Libraries

COLLABORATORS

	TITLE : Libraries	

ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		March 28, 2025	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

1 Libraries **1**

1.1 Amiga® RKM Libraries: 1 Introduction to Amiga System Libraries . . . 1
1.2 1 Introduction to Libraries / Programming in the Amiga Environment . . . 1
1.3 1 / Programming in the Amiga Environment / Multitasking . . . 2
1.4 1 / / Multitasking / What the System Does For You . . . 2
1.5 1 / / Multitasking / What the System Doesn't Do For You . . . 2
1.6 1 / Programming in the Amiga Environment / Libraries of Functions . . . 3
1.7 1 / / Libraries of Functions / Opening a Library in C . . . 4
1.8 1 / / Libraries of Functions / Opening a Library in Assembler . . . 5
1.9 1 / / Libraries of Functions / Another Kind of Function Library . . . 5
1.10 1 / / Libraries of Functions / Libraries, Devices and Resources . . . 6
1.11 1 / Programming in the Amiga Environment / Dynamic Memory Architecture . . . 7
1.12 1 / / Dynamic Memory Architecture / Exec: The System Executive . . . 9
1.13 1 / Programming in the Amiga Environment / Operating System Versions . . . 10
1.14 1 / / Operating System Versions / About Release 2 . . . 11
1.15 1 / Programming in the Amiga Environment / The Custom Chips . . . 11
1.16 1 / / The Custom Chips / Custom Chip Revisions . . . 11
1.17 1 / / The Custom Chips / Two Kinds of Memory . . . 12
1.18 1 Introduction to Amiga System Libraries / About the Examples . . . 12
1.19 1 Introduction to Libraries / General Amiga Development Guidelines . . . 13
1.20 1 / General Development Guidelines / 68010/020/030/040 Compatibility . . . 17
1.21 1 / General Development Guidelines / Hardware Programming Guidelines . . . 18
1.22 1 / General Guidelines / Additional Assembler Development Guidelines . . . 19
1.23 1 Introduction to Amiga System Libraries / 1.3 Compatibility Issues . . . 19
1.24 1 / 1.3 Compatibility Issues / Design Decisions . . . 19
1.25 1 / / Design Decisions / Transparent Release 2 Extensions . . . 20
1.26 1 / / Design Decisions / Conditional Code . . . 21
1.27 1 / / Design Decisions / ASL Requesters . . . 22
1.28 1 / / Design / DOS System(), CreateNewProc(), and CON: Enhancements . . . 22
1.29 1 / / Design Decisions / The Display Database . . . 22

1.30 1 / / Design Decisions / ARexx 23
1.31 1 / 1.3 Compatibility Issues / Compatible Libraries 23
1.32 1 / / Compatible Libraries / IFFParse Library 23
1.33 1 / / Compatible Libraries / Single Precision IEEE Math Libraries 24
1.34 1 / / Compatible Libraries / Third Party Compatible Libraries 24
1.35 1 / Introduction / Commodore Applications and Technical Support (CATS) 24
1.36 1 Introduction to Amiga System Libraries / Error Reports 24

Chapter 1

Libraries

1.1 Amiga® RKM Libraries: 1 Introduction to Amiga System Libraries

The Amiga, like other microcomputers, contains a ROM full of routines that make programming the machine easier. The purpose of this book is to show you how to use these routines. Perhaps the best way to learn Amiga programming is by following examples and that is the method used in this book. Before starting though it will be helpful to go over some Amiga fundamentals. This section presents some of the basics that all Amiga programmers need to know.

Programming in the Amiga Environment
About the Examples
General Amiga Development Guidelines
1.3 Compatibility Issues
Commodore Applications and Technical Support (CATS)
Error Reports

1.2 1 Introduction to Libraries / Programming in the Amiga Environment

To program in the Amiga's dynamic environment you need to understand these special features of the Amiga's design:

* Multitasking (without memory protection)

* Shared libraries of functions

* Dynamic memory architecture (no memory map)

* Operating system versions

* Custom chips with DMA access (two kinds of memory)

Multitasking Dynamic Memory Architecture The Custom Chips
Libraries of Functions Operating System Versions

1.3 1 / Programming in the Amiga Environment / Multitasking

The key feature of the Amiga's operating system design is multitasking. Multitasking means many programs, or tasks, reside in memory at the same time sharing system resources with one another. Programs take turns running so it appears that many programs are running simultaneously.

Multitasking is based on the concept that a program spends most of its time waiting for things to happen. A program waits for events like key presses, mouse movement, or disk activity. While a program is waiting, the CPU is idle. The CPU could be used to run a different program during this idle period if there was a convenient method for rapidly switching from one program to another. This is what multitasking does.

What the System Does For You
What the System Doesn't Do For You

1.4 1 / / Multitasking / What the System Does For You

The Amiga uses preemptive multitasking which means that the operating system keeps track of all the tasks in memory and decides which one should run. The system checks hundreds of times per second to see which task should be run based on whether or not it is waiting, and other factors. Since the system handles all the work of task switching, multitasking is transparent to the application. From the application's point of view, it appears to have the machine all to itself.

The Amiga OS also manages the sharing of resources between tasks. This is important because in order for a variety of tasks to run independently in the Amiga's multitasking environment, tasks must be prevented from interfering with one another. Imagine if five tasks were allowed to use the parallel port at the same time. The result would be I/O chaos. To prevent this, the operating system provides an arbitration method (usually a function call) for every system resource. For instance you must call a function, AllocMem(), to get exclusive access to a block of memory.

1.5 1 / / Multitasking / What the System Doesn't Do For You

The Amiga operating system handles most of the housekeeping needed for multitasking, but this does not mean that applications don't have to worry about multitasking at all. The current generation of Amiga systems do not have hardware memory protection, so there is nothing to stop a task from using memory it has not legally acquired. An errant task can easily corrupt some other task by accidentally overwriting its instructions or data. Amiga programmers need to be extra careful with memory; one bad memory pointer can cause the machine to crash (debugging utilities such as MungWall and Enforcer will prevent this).

In fact, Amiga programmers need to be careful with every system resource, not just memory. All system resources from audio channels to the floppy disk drives are shared among tasks. Before using a resource, you must ask the system for access to the resource. This may fail if the resource is

already being used by another task.

Once you have control of a resource, no other task can use it, so give it up as soon as you are finished. When your program exits, you must give everything back whether it's memory, access to a file, or an I/O port. You are responsible for this, the system will not do it for you automatically.

```
 ____________________________________________________________________________
|                                                                            |
| What Every Amiga Programmer Should Know:                                   |
| ----------------------------------------                                   |
| The Amiga is a multitasking computer.  Keep in mind that other             |
| tasks are running at the same time as your application.  Always ask        |
| the system for control of any resource you need; some other task may       |
| already be using it.  Give it back as soon as you are done; another        |
| task may want to use it.  This applies to just about every computing       |
| activity your application can perform.                                     |
|____________________________________________________________________________|
```

1.6 1 / Programming in the Amiga Environment / Libraries of Functions

Most of the routines that make up the Amiga's operating system are organized into groups called libraries. In order to call a function on the Amiga you must first open the library that contains the function. For example, if you want to call the Read() function to read data from disk you must first open the DOS library.

The system's master library, called Exec, is always open. Exec keeps track of all the other libraries and is in charge of opening and closing them. One Exec function, OpenLibrary(), is used to open all the other libraries.

Almost any program you write for the Amiga will have to call the OpenLibrary() function. Usage is as follows:

```
    struct Library *LibBase;      /* Global: declare this above main() */

    main()
    {
    LibBase = OpenLibrary("library.name",version);

    if(!LibBase) { /* Library did not open, so exit        */ }
    else         { /* Library opened, so use its functions */ }
    }
```

LibBase
    This is a pointer to the library structure in memory, often referred to as the library base. The library base must be global because the system uses it to handle the library's function calls. The name of this pointer is established by the system (you cannot use any name you want). Refer to the list below for the appropriate name.

library.name
    This is a C string that describes the name of the library you wish to open. The list of Amiga library names is given below.

version
    This should be set to the earliest acceptable library version. A value of 0 matches any version. A value of 33 means you require at least version 33, or a later version of the library. If the library version in the system is older than the one you specify, OpenLibrary() will fail (return 0).

The following table shows all the function libraries that are currently part of the Amiga system software. Column one shows the name string to use with OpenLibrary(); column two shows the name of the global variable you should use to hold the pointer to the library; column three shows the oldest version of the library still in use.

Table 1-1: Parameters to Use With OpenLibrary()

Library Name (library.name)*	Library Base Name (LibBase)	Oldest Version In Use (version)
asl.library	AslBase	36
commodities.library	CxBase	36
diskfont.library	DiskfontBase	33
dos.library	DOSBase	33
exec.library	SysBase	33
expansion.library	ExpansionBase	33
gadtools.library	GadToolsBase	36
graphics.library	GfxBase	33
icon.library	IconBase	33
iffparse.library	IFFParseBase	36
intuition.library	IntuitionBase	33
keymap.library	KeymapBase	33
layers.library	LayersBase	33
mathffp.library	MathBase	33
mathtrans.library	MathTransBase	33
mathieeedoubbas.library	MathIeeeDoubBasBase	33
mathieeedoubtrans.library	MathIeeeDoubTransBase	33
mathieeesingbas.library	MathIeeeSingBasBase	33
mathieeesingtrans.library	MathIeeeSingTransBase	33
rexxsyslib.library	RexxSysBase	36
translator.library	TranslatorBase	33
utility.library	UtilityBase	36
wb.library	WorkbenchBase	33

  * Other libraries may exist that are not supplied by Commodore since it is a feature of the operating system to allow such libraries.

Opening a Library in C
Opening a Library in Assembler
Another Kind of Function Library
Libraries, Devices and Resources

1.7 1 / / Libraries of Functions / Opening a Library in C

Call OpenLibrary() to open an Amiga function library. OpenLibrary() returns the address of the library structure (or library base) which you must assign to a specific global system variable as specified in the table above (case is important).

If the library cannot open for some reason, the OpenLibrary() function returns zero. Here's a brief example showing how it's used in C.

easy.c

1.8 1 / / Libraries of Functions / Opening a Library in Assembler

Here's the same example written in 68000 assembler. The principles are the same as with C: you must always open a library before using any of its functions. However, in assembler, library bases are treated a little differently than in C. In C, you assign the library base you get from OpenLibrary() to a global variable and forget about it (the system handles the rest). In assembler, the library base must always be in register A6 whenever calling any of the functions in the library.

You get the library base for any library except Exec, by calling OpenLibrary(). For Exec, you get the library base from the longword in memory location 4 ($0000 0004). Exec is opened automatically by the system at boot time, and its library base is stored there.

easy.asm

The Amiga library functions are set up to accept parameters in certain 68000 registers and always return results in data register D0. This allows programs and functions written in assembler to communicate quickly. It also eliminates the dependence on the stack frame conventions of any particular language.

Amiga library functions use registers D0, D1, A0 and A1 for work space and use register A6 to hold the library base. Do not expect these registers to be the same after calling a function. All routines return a full 32 bit longword unless noted otherwise.

1.9 1 / / Libraries of Functions / Another Kind of Function Library

The Amiga has two kinds of libraries: run-time libraries and link libraries. All the libraries discussed so far are run-time libraries. Run-time libraries make up most of the Amiga's operating system and are the main topic of this book.

There is another type of library known as a link library. Even though a link library is a collection of functions just like a run-time library, there are some major differences in the two types.

Run-time libraries
A run-time, or shared library is a group of functions managed by Exec

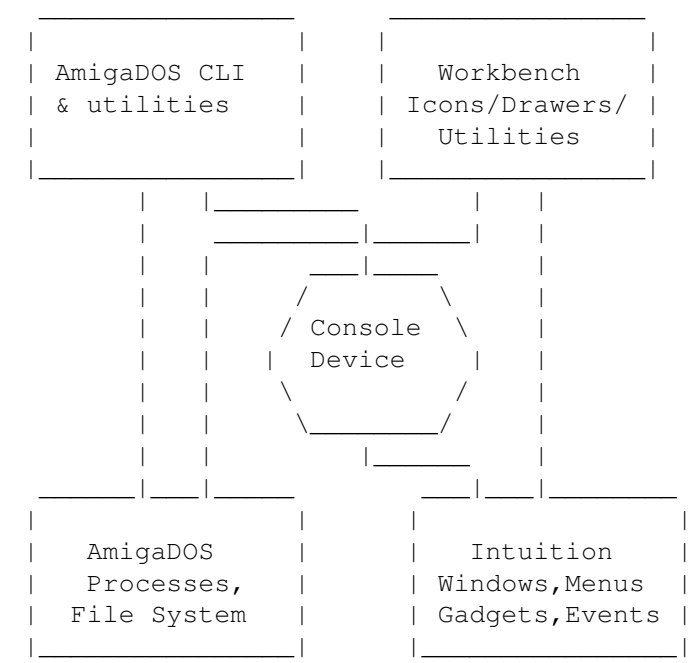
that resides either in ROM or on disk (in the LIBS: directory). A run-time library must be opened before it can be used (as explained above). The functions in a run-time library are accessed dynamically at run-time and can be used by many programs at once even though only one copy of the library is in memory. A disk based run-time library is loaded into memory only if requested by a program and can be automatically flushed from memory when no longer needed.

Link libraries
: A link library is a group of functions on disk that are managed by the compiler at link time. Link libraries do not have to be opened before they are used, instead you must link your code with the library when you compile a program. The functions in a link library are actually copied into every program that uses them. For instance the exit() function used in the C program listed above is not part of any of the libraries that make up the Amiga OS. It comes from the link library supplied with the compiler (lc.lib for SAS/Lattice C or c.lib for Manx Aztec C). The code that performs the exit() function is copied into the program when it is compiled.

1.10 1 / / Libraries of Functions / Libraries, Devices and Resources

Most of the Amiga's OS routines are organized into groups of shared run-time libraries. The Amiga also has specialized function groups called devices and resources that programmers use to perform basic I/O operations or access low-level hardware.

Devices and resources are similar in concept to a shared run-time library. They are managed by Exec and must be opened before they can be used. Their functions are separate from the programs that use them and are accessed dynamically at run time. Multiple programs can access the device or resource even though only one copy exists in memory (a few resources can only be used by one program at a time.)



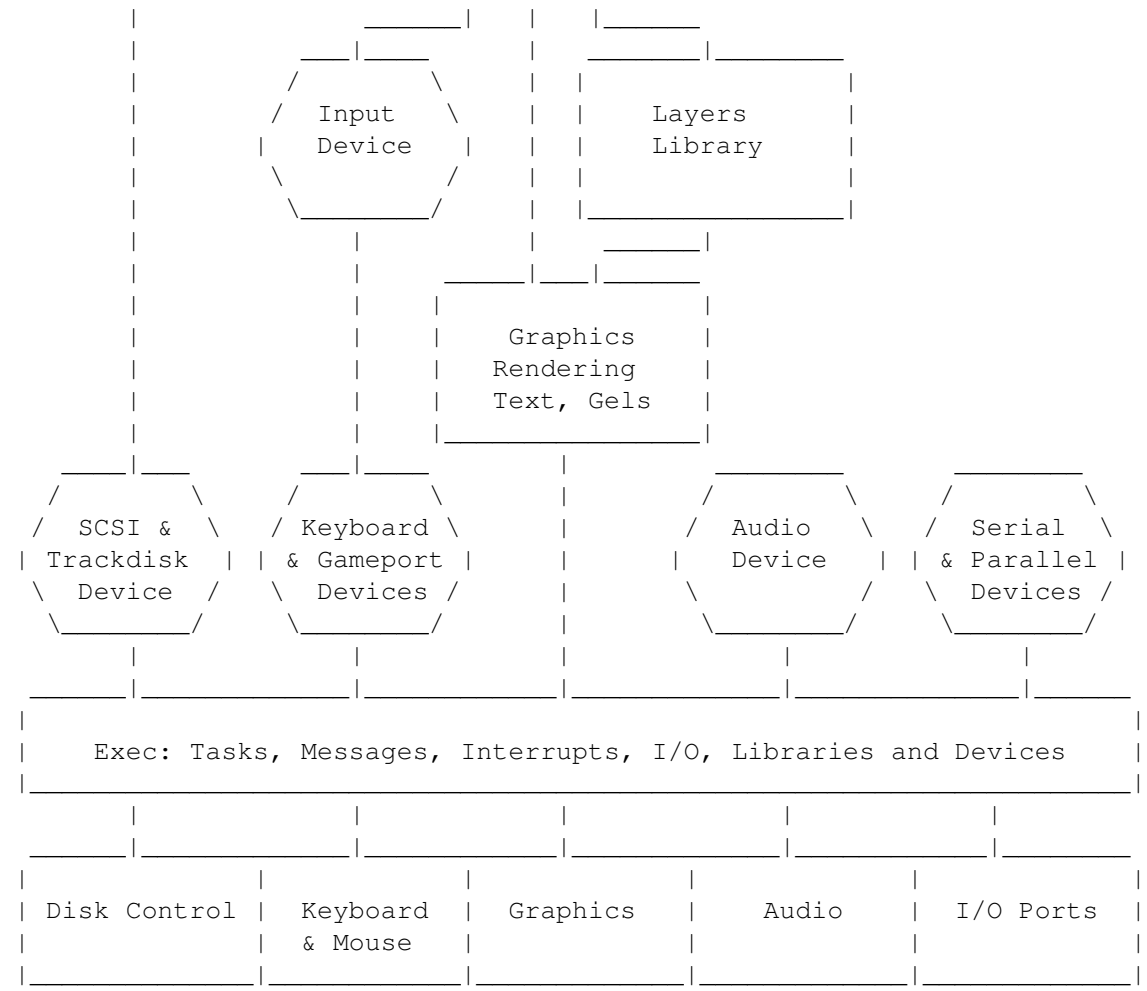


Figure 1-1: Amiga System Software Hierarchy

Devices and resources are managed by Exec just as libraries are. For more information on devices and resources, see the chapter on Exec Device I/O later in this book or refer to the Amiga ROM Kernel Reference Manual: Devices for detailed descriptions of each device.

```
| What Every Amiga Programmer Should Know:                          |
| ----------------------------------------                          |
| The functions in the Amiga OS are accessed through shared run-time |
| libraries.  Libraries must be opened before their functions may be |
| used.  The system's master library, Exec, is always open.  The Exec |
| function OpenLibrary() is used to open all other libraries.       |
```

1.11 1 / Programming in the Amiga Environment / Dynamic Memory Architecture

Unlike some microcomputer operating systems, the Amiga OS relies on absolute memory addresses as little as possible. Instead the Amiga OS uses a technique (sometimes referred to as soft machine architecture)

which allows system routines and data structures to be positioned anywhere in memory.

Amiga run-time libraries may be positioned anywhere in memory because they are always accessed through a jump table. Each library whether in ROM or loaded from disk has an associated Library structure and jump table in RAM.

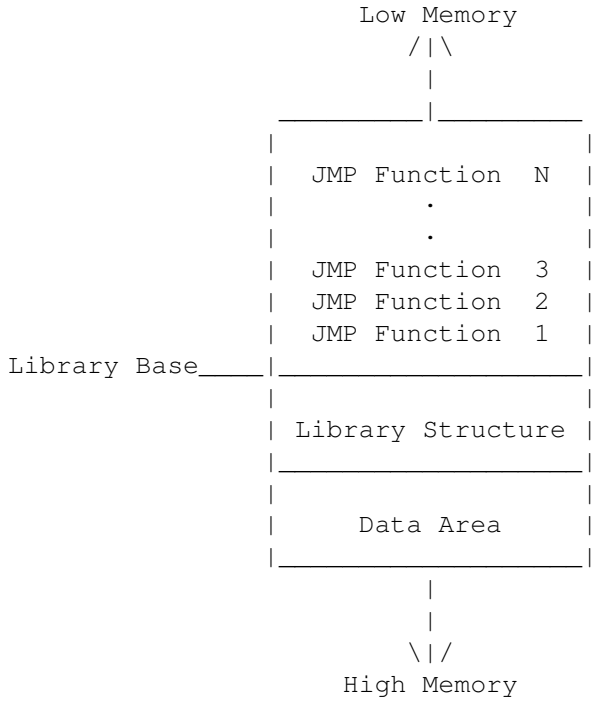


Figure 1-2: Amiga Library Structure and Jump Table

The system knows where the jump table starts in RAM because when a library is opened for the first time, Exec creates the library structure and keeps track of its location. The order of the entries in the library's jump table is always preserved between versions of the OS but the functions they point to can be anywhere in memory. Hence, system routines in ROM may be moved from one version of the OS to another. Given the location of the jump table and the appropriate offset into the table, any function can always be found.

Not only are system routines relocatable but system data structures are too. In the Amiga's multitasking environment, multiple applications run at the same time and each may have its own screen, memory, open files, and even its own subtasks. Since any number of application tasks are run and stopped at the user's option, system data structures have to be set up as needed. They cannot be set up ahead of time at a fixed memory location because there is no way to tell how many and what type will be needed.

The Amiga system software manages this confusion by using linked lists of information about items such as libraries, tasks, screens, files and available memory. A linked list is a chain of data items with each data item containing a pointer to the next item in the chain. Given a pointer to the first item in a linked list, pointers to all the other items in the

chain can be found.

Exec: The System Executive

1.12 1 / / Dynamic Memory Architecture / Exec: The System Executive

On the Amiga, the module that keeps track of linked lists is Exec, the system executive. Exec is the heart of the Amiga operating system since it also is in charge of multitasking, granting access to system resources (like memory) and managing the Amiga library system.

As previously discussed, memory location 4 ($0000 0004), also known as SysBase, contains a pointer to the Exec library structure. This is the only absolutely defined location in the Amiga operating system. A program need only know where to find the Exec library to find, use and manipulate all other system code and data.

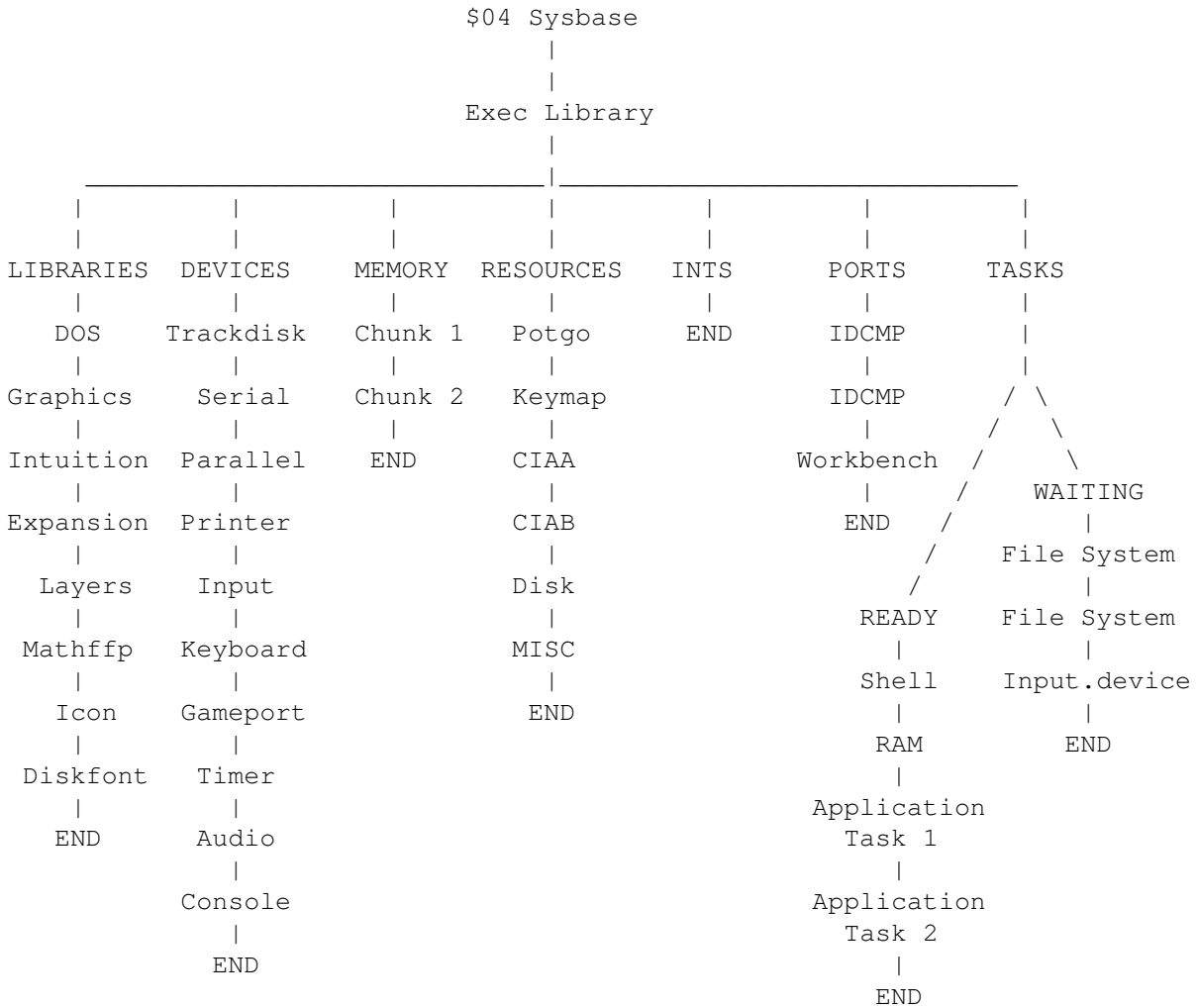


Figure 1-3: Exec and the Organization of the Amiga OS

The diagram above shows how the entire Amiga operating system is built as a tree starting at SysBase. Exec keeps linked lists of all the system libraries, devices, memory, tasks and other data structures. Each of these in turn can have its own variables and linked lists of data structures built onto it. In this way, the flexibility of the OS is preserved so that upgrades can be made without jeopardizing compatibility.

```
 ____________________________________________________________________________
|                                                                            |
| What Every Amiga Programmer Should Know:                                   |
| ----------------------------------------                                   |
| The Amiga has a dynamic memory map.  There are no fixed locations for      |
| operating system variables and routines.  Do not call ROM routines or      |
| access system data structures directly.  Instead use the indirect          |
| access methods provided by the system.                                     |
|____________________________________________________________________________|
```

1.13 1 / Programming in the Amiga Environment / Operating System Versions

The Amiga operating system has undergone several major revisions summarized in the table below. The latest revision is Release 2 (corresponds to library versions 36 and above).

System library version number	Kickstart release
0	Any version
30	Kickstart V1.0 (obsolete)
31	Kickstart V1.1 (NTSC only - obsolete)
32	Kickstart V1.1 (PAL only - obsolete)
33	Kickstart V1.2 (the oldest revision still in use)
34	Kickstart V1.3 (adds autoboot to V33)
35	Special Kickstart version to support A2024 high-resolution monitor
36	Kickstart V2.0 (old version of Release 2)
37	Kickstart V2.04 (current version of Release 2)

The examples listed throughout this book assume you are using Release 2.

Many of the libraries and functions documented in this manual are available in all versions of the Amiga operating system. Others are completely new and cannot be used unless you have successfully opened the appropriate version of the library.

To find out which functions are new with Release 2 refer to the ROM Kernel Reference Manual: Includes and Autodocs. The functions which are new are marked with (V36) or (V37) in the NAME line of the function Autodoc. These new functions require you to use a matching version number (36, 37, or higher) when opening the library.

Exit gracefully and informatively if the required library version is not available.

About Release 2

1.14 1 / / Operating System Versions / About Release 2

Release 2 first appeared on the Amiga 3000. This initial version corresponds to Kickstart V2.00, system library version number V36. Release 2 was subsequently revised and this older version is now considered obsolete.

Programs written for Release 2 should use only the later version corresponding to Kickstart V2.04, system library version number V37. If your system is using the earlier version of Release 2, you should upgrade your system. (Upgrade kits may be obtained from an authorized Commodore service center.)

```
 _____________________________________________________________________
|                                                                     |
| What Every Amiga Programmer Should Know:                            |
| ----------------------------------------                            |
| Some libraries or specific functions are not available in older     |
| versions of the Amiga operating system.  Be sure to ask for the     |
| lowest library version that meets the requirements of your program. |
|_____________________________________________________________________|
```

1.15 1 / Programming in the Amiga Environment / The Custom Chips

The most important feature of the Amiga's hardware design is the set of custom chips that perform specialized tasks independently of the CPU. Each of the custom chips (named Paula, Agnus, and Denise) is dedicated to a particular job:

```
Paula   (8364)            Audio, floppy disk, serial, interrupts
Agnus   (8361/8370/8372)  Copper (video coprocessor), blitter, DMA control
Denise  (8362)            Color registers, color DACs (Digital to Analog
                          Converters) and sprites
```

The custom chips can perform work independently of the CPU because they have DMA, or Direct Memory Access, capability. DMA means the custom chips can access special areas of memory by themselves without any CPU involvement. (On computer systems without DMA, the CPU must do some or all of the memory handling for support chips.) The Amiga's custom chips make multitasking especially effective because they can handle things like rendering graphics and playing sound independently, giving the CPU more time to handle the overhead of task-switching and other important jobs.

Custom Chip Revisions Two Kinds of Memory

1.16 1 / / The Custom Chips / Custom Chip Revisions

The custom chips have been revised as the Amiga platform has evolved and newer models of the Amiga developed. The latest revision of the Amiga custom chips is known as the Enhanced Chip Set, or ECS. Certain features of the Amiga operating system, such as higher resolution screens and special genlock modes, require the ECS version of the custom chips. In this book, features that require ECS are noted in the accompanying text. For more details about the special features of ECS, see Appendix C of the Amiga Hardware Reference Manual.

1.17 1 / / The Custom Chips / Two Kinds of Memory

To keep the Amiga running efficiently, the Amiga has two memory buses and two kinds of memory. Chip memory is memory that both the CPU and custom chips can access. Fast memory is memory that only the CPU (and certain expansion cards) can access. Since Chip memory is shared, CPU access may be slowed down if the custom chips are doing heavy-duty processing. CPU access to Fast memory is never slowed down by contention with the custom chips.

The distinction between Chip memory and Fast memory is very important for Amiga programmers to keep in mind because any data accessed directly by the custom chips such as video display data, audio data or sprite data must be in Chip memory.

```
 ____________________________________________________________________
|                                                                    |
| What Every Amiga Programmer Should Know:                           |
| ----------------------------------------                           |
| The Amiga has two kinds of memory: Chip memory and Fast memory.    |
| Use the right kind.                                                |
|____________________________________________________________________|
```

1.18 1 Introduction to Amiga System Libraries / About the Examples

For the most part, the examples in this book are written in C (there are a few 68000 assembly language examples too).

C examples have been compiled under SAS C, version 5.10a. The compiler options used with each example are noted in the comments preceding the code.

In general, the examples are also compatible with Manx Aztec C 68K, version 5.0d, and other C compilers, however some changes will usually be necessary. Specifically, all the C examples assume that the automatic Ctrl-C feature of the compiler has been disabled. For SAS C (and Lattice C revisions 4.0 and greater) this is handled with:

```
    /* Add this before main() to override the default Ctrl-C handling
     * provided in SAS (Lattice) C.  Ctrl-C event will be ignored */

    int CXBRK ( void )   { return(0); }
```

```
    int chkabort( void ) { return(0); }
```

For Manx Aztec C, replace the above with:

```
    /* Add this near the top */
            #include <functions.h>

    /* Add this before main() */
            extern int Enable_Abort;   /* reference abort enable */

    /* Add this after main(), as the first active line in the program */
            Enable_Abort=0;         /* turn off CTRL-C */
```

Other changes may be required depending on the example and the C compiler you are using. Most of the C examples in this book use the following special option flags of the SAS/C compiler (set the equivalent option of whatever compiler you are using):

```
-b1 = Small data model.
-cf = Check for function prototypes.
  i = Ignore #include statements that are identical to one already given.
  s = Store all literal strings that are identical in the same place.
  t = Enable warnings for structures that are used before they are defined.

 -v = Do not include stack checking code with each function.
 -y = Load register A4 with the data section base address on function
      entry.  The -v and -y flags are are generally only needed for parts
      of the program that are called directly by the system such as
      interrupt servers, subtasks, handlers and callback hook functions.
```

Except where noted, each example was linked with the standard SAS/C startup code c.o, the SAS/C linker library lc.lib and the Commodore linker library amiga.lib. The SAS/C compiler defaults to 32-bit ints. If your development environment uses 16-bit ints you may need to explicitly cast certain arguments as longs (for example 1L << sigbit instead of 1 << sigbit).

The 68000 assembly language examples have been assembled under the Innovatronics CAPE assembler V2.x, the HiSoft Devpac assembler V1.2, and the Lake Forest Logic ADAPT assembler 1.0. No substantial changes should be required to switch between assemblers.

1.19 1 Introduction to Libraries / General Amiga Development Guidelines

In the earlier sections of this chapter, the basic environment of the Amiga operating system was discussed. This section presents specific guidelines that all Amiga programmers must follow. Some of these guidelines are for advanced programmers or apply only to code written in assembly language.

* Check for memory loss. Arrange your Workbench screen so that you have a Shell available and can start your program without rearranging

any windows. In the Shell window type Avail flush several times (the flush option requires the Release 2 version of the Avail command). Note the total amount of free memory. Run your program (do not rearrange any windows other than those created by the program) and then exit. At the Shell, type Avail flush several times again. Compare the total amount of free memory with the earlier figure. They should be the same. Any difference indicates that your application is not freeing some memory it used or is not closing a disk-loaded library, device or font it opened. Note that under Release 2, a small amount of memory loss is normal if your application is the first to use the audio or narrator device.

* Use all of the program debugging and stress tools that are available when writing and testing your code. New debugging tools such as Enforcer, MungWall, and Scratch can help find uninitialized pointers, attempted use of freed memory and misuse of scratch registers or condition codes (even in programs that appear to work perfectly).

* Always make sure you actually get any system resource that you ask for. This applies to memory, windows, screens, file handles, libraries, devices, ports, etc. Where an error value or return is possible, ensure that there is a reasonable failure path. Many poorly written programs will appear to be reliable, until some error condition (such as memory full or a disk problem) causes the program to continue with an invalid or null pointer, or branch to untested error handling code.

* Always clean up after yourself. This applies for both normal program exit and program termination due to error conditions. Anything that was opened must be closed, anything allocated must be deallocated. It is generally correct to do closes and deallocations in reverse order of the opens and allocations. Be sure to check your development language manual and startup code; some items may be closed or deallocated automatically for you, especially in abort conditions. If you write in the C language, make sure your code handles Ctrl-C properly.

* Remember that memory, peripheral configurations, and ROMs differ between models and between individual systems. Do not make assumptions about memory address ranges, storage device names, or the locations of system structures or code. Never call ROM routines directly. Beware of any example code you find that calls routines at addresses in the $F0 0000 - $FF FFFF range. These are ROM routines and they will move with every OS release. The only supported interface to system ROM code is through the library, device, and resource calls.

* Never assume library bases or structures will exist at any particular memory location. The only absolute address in the system is $0000 0004, which contains a pointer to the Exec library base. Do not modify or depend on the format of private system structures. This includes the poking of copper lists, memory lists, and library bases.

* Never assume that programs can access hardware resources directly. Most hardware is controlled by system software that will not respond well to interference from other programs. Shared hardware requires programs to use the proper sharing protocols. Use the defined

interface; it is the best way to ensure that your software will continue to operate on future models of the Amiga.

* Never access shared data structures directly without the proper mutual exclusion (locking). Remember that other tasks may be accessing the same structures.

* The system does not monitor the size of a program's stack. (Your compiler may have an option to do this for you.) Take care that your program does not cause stack overflow and provide extra stack space for the possibility that some functions may use up additional stack space in future versions of the OS.

* Never use a polling loop to test signal bits. If your program waits for external events like menu selection or keystrokes, do not bog down the multitasking system by busy-waiting in a loop. Instead, let your task go to sleep by Wait()ing on its signal bits. For example:

```
    signals = (ULONG)Wait(  (1<<windowPtr->UserPort->mp_SigBit) |
                            (1<<consoleMsgPortPtr->mp_SigBit)  );
```

  This turns the signal bit number for each port into a mask, then combines them as the argument for the Exec library Wait() function. When your task wakes up, handle all of the messages at each port where the mp_SigBit is set. There may be more than one message per port, or no messages at the port. Make sure that you ReplyMsg() to all messages that are not replies themselves. If you have no signal bits to Wait() on, use Delay() or WaitTOF() to provide a measured delay.

* Tasks (and processes) execute in 680x0 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.

* Most system functions require a particular execution environment. All DOS functions and any functions that might call DOS (such as the opening of a disk-resident library, font, or device) can only be executed from a process. A task is not sufficient. Most other ROM kernel functions may be executed from tasks. Only a few may be executed from interrupts.

* Never disable interrupts or multitasking for long periods. If you use Forbid() or Disable(), you should be aware that execution of any system function that performs the Wait() function will temporarily suspend the Forbid() or Disable() state, and allow multitasking and interrupts to occur. Such functions include almost all forms of DOS and device I/O, including common stdio functions like printf().

* Never tie up system resources unless it is absolutely necessary. For example, if your program does not require constant use of the printer, open the printer device only when you need it. This will allow other tasks to use the printer while your program is running. You must provide a reasonable error response if a resource is not available when you need it.

* All data for the custom chips must reside in Chip memory (type

MEMF_CHIP). This includes bitplanes, sound samples, trackdisk buffers, and images for sprites, bobs, pointers, and gadgets. The AllocMem() call takes a flag for specifying the type of memory. A program that specifies the wrong type of memory may appear to run correctly because many Amigas have only Chip memory. (On all models of the Amiga, the first 512K of memory is Chip memory. In later models, Chip memory may occupy up to the first one or two megabytes).

However, once expansion memory has been added to an Amiga (type MEMF_FAST), any memory allocations will be made in the expansion memory area by default. Hence, a program can run correctly on an unexpanded Amiga which has only Chip memory while crashing on an Amiga which has expanded memory. A developer with only Chip memory may fail to notice that memory was incorrectly specified.

Most compilers have options to mark specific data structures or object modules so that they will load into Chip RAM. Some older compilers provide the Atom utility for marking object modules. If this method is unacceptable, use the AllocMem() call to dynamically allocate Chip memory, and copy your data there.

When making allocations that do not require Chip memory, do not explicitly ask for Fast memory. Instead ask for memory type MEMF_PUBLIC or 0L as appropriate. If Fast memory is available, you will get it.

* Never use software delay loops! Under the multitasking operating system, the time spent in a loop can be better used by other tasks. Even ignoring the effect it has on multitasking, timing loops are inaccurate and will wait different amounts of time depending on the specific model of Amiga computer. The timer device provides precision timing for use under the multitasking system and it works the same on all models of the Amiga. The AmigaDOS Delay() function or the graphics library WaitTOF() function provide a simple interface for longer delays. The 8520 I/O chips provide timers for developers who are bypassing the operating system (see the Amiga Hardware Reference Manual for more information).

* Always obey structure conventions!

    · All non-byte fields must be word-aligned. Longwords should be longword-aligned for performance.

    · All address pointers should be 32 bits (not 24 bits). Never use the upper byte for data.

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

    · All reserved or unused fields must be initialized to zero for future compatibility.

    · Data structures to be accessed by the custom chips, public data structures (such as a task control block), and structures which must be longword aligned must not be allocated on a program's stack.

· Dynamic allocation of structures with AllocMem() provides
longword aligned memory of a specified type with optional
initialization to zero, which is useful in the allocation of
structures.

For 68010/68020/68030/68040 Compatibility
Hardware Programming Guidelines
Additional Assembler Development Guidelines

1.20 1 / General Development Guidelines / 68010/020/030/040 Compatibility

Special care must be taken to be compatible with the entire family of
68000 processors:

* Do not use the upper 8 bits of a pointer for storing unrelated
information. The 68020, 68030, and 68040 use all 32 bits for
addressing.

* Do not use signed variables or signed math for addresses.

* Do not use software delay loops, and do not make assumptions about
the order in which asynchronous tasks will finish.

* The stack frame used for exceptions is different on each member of
the 68000 family. The type identification in the frame must be
checked! In addition, the interrupt autovectors may reside in a
different location on processors with a VBR register.

* Do not use the MOVE SR,<dest> instruction! This 68000 instruction
acts differently on other members of the 68000 family. If you want
to get a copy of the processor condition codes, use the Exec library
GetCC() function.

* Do not use the CLR instruction on a hardware register which is
triggered by Write access. The 68020 CLR instruction does a single
Write access. The 68000 CLR instruction does a Read access first,
then a Write access. This can cause a hardware register to be
triggered twice. Use MOVE.x #0, <address> instead.

* Self-modifying code is strongly discouraged. All 68000 family
processors have a pre-fetch feature. This means the CPU loads
instructions ahead of the current program counter. Hence, if your
code modifies or decrypts itself just ahead of the program counter,
the pre-fetched instructions may not match the modified instructions.
The more advanced processors prefetch more words. If self-modifying
code must be used, flushing the cache is the safest way to prevent
troubles.

* The 68020, 68030 and 68040 processors all have instruction caches.
These caches store recently used instructions, but do not monitor
writes. After modifying or directly loading instructions, the cache
must be flushed. See the Exec library CacheClearU() Autodoc for more
details. If your code takes over the machine, flushing the cache
will be trickier. You can account for the current processors, and
hope the same techniques will work in the future:

```
CACRF_ClearI  EQU     $0008     ;Bit for clear instruction cache
;
;Supervisor mode only. Use this only if you have taken over
;the machine.  Read and store the ExecBase processor AttnFlags
;flags at boot time, call this code only if the "68020 or
; better" bit was set.
;
ClearICache:  dc.w    $4E7A,$0002  ;MOVEC CACR,D0
              tst.w   d0           ;movec does not affect CC's
              bmi.s   cic_040      ;A 68040 with enabled cache!
              ori.w   #CACRF_ClearI,d0
              dc.w    $4E7B,$0002  ;MOVEC D0,CACR
              bra.s   cic_exit
cic_040:      dc.w    $f4b8        ;CPUSHA (IC)
cic_exit:
```

1.21 1 / General Development Guidelines / Hardware Programming Guidelines

If you find it necessary to program the hardware directly, then it is your responsibility to write code that will work correctly on the various models and configurations of the Amiga. Be sure to properly request and gain control of the hardware resources you are manipulating, and be especially careful in the following areas:

* Kickstart 2.0 uses the 8520 Complex Interface Adaptor (CIA) chips differently than 1.3 did. To ensure compatibility, you must always ask for CIA access using the cia.resource AddICRVector() and RemICRVector() functions. Do not make assumptions about what the system might be using the CIA chips for. If you write directly to the CIA chip registers, do not expect system services such as the trackdisk device to function. If you are leaving the system up, do not read or write to the CIA Interrupt Control Registers directly; use the cia.resource AbleICR(), and SetICR() functions. Even if you are taking over the machine, do not assume the initial contents of any of the CIA registers or the state of any enabled interrupts.

* All custom chip registers are Read-only or Write-only. Do not read Write-only registers, and do not write to Read-only registers.

* Never write data to, or interpret data from the unused bits or addresses in the custom chip space. To be software-compatible with future chip revisions, all undefined bits must be set to zeros on writes, and must be masked out on reads before interpreting the contents of the register.

* Never write past the current end of custom chip space. Custom chips may be extended or enhanced to provide additional registers, or to use bits that are currently undefined in existing registers.

* Never read, write, or use any currently undefined address ranges or registers. The current and future usage of such areas is reserved by Commodore and is subject to change.

* Never assume that a hardware register will be initialized to any

particular value. Different versions of the OS may leave registers set to different values. Check the Amiga Hardware Reference Manual to ensure that you are setting up all the registers that affect your code.

1.22 1 / General Guidelines / Additional Assembler Development Guidelines

If you are writing in assembly language there are some extra rules to keep in mind in addition to those listed above.

* Never use the TAS instruction on the Amiga. System DMA can conflict with this instruction's special indivisible read-modify-write cycle.

* System functions must be called with register A6 containing the library or device base. Libraries and devices assume A6 is valid at the time of any function call. Even if a particular function does not currently require its base register, you must provide it for compatibility with future system software releases.

* Except as noted, system library functions use registers D0, D1, A0, and A1 as scratch registers and you must consider their former contents to be lost after a system library call. The contents of all other registers will be preserved. System functions that provide a result will return the result in D0.

* Never depend on processor condition codes after a system call. The caller must test the returned value before acting on a condition code. This is usually done with a TST or MOVE instruction.

1.23 1 Introduction to Amiga System Libraries / 1.3 Compatibility Issues

This 3rd edition of the Amiga Technical Reference Series focuses on the Release 2 version of the Amiga operating system (Kickstart V2.04, V37). Release 2 of the operating system was first shipped on the Amiga 3000 and now available as an upgrade kit for the Amiga 500 and Amiga 2000 models to replace the older 1.3 (V34) operating system. Release 2 contains several new libraries and hundreds of new library functions and features to assist application writers.

Design Decisions
Compatible Libraries

1.24 1 / 1.3 Compatibility Issues / Design Decisions

The latest Amiga models, including all A3000's, are running Release 2. But many older Amigas are still running 1.3 at this time. Depending on your application and your market, you may choose to require the Release 2 operating system as a minimum platform. This can be a reasonable requirement for vertical markets and professional applications. Release 2 can also be a reasonable requirement for new revisions of existing

software products, since you could continue to ship the older 1.3-compatible release in the same package. If you have made the decision to require Release 2, then you are free to take advantage of all of the new libraries and features that Release 2 provides.

Throughout this latest edition of the Amiga Technical Reference Series, features, functions and libraries that are new for Release 2 are usually indicated by (V36) or (V37) in the description of the feature. Such features are not available on Amiga models that are running 1.3 (V34) or earlier versions of the OS. Unconditional use of Release 2 functions will cause a program to fail when it is run on a machine with the 1.3 OS. It is very important to remember this when designing and writing your code.

Developers of consumer-priced productivity, entertainment and utility software may not yet be ready to write applications that require Release 2, but even these developers can enhance their products by taking advantage of Release 2 while remaining 1.3 compatible.

There are three basic methods that will allow you to take advantage of enhanced Release 2 features while remaining 1.3 compatible:

* Transparent Release 2 Extensions

* Conditional Code

* Compatible Libraries

Transparent Release 2 Extensions
Conditional Code
ASL Requesters
DOS System(), CreateNewProc(), and CON: Enhancements
The Display Database
ARexx

1.25 1 / / Design Decisions / Transparent Release 2 Extensions

To provide Release 2 enhancements while remaining compatible with the older 1.3 version of the OS, several familiar 1.3 system structures have been extended to include an optional pointer to additional information. The new extended versions of such structures are generally defined in the same include file as the original structure. These extended structures are passed to the same 1.3 system functions as the unextended structure (e.g., OpenWindow(), OpenScreen(), AddGadget(), OpenDiskFont()). The existence of the extended information is signified by setting a new flag bit in the structure. (In one case, PROPNEWLOOK, only the flag bit itself is significant). These extensions are transparent to previous versions of the operating system. Only the Release 2 operating system will recognize the bit and act on the extended information.

The table below lists the flag bit for each structure to specify that extended information is present.

Original	Extended	Flag Field	Flag Bit	Defined In

```
NewScreen ExtNewScreen Type      NS_EXTENDED       <intuition/screens.h>
NewWindow ExtNewWindow Flags     WFLG_NW_EXTENDED  <intuition/intuition.h>
Gadget    Gadget       Flags     GFLG_STRINGEXTEND <intuition/intuition.h>
PropInfo  PropInfo     Flags     PROPNEWLOOK       <intuition/intuition.h
TextAttr  TTextAttr    tta_Style FSF_TAGGED        <graphics/text.h>
```

Through the use of such extensions, applications can request special Release 2 features in a 1.3-compatible manner. When the application is run on a Release 2 machine, the enhanced capabilities will be active.

The enhancements available through these extensions include:

```
   Screen: Overscan, 3D Look (SA_Pens), public screens, PAL/NTSC, new modes
   Window: Autoadjust sizing, inner dimensions, menu help
   Gadget: Control of font, pens, and editing of string gadgets
 PropInfo: Get 3D Look proportional gadgets when running under Release 2
TTextAttr: Control width of scaled fonts
```

Extensible longword arrays called TagItem lists are used to specify the extended information for many of these structures. Tag lists provide an open-ended and backwards-compatible method of growing system structures by storing all new specifications in an extendible array of longwords pairs.

Another transparent Release 2 extension is the diskfont library's ability to scale bitmap and outline fonts to arbitrary sizes and the availability of scalable outline fonts. Make sure that these new scalable fonts are available to your application by not setting the FPF_DESIGNED flag for AvailFonts() or OpenDiskFont(). Allow the user to create new font sizes by providing a way for her to manually enter the desired font size (the 1.3 OS returns the closest size, Release 2 returns the requested size).

See the Intuition and graphics library chapters, and the include file comments for additional information. See the "Utility Library" chapter for more information on TagItems and tag lists.

1.26 1 / / Design Decisions / Conditional Code

Conditional code provides a second way to take advantage of Release 2 enhancements in a 1.3-compatible application. The basic idea is to add low overhead conditional code, based on library version, to make use of selected Release 2 features if they are available. There are some powerful and beneficial Release 2 features which are definitely worth conditional code.

The control flow for such conditional code is always based on whether a particular version of a library is available. Failure of OpenLibrary() (i.e., return value of NULL) means that the library version requested is not available. The version number of a library that successfully opened can be checked by testing LibBase->lib_Version. Always check for a version greater or equal to the version you need.

```
Examples of conditional library checking code:

/* Checking for presence of a new Release 2 library */
if( AslBase = OpenLibrary("asl.library", 37L) )
        { /* OK to use the ASL requester */ }
else
        { /* Must use a different method */ }

/* Check version of an existing library with new Release 2 features */
if(GfxBase->lib_Version >= 37)   { /* then allow new genlock modes */}
```

1.27 1 / / Design Decisions / ASL Requesters

The Release 2 ASL library provides standard file and font requesters. Allocation and use of an ASL requester can be handled by coding a simple subroutine to use the ASL requester if available. Otherwise use fallback code or a public domain requester. By now, many of you have probably coded your own requesters and you may be quite attached to them. In that case, at least give your users the option to use the ASL requester if they wish. By using the ASL requesters, you can provide a familiar interface to your users, gain the automatic benefit of all ASL file requester improvements, and stop maintaining your own requester code.

1.28 1 / / Design / DOS System(), CreateNewProc(), and CON: Enhancements

If your program currently uses the 1.3 AmigaDOS Execute() or CreateProc() functions, then it is definitely worth conditional code to use their V37 replacements when running under Release 2. The System() function of Release 2 allows you to pass a command line to AmigaDOS as if it had been typed at a Shell window. System() can run synchronously with return values or asynchronously with automatic cleanup and it also sets up a proper stdio environment when passed a DOS filehandle for SYS_Input and NULL for SYS_Output. In combination with enhanced Release 2 CON: features, System() can provide a suitable execution environment on either Workbench or a custom screen. The CreateNewProc() function provides additional control and ease in process creation.

CON: input and output in custom Intuition screens and windows is now supported. New options in the Release 2 console handler (CON:) provide the ability to open a CON: on any public Intuition screen, or to attach a CON: to an existing Intuition window. Additional options can add a close gadget or create an AUTO console window which will only open if accessed for read or write. Add conditional code to use these system-supported methods when running under Release 2 or later versions of the OS. Note that additional CON: option keywords can be easily removed under 1.3 at runtime by terminating the CON: string with NULL after the window title. Consult The AmigaDOS Manual by Bantam Books for additional information on Release 2 CON: and DOS features.

1.29 1 / / Design Decisions / The Display Database

The Release 2 graphics library and the Enhanced Chip Set (ECS) provide programmable display modes and enhanced genlock capabilities. Users with Release 2 and ECS may wish to use your application in one of the newer display modes. The Release 2 display database provides information on all of the display modes available with the user's machine and monitor. In addition, it provides useful information on the capabilities and aspect ratio of each mode (DisplayInfo.Resolution.x and .y). A new function named ModeNotAvailable() allows you to easily check if particular modes are available.

The ExtNewScreen structure used with Intuition's OpenScreen() function allows you to specify new display modes with the SA_DisplayID tag and a longword ModeID. The Release 2 graphics library VideoControl() function provides greatly enhanced genlock capabilities for machines with ECS and a genlock. Little conditional code is required to support these features. See the graphics library chapters and Autodocs for more information.

1.30 1 / / Design Decisions / ARexx

Add conditional ARexx capabilities to your program. ARexx is available on all Release 2 machines, and many 1.3 users have purchased ARexx separately. ARexx capability adds value to your product and allows users and vertical market developers to create custom and hybrid applications. Add the ability to control your application externally via ARexx, and internally via ARexx macros. Allow the user to execute ARexx scripts to control other programs, including the ability to pass information from your program to other applications. For more information on adding ARexx functionality to your application, see the Amiga Programmer's Guide to ARexx, a publication by Commodore Applications and Technical Support (CATS). Contact your local Commodore support organization for information on ordering this book.

1.31 1 / 1.3 Compatibility Issues / Compatible Libraries

Compatible libraries provide a third method for using Release 2 while remaining 1.3-compatible. Some Release 2 libraries are 1.3-compatible and may be distributed with your product if you have a 1.3 Workbench License and an amendment to distribute the additional library.

IFFParse Library
Single Precision IEEE Math Libraries
Third Party Compatible Libraries

1.32 1 / / Compatible Libraries / IFFParse Library

The new IFFParse library is compatible with both Release 2 and the 1.3 version of the OS. IFFParse is a run-time library which provides low level code for writing, reading, and parsing IFF files. Use of IFFParse library and the new IFF example code modules can significantly reduce your

development and debugging time. In addition, the IFFParse code modules provide effortless handing of the clipboard device. See the IFFParse Library chapter in this book and the IFF Appendix of the Amiga ROM Kernel Reference Manual: Devices for additional information.

1.33 1 / / Compatible Libraries / Single Precision IEEE Math Libraries

The Release 2 single precision IEEE math libraries are also compatible with 1.3. These libraries provide single-precision math functions that will use a math coprocessor if available.

1.34 1 / / Compatible Libraries / Third Party Compatible Libraries

Developers of new code may wish to take advantage of the ease with which a user interface can be created using the Release 2 GadTools and ASL support libraries. These new libraries are not 1.3-compatible but there are some third party development efforts towards providing 1.3-compatible versions of them. You may wish to explore this possibility.

1.35 1 / Introduction / Commodore Applications and Technical Support (CATS)

Commodore maintains a technical support group dedicated to helping developers achieve their goals with the Amiga. Currently, technical support programs are available to meet the needs of both smaller, independent software developers and larger corporations. Subscriptions to Commodore's technical support publication, Amiga Mail, is available to anyone with an interest in the latest news, Commodore software and hardware changes, and tips for developers.

To request an application for Commodore's developer support program, or a list of CATS technical publications send a self-addressed, stamped, 9" x 12" envelope to:

```
CATS-Information
1200 West Wilson Drive
West Chester, PA  19380-4231
```

1.36 1 Introduction to Amiga System Libraries / Error Reports

In a complex technical manual, errors are often found after publication. When errors in this manual are found, they will be corrected in a subsequent printing. Updates will be published in Amiga Mail, Commodore's technical support publication.

Bug reports can be sent to Commodore electronically or by mail. Submitted reports must be clear, complete, and concise. Reports must include a

telephone number and enough information so that the bug can be quickly verified from your report (i.e., please describe the bug and the steps that produced it).

```
Amiga Software Engineering Group
ATTN: BUG REPORTS
Commodore Business Machines
1200 Wilson Drive
West Chester, PA  19380-4231
USA


BIX:    amiga.com/bug.reports(Commercial developers)
        amiga.cert/bug.reports(Certified developers)
        amiga.dev/bugs(Others)


USENET: bugs@commodore.COM  or  uunet!cbmvax!bugs
```