEEDPMul

Multiply two IEEE double precision variables

Usage: f1 = IEEEDPMul(f2, f3);

IEEEDPDiv

Divide two IEEE double precision variables

Usage: f1 = IEEEDPDiv(f2, f3);

Be sure to include proper data type definitions as shown in the example below.

```
int MathIeeeDoubBasBase;

main()
{
      double f1, f2, f3;
      int    i1, i2, i3;
      char lib_name[] = "mathieeedoubbas.library";

      if((MathIeeeDoubBasBase = OpenLibrary(lib_name, 0)) < 1 ) {
            printf("Can't open %s: vector = %08x\n", lib_name,
                  MathIeeeDoubBasBase);
            exit(); }

      i1 = IEEEDPFix(f1);                    /* Call IEEEDPFix entry */
      fi = IEEEDPFlt(i1);                    /* Call IEEEDPFlt entry */
      switch (IEEEDPCmp(f1, f2)) {};         /* Call IEEEDPCmp entry */
      switch (IEEEDPTst(f1)) {};             /* Call IEEEDPTst entry */
      f1 = IEEEDPAbs(f2);                    /* Call IEEEDPAbs entry */
      f1 = IEEEDPNeg(f2);                    /* Call IEEEDPNeg entry */
      f1 = IEEEDPAdd(f2, f3);                /* Call IEEEDPAdd entry */
      f1 = IEEEDPSub(f2, f3);                /* Call IEEEDPSub entry */
      f1 = IEEEDPMul(f2, f3);                /* Call IEEEDPMul entry */
      f1 = IEEEDPDiv(f2, f3);                /* Call IEEEDPDiv entry */
}
```

The Amiga assembly language interface to the IEEE double precision floating point basic math routines is shown below, including some details about how the system flags are affected by each operation. Note that the access mechanism from assembly language is:

```
LEA    _LVOIEEEDPFix,A6
JSR    _MathIeeeDoubBasBase(A6)
```

_LVOIEEEDPFix - Convert IEEE double precision to integer

Inputs:	D0/D1 = IEEE double precision argument
Outputs:	D0 = Integer (2's complement) result
Condition codes:	N = 1 if result is negative
	Z = 1 if result is zero
	V = 1 if overflow occurred
	C = undefined
	X = undefined

_LVOIEEEDPFlt - Convert integer to IEEE double precision

Inputs:	D0 = Integer (2's complement) argument
Outputs:	D0/D1 = IEEE double precision result
Condition codes:	N = 1 if result is negative
	Z = 1 if result is zero
	V = 0
	C = undefined
	X = undefined

_LVOIEEEDPCmp - Compare two IEEE double precision values

Inputs:	D0/D1 = IEEE double precision argument 1
	D2/D3 = IEEE double precision argument 2
Outputs:	D0 = +1 if arg1 < arg2
	D0 = -1 if arg1 > arg2
	D0 = 0 if arg1 = arg2
Condition codes:	N = 0
	Z = 1 if result is zero
	V = 0
	C = undefined
	X = undefined
	GT = arg2 > arg1
	GE = arg2 >= arg1
	EQ = arg2 = arg1
	NE = arg2 <> arg1
	LT = arg2 < arg1
	LE = arg2 <= arg1

_LVOIEEEDPTst - Test an IEEE double-precision value against zero

Inputs:	D0/D1 = IEEE double precision argument
Outputs:	D0 = +1 if arg > 0.0
	D0 = -1 if arg < 0.0
	D0 = 0 if arg = 0.0
Condition codes:	N = 1 if result is negative
	Z = 1 if result is zero
	V = 0
	C = undefined
	X = undefined
	EQ = arg = 0.0
	NE = arg <> 0.0
	PL = arg >= 0.0
	MI = arg < 0.0

_LVOIEEEDPAbs - Absolute value

Inputs:	D0/D1 = IEEE double precision argument
Outputs:	D0/D1 = IEEE double precision absolute value result
Condition codes:	N = 0
	Z = 1 if result is zero
	V = 0
	C = undefined
	X = undefined

_LVOIEEEDPNeg - Negate

Inputs:	D0/D1 = IEEE double precision argument
Outputs:	D0/D1 = IEEE double precision negated result
Condition codes:	N = 1 if result is negative
	Z = 1 if result is zero
	V = 0
	C = undefined
	X = undefined

_LVOIEEEDPAdd - Addition

Inputs:	D0/D1 = IEEE double precision argument 1
	D2/D3 = IEEE double precision argument 2
Outputs:	D0/D1 = IEEE double precision addition of arg1+arg2 result
Condition codes:	N = 1 if result is negative
	Z = 1 if result is zero
	V = 1 if result overflowed
	C = undefined
	Z = undefined

_LVOIEEEDPSub - Subtraction

Inputs:	D0/D1 = IEEE double precision argument 1 D2/D3 = IEEE double precision argument 2
Outputs:	D0/D1 = IEEE double precision subtraction of arg1-arg2 result
Condition codes:	N = 1 if result is negative Z = 1 if result is zero V = 1 if result overflowed C = undefined Z = undefined

_LVOIEEEDPMul - Multiply

Inputs:	D0/D1 = IEEE double precision argument 1 D2/D3 = IEEE double precision argument 2
Outputs:	D0/D1 = IEEE double precision multiplication of arg1*arg2 result
Condition codes:	N = 1 if result is negative Z = 1 if result is zero V = 1 if result overflowed C = undefined Z = undefined

_LVOIEEEDPDiv - Divide

Inputs:	D0/D1 = IEEE double precision argument 1 D2/D3 = IEEE double precision argument 2
Outputs:	D0/D1 = IEEE double precision division of arg1/arg2 result
Condition codes:	N = 1 if result is negative Z = 1 if result is zero V = 1 if result overflowed C = undefined Z = undefined

Chapter 2

Workbench

This chapter shows how to use the Workbench facilities in your applications. For information about *IconEd*, the icon editor for making Workbench icons, see the appendixes of the *Introduction to Amiga* manual for revision 1.1 of the system software.

2.1. INTRODUCTION

Workbench is both an application program and a screen where other applications can run. Workbench allows users to interact with the Amiga file system by using icons, and it gives the programmer access to a body of library functions for manipulating the application's objects and icons.

Here are definitions of some terms that may be unfamiliar or used in unfamiliar ways.

Workbench object
: A Workbench object contains all the information that Workbench needs to display and use a project, tool, drawer, etc. The two kinds of Workbench objects are **WBObject** (as Workbench uses objects) and **DiskObject** (as most other users will view objects in memory or in a file on disk).

icon
: This is a shorthand name for a Workbench object. An icon may be in memory or on disk or both.

info file
: The disk representation of an icon. The format of an icon on disk is slightly different from an icon in memory, but one is obtainable from the other.

strings
: A null-terminated sequence of bytes.

activating
The act of starting a tool, opening a drawer, and so on. The term *opening* is reserved for windows and files.

tool
An application program or system utility.

project
Something produced by an executable program and associated with an executable program. For example, a text file or a drawing.

drawer
A disk-based directory.

2.2. THE ICON LIBRARY

The icon library, "icon.library", library has memory managment routines, icon input and output routines, and string manipulation routines. The function appendix to this manual contains the reference pages for this library.

2.3. THE INFO FILE

The *info file* is the center of interaction between applications and Workbench. This file stores all the necessary information to display an icon and to start up an application. An info file can contain several different types of icons, as shown in table 2-1.

Table 2-1: Contents of a Workbench Info File

Icon Name	*Object*
WBDISK	the root of a disk
WBDRAWER	a directory on the disk
WBTOOL	a directly runnable program
WBPROJECT	a data file of some sort
WBGARBAGE	the trash can directory
WBKICK	a non-DOS disk

The actual data present in the info file depends on the icon type. Note that *any* graphical image can be used for any icon type in the info file. In fact, the graphical image need not be unique for each type of icon. However, it is strongly recommended as a matter of

programming style that each different type of icon have a unique graphical image associated with it. In fact, you may want to have several unique images associated with an icon type. For example, you can have several different images associated with the WBTOOL type of icon info file.

Most people will not access the info file directly. The icon manipulation library does all the work needed to read and write info files. Three routines are especially helpful: **GetDiskObject()**, **PutDiskObject()**, **and FreeDiskObject()**. The calling sequence of each of these is given in the icon library reference pages in the function appendix.

The DiskObject Structure

The **DiskObject** structure is at the beginning of all info files, and is used in **GetDiskObject()**, **PutDiskObject()**, and **FreeDiskObject()**. The structure is defined in *include/workbench/workbench.h* and contains the following elements:

do_Magic
A magic number that the icon library looks for to make sure that the file it is reading really contains an icon. It should be the manifest constant WB_DISKMAGIC. **PutDiskObject()** will put this value in the structure, and **GetDiskObject** will not believe that a file is really an icon unless this value is correct.

do_Version
This provides a way to enhance the info file in an upward-compatible way. It should be WB_DISKVERSION. The icon library will set this value for you, and will not believe weird values.

do_Gadget
This contains all the imagery for the icon. See the "Gadget Structure" section for more details.

do_Type
The type of the icon (WBTOOL, WBPROJECT, and so on).

do_DefaultTool
Default tools are used for projects and disks. For projects the default tool is the program invoked when the project is activated. This tool may either be absolute (DISK:file), relative to the root of this disk (:file), or relative to the project (file). If the icon is of type WBDISK, the default tool is the diskcopy program that will be used when this disk is the source of a copy.

Note that if the tool is run via the default tool mechanism (for example, a project was activated, not a tool) then all the information in the project's info file is used, and the tool's info file is ignored. This is especially important for variables like

StackSize and **ToolWindow**.

do_ToolTypes
ToolTypes is an array of free-format strings. Workbench does not enforce any rules on these strings, but they are useful for passing environment information. See the "ToolTypes" section for more information.

do_CurrentX, do_CurrentY
Drawers have a virtual coordinate system. The user can scroll around in this system using the scroll gadgets on the "drawers" window. Each icon in the drawer has a position in the coordinate system. **CurrentX** and **CurrentY** are the icon's current position in the drawer.

do_DrawerData
If the icon is capable of being opened as a drawer (WBDISK, WBDRAWER, WBGARBAGE) then it needs a **DrawerData** structure to go with it. This structure contains an Intuition **NewWindow** structure[1]; Workbench uses this to hold the current window position and size of the window so it will reopen in the same place. The **CurrentX** and **CurrentY** of the origin of the window is also stored.

do_ToolWindow
By default, Workbench will start a program without a window. If **ToolWindow** is set, this file will be opened and made the standard input and output of the program. This window will also be put into the process's **pr_WindowPtr** variable and will be used for all system requesters. Note that this work is actually done in the language-dependent startup script; if you are coding in assembly or an unsupported language, you will have to do the work yourself. The only two files that it makes sense to open are "CON:" or "RAW:". See the AmigaDOS manual for the full syntax that these devices accept.

do_StackSize
This is the size of the stack used for running the tool. If this is null, then Workbench will use a reasonable default stack size (currently 4K bytes).

The Gadget Structure

Workbench uses an Intuition **Gadget** structure, defined in *include/intuition/intuition.h* or *include/intuition/intuition.i* for the assembly language version, to hold the icon's image. Workbench restricts some of the values of the gadget. Any unused field should be set to 0. For clarity in presentation, you can use the assembly language version of these structures,

[1] See the manual called *Intuition: The Amiga User Interface* for more information about windows.

NOTE: The C version has the leading "gg_" stripped off. (Workbench structure members have the same name in all languages supported by Amiga). The Intuition gadget structure members that Workbench pays attention to are

gg_Width
This is the width (in pixels) of the active icon's active region. Any mouse button press within this range will be interpreted as having selected this icon.

gg_Height
The same as **Width**, only in the vertical direction.

gg_Flags
Currently the gadget *must* be of type GADGIMAGE. There are three highlight modes supported: GADGHCOMP, GADGHIMAGE, and GADGBACKFILL. GADGHCOMP complements the image specified (as opposed to Intuition, which complements the select box). GADGHIMAGE uses an alternate selection image. GADGBACKFILL is similar to GADGHCOMP, but ensures that there is no "orange ring" around the selected image. It does this by first complementing the image, and then flooding all orange pixels that are on the border of the image to blue. (In case you do not use the default colors, orange is color 3 and blue is color 0.) All other flag bits should be 0.

gg_Activation
The activation should have only RELVERIFY and GADGIMMEDIATE set.

gg_Type
The gadget type should be BOOLGADGET.

gg_GadgetRender
Set this to an appropriate **Image** structure.

gg_SelectRender
Set this if and only if the highlight mode is GADGHIMAGE.

The **Image** structure is typically the same size as the gadget, except that **ig_Height** is often one pixel less than the gadget height. This allows a blank line between the icon image and the icon name. The image depth *must* be 2; **ig_PlanePick** *must* be 3; and **ig_PlaneOnOff** should be 0. The **ig_NextImage** field should be null.

Icons with No Position

Picking a position for a newly created icon can be tricky. NO_ICON_POSITION is a magic value for **do_CurrentX** and **do_CurrentY** that instructs Workbench to pick a reasonable place for the icon. Workbench will place the icon in an unused region of the drawer. If there is no space in the drawers window, the icon will be placed just to the right of the visible region.

2.4. WORKBENCH ENVIRONMENT

When a user activates a tool or project, Workbench runs a program. This program is a separate process, and runs asynchronously to Workbench. This allows the user to take advantage of the multiprocessing features of the Amiga.

The environment for a tool under the Workbench is quite different from the environment when a tool is run from the CLI. The CLI does not create a new process for a program; it jumps to the program's code and the program shares the process with the CLI. This means that the program has access to all the CLI's environment, but the program must be very careful to restore all the correct defaults before returning. Workbench starts a tool from scratch and explicitly passes the environment to the tool.

One of the things that a workbench program must set up is stdin and stdout. By default workbench program does not have a window for its output to go to. Therefore stdin and stdout do not point to legal file handles. If one attempts to **printf()** one will destroy the system.

Startup Message

Right after the tool is started, Workbench sends the tool a message, which is posted to the message port in the tool's process. This message contains the environment and the arguments for the tool.

Each icon that is selected in the Workbench is passed to the tool. The first argument is the tool itself. If the tool was derived from a default tool, then this is passed in addition to the project. All other arguments are passed in the order in which the user selected them; the first icon selected will be first.

The tool may do what it wishes with the startup message; however, it must deallocate the message sooner or later. If the message is replied to Workbench, then Workbench will take care of all the cleanup. The tool should not do this until it finishes executing because part of the cleanup is freeing the tool's data space.

The startup message, whose structure is outlined in *include/workbench/startup.h*, has the following structure elements.

sm_Message
A standard Exec message. The reply port is set to the Workbench.

sm_Process
The process descriptor for the tool (as returned by **CreateProcess()**).

sm_Segment
The loaded code for the tool (returned by **LoadSeg()**).

sm_NumArgs
The number of arguments in **sm_ArgList**

sm_ToolWindow
This is the same string as the **DiskObject's do_ToolWindow**. It is passed here so the tool's startup code can open a window for the tool. If it is null, no default window is opened.

sm_ArgList
This is the argument list itself.

Each argument has two parts to it. The **wa_Name** element is the name of the argument. If this is not a default tool or a drawer-like object, this will be the same as the string displayed under the icon. A default tool will have the text of the **wo_DefaultTool** pointer; a drawer will have a null name passed. The **wa_Lock** is always a lock on a directory, or is NULL (if that object type does not support locks).

The following code fragment will work for all arguments (assuming that open will work on them at all).

```
LockArg( arg )
struct WBArg *arg;
int openmode;
{
    LONG olddir;
    LONG lock;

    /* see if this type can be locked */
    if( arg->wa_Lock == NULL ) {
        /* cannot lock it -- it must be a device (for example, DF0:) */
        return( NULL );
    }

    /* change directory to where the argument is */
    olddir = CurrentDir( arg->wa_Lock );

    /* open the argument up */
    lock = Lock( arg->wa_Name, SHARED_LOCK );
    if( lock == NULL ) {
        /* who knows: maybe the user canceled a disk insertion
         * request.  The real reason can be gotten  by IoErr()
         */
        return( NULL );
    }

    /* set the directory back */
    CurrentDir( olddir );

    return( lock );
}
```

For more routines to manipulate Workbench arguments, see the function appendix.

The Standard Startup Code

The standard startup code handles the worst of the detail work of interfacing with the system. The C startup code (**startup.obj**) waits for the startup message, opens the tool window (if one has been requested), sets up **SysBase** and **DOSBase**, and passes the startup message on to **main()**. When **main()** returns (or **exit()** is called) it replies the message back to Workbench.

The **main()** procedure is called with two parameters: **argv** and **argc**. If **argc** is not NULL, you have been called from the CLI. If **argc** is NULL, you have been called from Workbench. The global variable **WBenchMsg** points to the Workbench startup message.

NOTE: A word of warning for those of you who don't use the standard startup sequence: you *must* turn off task switching (with **Forbid()**) before replying the message to Workbench. This will prevent Workbench from unloading your code before you can tell the DOS that you want to exit. See the C startup code in the "Example Programs" section.

2.5. TOOLTYPES

This section shows how the **ToolTypes** array should be formated, and describes the standard entries in the **ToolTypes** array. In brief, **ToolTypes** is an array of strings. These strings can be used to encode information about the icon that will be available to all who wish to use it. The formats are user-definable and user-extensible.

Workbench does not enforce very much about the **ToolTypes** array, but some conventions are strongly encouraged. A string may be up to 32K bytes large, but you should not make it over a line long. The alphabet is 8-bit ANSI (for example, normal ASCII with foreign language extensions). To see what it looks like, try typing with the <alt> key held down. Avoid special or non-printing characters. The case of the characters is significant. The general format is

<name>=<value>[|<value>]*

where <name> is the field name and <value> is the text to associate with that name. If the ID has multiple values, the values may separated by a vertical bar. Currently, the value should be the name of the application that understands this file. For example, a basic program might be

FILETYPE=ABasiC.program|text

This notifies the world that this file is acceptable to a program that is either expecting any arbitrary type of text (for example, an editor) or to someone who only understands a basic program.

There are two routines provided to help you deal with the tool type array. **FindToolType()** returns the value of a tool type element. Using the above example, if you are looking for FILETYPE, the string "ABasic.program|text" will be returned.

MatchToolValue() returns nonzero if the specified string is in the reference value string. This routine knows how to parse vertical bars. For example, using the reference value string of "ABasic.program|text", **MatchToolValue()** will return TRUE for "text" and "ABasic.program", and FALSE for everything else.

2.6. EXAMPLE PROGRAMS

Some example programs, including a startup sequence, are shown below in the following sections.

FriendlyTool

This program tells the application if it can understand a particular object.

```
/*
*
* INPUTS
*      diskobj -- a workbench DiskObject (a returned by GetDiskObject)
*      id -- the application identifier
*
* OUTPUTS
*      nonzero if it understands this object's type
*
*/

#include "exec/types.h"
#include "workbench/workbench.h"
#include "workbench/icon.h"

LONG IconBase;

FriendlyTool( diskobj, id )
struct DiskObject *diskobj;
char *id;
{
   char **toolarray;
   char *value;

   /* default return value is failure */
   int isfriendly = 0;

   /* this assumes that you have not already opened the icon library
    * else where in your program... You undoubtedly have, because
    * you managed to get a diskobject structure...
    */
   IconBase = OpenLibrary( ICONNAME, 1 );
   if( IconBase == NULL ) {
      /* couldn't find the library??? */
```

```
        return( 0 );
    }

    /* extract the tool type value array */
    toolarray = diskobj->do_ToolType;


    /* find the FILETYPE entry */
    value = FindToolType( toolarray, "FILETYPE" );
    if( value ) {
        /* info file did define the FILETYPE entry */

        isfriendly = MatchToolValue( value, id );
    }

    Close( IconBase );

    /* protect ourselves from inadvertent use */
    IconBase = -1;

    return( isfriendly );
}
```

ReadInfoFile

This program reads in an icon's info file from a Workbench argument structure.

```
/*
*
*     don't forget to FreeDiskObject() the object when you are done
*     with it...
*
* INPUTS
*     wbarg -- a workbench argument structure
*
* OUTPUTS
*     a disk object structure if successful, else null.
*
*/

#include "exec/types.h"
#include "exec/memory.h"
#include "workbench/workbench.h"
#include "workbench/icon.h"
```

```
LONG IconBase;

struct DiskObject *
ReadInfoFile( wbarg )
struct WBArg wbarg;
{
    struct DiskObject *diskobj = NULL;      /* assume failure */
    LONG olddir;

    /* this assumes that you have not already opened the icon library
     * else where in your program...
     */
    IconBase = OpenLibrary( ICONNAME, 1 );
    if( IconBase == NULL ) {
        /* couldn't find the library??? */
        return( NULL );
    }

    /* check for those things that do not have info files */
    if( wbarg->wa_Lock == NULL ) {
        /* must be a device */
        goto end;
    }

    olddir = CurrentDir( wbarg->wa_Lock );

    if( wbarg->wa_Name[0] != ' ' ) {
        /* this is a tool or a project */
        diskobj = GetDiskObject( wbarg->wa_Name );
    } else {
        /* this is a drawer-type object.  The lock points to the
         * object itself.  we must go up a level to get the info
         * file.
         */

        diskobj = getDrawerObject( wbarg );
    }

    CurrentDir( olddir );

end:
    Close( IconBase );

    /* protect ourselves from inadvertent use */
    IconBase = -1;

    return( diskobj );
}

/*
```

```
* this routine is split off because getting a drawer object's
* info file is something most applications won't want to do,
* because they won't be able to do anything useful with a
* drawer.  These applications will probably just give an
* error at this point.
*/
getDrawerObject( wbarg )
struct WBArg wbarg;
{
    struct FileInfoBlock *fib;
    LONG oldlock, parentlock;
    struct DiskObject *diskobj = NULL;      /* assume failure */

    fib = (struct FileInfoBlock *fib)
        AllocMem( sizeof( struct FileInfoBlock ), MEMF_CLEAR );

    /* check to see if there was enough memory */
    if( fib == NULL ) return( NULL );

    /* use examine to get the object's name */
    if( ! Examine( wbarg->wa_Lock, fib ) ) goto err;

    parentlock = ParentDir( wbarg->wa_Lock );
    if( parentlock ) {

        /* this is a normal drawer -- it has a parent directory */
        oldlock = CurrentDir( parentlock );
        diskobj = GetDiskObject( fib->fib_FileName );
        CurrentDir( oldlock );
        UnLock( parentlock );

    } else {

        /* ParentDir failed.  Either something is seriously wrong,
         * or we were fed the root of a volume.
         */
        if( IoErr() == NULL ) {
            /* this is the root */
            diskobj = GetDiskObject( "Disk" );
        }
    }

err:
    FreeMem( fib, sizeof( struct FileInfoBlock ) );
    return( diskobj );
}
```

Startup Program

```
*****************************************************************************
*
*     C Program Startup/Exit (Combo Version: CLI and WorkBench)
*
*****************************************************************************

******* Included Files ************************************************

        INCLUDE "exec/types.i"
        INCLUDE "exec/alerts.i"
        INCLUDE "exec/nodes.i"
        INCLUDE "exec/lists.i"
        INCLUDE "exec/ports.i"
        INCLUDE "exec/libraries.i"
        INCLUDE "exec/tasks.i"
        INCLUDE "libraries/dos.i"
        INCLUDE "libraries/dosextens.i"
        INCLUDE "workbench/startup.i"

******* Imported ******************************************************

xlib    macro
        xref    _LVO1
        endm

        xref    _AbsExecBase
        xref    _Input
        xref    _Output

        xref    _main                   ; C code entry point

        xlib    Alert
        xlib    FindTask
        xlib    Forbid
        xlib    GetMsg
        xlib    OpenLibrary
        xlib    CloseLibrary
        xlib    ReplyMsg
        xlib    Wait
        xlib    WaitPort

        xlib    CurrentDir
        xlib    Open
```

```
******* Exported ********************************************************

        xdef  _SysBase
        xdef  _DOSBase

        xdef  _errno
        xdef  _stdin
        xdef  _stdout
        xdef  _stderr

        xdef  _exit             ; standard C exit function

callsys     macro
        CALLLIB  _LVO1
        endm

*****************************************************************************
*
*       Standard Program Entry Point
*
*****************************************************************************
*
*       main (argc, argv)
*         int  argc;
*         char *argv[];
*
*****************************************************************************

startup:                        ; reference for Wack users
            move.l      sp,initialSP        ; initial task stack pointer
            move.l      d0,dosCmdLen
            move.l      a0,dosCmdBuf
            clr.l   returnMsg

        ;------ get Exec's library base pointer:
            move.l      _AbsExecBase,a6
            move.l      a6,_SysBase

        ;------ get the address of our task
            suba.l      a1,a1
            callsys     FindTask
            move.l      d0,a4

        ;------ are we running as a son of Workbench?
            tst.l   pr_CLI(A4)
            beq     fromWorkbench

 ;=========================================================================
 ;          CLI Startup Code
 ;=========================================================================
```

```
fromCLI:
        ;------ attempt to open DOS library:
                bsr   openDOS

        ;------ find command name:
                move.l      pr_CLI(a4),a0
                add.l   a0,a0           ; bcpl pointer conversion
                add.l   a0,a0
                move.l      cli_CommandName(a0),a0
                add.l   a0,a0           ; bcpl pointer conversion
                add.l   a0,a0

        ;------ create buffer and array:
*               link    a6,#-(100+16*4+2*4)
                movem.l d2/a2/a3,-(sp)
                lea     argvBuffer,a2
                lea     argvArray,a3
*               move.l      a3,16(sp)   ; save
                moveq.l     #1,d2               ; param counter

        ;------ fetch command name:
                moveq.l     #0,d0
                move.b      (a0)+,d0    ; size of command name
                move.l      a2,(a3)+    ; ptr to command name
                bra.s 1$
2$:             move.b      (a0)+,(a2)+
1$:             dbf   d0,2$
                clr.b (a2)+

        ;------ collect parameters:
                move.l      dosCmdLen,d0
                move.l      dosCmdBuf,a0

        ;------ skip control characters and space:
3$:             move.b      (a0)+,d1
                subq.l      #1,d0
                ble.s parmExit
                cmp.b       #' ',d1
                ble.s 3$

        ;------ copy parameter:
                addq.l      #1,d2
                move.l      a2,(a3)+
                bra.s 5$
4$:             move.b      (a0)+,d1
                subq.l      #1,d0
                cmp.b       #' ',d1
                ble.s 6$
5$:             move.b      d1,(a2)+
                bra.s 4$
```

```
6$:
            clr.b  (a2)+
            bra.s  3$
parmExit:   clr.b  (a2)+
            clr.l  (a3)+

            move.l      d2,d0
            movem.l     (sp)+,d2/a2/a3
            pea    argvArray
            move.l      d0,-(sp)


*
* The above code relies on the end of line containing a control
* character of any type, i.e. a valid character must not be the
* last.  This fact is ensured by DOS.
*


        ;------ get standard input handle:
            jsr    _Input
            move.l      d0,_stdin

        ;------ get standard output handle:
            jsr    _Output
            move.l      d0,_stdout
            move.l      d0,_stderr

        ;------ call C main entry point
            jsr    _main

        ;------ return success code:
            moveq.l     #0,D0
            move.l      initialSP,sp        ; restore stack ptr
            rts

;=============================================================================
;       Workbench Startup Code
;=============================================================================
fromWorkbench:
        ;------ open the DOS library:
            bsr    openDOS

        ;------ we are now set up.  wait for a message from our starter
            bsr    waitmsg

        ;------ save the message so we can return it later
            move.l      d0,returnMsg    NOTE: no GetMsg performed

        ;------ push the message on the stack for wbmain
```

```
        clr.l  -(SP)        indicate: run from Workbench
        move.l      d0,-(SP)

    ;------ get the first argument
        move.l      d0,a2
        move.l      sm_ArgList(a2),d0
        beq.s docons

    ;------ and set the current directory to the same directory
        move.l      _DOSBase,a6
        move.l      d0,a0
        move.l      wa_Lock(a0),d1
        callsys     CurrentDir

docons:
    ;------ get the toolwindow argument
        move.l      sm_ToolWindow(A2),d1
        beq.s domain

    ;------ open up the file
        move.l      #MODE_OLDFILE,d2
        callsys     Open

    ;------ set the C input and output descriptors
        move.l      d0,_stdin
        move.l      d0,_stdout
        move.l      d0,_stderr
        beq.s domain

    ;------ set the console task (so Open( "*", mode ) will work
    ;       waitmsg has left the task pointer in A4 for us
        lsl.l   #2,d0
        move.l      d0,a0
        move.l      fh_Type(a0),pr_ConsoleTask(A4)

domain:
        jsr    _main
        moveq.l     #0,d0           Successful return code
        bra.s exit2


*****************************************************************************
*
*     C Program Exit Function
*
*****************************************************************************
*
* Warning: this function really needs to do more than this.
*
*****************************************************************************
```

```
_exit:
            move.l      4(SP),d0    ; extract return code
exit2:
            move.l  initialSP,SP    ; restore stack pointer
            move.l      d0,-(SP)    ; save return code

      ;------ close DOS library:
            move.l      _AbsExecBase,A6
            move.l      _DOSBase,a1
            callsys     CloseLibrary

      ;------ if we ran from CLI, skip workbench cleanup:
            tst.l  returnMsg
            beq.s exitToDOS

      ;------ return the startup message to our parent
      ;     we forbid so workbench can't UnLoadSeg() us
      ;     before we are done:
            callsys Forbid
            move.l      returnMsg,a1
            callsys     ReplyMsg

      ;------ this rts sends us back to DOS:
exitToDOS:
            move.l      (SP)+,d0
            rts

;---------------------------------------------------------------------
noDOS:
            ALERT   (AG_OpenLib!AO_DOSLib)
            moveq.l     #100,d0
            bra.s       exit2


;---------------------------------------------------------------------
; This routine gets the message that workbench will send to us
; called with task id in A4

waitmsg:
            lea   pr_MsgPort(A4),a0     * our process base
            callsys     WaitPort
            lea   pr_MsgPort(A4),a0     * our process base
            callsys GetMsg
            rts

;---------------------------------------------------------------------
;  Open the DOS library:

openDOS
            lea   DOSName,A1
```

```
            callsys OpenLibrary
            move.l      D0,_DOSBase
            beq   noDOS
            rts


*****************************************************************************

      DATA

*****************************************************************************

VerRev          dc.w 1,0

_SysBase  dc.l  0
_DOSBase dc.l  0

_errno          dc.l  0
_stdin          dc.l  0
_stdout         dc.l  0
_stderr         dc.l  0

initialSP   dc.l  0
returnMsg dc.l  0

dosCmdLen       dc.l  0
dosCmdBuf       dc.l  0

argvArray  ds.l  32
argvBuffer ds.b  256

DOSName         DOSNAME


      END
```

ReadInfoFile

```
/*** echo.c ***************************************************************
*
* Echo out all of our arguments
*
*************************************************************************/

#include "exec/types.h"
```

```
#include "workbench/startup.h"

extern struct WBStartup *WBenchMsg;

main( argc, argv )
int argc;
char **argv;
{
   LONG file;
   BYTE c;

   if( argc ) {

      printcliargs( Output(), argc, argv );

   } else {

      file = Open( "RAW:50/20/440/150/Echo Window", MODE_OLDFILE );
      if( file == NULL ) {
         return;
      }

      printwbargs( file, WBenchMsg );

      /* wait for some input */
      Read( file, &c, 1 );

      Close( file );
   }
}

printwbargs( file, msg )
ULONG file;
struct WBStartup *msg;
{
   struct WBArg *arg;
   int i;

   arg = msg->sm_ArgList;
   for( i = 0; i < msg->sm_NumArgs; i++, arg++ ) {
      printarg( i, arg, file );
   }
}

printarg( i, arg, file )
int i;
struct WBArg *arg;
LONG file;
{
   fprintf( file, "arg %2ld: lock 0x%06lx name <%s>\r\n",
```

```
        i, arg->wa_Lock, arg->wa_Name );
}

printcliargs( file, argc, argv )
LONG file;
int argc;
char **argv;
{
    int i;

    for( i = 1; i < argc; i++ ) {
        if( i != 1 ) fprintf( file, " " );
        fprintf( file, "%s", argv[i] );
    }
    fprintf( file, "\n" );
}
```

Index

Commodore Business Machines, Inc.
1200 Wilson Drive, West Chester, PA 19380

Commodore Business Machines, Limited
3370 Pharmacy Avenue, Agincourt, Ontario, M1W 2K4

327271-04