

## **Libraries**

**COLLABORATORS**

	<i>TITLE :</i> Libraries		
<i>ACTION</i>	<i>NAME</i>	<i>DATE</i>	<i>SIGNATURE</i>
WRITTEN BY		March 28, 2025	

**REVISION HISTORY**

NUMBER	DATE	DESCRIPTION	NAME

# Contents

<b>1</b>	<b>Libraries</b>	<b>1</b>
1.1	Amiga® RKM Libraries: 14 Workbench and Icon Library	1
1.2	14 Workbench and Icon Library / The Info File	1
1.3	14 Workbench and Icon Library / Workbench Environment	2
1.4	14 / Workbench Environment / Argument Passing In Workbench	3
1.5	14 // Argument Passing In Workbench / One Argument	3
1.6	14 // Argument Passing In Workbench / Two Arguments	3
1.7	14 // Argument Passing In Workbench / Multiple Arguments	3
1.8	14 / Workbench Environment / WBStartup Message	4
1.9	14 Workbench and Icon Library / The Icon Library	5
1.10	14 / The Icon Library / Icon Library Data Structures	6
1.11	14 // Icon Library Data Structures / The DiskObject Structure	6
1.12	14 // Icon Library Data Structures / The Gadget Structure	8
1.13	14 / The Icon Library / Icon Library Functions	9
1.14	14 / The Icon Library / The Tool Types Array	10
1.15	14 Workbench and Icon Library / The Workbench Library	11
1.16	14 / The Workbench Library / Workbench Library Functions	11
1.17	14 Workbench and Icon Library / Workbench and the Startup Code Module	12
1.18	14 / Workbench and the Startup Code Module / Workbench Startup	12
1.19	14 / Workbench and the Startup Code Module / Shell Startup	13
1.20	14 Workbench and Icon Library / Function Reference	14

# Chapter 1

## Libraries

### 1.1 Amiga® RKM Libraries: 14 Workbench and Icon Library

Workbench is the graphic user interface to the Amiga file system that uses symbols called icons to represent disks, directories and files. This chapter shows how to use Workbench and its two support libraries `workbench.library` and `icon.library`.

Workbench is both a system program and a screen. Normally it is the first thing the user sees when the machine is booted providing a friendly operating environment for launching applications and performing other important system activities like navigating through the Amiga's hierarchical filing system.

All application programs should be compatible with Workbench. There are only two things you need to know to do this: how to make icons for your application, data files and directories; and how to get arguments if your application is launched from Workbench.

The Info File	The Workbench Library
Workbench Environment	Workbench and the Startup Code Module
The Icon Library	Function Reference

### 1.2 14 Workbench and Icon Library / The Info File

The iconic representation of Amiga filing system objects is implemented through `.info` files. In general, for each file, disk or directory that is visible in the Workbench environment, there is an associated `.info` file which contains the icon imagery and other information needed by Workbench.

Icons are associated with a particular file or directory by name. For example, the icon for a file named `myapp` would be stored in a `.info` file named `myapp.info` in the same directory.

To make your application program accessible (and visible) in the Workbench environment, you need only supply a `.info` file with the appropriate name and type. There are four main types of icons (and `.info` files) used to represent Amiga filing system

---

objects (Table (14-1)).

Table 14-1: Basic Workbench Icon Types

Workbench Icon Type -----	Filing System Object -----	Result When Icon Is Activated -----
Disk	The root level directory	Window opens showing files and subdirectories
Drawer	A subdirectory	Window opens showing files and subdirectories
Tool	An executable file (i.e., an application)	Application runs
Project	A data file	Typically, the application that created the data file runs and the data file is automatically loaded into it.

Figure 14-1: Basic Workbench Icon Types

Icons can be created with the IconEdit program (in the Tools directory of the Extras disk), or by copying an existing .info file of the correct type. Icons can also be created under program control with PutDiskObject(). See the discussion of the icon library functions below for more on this.

For an executable file the icon type must be set to tool. For a data file the icon type must be set to project. Create icons for your application disk and directories too. For a directory, the icon is stored in a .info file at the same level where the directory name appears (not in the directory itself). The icon type should be set to drawer. The icon for a disk should always be stored in a file named disk.info at the root level directory of the disk. The icon type should be set to disk. (The icon type can be set and the icon imagery edited with the IconEdit program.)

### 1.3 14 Workbench and Icon Library / Workbench Environment

On the Amiga there are at least two ways to start a program running:

- \* By activating a tool or project icon in Workbench (an icon is activated by pointing to it with the mouse and double-clicking the mouse select button.)
- \* By typing the name of an executable file at the Shell (also known as the CLI or Command Line Interface)

In the Workbench environment, a program is run as a separate process. A process is simply a task with additional information needed to use DOS

library.

By default, a Workbench program does not have a window to which its output will go. Therefore, `stdin` and `stdout` do not point to legal file handles. This means you cannot use `stdio` functions such as `printf()` if your program is started from Workbench unless you first set up a `stdio` window.

Some compilers have options or defaults to provide a `stdio` window for programs started from Workbench. In Release 2, applications can use an auto console window for `stdio` when started from Workbench by opening "CON:0/0/640/200/auto/close/wait" as a file. An auto console window will only open if `stdio` input or output occurs. This can also be handled in the startup code module that comes with your compiler.

Argument Passing In Workbench  
WBStartup Message  
Example of Parsing Workbench Arguments

## 1.4 14 / Workbench Environment / Argument Passing In Workbench

Applications started from Workbench receive arguments in the form of a `WBStartup` structure. This is similar to obtaining arguments from a command line interface through `argc` and `argv`. The `WBStartup` message contains an argument count and a pointer to a list of file and directory names.

One Argument      Two Arguments      Multiple Arguments

## 1.5 14 // Argument Passing In Workbench / One Argument

A program started by activating its tool icon gets one argument in the `WBStartup` message: the name of the tool itself.

## 1.6 14 // Argument Passing In Workbench / Two Arguments

All project icons (data files) have a default tool field associated with them that tells Workbench which application tool to run in order to operate on the data that the icon represents. When the user activates a project icon, Workbench runs the application specified in the default tool field passing it two arguments in the `WBStartup` message: the name of the tool and the project icon that the user activated.

## 1.7 14 // Argument Passing In Workbench / Multiple Arguments

With extended select, the user can activate many icons at once. (Extended select means the user holds down the Shift key while clicking the mouse select button once on each icon in a group, double-clicking on the last

---

icon.) If one of the icons in a group activated with extended select is an application tool, Workbench runs that application passing it the name of all the other icons in the group. This allows the user to start an application with multiple project files as arguments. If none of the icons in a group activated with extended select is a tool icon, then Workbench looks in the default tool field of each icon in the order they were selected and runs the first tool it finds.

## 1.8 14 / Workbench Environment / WBStartup Message

When Workbench loads and starts a program, it sends the program a WBStartup message containing the arguments as summarized above. Normally, the startup code supplied with your compiler will place a pointer to WBStartup in argv for you, set argc to zero and call your program.

The WBStartup message, whose structure is outlined in <workbench/startup.h>, has the following structure elements:

```
struct WBStartup
{
    struct Message    sm_Message;    /* a standard message structure */
    struct MsgPort *  sm_Process;    /* process descriptor for you */
    BPTR              sm_Segment;    /* a descriptor for your code */
    LONG              sm_NumArgs;    /* number of elements in ArgList */
    char *            sm_ToolWindow; /* reserved for future use */
    struct WBArg *    sm_ArgList;    /* the arguments themselves */
};
```

The fields of the WBStartup structure are used as follows.

```
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
    CreateProc())

sm_Segment
    The loaded code for the tool (returned by LoadSeg())

sm_NumArgs
    The number of arguments in sm_ArgList

sm_ToolWindow
    Reserved (not currently passed in startup message)

sm_ArgList
    This is the argument list itself. It is a pointer to an array
    of WBArg structures with sm_NumArgs elements.
```

Workbench arguments are passed as an array of WBArg structures in the sm\_ArgList field of WBStartup. The first WBArg in the list is always the tool itself. If multiple icons have been selected when a tool is activated, the selected icons are passed to the tool as additional WBArgs. If the tool was derived from a default tool, the project will be the

second WBArg. If extended select was used, arguments other than the tool are passed in the order of selection; the first icon selected will be first (after the tool), and so on.

Each argument is a struct WBArg and has two parts: wa\_Name and wa\_Lock.

```
struct WBArg
{
    BPTR      wa_Lock;    /* a lock descriptor */
    BYTE *    wa_Name;    /* a string relative to that lock */
};
```

The wa\_Name element is the name of an AmigaDOS filing system object. The wa\_Name field of the first WBArg is always the name of your program and the wa\_Lock field is an AmigaDOS Lock on the directory where your program is stored.

If your program was started by activating a project icon, then you get a second WBArg with the wa\_Name field containing the file name of the project and the wa\_Lock containing an AmigaDOS Lock on the directory where the project file is stored.

If your program was started through extended select, then you get one WBArg for each icon in the selected group in the order they were selected. The wa\_Name field contains the file name corresponding to each icon unless the icon is for a directory, disk, or the Trashcan in which case the wa\_Name is set to NULL. The wa\_Lock field contains an AmigaDOS Lock on the directory where the file is stored. (For disk or drawer icons the wa\_Lock is a lock on the directory represented by the icon. Or, wa\_Lock may be NULL if the icon type does not support locks.)

Workbench Locks Belong to Workbench.

-----  
You must never call UnLock() on a wa\_Lock. These locks belong to Workbench, and Workbench will UnLock() them when the WBStartup message is replied by your startup code. You must also never UnLock() your program's initial current directory lock (i.e., the lock returned by an initial CurrentDir() call). The classic symptom caused by unlocking Workbench locks is a system hang after your program exits, even though the same program exits with no problems when started from the Shell.

You should save the lock returned from an initial CurrentDir(), and CurrentDir() back to it before exiting. In the Workbench environment, depending on your startup code, the current directory will generally be set to one of the wa\_Locks. By using CurrentDir(wa\_Lock) and then referencing wa\_Name, you can find, read, and modify the files that have been passed to your program as WBArgs.

## 1.9 14 Workbench and Icon Library / The Icon Library

The .info file is the center of interaction between applications and Workbench. To help support the Workbench iconic interface and manage .info files, the Amiga operating system provides the icon library. The icon library allows you to create icons for data files and directories

---

under program control and examine icons to obtain their Tool Types and other characteristics.

Icon Library Data Structures  
 Icon Library Functions  
 The Tool Types Array  
 Example of Reading Icons and Parsing Tool Types

## 1.10 14 / The Icon Library / Icon Library Data Structures

The preceding sections discussed how icons are used to pass file name arguments to an application run from the Workbench. Workbench allows other types of arguments to be passed in the Tool Types array of an icon. To examine the Tool Types array or find other characteristics of the icon such as its type, applications need to read in the .info file for the icon.

The DiskObject Structure      The Gadget Structure

## 1.11 14 // Icon Library Data Structures / The DiskObject Structure

The actual data present in the .info file is organized as a DiskObject structure which is defined in the include file <workbench/workbench.h>. For a complete listing, see the Amiga ROM Kernel Reference Manual: Includes and Autodocs. The DiskObject structure contains the following elements:

```
struct DiskObject
{
    UWORD                do_Magic;    /* magic number at start of file */
    UWORD                do_Version; /* so we can change structure */
    struct Gadget        do_Gadget;   /* a copy of in core gadget */
    UBYTE               do_Type;
    char                 *do_DefaultTool;
    char                 **do_ToolTypes;
    LONG                do_CurrentX;
    LONG                do_CurrentY;
    struct DrawerData *do_DrawerData;
    char                 *do_ToolWindow; /* only applies to tools */
    LONG                do_StackSize; /* only applies to tools */
};
```

### 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 upwardly-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 below for more details.

`do_Type`

The type of the icon; can be set to any of the following values.

<code>WBDISK</code>	The root of a disk
<code>WBDRAWER</code>	A directory on the disk
<code>WBTOOL</code>	An executable program
<code>WBPROJECT</code>	A data file
<code>WBGARBAGE</code>	The Trashcan directory
<code>WBKICK</code>	A Kickstart disk
<code>WBAPPICON</code>	Any object not directly associated with a filing system object, such as a print spooler (new in Release 2).

Table 14-2: Workbench Object Types

`do_DefaultTool`

Default tools are used for project and disk icons. For projects (data files), the default tool is the program Workbench runs when the project is activated. Any valid AmigaDOS path may be entered in this field such as "SYS:myprogram", "df0:mypaint", "myeditor" or ":work/mytool".

For disk icons, the default tool is the diskcopy program ("SYS:System/DiskCopy") that will be used when this disk is the source of a copy.

`do_ToolTypes`

This 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 section on "The Tool Types Array" below 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 window that opens when the drawer is activated. Each icon in the drawer has a position in the coordinate system. `CurrentX` and `CurrentY` contain the icon's current position in the drawer. Picking a position for a newly created icon can be tricky. `NO_ICON_POSITION` is a system constant 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.

`do_DrawerData`

If the icon is associated with a directory (`WBDISK`, `WBDRAWER`, `WBGARBAGE`), it needs a `DrawerData` structure to go with it. This structure contains an `Intuition NewWindow` structure (see the

"Intuition Windows" chapter for more information):

```

struct DrawerData
{
    struct NewWindow dd_NewWindow; /* structure to open window */
    LONG             dd_CurrentX; /* current x coordinate of */
                                /* origin */
    LONG             dd_CurrentY; /* current y coordinate of */
                                /* origin */
};

```

Workbench uses this to hold the current window position and size of the window so it will reopen in the same place.

do\_ToolWindow

This field is reserved for future use.

do\_StackSize

This is the size of the stack (in bytes) used for running the tool. If this is NULL, then Workbench will use a reasonable default stack size (currently 4K bytes).

Stack Size is Taken from the Project Icon.

-----  
When a tool is run via the default tool mechanism (i.e., a project was activated, not the tool itself), Workbench uses the stack size specified in the project's .info file and the tool's .info file is ignored.

## 1.12 14 // Icon Library Data Structures / The Gadget Structure

To hold the icon's image, Workbench uses an Intuition Gadget structure, defined in <intuition/intuition.h>. Workbench restricts some of the values of the gadget. All unused fields should be set to 0 or NULL. The Intuition gadget structure members that Workbench icons use are listed below.

Gadget Names in Assembly Language Are Different.

-----  
The assembly language version of the Gadget structure has leading "gg\_" for each variable name.

Width

This is the width (in pixels) of the icon's active region. Any mouse button press within this range will be interpreted as having selected this icon.

Height

This is the height (in pixels) of the icon's active region. Any mouse button press within this range will be interpreted as having selected this icon.

Flags

The gadget must be of type GADGIMAGE. Three highlight modes are supported: GADGHCOMP, GADGHIMAGE, and GADGBACKFILL. GADGHCOMP

complements everything within the area defined by CurrentX, CurrentY, Width, Height. GADGHIMAGE uses an alternate selection image. GADGBACKFILL is similar to GADGHCOMP, but ensures that there is no "ring" around the selected image. It does this by first complementing the image, and then flooding all color 3 pixels that are on the border of the image to color 0. All other flag bits should be 0.

#### Activation

The activation should have only RELVERIFY and GADGIMMEDIATE set.

#### Type

The gadget type should be BOOLGADGET.

#### GadgetRender

Set this to an appropriate Image structure.

#### SelectRender

Set this to an appropriate alternate Image structure if and only if the highlight mode is GADGHIMAGE.

The Image structure is typically the same size as the gadget, except that 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; PlanePick must be 3; and PlaneOnOff should be 0. The NextImage field should be null.

## 1.13 14 / The Icon Library / Icon Library Functions

The icon library functions do all the work needed to read, write and examine an icon's .info file and corresponding DiskObject structure:

```

struct DiskObject *GetDiskObject(UBYTE *name);
struct DiskObject *GetDiskObjectNew(UBYTE *name);           (V36)
BOOL               PutDiskObject(UBYTE *name, struct DiskObject *diskobj);
void               FreeDiskObject(struct DiskObject *diskobj);
BOOL               DeleteDiskObject(UBYTE *);              (V37)

UBYTE              *FindToolType(UBYTE **toolTypeArray, UBYTE *typeName);
BOOL               MatchToolValue(UBYTE *typeString, UBYTE *value);

struct DiskObject *GetDefDiskObjectNew(LONG type);         (V36)
BOOL               PutDefDiskObject(struct DiskObject *diskobj); (V36)

UBYTE              *BumpRevision(UBYTE *newbuf, UBYTE *oldname);

```

The icon library routine GetDiskObject() reads an icon's .info file from disk into a DiskObject structure it creates in memory where it can be examined or altered. PutDiskObject() writes the DiskObject out to disk and FreeDiskObject() frees the memory it used. If you modify any pointers in a DiskObject acquired via GetDiskObject(), replace the old pointers before calling FreeDiskObject() so that the proper memory will be freed.

Release 2 includes a new function named `GetDiskObjectNew()` that works the same as `GetDiskObject()`, except that if no `.info` file is found, a default `DiskObject` will be created for you. Also new for Release 2 is `DeleteDiskObject()` for removing `.info` files from disk, and the functions `GetDefDiskObject()` and `PutDefDiskObject()` which allow the default icons in ROM to be copied or replaced with new defaults in RAM.

Once an icon's `.info` file has been read into a `DiskObject` structure, the functions `FindToolType()` and `MatchToolValue()` can be used to examine the icon's `Tool Types` array.

## 1.14 14 / The Icon Library / The Tool Types Array

Earlier sections discussed how Workbench passes filenames as arguments to a program that's about to run. Workbench also allows other types of arguments to be passed in the `Tool Types` array of an icon. The `Tool Types` array is found in the `do_ToolTypes` field of the icon's `DiskObject` structure.

In brief, `Tool Types` is an array of pointers to strings that contain any information an application wants to store such as the program options that were in effect when the icon was created. These strings can be used to encode information which will be available to all applications that read the icon's `.info` file. Users can enter and change a selected icon's `Tool Types` by choosing `Information` in the `Workbench Icons` menu.

Workbench does not place many restrictions on the `Tool Types` array, but there are a few conventions you should follow. A string may be no more than 128 bytes long. The alphabet used is 8-bit ANSI (for example, normal ASCII with foreign-language extensions). This means that users may enter `Tool Type` strings containing international characters. Avoid special or nonprinting characters. The case of the characters is currently significant, so the string `"Window"` is not equal to `"WINDOW"`.

The general format for a `Tool Types` entry is `<name>=<value>[|<value>]`, where `<name>` is the field name and `<value>` is the text to associate with that name. Multiple values for one name may be separated by a vertical bar. The values may be the type of the file, programs that can access the data, parameters to be passed to an application, etc. For example, a paint program might set:

```
FILETYPE = PaintProgram | ILBM
```

This `Tool Type` indicates that the file is an `ILBM`, perhaps with some additional chunks of data specific to `PaintProgram`.

`Tool Type` strings have few restrictions but there are some reserved `Tool Types` that are parsed by Workbench itself when an application is started from an icon. The reserved `Tool Types` are `TOOLPRI=n` (sets the `Exec` task priority at which Workbench will start the application), `STARTPRI=n` (sets the starting order for icons in the `Wbstartup` drawer), and `DONOTWAIT` (tells Workbench not to wait for the return of a program started via an icon in the `Wbstartup` drawer). In addition to the reserved `Tool Types`, which applications should not use, there are standard `Tool Types`, which applications should use only in the standard way. For a list of standard

Tool Types refer to the Amiga User Interface Style Guide.

Two routines are provided to help you deal with the Tool Types array. FindToolType() returns the value of a Tool Type element. Using the above example, if you are looking for FILETYPE, the string "PaintProgram|ILBM" 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 strings of "PaintProgram" or "ILBM", MatchToolValue() will return TRUE for "ILBM" and "PaintProgram" and FALSE for everything else.

## 1.15 14 Workbench and Icon Library / The Workbench Library

Workbench arguments are sent to an application when it is started. There are also special facilities in Release 2 of Workbench that allow an application that is already running to get additional arguments. These special facilities are known as AppWindow, AppIcon and AppMenuItem.

An AppWindow is a special kind of window that allows the user to drag icons into it. Applications that set up an AppWindow will receive a message from Workbench whenever the user moves an icon into the AppWindow. The message contains the name of the file or directory that the icon represents.

An AppIcon is similar to an AppWindow. It is a special type of icon that allows the user to drag other icons on top of it. Like AppWindows, an application that sets up an AppIcon will receive a message from Workbench whenever the user moves another icon on top of the AppIcon. The message contains the name of the file or directory that the moved icon represents.

An AppMenuItem allows an application to add a custom menu item to the usual set of menu choices supported by Workbench. An application that sets up an AppMenuItem will receive a message from Workbench whenever the user picks that item from the Workbench menus.

When an application receives the messages described above, the message will include struct WBArg \*am\_ArgList containing the names (wa\_Name) and directory locks (wa\_Lock) of all selected icons that were passed as arguments by the user. This am\_ArgList has the same format as the sm\_ArgList of a WBStartup message.

Workbench Library Functions	An AppMenuItem Example
An AppIcon Example	An AppWindow Example

## 1.16 14 / The Workbench Library / Workbench Library Functions

AppWindows, AppIcons and AppMenuItems extend the user's ability to perform operations with the Workbench iconic interface. They all provide graphical methods for passing arguments to a running application. In order to manage AppWindows, AppIcons and AppMenuItems, the Amiga OS includes these Workbench library functions:

---

```

struct AppIcon      *AddAppIconA( ULONG, ULONG, char *, struct MsgPort *,
                                struct FileLock *, struct DiskObject *,
                                struct *TagItem );
struct AppMenuItem *AddAppMenuItemA( ULONG, ULONG, char *,
                                      struct MsgPort *, struct *TagItem);
struct AppWindow   *AddAppWindowA( ULONG, ULONG, struct Window *,
                                    struct MsgPort *, struct *TagItem);

BOOL                RemoveAppIcon(struct AppIcon *);
BOOL                RemoveAppMenuItem(struct AppMenuItem *);
BOOL                RemoveAppWindow(struct AppWindow *);

```

The functions `AddAppMenuItemA()`, `AddAppWindowA()` and `AddAppIconA()` have alternate entry points using the same function name without the trailing `A`. The alternate functions accept any `TagItem` arguments on the stack instead of from an array. See the listings below for examples.

## 1.17 14 Workbench and Icon Library / Workbench and the Startup Code Module

Standard startup code handles the detail work of interfacing with the arguments and environment of Workbench and the Shell (or CLI). This section describes the behavior of standard startup modules such as the ones supplied with SAS (Lattice) C and Manx Aztec C.

The environment for a program started from Workbench is quite different from the environment for a program started from the Shell. The Shell does not create a new process for a program; it jumps to the program's code and the program shares the process with the Shell. Programs run under the Shell have access to all the Shell's environment, including the ability to modify that environment. (Programs run from the Shell should be careful to restore all values that existed on startup.) Workbench starts a program as a new DOS process, explicitly passing the execution environment to the program.

Workbench Startup      Shell Startup

## 1.18 14 / Workbench and the Startup Code Module / Workbench Startup

When the user activates a project or tool icon, the program is run as a separate process asynchronous to Workbench. This allows the user to take full advantage of the multitasking features of the Amiga. A process is simply a task with additional information needed to use DOS library.

When Workbench loads and starts a program, it sends the program a `WBStartup` message containing the arguments as described earlier. The `WBStartup` also contains a pointer to the new `Process` structure which describes the execution environment of the program. The `WBStartup` message is posted to the message port of the program's `Process` structure.

The `Process` message port is for the exclusive use of DOS, so this message

must be removed from the port before using any DOS library functions. Normally this is handled by the startup code module that comes with your compiler so you don't have to worry about this unless you are writing your own startup code.

Standard startup code modules also set up SysBase, the pointer to the Exec master library, and open the DOS library setting up DOSBase. That is why Exec and AmigaDOS functions can be called by C applications without first opening a library; the startup code that applications are linked with handles this. Some special startups may also set up NIL: input and output streams, or may open a stdio window so that the Workbench applications can use stdio functions such as printf().

The startup code can tell if it is running in the Workbench environment because the pr\_CLI field of the Process structure will contain NULL. In that case the startup code removes the WBStartup message from the Process message port with GetMessage() before using any functions in the DOS library.

Do Not Use the Process Message Port for Anything Else.

-----  
The message port in a Process structure is for the exclusive use of the DOS library.

Standard startup code will pass the WBStartup message pointer in argv and 0 (zero) in argc if the program is started from Workbench. These values are pushed onto the stack, and the startup code calls the application code that it is linked with as a function. When the application code exits back to the startup code, the startup code closes and frees all opens and allocations it made. It will then Forbid(), and ReplyMsg() the WBStartup message, notifying Workbench that the application Process may be terminated and its code unloaded from memory.

Avoid the DOS Exit() function.

-----  
The DOS Exit() function does not return an application to the startup code that called it. If you wish to exit your application, use the exit function provided by your startup code (usually lower-case exit(), or \_exit for assembler), passing it a valid DOS return code as listed in the include file <libraries/dos.h>.

## 1.19 14 / Workbench and the Startup Code Module / Shell Startup

When a program is started from the Shell (or a Shell script), standard startup modules will parse the command line (received in A0, with length in D0) into an array of pointers to individual argument strings placing them in argv, and an argument count in argc.

If a program is started from the Shell, argc will always equal at least one and the first element in argv will always be a pointer to the command name. Other command line arguments are stored in turn. For example, if the command line was:

```
df0:myprogram "my file1" file2 ;this is a comment
```

then argc will be 3, argv[0] will be "df0:myprogram", argv[1] will be "my

file1", and argv[2] will be "file2". Correct startup code will strip spaces between arguments and trailing spaces from the last argument and will also properly deal with quoted arguments with embedded spaces.

As with Workbench, standard startup code for the Shell sets up SysBase, the pointer to the Exec master library, and opens the DOS library setting up DOSBase. C applications that are linked with standard startup code can call an Exec or AmigaDOS functions without opening the library first.

The startup code also fills in the stdio file handles (\_stdin, \_stdout, etc.) for the application. Finally argv and argc, are pushed onto the stack and the application is called via a JSR. When the application returns or exits back to the startup code, the startup code closes and frees all opens and allocations it has made for the application, and then returns to the system with the whatever value the program exited with.

Link your applications only with standard, tested startup code of some type such as the module supplied with your compiler. Startup code provides your programs with correct, consistent handling of Shell command line and Workbench arguments and will perform some initializations and cleanups which would otherwise need to be handled by your own code. Very small startups can be used for programs that do not require command line arguments.

A few words of warning for those of you who do not use standard startup code:

- \* If you are started as a Workbench process, you must GetMsg() the WBStartup message before using any functions in the DOS library.
- \* You must turn off task switching (with Forbid()) before replying the WBStartup message from Workbench. This will prevent Workbench from unloading your code before you can exit properly.
- \* If you do your own command line parsing, you must provide the user with consistent and correct handling of command line arguments.

## 1.20 14 Workbench and Icon Library / Function Reference

The following are brief descriptions of the functions in workbench.library and icon.library. See the Amiga ROM Kernel Reference Manual: Includes and Autodocs for details on each function call.

Table 14-3: Icon Library Functions

Function	Description
GetDiskObject()	Read the .info file of an icon into a DiskObject structure
GetDiskObjectNew()	Same as GetDiskObject() but returns a default icon if none exists
PutDiskObject()	Write a DiskObject structure to disk as a .info file

FreeDiskObject()	Free the DiskObject structure created by GetDiskObject()
DeleteDiskObject()	Deletes a given .info file from disk
-----	
FindToolType()	Return the value of an entry in the icon's Tool Type array
MatchToolValue()	Check a Tool Type entry against a given value
-----	
GetDefDiskObject()	Read the default icon for a given icon type
PutDefDiskObject()	Replace the default icon for a given icon type (V36)
-----	
AddFreeList()	Add memory you have allocated to a FreeList
FreeFreeList()	Free all the memory for entries in the FreeList
BumpRevision()	Create a new name for a second copy of a Workbench object

Table 14-4: Workbench Library Functions

Function	Description
AddAppIcon()	Add an AppIcon to Workbench
AddAppMenuItem()	Add an AppMenuItem to the Workbench Tools menu
AddAppWindow()	Add an AppWindow to Workbench
-----	
RemoveAppIcon()	Remove an AppIcon to Workbench
RemoveAppMenuItem()	Remove an AppMenuItem to the Workbench Tools menu
RemoveAppWindow()	Remove an AppWindow to Workbench