Hardware Reqt.: Apple IIGS/1 MEG, 2 or More 3.5 DRIVES AND/OR HARD DRIVE.

Call-Box

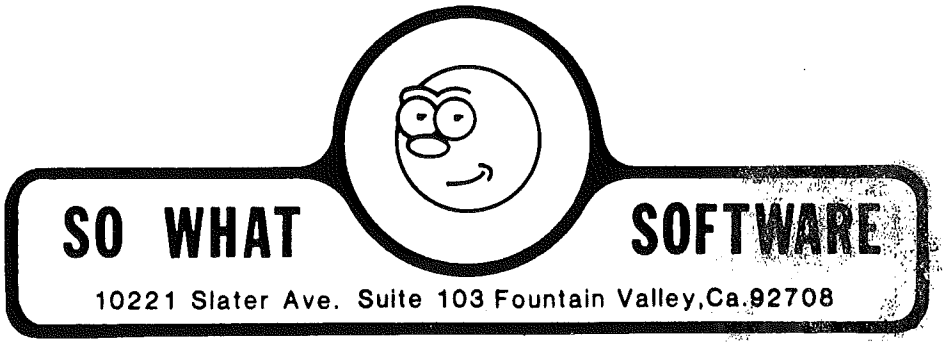
INCLUDES . . . LAUNCHING SHELL, EDITORS and BASIC INTERFACE

The Toolbox Programming System . . .

Call Box™ TPS

The Toolbox Programming System

Version 2.0 15-Jan-90



NOTICE

So What Software reserves the right to make improvements in the product described in this manual at any time without notice.

So What Software
10221 Slater Ave.
Suite 103,
Fountain Valley CA.
92708

So What Software provides a 90 day warranty against mechanical failure and physical defects in this product from the date of purchase. The warranty card must be filled out and sent back to So What Software before this warranty can be honored. So What Software makes no warranties with respect to this product, its quality, performance, merchantability or fitness for any particular purpose.

Software Design: SHELL- William Stephens, Eric Joham BASIC- William Stephens, Eric Joham EDITORS- William Stephens, Joe Jaworski DEMOS- Ed Rambeau, Eric Joham, William Stephens

Manual Design: Don Druce, William Stephens, Eric Joham

This software package was created using the following software and hardware products: Apple IIgs /W 1.5M & GS/OS V5.0, Applied Ingenuity 40M Inner Drive, Apple Laserwriter IINT, Apple 3.5 drives, Apple LocalTalk network, Applied Engineering TranswarpGS, Byte Works Orca/M assembler/linker, Claris AppleworksGS.

So What Software Product #M400-001A

Welcome

Welcome to the **Call Box TPS** (*Toolbox Programming System*). The Call Box system introduces a toolbox driver which allows you to create launchable desktop applications in enhanced **Applesoft BASIC.**
This driver can act as the ideal prototyping language for the professional and can open up the mysteries of the IIgs for the amatuer and beginner. This driver is supported by **WYSIWYG** (*What You See Is What You Get*) editors which create commonly used toolbox data structures which up till now required hours of tedious planning, guesswork and re-work to make. You can create Windows, Dialogs, Menus, Icons, Cursors and pixel images in a fraction of the time that it usually takes and incorporate them into any language in a variety of forms.

The Call Box TPS is designed to grow with new advancements in the IIgs and upgrades will be made available to registered owners of this software as they become available. **Be sure to send in your warranty card when you receive this product so we can notify you when upgrades are available...** there is really no way we can find you without this.

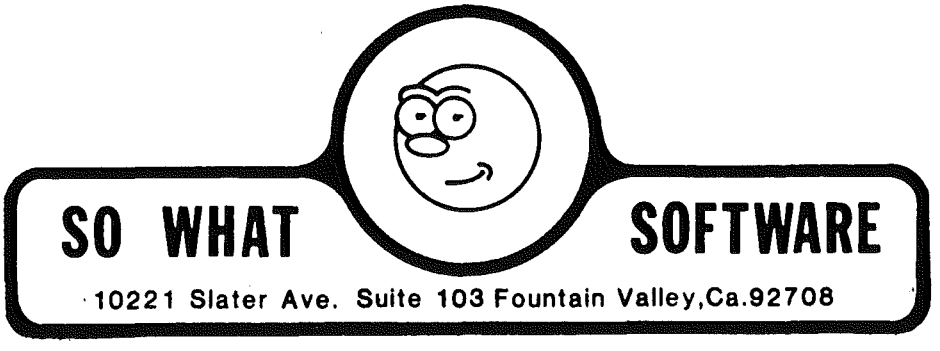
This manual is divided into 3 sections (*Launcher, BASIC Driver and Editors*). In order to take full advantage of all the features in this software we recommend that you actually **READ** this manual to serve as a resource for any questions you may have.

IMPORTANT NOTE:

The version 2.0 release of this software is full featured as advertised. We would like to thank the customers who purchased the Version 1.0 Call Box TPS for their patience and understanding with the prior limitations to this system. All registered owners of Version 1.0 are entitled to a free upgrade to Version 2.0.

Launching Shell

Version 2.0 15-Jan-90



CHAPTER 1 - OVERVIEW

OVERVIEW

The Call Box launching shell gives the user access to the various functions of the Call Box Toolbox programming system. This shell has 3 major functions or divisions.

SYSTEM: This group of functions are found in the File menu of the launchers menu bar. These functions include a programmable launcher, file utilities such as delete, rename, set filetype, set auxtype and file access bits, launcher desktop preferences, system installer, and eject drive(s).

EDITORS: This is where the WYSIWYG editors are launched from. These editors create source, object code or resources segments for use by any programming language. The editors create Window, Dialog, Menu, Icon, Cursor and pixel image data structures. Other editors will be released in the future and this menu will grow automatically as the need presents itself. (See. The EDITORS MANUAL for details)

BASIC: This is where the Call Box BASIC Interface resides. You can start Call Box BASIC, or Applesoft BASIC from their menu items and can run the Call Box BASIC Deom/tutorial as well. Another utility is provided to edit the CB.VARS file used with the BASIC interface.(See. The BASIC INTERFACE MANUAL)

This manual will explain how to use the utilities found in the launching shell as well as how to use this shell with several different system configurations. The exact use of the EDITORS and BASIC Interface is described in their own manuals.

THE DESKTOP

When the CALL BOX LAUNCHING SHELL disk is booted up a menu bar with the selections (APPLE), File, Edit, Editors and BASIC will appear and a 320 mode graphic image will appear as your desktop. This image is in fact 2-65 block type $C1 super hi-res pictures named XXX and XXXX. These pictures can be edited by any paint program or can be replaced by any ones that you have created. Just a portion of the second picture (XXXX) is actually displayed by selecting the menu selection Apple-About CALL BOX...

The desktop can be displayed in any one of 4 different ways... 320 mode picture desktop, 320 mode standard desktop (periwinkle blue) 640 mode picture desktop or 640 mode standard desktop. Select File-Preferences to set the way you want the desktop to appear. (See. Figure 1.1)

The next time the launcher is booted or launched your preferences will take effect.

If you exclude either or both of the pictures from your disk then the standard desktop appears by default. Eliminating the pictures will free up 130 blocks of disk space and will eliminate 2 or 3 seconds of loading time.... I prefer to have a picture type of desktop, it looks fancier and the time difference is negligeable.

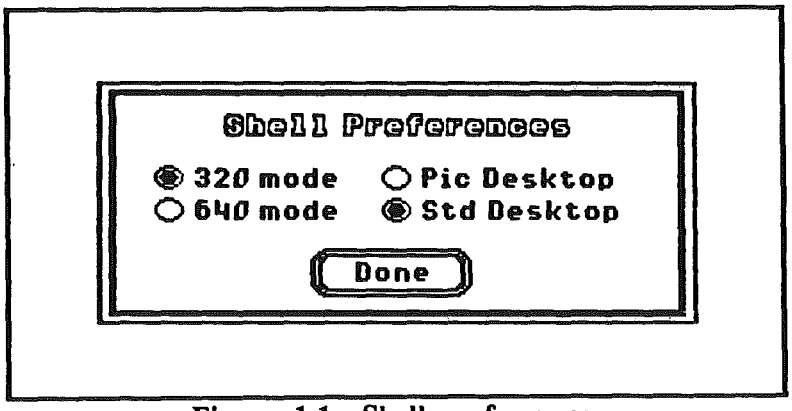


Figure 1.1 Shell preferences

FILE UTILITIES

The selection File-File Utilities will bring up a Standard file selector box. Use the disk button to display the drive that contains the file you want to edit. use OPEN and the scroll bar to put this file in the scroll window. Point and click the arrow cursor on the file in question and it will hilite, then click the OPEN button. (You can just double-click the files title to accomplish the same thing).

The standard file box will disappear and the file utilities box will appear. (See Figure 1.2) This "Mini-utility" allows you to do five things to a ProDOS file.

1. To rename a file press the DELETE key and then type in your new filename, it will appear in the line edit box named Filename: Press RETURN or click the OK button to accept the new name.

2. To change the filetype of your file triple-click the line edit box named Filetype, press delete and type in the new filetype. You can enter the filetypes hex number (ie: $04.. $C1) or enter any of the standard 3 letter designators (ie: BAS, BIN, S16...) Press RETURN or click the OK button to accept the new name.

3. To change the Aux Type of your file follow the same procedure as outlined in the previous description. You can only enter the hex number (minus the $...dollar sign) and it must be 4 digits. Press RETURN or click the OK button to accept the new name.

4. To set the access bits set the check boxes named Destroy, Rename, Backup, Write or Read to reflect the desired settings. A checked box "enables" the access and an unchecked one"disables" it. Press RETURN or click the OK button to accept the new name.

5. To delete a file click the DELETE button. A second box will appear as a safety which allows you to change your mind before doing something permanent and possibly destructive.

NOTE: If you accidentally delete a file you did not want deleted, all is not lost. STOP ALL WRITING TO DISK NOW!!! Utilities such as Disc Commander or Copy II Plus have undelete functions which wil fix things up, but they are ineffective if you have written to disk after the file is deleted.

The File utilities box has one more button and that is CANCEL.... its function should be obvious.

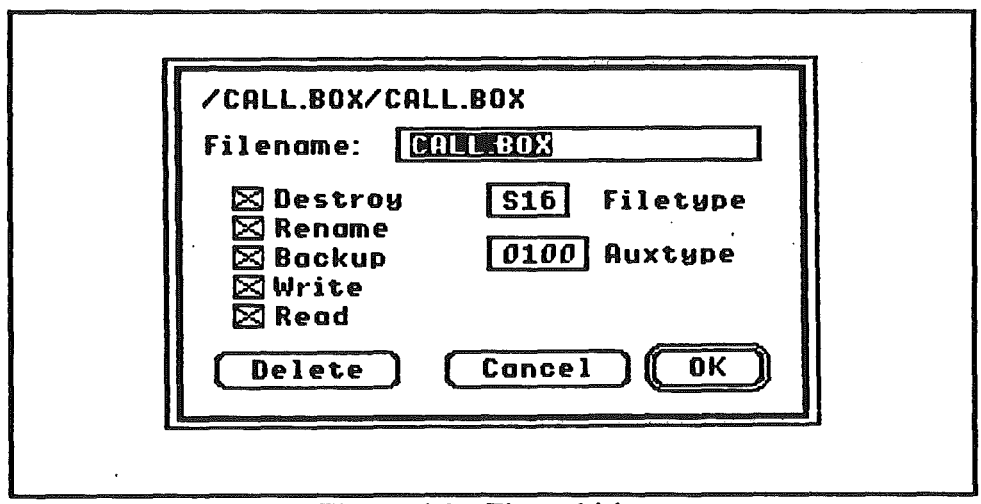


Figure 1.2 File utilities

MY APPLICATION

The menu selection FILE-MY APPLICATION will allow you to launch your own program from this launching shell. Your program may be your programming language environment, the finder, or any launchable application. This selection provides you quick access to another program which will return to the CALL BOX launching shell when the other application is quit.

To set which program you want to launch with this feature select FILE-SET MY APPLICATION and a standard file box will appear. (See Figure 1.3) Use the disk button to display the drive that contains the file you want to launch. Use OPEN and the scroll bar to put this file in the scroll window. Point and click the arrow cursor on the file in question and it will hilite, then click the OPEN button. (You can just double-click the files title to accomplish the same thing).

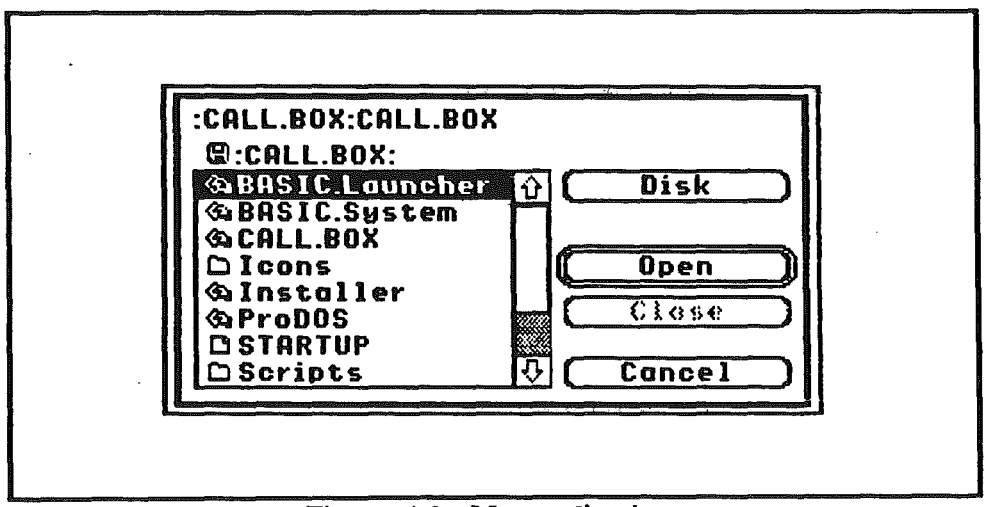


Figure 1.3 My application

CONFIGURE CB.VARS

This utility allows the user to alter the variable names assigned for the functions of the CALL BOX BASIC interface. These variable names are contained in a variables file named CB.VARS. This file is RESTOREd at the beginning of each CB BASIC program to link the program to the CALL BOX BASIC Interface.

The selection BASIC-CONFIGURE CB.VARS will bring up a Standard file selector box. Use the disk button do display the drive that contains the CB.VARS file you want to edit. Use OPEN and the scroll bar to put this file in the scroll window. Point and click the arrow cursor on the file in question and it will hilite, then click the OPEN button. (You can just double-click the files title to accomplish the same thing).
The standard file selector box will disappear and the CB.VARS configuration box will appear. To alter a variable triple-click the desired variable, press DELETE and then type in the new variable. CALL BOX BASIC interface variables must be "Real" and have 2 letters. A real variable is one in floating point format. (See Figures 1.4)

Repeat this process for each variable you want to alter. When done either click the OK button or press RETURN.

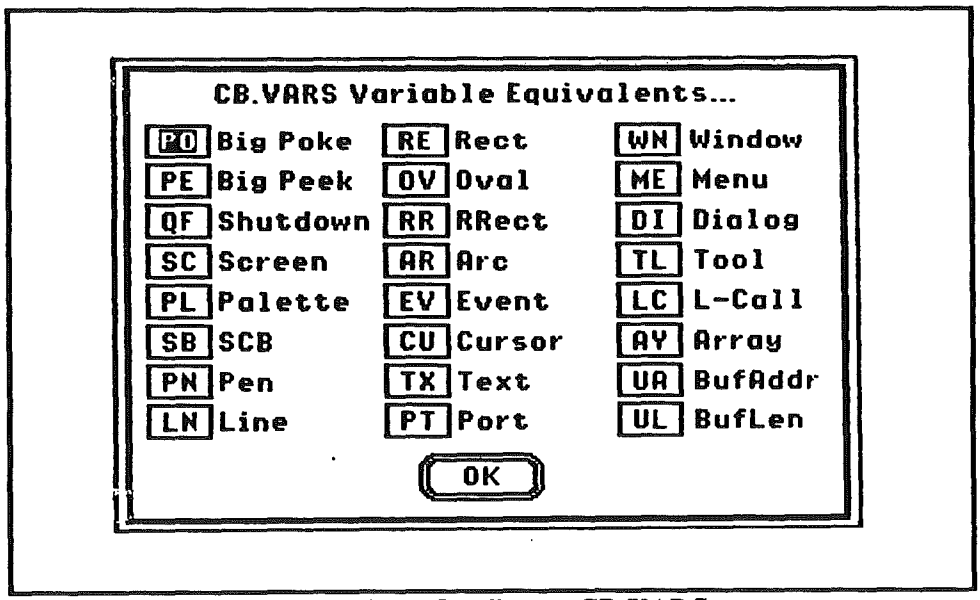


Figure 1.4 Configure CB.VARS

1 DRIVE OPERATION

Single drive operation of the CALL BOX Toolbox Programming System is the least recommended type of operation. You can successfully operate this system on one drive but you will have to eject and insert disks quite often.

NOTE: Always use backup copys of the CALL BOX disks when operating from disk. Many things can happen to a computer system while operating it such as power transients or bad keypresses and even bad programming procedures. Any of these occurrences can cause disk damaged. Play it safe and use only backup disks when programming.... this applies to all software, not just ours.

Boot-up the Launching Shell disk and the desktop will appear. The EDITORS menu selection will be disabled at this time and the CALL BOX BASIC specific selections in the BASIC menu selection will be disabled as well.

To launch an editor eject the launching shell disk and insert the editors disk... the EDITORS menu selection will become selectable. Pull-down and select the desired editor and it will be launched. In the launching process the editor will need to load or access some things in the launching shell disk and you will be prompted to insert the needed disk(s) as applicable.

While in the editor you will need to load or save editor data to or from your own disk. Eject the EDITORS disk and insert your disk to load or save things. The EDITORS disk can remain out of the drive during the editing session because the editors are loaded entirely in memory and need no further disk access to operate once up and running.

When you QUIT the Editor you will need the Launching Shell disk back in the drive, if you forget, the system will prompt you to do so.

This procedure holds true for the BASIC Interface disk as well. The system is designed so you can not select a system function that is not on-line currently.

The FILE-EJECT.. functions are provided to eject the disks from the drives. You can just press the button on the face of your disk drive to accomplish the same function.

There are 3 CALL BOX disks, the names of the disks differ from their volume names. When prompted for a different disk the actual volume name is requested and not the name on the disk label.

Launching Shell = /CALL.BOX

Editors = /CALL.BOX.2

BASIC Interface = /CALL.BOX.3

2 DRIVE OPERATION

Double drive operation of the CALL BOX Toolbox Programming System is quite similar to single drive operation except that you will have to swap disks less frequently.

NOTE: Always use backup copies of the CALL BOX disks when operating from disk. Many things can happen to a computer system while operating it such as power transients or bad keypresses and even bad programming procedures. Any of these occurrences can cause disk damaged. Play it safe and use only backup disks when programming.... this applies to all software, not just ours.

Boot-up the Launching Shell disk and insert either the EDITORS or the BASIC Interface disk in drive 2. The system will automatically sense which disk(s) is on-line and will enable the appropriate menu selections.

Removing either the EDITORS disk or the BASIC Interface disk will disable the menu selections automatically... reinserting them will enable the selections.

You can boot-up the Launching shell disk and when the desktop comes up you can then eject this disk and put the EDITORS and BASIC interface disks in the 2 drives. No matter how you use your drives the system will prompt you when a disk that is not currently on-line is needed.

There are 3 CALL BOX disks, the names of the disks differ from their volume names. When prompted for a different disk the actual volume name is requested and not the name on the disk label.

Launching Shell = /CALL.BOX

Editors = /CALL.BOX.2

BASIC Interface = /CALL.BOX.3

HARD DRIVE OPERATION/INSTALLATION

The CALL BOX TPS was designed with the hard drive user in mind. A large storage device such as a hard drive is basically necessary for any serious program development task. The limitations of disk drives becomes apparent when developing software on a par with commercial and professional applications.

An installer script is provided to install the system on a hard drive. The only prerequisite is that the hard drive volume must be GS/OS V5.0 minimum. Using this software with versions prior to 5.0 will create all kinds of problems and will probably not work. The Shell and Editors take advantage of NEW GS/OS and toolbox calls plus utilize the Resource Manager which is not present on earlier versions.

The Launching Shell disk has a minimal GS/OS V5.0 system on it which does not include many of the segments that a real GS/OS V5.0 system disk does. There is only enough of the system present to make it boot and support the functions of this software. If you do not already have GS/OS V5.0 min. Installed in your hard drive, go to your local Apple dealer and purchase a copy and install it.

To install the CALL BOX TPS on your hard drive select FILE-INSTALLER from the launching shell menu. Click the INSTALL button from the installer program and follow the prompts as they come up.... That's all there is to it!

There are 3 CALL BOX disks, the names of the disks differ from their volume names. When the installer asks for a different disk the actual volume name is requested and not the name on the disk label.

Launching Shell = /CALL.BOX

Editors = /CALL.BOX.2

BASIC Interface = /CALL.BOX.3

A subdirectory named CALL.BOX will be created in your root directory and will contain the Launching Shell, Editors and the BASIC Interface all together. Segments from all 3 disks will be required for this installation procedure. An additional tool will be installed and the basic.launcher program will be overwritten with a different version.... the finder will not be affected by this.

To use the CALL BOX TPS Launch the file named CALL.BOX in the CALL.BOX subdirectory with whatever launcher you have in the START position of your hard drive volume.

ICONS

This disk contains special icons for use by the "Finder" or other icon based programs. These icons are CALL BOX specific and are in 640 mode using dithered colors. The installer script for putting CALL BOX on a hard drive automatically installs them in the icons folder of you hard drive root directory. If you will be using disk drives instead of a hard drive then copy the file CB.ICONS from the CALL BOX icons folder to the same folder on your system disk so the "Finder" on that disk can bring them up.

These icons are simply cosmetic and do not affect the operation of CALL BOX in any way.

TOOL053

There is a "NEW" tool on the CALL.BOX disk called Tool053. This tool provides the user interface for loading and saving resources.

All CALL BOX editors depend on the presence of this tool and will not operate without it. You must include this tool in any system disk that will be used in conjunction with the CALL BOX editors. This tool goes in the SYSTEM/TOOLS folder of your boot volume.

There are 2 functions in addition to the normal housekeeping functions for tools, these functions are RFPutFile and RFGetFile.

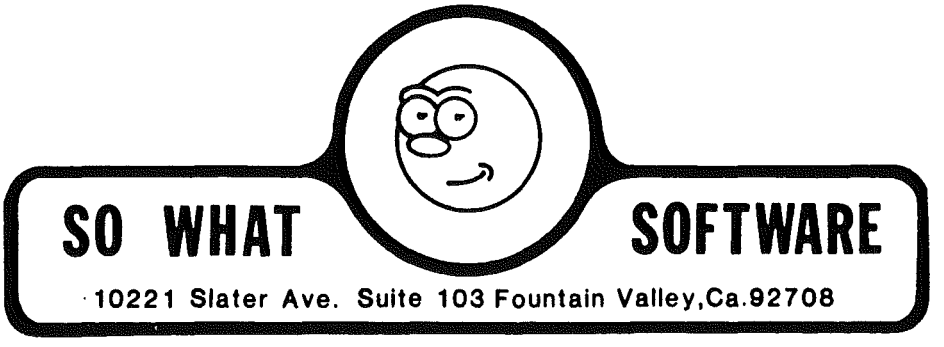
RFPutFile will provide the user with point-and-click access to any resource fork, showing the resource I.D.'s for any specified resource type. (See the save sections for each editor in the EDITORS manual for detailed operating procedures)

RFGetFile will provide the user with point-and-click access to any resource fork, showing the resource I.D.'s for any specified resource type. (See the load sections for each editor in the EDITORS manual for detailed operating procedures)

A complete Tool053 reference document is available separately from us for those of you who wish to incorporate this tool in your own programs. (product #M400-004)

Call Box™ BASIC

Version 2.0 15-Jan-90



Call Box™ BASIC Contents

OVERVIEW

The **Call Box BASIC** driver gives **Applesoft BASIC** new commands and capabilities that utilize the **Apple IIgs Toolbox**. These tools give you the ability to draw in either 320 or 640 mode plus make use of Icons, Cursors, Fonts, Dialogs, Menus, Windows... In fact, almost any tool call is made possible with the Call Box BASIC Interface.

Call Box BASIC in the Launching Shell

The Call Box TPS (*Toolbox Programming System*) launching shell has a menu bar selection named BASIC. This menu contains selections which deal with Call Box BASIC and Applesoft.

Applesoft BASIC	This selection puts you in Applesoft BASIC and ProDOS 8. All of the usual Applesoft and **ProDOS 8** functions are available here.
Call Box BASIC	This selection puts you in **Call Box BASIC**, **ProDOS 8** and **SoDOS**. Call Box BASIC is an enhanced Applesoft BASIC and SoDos is a GS/OS emulator.
Configure CB.VARS	This selection allows you to change the variable names given to the Call Box functions.
Call Box Demo	This selection runs a Demo/Tutorial on the Call Box BASIC driver. This program demonstrates by example each interface command and shows programming line examples with each demonstration.

The launching shell provides you with a convienent environment from which you can create Call Box BASIC programs. Call Box BASIC programs can be used on bootable 3.5 inch disks and can be launched from any program launcher using RAM or Hard Disk Drives.

Bootable 3.5" Call Box Program Disks

A bootable program disk can be made by inserting a blank disk in Slot 5 Drive 1 and then selecting **FILE-FORMAT DISK** from the Call Box Launching Shell. This will format the disk as volume **CB.BASIC**. Next select **FILE-INSTALLER** and run the script named **INITIALIZE CB.BASIC**, this will install GS/OS and all the necessary Call Box files to make-up a bootable disk. When this disk is booted it will result in running a mock **STARTUP** program using Call Box BASIC. Replace this program with your own program(s)... your ready to go!

Launchable Call Box Programs

Call Box BASIC programs can be launched by any program launcher capable of launching BASIC programs. Launchers that use the desktop like HyperLaunch and the Finder need no special handling but launchers that use the text screen display like ProSel will need to run the **CB.PreLaunch** program before running Call Box BASIC programs in order to install and initialize the desktop tools. The boot volume must contain the init file called **CB.Init** in any case. Your Call Box BASIC programs should be in their own subdirectory which must also contain the files **CB** and **CB.VARS**. More than one Call Box style subdirectory can exist and any kind of file can be in these subdirectorysas long as the minimum required files are present, which are CB and CB.VARS. The file CB.PreLaunch can be put anywhere.

OVERVIEW cont.

Call Box BASIC and Desk Accessories

Classic Desk Accessories (CDA's) are always available by pressing **OPEN APPLE-CONTROL-ESCAPE** and 3 are supplied with the Call Box Toolbox Programming System.

Reveal: This one will show the text screen while the Super Hi-res screen is active.

Applesoft Memory: This one shows the current boundaries and locations in Applesoft BASIC.

Call Box Memory: This one shows special locations and data behind the scenes in Call Box BASIC.

New Desk Accessories (NDA's)

Call Box BASIC gives you the ability to display and run desktops from Applesoft BASIC. The system menu bar in a desktop application has a (*colored*) Apple menu selection which contains all of the active New Desk Accessories in the **:SYSTEM:DESK.ACCS** subdirectory of your boot volume. Some caution must be observed with NDA's... NDA's that access disks or ones that show GS/OS or P16 system status will probably hang or crash while Call Box BASIC is active. As an example, one of our machines has the following NDA's in it: **Memory, Analog Clock, Clock, System Control and Control Panel.** Everything operates except the Control Panel NDA and the STATUS function of the System Control NDA. Trial and error will sort out what will work and what will not.

Call Box BASIC Program Structure

The file **CB** must be run to startup the Call Box BASIC driver. This file loads into bank 0 at $2000 and then executes itself. When it is done this area is freed up and can be used by Applesoft code. Many programs will be small enough to fit under this area and can run CB from within them. Larger programs should have a smaller program initialize CB and load in the entities, then run a second program (*which can be up to 31K in length*) which can subsequently **RUN** or **CHAIN** other programs... all without shutting down Call Box BASIC. **VARS** files can be used to pass variables from program segment to program segment and you need only shutdown Call Box BASIC when you **QUIT** or **BYE** from the program(s). This allows you to create Applesoft programs of incredible length and complexity. The loading of Entities, Fonts, Icons, Cursors, and Pictures from within any of the program segments will not interfer with even the largest of segments. The demo in the Call Box Launching Shell is a good example of this. The actual structure of the program code is really up to you... (*spaghetti code* works just fine!*)

Each Applesoft program segment must **RESTORE CB.VARS** if they are **RUN**, if these Applesoft segments are **CHAIN**ed then the variables are preserved from segment to segment and only an initial RESTORE CB.VARS is needed in the first segment.

Spaghetti code... Program code that is written without a plan, resulting in redundant routines, patches and entangled program flow.

OVERVIEW cont.

BASIC Concepts

The Apple IIgs has a vast memory area of which only a small portion is used for Applesoft BASIC. Applesoft does not run under ProDOS 16 or GS/OS and tools do not run under ProDOS 8... well this is not exactly true, most tools need to be installed under ProDOS 16, after that any system can use them as long as certain rules and restrictions are observed.

The Call Box BASIC driver can be thought of as a tool manager for Applesoft BASIC giving the novice and advanced programmer alike the ability to use most of the advanced features of the Apple IIgs with ease. The BASIC Interface also handles memory allocation/de-allocation and organization in those areas not under the control of ProDOS 8 or Applesoft. Namely bank 1 and up.

The memory area above bank 0 can contain tools, desk accessories, handles, pointers, flags and various and sundry pieces of data or code that makes your IIgs what it is. There are special memory areas and ROM up there as well. You do not need to know about these things to create Call Box driven programs, but if you want more information try the Apple IIgs reference manuals published by Addison-Wesely as a starting point.

Entities

The Call Box BASIC driver has to have a way of knowing what type of data it is dealing with be it a font, icon, window, dialog or menu etc... To accomplish this task, the data is broken down into a particular type called an entity. Each entity has a kind and an I.D. associated with it. You need only use the entity I.D. when writing your programs but each kind of entity is restricted in the number of I.D's it can have so it is important to know these as well.

Type	Kind	I.D.	Description of reserved I.D.'s
GrafPort	Port	0-31	Port 0 = SHGR screen in $E1
Window	Port		
Dialog	Port		
Menu	Port		
Font	Image	0-15	Font 0 is Shaston 8
Cursor	Image	0-15	
Icon	Image	0-63	

Figure 1.1 Entity types

The Entities come in 2 kinds:

Port A port is a special record that Quickdraw II keeps to define the drawing environment. Graphic ports are always 200 x 320 or 200 x 640 pixels in size, depending on the screen mode. Ports need graphic images drawn to them, otherwise their contents appear as a blank rectangle. Drawing takes the form of either loading in a pixel image or filetype $C1 picture from disk, or drawing directly to the port using Quickdraw II commands. Ports are also Managed screen items such as Windows, Dialogs and Menus, all of these items are handled in similar ways by the toolbox managers.

Image An image is a data segment that can reside anywhere in memory "as is" and has no absolute references. The file format for images is binary.

Port entity I.D. 0 is reserved for the super hi-res screen. This allows up to 31 port entities that can be defined elsewhere in memory. Each image entity has its own set of I.D.'s (*0 - 15 with some pre-defined, refer to* **Figure 1.1**) so there can be **16** of each type of image entity with the exception of Icon image entities of which there can be **64** in addition to the **32** port entities already mentioned.

You do not need to worry about where the entities you will use in your program reside in memory. The Call Box BASIC driver manages this for you, however, you must specify the particular I.D. number of the entity you wish to use so the BASIC driver will know where to look as well as determine the kind of entity being dealt with.

Bank Zero Memory Use

The Call Box BASIC driver and various tools need some pages in bank zero. Normally, these pages would be dynamically allocated by the memory manager (*Toolset #2*). Applesoft BASIC and ProDOS 8 use most of the available memory in bank zero. The only safe and trackable area of bank zero common to both P8 and the memory manager is the area between ProDOS 8 and its buffers, see **Figure 1.2.** The exact address at which this allocation takes place can vary depending on how the memory is being used when the Call Box BASIC driver is initialized. If another application is using the same area, the BASIC driver will fit around it automatically. Additional memory for other tools is allocated between the already allocated pages and the ProDOS buffers. This causes the ProDOS buffers and HIMEM to move down in memory.

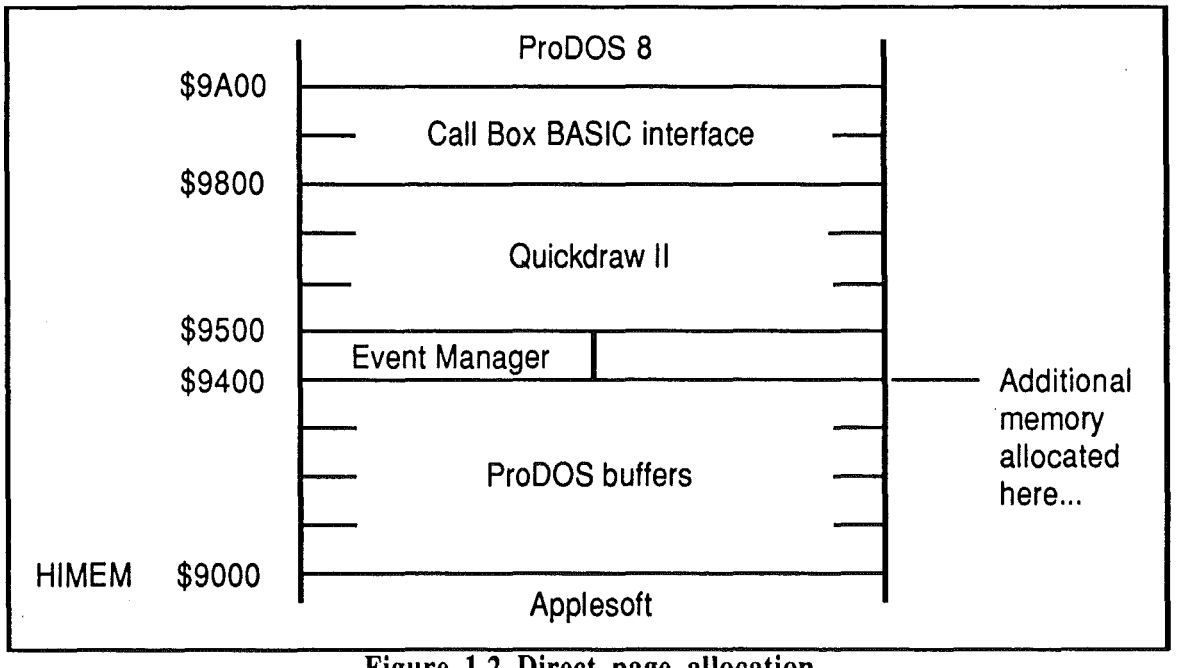


Figure 1.2 Direct page allocation

User Buffer

Every program needs some space to use for storing bits of data and to use as a "*scratchpad*" workspace. This space is located at the bank zero (*this means that you can directly peek and poke it*) address specified by variable **UA** and is as big as variable **UL** specifies.

When using the Long Call command (*see CALL LC*) you will often need to pass a pointer to a string or for a place to put a string. You would use the User Buffer for this purpose. Sometimes you need to pass a pointer to a pattern or table. You would write the pattern or table in the User Buffer and then pass the address of the buffer to the Long Call routine.

By using the User Buffer and Long Call you can operate the toolbox with a precision only found in assembly language.

Making a Call Box BASIC Desktop Application

A standard sequence of steps is used to create a desktop application, the first of which is to make a **plan**! This plan looks like drawings, each one being a different screen display which are linked by a logic diagram describing how the program will operate (*the exact form that the plan takes is up to you, this is just my own personal conceptualization*).

From this plan you use the **Editors** in the **Call Box Launching Shell** to create the entities you will be needing based on your drawings. Make sure that when you create an **entity**, you record any I.D. numbers associated with controls or items it may contain. You will need these for your Applesoft code.

You now would enter **Call Box BASIC** and start writing your program. This program will start off by starting up Call Box BASIC, RESTORING CB.VARS, HOMEing the screen and setting up the super hi-res screen and the desktop. Next you would load all of the entities you will be using in your program and start-up the system menu bar and anything else that needs to be up at startup.

After this "*housekeeping*" is finished you call **TaskMaster** (*see CALL EV...*) and check the results to see where or what was selected by the mouse or the keyboard. The main thing in a desktop application is to check and see if the mouse was pressed in the system menu bar, you would also check and see if the click was in the close region of the currently active window as well as other things determined by your particular program structure. If none of these things have occured then you would loop back to the TaskMaster call. If you had a click in the system menu bar then you would run a subroutine indexed by the menu item I.D. returned to you by TaskMaster. When the subroutine was finished you would UnHilite the menu bar selection and loop back to the TaskMaster call. This type of action keeps occuring until you select the menu item that you have setup as Quit. At this point you would close any open entities, shutoff Call Box BASIC and either **END** or **BYE** the program.

The previous description is very generalized and simple... *however*...this is the fundamental structure for all or at least most desktop applications. Reference the commands Event, Window, Menu and Dialog in this manual for more exact programming examples and examine the programs on the Call Box disks (*the Applesoft ones...filetype BAS*) to get an understanding of how to get one of these things up and running.

Using Call Box BASIC

The principle behind the BASIC driver is twofold: to make toolbox calls accessible to Applesoft and to make them as simple as possible to use. In order to keep things orderly, while not compromising on power and flexibility, the following method is used. To access the toolbox, you issue a **CALL** statement from BASIC. Some CALLs are predefined to access a particular tool function, while some allow you complete access to the entire toolsets, as long as you understand how the toolset works.

CALLs take the form of a variable name and parameters if necessary.
For example:

```
100 CALL QF
```

This command would shutdown the BASIC interface (*QF stands for Quit Function*) whereas:

```
100 CALL SC,1
```

This command would turn on the super hi-res screen (*SC for Screen*) but more on commands later.

The BASIC driver is the file named **CB**. It is a "System" file of type $FF. You talk to the BASIC interface thru the variables in the file **CB.VARS** which is an Applesoft/ProDOS 8 variable file. The BASIC driver (*CB*) is initialized by running the file CB directly from the keyboard or from within a program:

```
PRINT CHR$(4) ;"-CB"
```

This installs the Call Box BASIC driver, allocates some memory, starts-up the Tool Locator, Memory Manager, Misc. Tools, Integer Math, Quickdraw II, Event Manager, and Quickdraw II aux. Other tools are started-up as needed using the **CALL TL** (*tools*) commands.

CB bootstraps in from address $00/2000 so if you are executing it from a program, the size of the program should be smaller than $1800 (to allow room for simple variables) or the -CB will overwrite the BASIC program. Once CB has finished executing, the entire range from $800 to HIMEM is free for Applesoft BASIC.

A usual tactic for installing **CB** is from within a short **STARTUP** program. This technique is just fine, but it should be noted that editing while CB is active causes several bus problems which, while not terminal, can be quite annoying - like, no more repeat key function. CB will shutdown itself automatically when a non-Applesoft error occurs but you will have to handle the **CALL QF** in all other circumstances.

The other half of this equation is the file**CB.VARS**. Every program that uses CB needs to include the following line before any CB calls are made:

```
PRINT CHR$(4) ;"RESTORE CB.VARS"
```

This installs the variables that you need for the Call Box BASIC driver.

Many times when you are making an Applesoft program it is necessary to hit the old **CONTROL-RESET** and break out of some endless loop. This type of event is quite destructive to the tools and will most probably result in a bouncing apple system error message **$0206**... not the Applesoft cursor like you expected.

The last little hint is to be sure to issue a **CALL QF** before exiting to anywhere... This is analogous to putting your toys away, If you don't you will get away with it for a little while but later on when your father (*GS/OS*) gets home he'll CRASH the whole system!

Error Messages

Errors are returned in text mode regardless of the display mode when the error occured. A typical error message would look like this:

```
Tool not supported
Error in line ->75
]
```

You will be in Applesoft immediate mode and all variables will be null and the Call Box BASIC driver will be shut down.. You will have to re-initialize the Call Box BASIC driver to run your program again.
The following is a list of Call Box generated error messages... Applesoft returns its own messages and can be distinguished from Call Box error messages by their appearance. Presently, you cannot trap Call Box errors.

I/O Error
Pathname has invalid syntax
Path to files subdirectory is bad
Volume directory not found
Damaged disk
Access refused
Disk full
Disk is write protected
Font not found
Template not found
Resource not found
Out of memory (mem.mgr)
Tool not found
Bad parameter
Tool not supported
Not Dialog
Not Window
Not Menu
Entity is already assigned

Command Structure

This section describes the Call Box BASIC interface commands. These commands are arranged in alphabetical order. Before you look at the commands let's review the command syntax first.

Call Box commands are Calls to a global page address followed by parameters, separated by commas. The parameters can be of several different types as described in **Figure 1.3** and the global page addresss are automatically installed by **RESTORE**-ing **CB.VARS** described in **Figure 1.4.**

A typical command line may look like this...

```
CALL SC,0,640 : CALL SB,0,!10000000,0,200 : CALL SC,3,$FFFF
```

This line sets the screen mode to 640, sets the SCBs to use 640 mode and palette 0, and finally clears the screen to white.

CONSTANT:	
	Decimal (0123456789)
	Hex ($0123456789ABCDEF)
	Binary (!01)
VARIABLE:	
	Floating point (A.. ZZ)
	Integer (A%.. ZZ%)
STRING:	
	String ("string text")
	String variable (A$.. ZZ$)
MATH EXPRESSION:	
	Must start with a numeric value and may not contain hex or binary representations.

Figure 1.3 Legal Parameter Types

The types can be mixed in a single call without incident. The numeric limits are the same as Applesofts.

AR = Arc	LN = Line	PO = Big Poke	SC = Screen
AY = Super Array	ME = Menu	PT = Port	TL = Tool
CU = Cursor	OV = Oval	QF = Shutdown	TX = Text
DI = Dialog	PE = Big Peek	RE = Rectangle	UA = Buffer Addr.
EV = Event	PL = Palette	RR = RRectangle	UL = Buffer length
LC = Long Call	PN = Pen	SB = SCB's	WN = Window

Figure 1.4 CB.VARS variable equivalents

OVERVIEW cont.

A command has 3 basic parts:

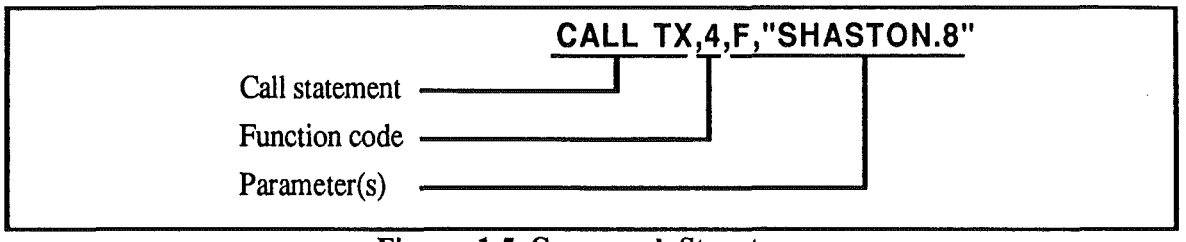


Figure 1.5 Command Structure

Call statement: This is a standard Applesoft CALL statement using the variables supplied by the CB.VARS file.

Function code: Most calls have function codes. The function code identifies the particular operation you want.

Parameters: This is any data needed to complete the call such as color number, height, width, mode, etc. etc... All parameters are separated by commas.

Be sure to RESTORE CB.VARS the very first thing in your program to assure that the Call variables are available... changing a program line, issuing a CLEAR or a NEW will wipe out these variables.

From within an Applesoft program:

```
PRINT CHR$(4) ;"RESTORE CB.VARS"
```

or from the keyboard directly:

```
RESTORE CB.VARS
```

Observe the usual rules of good programming in Applesoft BASIC like, put frequently used subroutines at the beginning of your program, don't go overboard with REM statements and use the X = FRE(0) to clean up string storage when you are concatenating strings a lot and so on.

The following pages describe in detail each Call Box BASIC driver command.

CALL AR the ARC commands

There are **5** commands that draw arcs. The arcs are drawn with the pen in the current port.

Note: Any command that draws something, draws to a graphics port. You must make sure that the port you want to draw in is the current port before drawing to it. Because you are drawing to a port and not necessarily the screen, the coordinates used to draw are in a coordinate system unique to the port you have selected. This coordinate system is called **LOCAL. GLOBAL** coordinates on the other hand refer to the screen coordinates... or the current cursor position. If you choose to draw to the screen then the **LOCAL** coordinates will equal the **GLOBAL** coordinates.

CALL AR,0,X,Y,W,H,SA,AA	**FRAME ARC:** draws an arc outline in the current pen color.
CALL AR,1,X,Y,W,H,SA,AA	**PAINT ARC:** draws an arc and fills it with the current pen color.
CALL AR,2,X,Y,W,H,SA,AA	**ERASE ARC:** draws an arc and fills it with color 0.
CALL AR,3,X,Y,W,H,SA,AA	**INVERT ARC:** draws an arc and inverts the pixels colors.
CALL AR,4,X,Y,W,H,SA,AA,P	**FILL ARC:** draws an arc and fills it with the current pen pattern.

X
Left edge of the enclosing rectangle for the arc in LOCAL coordinates.

Y
Top edge of the enclosing rectangle for the arc in LOCAL coordinates.

W
Width of the enclosing rectangle for the arc in pixels.

H
Height of the enclosing rectangle for the arc in pixels.

SA
Starting angle in degrees

AA
Ending angle in degrees

P
Pattern used to fill arc (*0-15*)

CALL AR the ARC commands *(continued)*

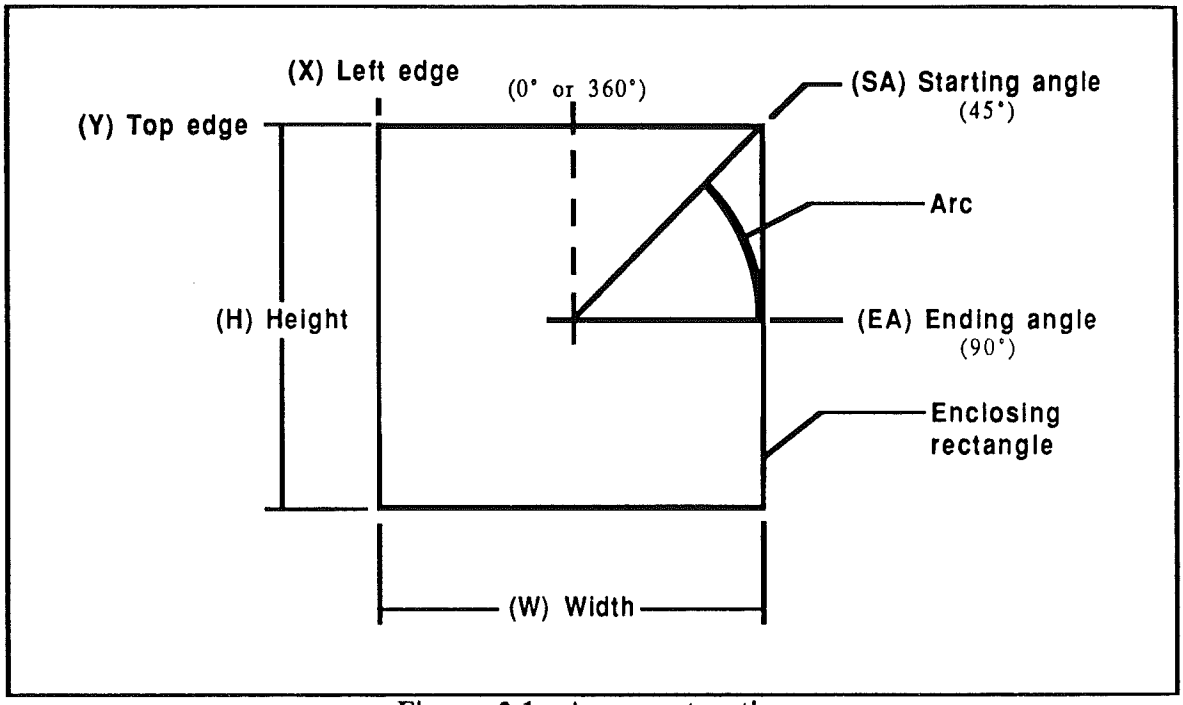


Figure 2.1 Arc construction

Arcs are constructed inside of an imaginary rectangle (*Enclosing rectangle*) specified by **X,Y,W** and **H**. The starting angle (**SA**) and the ending angle (**EA**) are specified in degrees (*0° to 360°*). 0 degrees is at the 12 o-clock position and the angle increases clockwise around the center of the enclosing rectangle, refer to **Figure 2.1**. Arcs made in 320 mode will follow a circular path if the width and height are the same, however arcs created in 640 mode will have to have the width twice the height for the same effect.

Paint, Erase, Invert, and **Fill** will create filled arcs as shown by the (*pie-wedged*) example in **Figure 2.1**. **Frame** will produce just the curved portion of the arc.

Specifying 0° for the starting angle and 360° for the ending angle will produce an Oval or Circle depending on the shape of the enclosing rectangle.

CALL AY the SUPER ARRAY commands

Due to the limited Applesoft program code area (*about 31K*) it is advantageous to put as much of the programs support data as possible out of this area (*bank 0*). Super Arrays allow you to put all of your Arrayed data up in the upper banks of your computers memory, this frees up a lot of room down in bank 0 and allows for bigger program code. These arrays also provide the capability of much larger arrays than was possible before... their size is directly dependent on how much memory your IIgs has available.

There are **5** commands that control arrays.

CALL AY,0,N	**UNDIMENSION ARRAY:** Remove an array from the array table.
CALL AY,1,N,{0,1,...,87}	**DIMENSION ARRAY:** Reserve memory for an array (*N{0,1,...,87}*).
CALL AY,2,N,{0,1,...,87},V	**GET VALUE:** gets a value (*V*) from array (*N{0,1,...,87}*).
CALL AY,3,N,{0,1,...,87},V	**SET VALUE:** sets a value (*V*) in array (*N{0,1,...,87}*).
CALL AY,4,N,V	**SET ALL VALUES:** sets all entries in array (*N*) to value (*V*).

N
The name of an array. Should be a valid Applesoft type variable (*real, integer, or string*). Once a type is set,
all subsequent calls to that array should use the same type and same name for the array.

{0,1,...,87}
Array subscripts. Use to define the size of the array or to get/set a value. Each number represents a particular element within the array. You must specify at least one element. Subscripts can be any valid numeric type (*including decimal, binary, and hexadecimal*) and are limited in size to 64K (*although your particular memory configuration may impose a greater limit*). Also, you are limited to only 88 dimensions.

V
Array value. Should be a variable of the same type as the array. In other words, if your array is a string array then V should be a string variable.

Super Array Operation

Super Arrays follow similar conventions to Applesoft arrays. However, you have the added abillity to "**undimension**" a super array (*i.e. de-allocate the memory associated with it*) and to set all values in a super array to the same value. To use a super array use should first use the dimension command. You would define a real array called "A" in the following manner:

```
100 CALL AY,1,A,{4,4,4,4}
```

The above line will dimesion a 5 by 5 by 5 by 5 element array for a total of 625 elements. If you

want to set all the values in this array to 0 you would use the Set All Values call with a value of zero.

```
110 CALL AY,4,A,0
```

Suppose you wanted to set the value of the {1,0,3,2} element to 15. Use the Set Value call to do this:

```
120 CALL AY,3,A,{1,0,3,2},15
```

If you wanted to examine the same element you would use the Get Value call as follows:

```
130 CALL AY,2,A,{1,0,3,2},V
```

The value of the element will be in the variable "V." Make sure that any variable or value used to get or set a value should be of the same type as the array. Otherwise you will get a "Wrong Super Array Type" error. If you wanted to dispose of a super array to free up some memory, you would use the undimension array call.

```
140 CALL AY,0,A
```

That's all there is to it! You can have as many super arrays as you have variables for within memory constraints (*in addition to any Applesoft arrays your program has dimensioned*).

CALL CU the CURSOR and ICON commands

There are **6** commands that control cursors, and **2** commands the control icons. There are 2 system cursors, the primary cursor is the **arrow cursor** and the second is the **wait cursor** (*wrist watch*). You can load up to **16** more cursors making a total of **18** possible cursors in any given application. You can have **64** icons as well.

Cursor #0 is the arrow cursor and Cursor #1 is the wait cursor. Cursors 0 thru 15 are available for loading. There are no standard icons so icons 0-63 are available for loading.

Cursors and icons can be created and edited using the **CALL BOX Image Editor.**

CALL CU,0	**CURSOR OFF:** this makes the current cursor invisible.
CALL CU,1	**CURSOR ON:** this makes the current cursor visible.
CALL CU,2	**ARROW:** this makes the current cursor the system arrow cursor.
CALL CU,3	**WAIT:** this makes the current cursor the system wait cursor.
CALL CU,4,N	**SET CURSOR:** this sets the current cursor to the user defined cursor (*0-15*).
CALL CU,5,N,"*pathname*"	**LOAD CURSOR:** this will load a cursor from disk as cursor I.D. 0-15.
CALL CU,6,N,M,X,Y	**PLOT ICON:** this plots an icon (*0-63*) at coordinates X and Y in mode M.
CALL CU,7,N,"*pathname*"	**LOAD ICON:** this loads an icon from disk as icon I.D. 0-63.

N
User cursor or icon I.D. number. There are 16 possible I.D. numbers for cursors (*0-15*) and 64 for icons (*0-63*).

"*pathname*"
ProDOS pathname for the cursor or icon file to load.

X
Horizontal icon plotting position.

Y
Vertical icon plotting position.

M
Icon plotting mode.(*see* **Figure 2.2**)

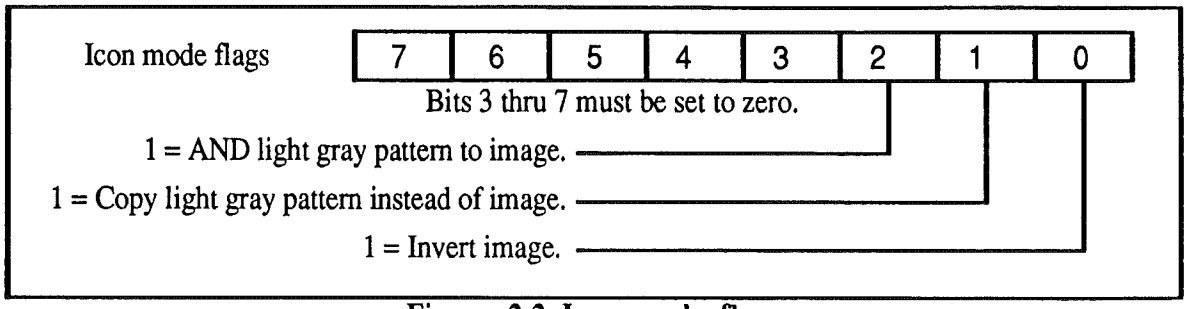


Figure 2.2 Icon mode flags

Cursors and Icons have similar structures (see **Figure 2.3**) but are handled by the IIgs toolbox differently. Cursors have several "*special*" considerations that need to be taken care of for them to work properly. The **Image Editor** chapter in the **Call Box Editors manual** outlines these peculiarities.

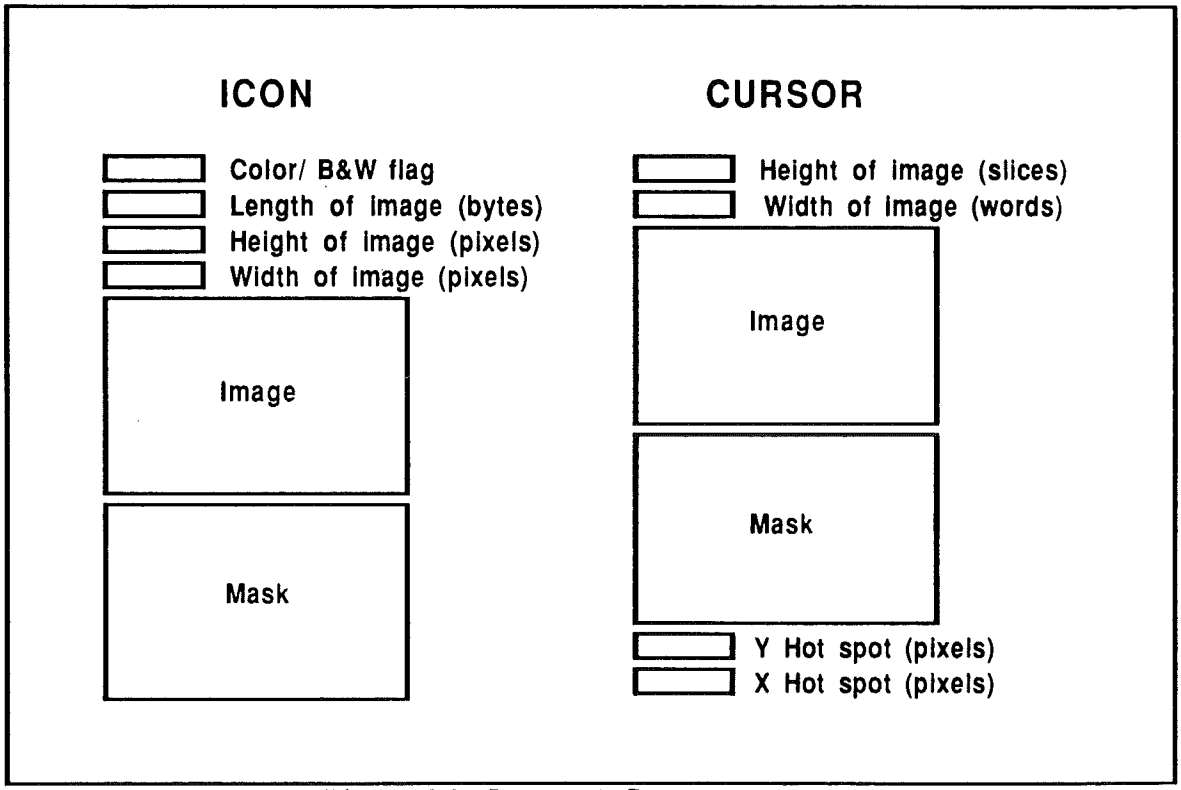


Figure 2.3 Icon and Cursor structures

CALL DI the DIALOG commands

Dialogs are high level tool functions which are dependent on other toolbox functions as well as GS/OS (*ProDOS16*) commands. The Dialog functions need to be initialized by using the "high level" startup command **CALL TL,2,"Desk"** (*see CALL TL in this manual for a complete description*). This call will startup all tools needed for desktop applications... the Dialog Manager is one of them.

Dialogs are designated as "*entities*" in the Call Box BASIC driver and these entities are created using the Call Box TPS **Dialog Editor**. The output type "*object*" must be used for dialogs that are to be used by the BASIC driver.

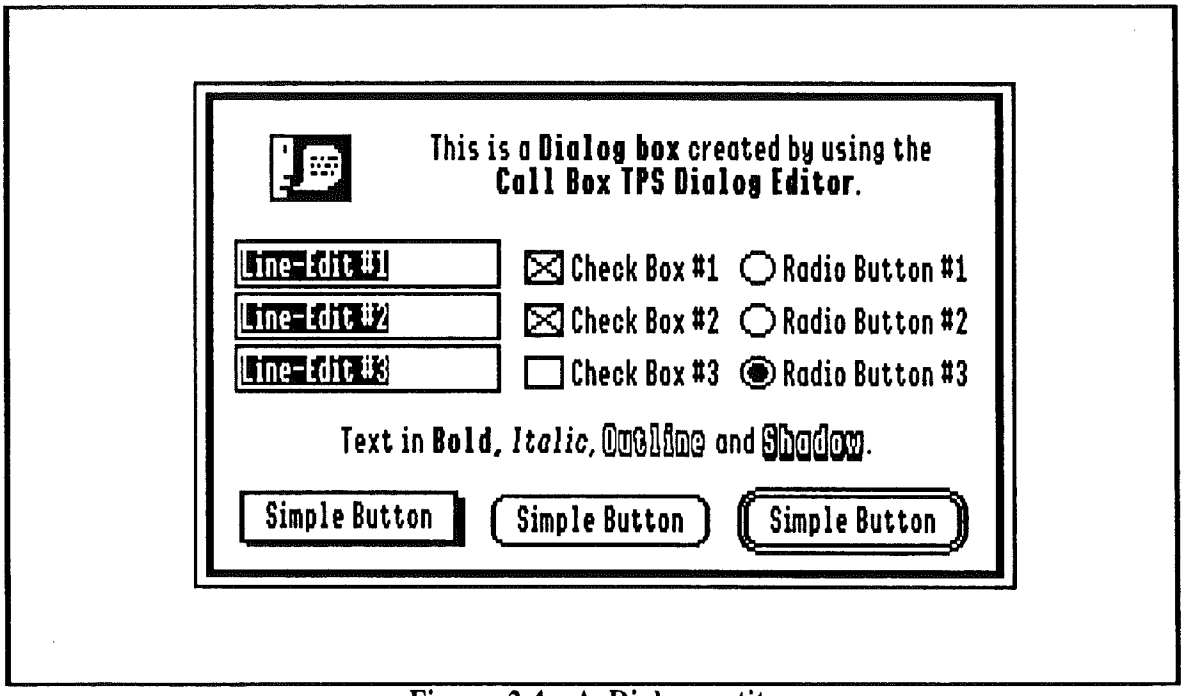


Figure 2.4 A Dialog entity

Operating a Dialog is semi-automatic. You must load it, open it and operate it. At this point the Dialog Manager has control of program execution and keeps it until you select something by clicking it or pressing the return key. Once a selection is made control is passed back to your application as well as the I.D. number for the item selected. Your application can take action based on this I.D. number and either close the Dialog or go back to it for further actions. This is a "**Modal**" type of access and is the most common type used with Dialogs. There is also a "**Modeless**" type of dialog (*not supported directly*) which allows you to choose items such as menu bar selections or other open modeless dialogs at the same time your Dialog is open. Open modal Dialogs must be closed before other desktop actions can take place.

The dialog can contain several types of controls (see **Figure 2.4**), each of which serve different purposes and are outlined as follows:

CALL DI the DIALOG commands *(continued)*

Simple button: This type of control is used to select an action. Simple buttons contain the text of the action to take such as "**Continue**", "**OK**", "**Load**", **Save**" etc. A simple button with a double outline is the default button and aside from responding to a mouse click it will also respond to pressing the return key. The I.D. number of this type of button is always 1.

Check box: This type of control is used to select an "**ON-OFF**" type of status such as which items out of a group of items should be enabled. Each check box has some text associated with it which describes the significance of the check box.

Radio button: This type of control is used like a check box except that only one item out of a group of items can be selected at any one time (*like the buttons on a car radio*). A group of buttons is called a "**family**" and a dialog can contain several families of radio buttons where only one button in each family can be set at any given time.

Line Edit: This type of control is used to enter text. This control obeys the standard Apple rules for text entry like click to position the typing cursor, double clicking to select a word or triple clicking to select a complete sentence. The delete key will remove all selected text and text is entered directly from the keyboard in insert mode.

Icon: This type of control is not really a control but a picture instead. Its purpose is purely decorative or used as a symbolic title such as a "**stop sign**" or "**caution sign**" to alert the user of possibly destructive actions.

Static Text: This is not a control either, but is a word or phrase used to identify the dialogs purpose. This would probably be the title of the dialog box.

Note: Functions such as **Hi-liting** or **dimming** controls is accomplished by using direct Control Manager commands via the **CALL LC** command. The use of "*hook*" procedures is possible as well by using the CALL LC command. The vast majority of applications will need to use the CALL DI commands for all of their functions, the more exotic control procedures however are possible but this manual will not describe them. Use the **Apple IIgs Toolbox reference manuals** (*vol. 1,2 and 3*) which outline all of the toolbox commands for more sophisticated programming procedures. These manuals are essential for a complete understanding of the vast number of toolbox calls available.

Dialog Controls (Items) and Item I.D.'s

The Call Box TPS Dialog Editor allows you to create dialogs by arranging dialog controls (items) in a dialog window. These items each have a unique I.D. number automatically assigned by the editor. It's important that you know what these item I.D.'s are because after you select something in a dialog the I.D. number of what you selected is returned to you and you will need to take some action based on which item I.D. is returned.

CALL DI the DIALOG commands *(continued)*

.The I.D. numbers assigned by the Dialog Editor are shown in **Figure 2.5.**

Type	I.D.
Simple Button	1 - 16
Radio Button	17 - 32
Check Box	33 - 48
Icon	49, 50
Line Edit	51 - 66
Static Text	67 - 82

Figure 2.5 Dialog I.D.s

Check boxes and **Radio buttons** do not hilite and un-hilite automatically. Each time one of them is selected control passes back to you and you must check or un-check etc...by setting (1) or un-setting (0) the items value by using the**SetValue** and**GetValue** commands. Once you have handled one of these items you loop back to the **OperateDialog** command and wait for another event. **Line Edit** items will operate automatically but you must fetch their contents (*strings*) before you close the dialog box (*this applies to any item value you are interested in as well*) with the **GetText** command. **Simple buttons** also operate automatically which leaves us with **Icons** and **Static Text** which are not controls.

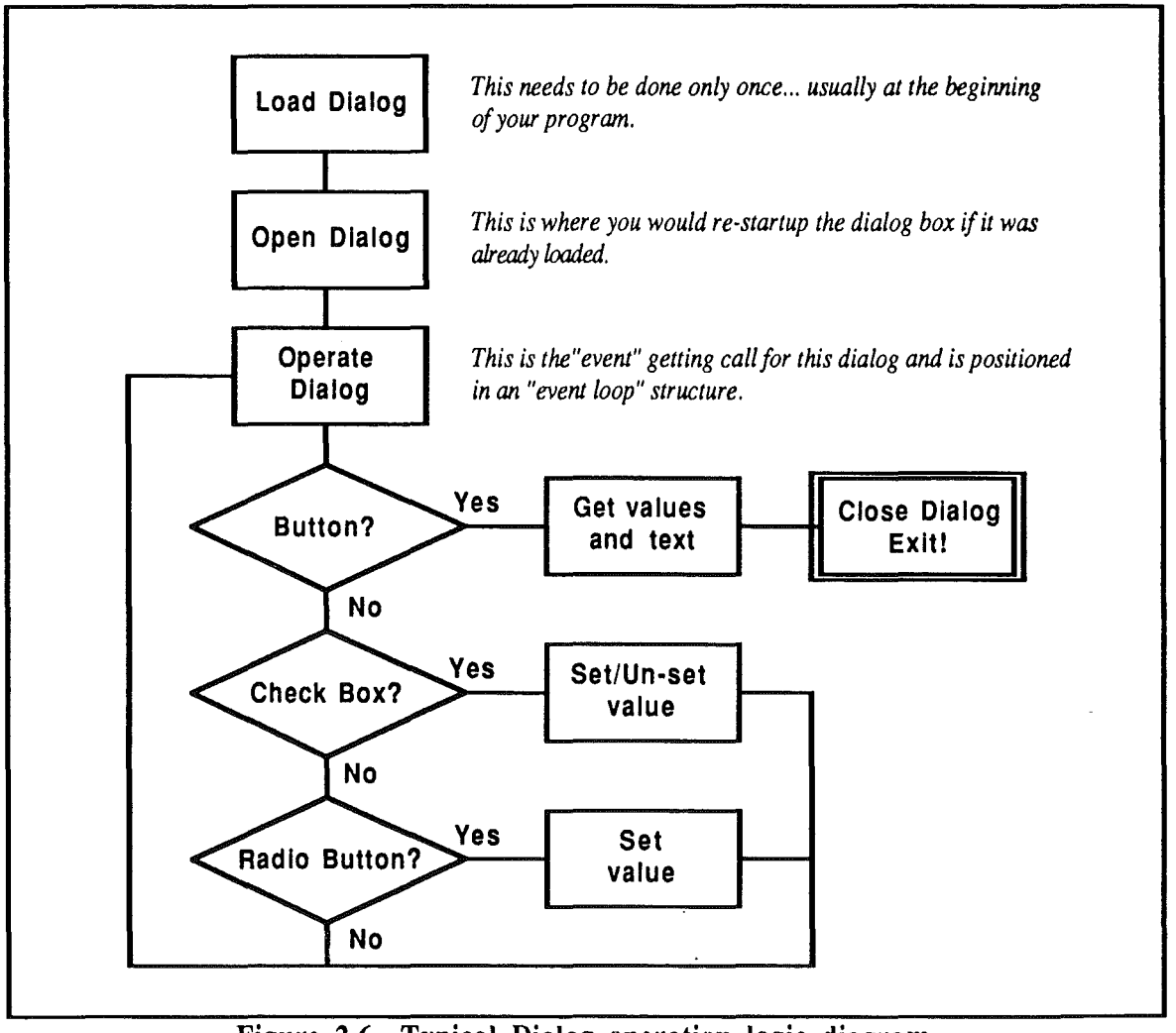


Figure 2.6 Typical Dialog operation logic diagram

CALL DI the DIALOG commands *(continued)*

Dialog Operation

The logic diagram in **Figure 2.6** shows the normal operation of a dialog box. Sometime before you need it you load a dialog entity created by the Call Box TPS Dialog editor.

```
100 CALL DI,0,N,"MyDialog"
```

Note: At this point you could preset some values and text in the dialog box. Use the **SetValue** and **SetText** commands to preset item data before using your dialog box.

Some action in your program such as a menu selection occurs which calls for your dialog box to come up and you open the dialog (*which makes it visible on the screen*). **N** is the dialog boxes "Entity" or port number, for our purposes make **N = 1.**

```
110 CALL DI,1,N
```

The next step is to set up an "event loop" type of programming structure to operate the dialog from. This loop centers around the command **OperateDialog (CALL DI,2...).**

```
120 CALL DI,2,N,I
```

This call works like **CALL EV (Event)** command except that it maintains control of program execution until you select an item in the dialog box. Once you have selected something the I.D. number of that item is returned to you in the variable **I**. You must "test" this number to see which item you have selected by comparing it to known values. If your **I** value is greater than 0 and it's less than 17 then the item selected was a button, if this value is greater than 16 but less than 33 then a radio button was selected, if it's greater than 32 but less than 49 then it was a check box. (See **Figure 2.5**). If none of the above is true then loop back to the **OperateDialog** command.

```
130 IF I > 0 AND I < 17 THEN 200
140 IF I >16 AND I < 33 THEN 300
150 IF I >32 AND I < 49 THEN 400
160 GOTO 120
```

If line 150 is "true" (*A check box was selected*) then you need to toggle the state of the check box that was clicked. First get the items value with **GetValue (CALL DI,4...)** and then check if it is 0 or not. If it's 0 then set it to 1 by using the **SetValue (CALL DI,5...)** command. If it's 1 then set it to 0 using the **SetValue** command as well.

```
400 CALL DI,4,N,I,V : CALL DI,5,N,I,1-V : GOTO 120
```

If line 140 is true (*A radio button was selected*) then you need only set the items value to 1 (*the Control Manager will unset the currently set button in the radio family automatically*).

```
300 CALL DI,5,N,I,1 : GOTO 120
```

If line 130 is true (*A simple button was selected*) you will probably be exiting the dialog box such as if the **OK** or **Continue** button was clicked. Before you do this remember to save all the current values of the items in the dialog box, including the text strings in any line edit items by using the **GetValue** and **GetText** commands for each item of interest. After you have retrieved your data you must close the dialog box.

```
200 REM  Get all the returned values here...
210 CALL DI,3,N : END
```

The dialog will stay in memory until you shut down your program and is restartable by simply opening it again. The settings made to radio buttons and check boxes as well as the text in line edit items is preserved from usage to usage. Your dialogs will remember their previous settings.

Dialog Commands

There are **9** commands that control dialogs. Dialogs must be used only when the**desktop** is active. (*See CALL WN in this manual for a complete description*)

CALL DI,0,N,"*pathname*"	**LOAD DIALOG:** this will load a dialog from disk as entity (*N = 1-31*).
CALL DI,1,N	**OPEN DIALOG:** this makes the current dialog visible.
CALL DI,2,N,I	**OPERATE DIALOG:** this routine returns the item I.D. (*I*) of the item clicked in the dialog.
CALL DI,3,N	**CLOSE DIALOG:** this removes the (*N*) dialog from the screen.
CALL DI,4,N,I,V	**GET VALUE:** this gets the value of item (*I*) returned in variable (*V*).
CALL DI,5,N,I,V	**SET VALUE:** this sets the value of item (*I*) specified by variable (*V*).
CALL DI,6,N,I,A$	**GET TEXT:** this gets the text contained in line edit item (*I*) and puts it in variable (*A$*).
CALL DI,7,N,I,A$	**SET TEXT:** this sets the text in line edit item (*I*) to the variable or string (*A$*).
CALL DI,8,N,P	**GET POINTER:** this returns the pointer to the current dialog.

CALL DI the DIALOG commands *(continued)*

N
Entity number for this dialog *(1-31)*

I
Item number of an item in the current dialog box. (*See* **Figure 2.5**)

V
The value of an item... 1 = checked (*check box*), hilited (*radio button*) and 0 = unchecked (*check box*),unhilited (*radio button*).

"pathneme"
This is the ProDOS pathname of a dialog (*filetype $B1... OBJ*) created with the Call Box TPS Dialog Editor.

P
This is the pointer to the current Dialog Box. This pointer is needed by various toolbox calls accessable thru the Long Call (*CALL LC*) command.

CALL EV the EVENT MANAGER commands

There are 2 commands that control the Event Manager. These calls are used to get information on the system such as if a key was pressed and what it was, or what were the mouse coordinates when the button was pressed. This first call is used when the desktop in not active.

GetNextEvent

CALL EV,X,Y,B,M,K,T

GET NEXT EVENT: returns the mouse coordinates, button status, modifier key code, standard key code and tick count.

X
Horizontal mouse position in GLOBAL coordinates.

Y
Vertical mouse position in GLOBAL coordinates.

B
Button status. 2 = down 0 = up

M
Modifier key code.(*see* **Figure 2.7**)

K
Keypress code equals an ASCII character value less than 128. Values greater than 128 represents a repeat key event.

T
Tick count.

To determine if a double click of the mouse has occured use **CALL EV** three times as follows.

```
10 CALL EV,X,Y,B,M,K,T : IF B <> 2 THEN GOTO 10 :REM Wait for click
20 CALL EV,X,Y,B,M,K,T1: IF B <> 0 THEN GOTO 20: REM Wait for up
30 CALL EV,X,Y,B,M,K,T1: IF B <> 2 THEN GOTO 20: REM Wait for down
```

Now that the time value between mouse up and mouse down is in the variables T and T1 respectively, use the longcall command **CALL LC** to execute the Event Manager Function **GetDblTime**. GetDblTime returns the maximum difference in ticks between mouse down and mouse up events allowed for a double click.

```
40 CALL LC,_0\$1106\_MT: REM MT = maximum tick value.
50 IF MT > T1 - T THEN PRINT "Valid Double Click": REM  double click
```

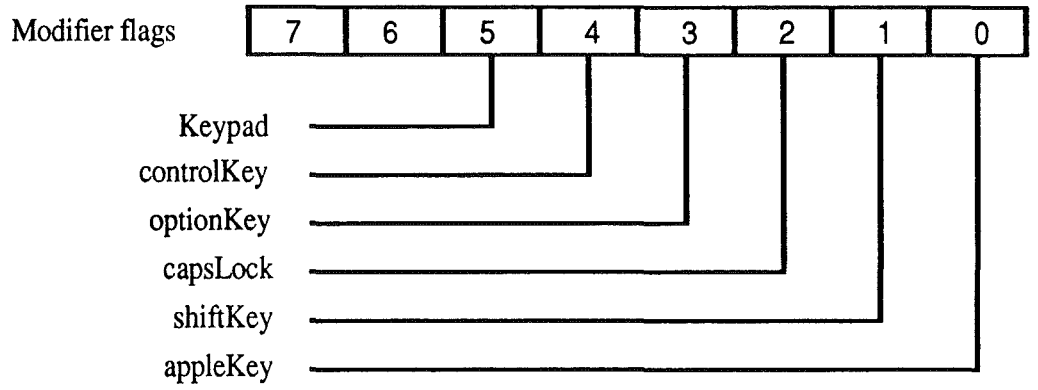
The value T1 - T holds the time between mouse up and mouse down events obtained in lines 10,20 and 30 above. The longcall command returns the maximum allowable tick time in the variable MT. (*For more information on CALL LC refer to that section in this manual.*) If MT is greater then T1-T, a valid double click has occured.

CALL EV the EVENT MANAGER commands *(continued)*

TaskMaster

The second Event call is a call to **TaskMaster** which **must** be used whenever the desktop is active... such as when you are using windows, menus and dialogs. The TaskMaster call returns all the information that the Get Event call does but it also supplies 2 more values which return desktop region information. The double click technique works with this call as well.

CALL EV,@,X,Y,B,M,K,T,C,D **TASKMASTER:** returns the mouse coordinates, button status, modifier key code, standard key code, tick count, TaskMaster code and data.



KeyPad: 1 = keypress on keypad 0 = keypress on keyboard
controlKey: 1 = control key is down 0 = control key is not down
optionKey: 1 = option key is down 0 = option key is not down
capsLock: 1 = caps lock key is down 0 = caps lock key is not down
shiftKey: 1= shift key is down 0 = shift key is not down
appleKey: 1 = apple key is down 0 = apple key is not down

Figure 2.7 Modifier key flags

16	In desktop	24	In info bar
17	In system menu bar	25	Item ID selected was 250-255
18	System click called	26	Item ID selected was 1-249
19	In content region	27	In window frame
20	In drag region	28	Inactive menu item selected
21	In grow region	29	Desk accessory closed
22	In go-away region	30	Inactive menu item selected
23	In zoom region		

Figure 2.8 TaskMaster codes

CALL EV the EVENT MANAGER commands *(continued)*

X
Horizontal mouse position in GLOBAL coordinates.

Y
Vertical mouse position in GLOBAL coordinates.

B
Button status. 2 = down 0 = up

M
Modifier key code.(*see* **Figure 2.7**)

K
Keypress code equals an ASCII character value less than 128. Values greater than 128 represents a repeat key event.

T
Tick count.

C
TaskMaster code.(*see* **Figure 2.8**)

D
Data, if the TaskMaster code is in the system menu bar then this value would be two words the first of which is the menu bar item number and the second is the menu item number selected. Parsing these numbers goes like this:
(menu item number) = **D - INT(D/65536)*65536**
(menu bar item number) = **INT(D/65536)**

If the TaskMaster code shows that a window item was selected then **D** is a pointer to the window. You can derive the entity number of the window by using this value in the Get Window Pointer/ Entity Number command in the window commands.

CALL EV the EVENT MANAGER commands *(continued)*

CALL LC the LONG CALL command

There is only **1** command associated with the LONG CALL. It allows you to directly access any tool you want. You must know the specific requirements of the tool being called in order to avoid problems, such as a system CRASH!

The basic format of the call is to pass any necessary parameters, pass the function number and tool set number, and to provide variables for any returned values.

CALL LC,(*par1*),*par2*),... ,(*parN*)**$TFTN**\(*var1*),(*var2*),... ,(*varN*)

(*par*)
A parameter that is to be passed to the tool. The call will pass values until it encounters the "\" delimiting character.

$TFTN
The function (*TF*) and the toolset number (*TN*) that is to be called. It is best to specify this value in hex.

(*var*)
The variables that will contain any returned values from the call to the toolset.

This is perhaps the most powerful call provided. However, you must know exactly what parameters the tool needs passed to it and what values it will pass back in order to use this call correctly. Otherwise unpredictable results may occur, one of which may be a system crash. (*This information can be obtained from the Toolbox Reference Manuals published by Addison-Wesely*).

Each parameter passed, through CALL LC is one word in length (*or 2 bytes*) unless otherwise specified. To specify a longword value (*4 bytes*) or variable, use the underline character "_" before the value or variable. In other words, a long value of zero would be denoted as "_0" and a long variable could be denoted as "_A".

Example:
Suppose you wanted to display in hex the total memory in your system on the super hi-res screen. You need to access three different tools to accomplish this. These tools are:

Memory Manager (always active)
TotalMem ($TFTN = $1D02)

IntegerMath (always active)
Long2Hex ($TFTN = $230B)

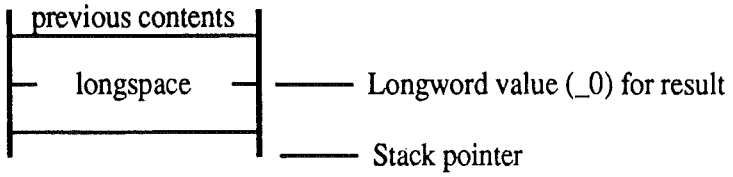
Quickdraw II (always active)
DrawText ($TFTN = $A704)

You will also need to use a few Call Box calls to set the background and foreground colors, as well as the vertical and horizontal position for the text plot.

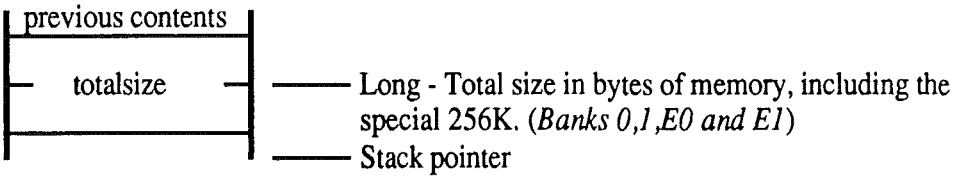
TotalMem
Find the total memory in your system using the Memory Manager function TotalMem ($1D02).

The toolbox reference manual specifies making this call in this manner:

Stack before call



Stack after call



If you don't know what the stack is or what it does, don't worry about it. All you need to know is the order of the parameters which is specified by the stack diagram. Therefore, to pass the parameter through the LONG CALL and return the memory size to your BASIC program you would do the following:

```
90 CALL LC,_0\$1D02\_S
```

The "_0" is a long "result space," which makes room for the result being returned. The "_S" is a long variable. The underline is just a convention that is used to denote that you expect a long value returned from the call. The variable name is actually "S" and you would use it just as you would any other variable.

At this point, you could actually convert the variable into an Applesoft string if you wanted to print out the value in decimal rather than hex. To do this, you would use the following statements:

```
100 S$= STR$(S) : REM Convert the variable to a string
110 CALL TX,1,0,15 : REM Set black text, white bkgnd.
120 CALL PN,2,40,20 : REM Set position to H=40, V=20.
130 CALL TX,0,0,S$ : REM Plot the string using def. font
```

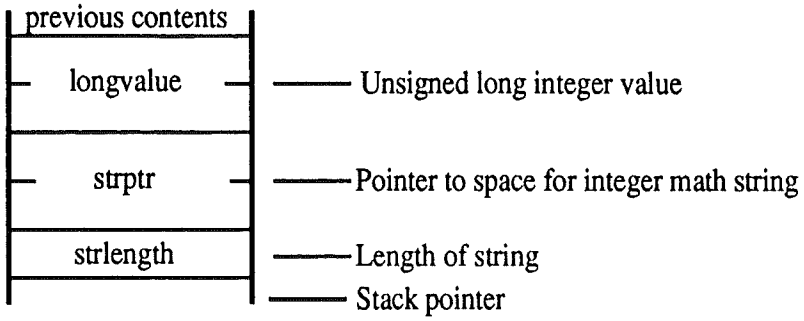
To print out the string in hex you would ignore the previous Applesoft program lines and continue as shown below.

CALL LC the LONG CALL command *(continued)*

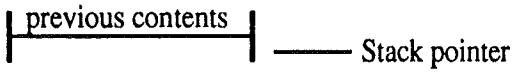
Long2Hex

This tool converts a Long Integer value into an Integer Math hex string. The toolbox reference manual specifies the call in this fashion:

Stack before call



Stack after call



No value is returned from this call. The value is instead put in a buffer, assigned by you. For our purposes, since no DOS operation will take place durring the plotting, we will use the User DOS Buffer as a temporary string storage area. To do this, we need to know where the buffer is located. The variable "**UA**" points to the memory location that holds the starting address of the buffer. We can use a long peek command to get the address into the variable "**BF**".

The calls look like this:

```
150 CALL PE,2,UA,BF : REM Get address of user buffer
160 CALL LC,_S,_BF,8\$230B\ : REM Make a Hex String
```

The long peek call "**CALL PE**" gets the two byte value (*denoted by the number two following the call statement*) which is stored at the memory location held by the variable **UA**. The two byte value that is returned from the call is placed in the variable **BF**.

The "**CALL LC**" takes the long word value of the variable "**S**", makes an Integer Math Hex string out of it, and places the string at the location held by the value of the variable "**BF**". The "**_S**" holds the Total Memory in your system. It is the long size variable returned from the TotalMem call above. The "**_BF**" holds the user buffer address which serves as our temporary string buffer. The "**8**" is the length of the string in characters (*hex digits are two characters each*).

CALL LC the LONG CALL command *(continued)*

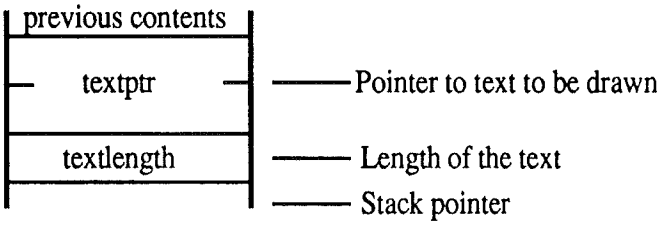
At this point, you need to set the current pen location as well as the foreground and background colors. To do this, you would use the following statements:

```
210 CALL TX,1,0,15 : REM Set black text, white bkgnd.
220 CALL PN,2,40,20 : REM Set position to H=40, V=20.
```

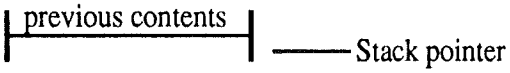
Note: You don't need to set the foreground and background colors each time you draw with the pen. They will retain the same values that they had the last time that you used them. This goes for the pen position as well.

DrawText
This tool function will draw specified text at the current pen location and updates the pen location.

Stack before call



Stack after call



This call will finish up by placing the string in the user buffer onto the Super Hi-Res Screen. The call looks like this:

```
300 CALL LC,_BF,8\$A704\
```

You now should have the size of your systems memory on the super hi- res screen in hex (*and/or decimal if you used the first example*)!

You can follow this format to make similar calls to any toolbox function provided you know the parameters expected by the tool. The best way to get this information is to use the Apple IIgs Toolbox Reference Manuals published by Addison-Wesely and available from A.P.D.A. (*Apple Programers and Developers Association*).

CALL LN the LINE command

There is 1 command that draws lines... this command will also draw points if you make the two sets of coordinates the same. The Line is drawn with the pen in the current port.

Note: Any command that draws something, draws to a graphics port. You must make sure that the port you want to draw in is the current port before drawing to it. Because you are drawing to a port and not necessarily the screen, the coordinates used to draw are in a coordinate system unique to the port you have selected. This coordinate system is called **LOCAL. GLOBAL** coordinates on the other hand refer to the screen coordinates... or the current cursor position. If you choose to draw to the screen then the **LOCAL** coordinates will equal the **GLOBAL** coordinates.

CALL LN,H1,V1,H2,V2

DRAW LINE: draws a line from H1,V1 to H2,V2 in the current pen mode and pattern.

V1
Vertical starting position

H1
Horizontal starting position

V2
Vertical ending position

H2
Horizontal ending position

CALL LN the LINE command *(continued)*

CALL ME the MENU commands

Menus are high level tool functions which are dependent on other toolbox functions as well as GS/OS (*ProDOS16*) commands. The menu functions need to be initialized by using the "high level" startup command **CALL TL,2,"Desk"** (*see CALL TL in this manual for a complete description*). This call will startup all tools needed for desktop applications... the Menu Manager is one of them.

Menus are designated as "*entities*" in the Call Box BASIC driver and these entities are created using the Call Box TPS **Menu Editor**. The output type "*object*" must be used for menus that are to be used by the BASIC driver.

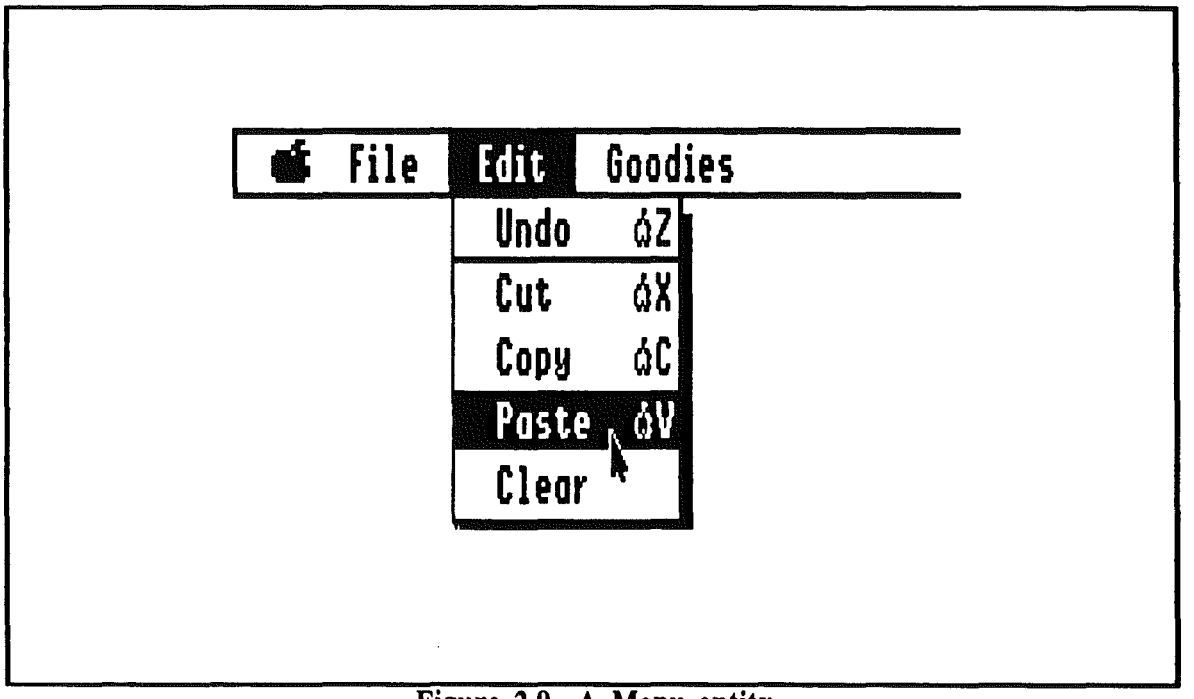


Figure 2.9 A Menu entity

There are different kinds of menus, some of which appear in windows, behind buttons (*pop-ups*) or in most any other location. The BASIC driver directly supports the "**System Menu**" which is the menu that appears across the top of the screen when you have a desktop application active. The first item in a system menu bar is the "**Apple**" selection. This menu bar selection is where **NDA's** can be selected from... simply have this selection avaliable and the BASIC driver (*and GS/OS*) will put all of the NDAs in your system/desk.accs folder in this menu.

Operating a Menu is automatic, you must load it and open it. The window manager actually operates the menu and the menu item numbers are returned with **TaskMaster** (*see CALLEV*) calls in the main loop of your program.

CALL ME the MENU commands *(continued)*

Menu Items and Item I.D.'s

I.D. numbers are automatically assigned by the Call Box Menu Editor when you create a menu. There are 3 ranges of I.D. numbers, each one handles a different group of menu items.

I.D.'s 1 thru 249 are reserved for menu bar items.

I.D.'s 250 thru 255 are for the **Standard EDIT menu** which supports NDA's with the functions UNDO, CUT, COPY, PASTE, CLEAR and CLOSE.

I.D.'s 256 and up are for menu items

If you create a menu bar which has (*from left to right*) colored Apple, File, Alter, Special, and Goodies then the File would be I.D. #1, Alter would be I.D. #2, Special would be I.D. #3 and Goodies would be I.D. #4. The menus that these menu bar titles select follow the same rules but the numbering starts at 256 instead of 1. Suppose that this menu had an About... item under the colored Apple, a Load... a Save... and a Quit under the File, Create and Destroy under the Alter and Midnight under Special. The I.D. for About... would be 256, Load... would be 257, Save... would be 258, Quit would be 259, Create would be 260, etc. etc.

Standard Edit Menu

This menu is avaliable to accomodate special functions reserved by Apple for the Human Interface of the toolbox. The functions **UNDO, CUT, COPY, PASTE, CLEAR** and **CLOSE** have been given special I.D. numbers (*250 thru 255*) which never change. Whenever a system window is up (*like NDA's*) these menu items should become selectable... when there is no system window active then they should be un-selectable (*dimmed*). **Check Menu** (*CALL ME,2,..*) will activate or de-activate these menu items depending on whether there is a system window active at the time. Place this call in your programs event loop if you are using a Standard Edit Menu.

The Un-Hilite Consideration

When a menu is "pulled-down" the menu bar item is hi-lited (*white on black or inverted*) and if you select a menu item the item blinks and then the menu closes. At this point you would do the task specified by the menu item. All this time however the menu bar item remains hi-lited and never un-hilites. This works that way so that you can see by looking at the menu bar that a process is still going on. When your task is complete you must un-hilite this menu bar item yourself... (*using the D value from the TaskMaster call*)

```
V = INT(D/65536) : CALL ME,7,N,V,0
```

Failure to do this will result in items randomly being hilited or un-hilited... a real mess, just remember to handle this little detail and everything will look good.

Menu Operation

After starting-up and initializing Call Box BASIC load a Menu entity created by the Call Box Menu Editor.

```
10 CALL ME,0,N,"MyMenu"
```

Next you need to build and display the menu, adding in any NDA's in the colored Apple menu bar selection.

```
12 CALL ME,1,N
```

Menus are actually operated by the Window Manager, all you do is check TaskMaster and see if a mouse click or event happened in the system menu bar (*Task Code = 17*) and take some action based on the menu item I.D. returned in the data variable of the TaskMaster call. If the Task Code indicates a 17 then derive the menu item I.D. and the menu bar item I.D. from the TaskMaster data value.:

(*Menu Bar item I.D.*) = **INT(D/65536)**
(*Menu item I.D.*) = **D - INT(D/65536)*65536**

Menu Commands

There are **10** commands that control menus. Menus must be used only when the **desktop** is active. (*see CALL WN in this manual for a complete description*)

CALL ME,0,N,*"pathname"*	**LOAD MENU:** this will load a menu from disk as entity (*N = 1-31*).
CALL ME,1,N	**OPEN MENU:** this builds and displays the system menu.
CALL ME,2,N	**CHECK MENU:** this hilites or un-hilites the standard Edit menu functions.
CALL ME,3,N	**CLOSE MENU:** this removes the menu from the screen.
CALL ME,4,N,V	**VISIBLE MENU:** this will show (*V = 1*) or hide (*V = 0*) the menu.
CALL ME,5,N,I,V	**ENABLE/DISABLE ITEM:**this will enable (*V = 1*) or disable (*V = 0*) the menu item specified by (*I*).
CALL ME,6,N,I,V	**SYMBOL ITEM:** this puts no symbol (*V = 0*) or a check mark (*V = 1*) or an ASCII symbol (*V = 2-255*) to the left of menu item specified by (*I*).
CALL ME,7,N,I,V	**HILITE/UNHILITE ITEM:** this hilites (*V = 1*) or unhilites (*V = 0*) the menu item specified by (*I*) .
CALL ME,8,N,I,V	**STYLE ITEM:** this sets the text style (*V*) (*see CALL TX*) of the menu item specified by (*I*).
CALL ME,9,N,P	**GET POINTER:** this returns the pointer to the menu manager port.

CALL ME the MENU commands *(continued)*

N
Entity number for this menu *(1-31)*

I
Item number of an item in the current menu.

V
The value to fetch or put for an item, or action to take.

P
The pointer of the menu manager port.

"*pathname*"
This is the ProDOS pathname of a menu (*filetype $B1... OBJ*) created with the Call Box TPS Menu Editor.

CALL OV the OVAL commands

There are **5** commands that draw ovals. The ovals are drawn with the pen in the current port.

Note: Any command that draws something, draws to a graphics port. You must make sure that the port you want to draw in is the current port before drawing to it. Because you are drawing to a port and not necessarily the screen, the coordinates used to draw are in a coordinate system unique to the port you have selected. This coordinate system is called **LOCAL. GLOBAL** coordinates on the other hand refer to the screen coordinates... or the current cursor position. If you choose to draw to the screen then the **LOCAL** coordinates will equal the **GLOBAL** coordinates.

CALL OV,0,X,Y,W,H	**FRAME OVAL:** draws an oval outline in the current pen color.
CALL OV,1,X,Y,W,H	**PAINT OVAL:** draws an oval and fills it with the current pen color.
CALL OV,2,X,Y,W,H	**ERASE OVAL:** draws an oval and fills it with color 0.
CALL OV,3,X,Y,W,H	**INVERT OVAL:** draws an oval and inverts the pixels colors.
CALL OV,4,X,Y,W,H,P	**FILL OVAL:** draws an oval and fills it with the current pen pattern.

X
Left edge of the enclosing rectangle for the oval in LOCAL coordinates.

Y
Top edge of the enclosing rectangle for the oval in LOCAL coordinates.

W
Width of the enclosing rectangle for the oval in pixels.

H
Height of the enclosing rectangle for the oval in pixels.

P
Pattern used to fill the oval (*0-15*)

CALL OV the OVAL commands *(continued)*

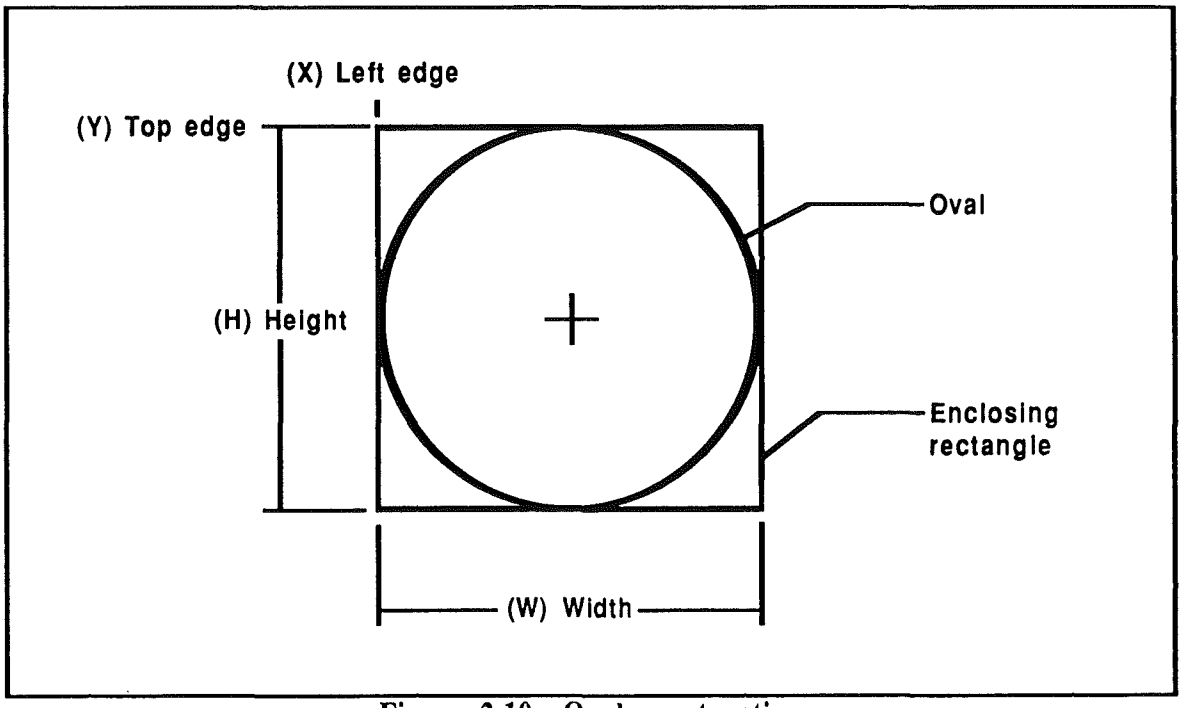


Figure 2.10 Oval construction

Ovals are constructed inside of an imaginary rectangle (*Enclosing rectangle*) specified by **X,Y,W** and **H**. (refer to**Figure 2.10**) Ovals made in 320 mode will follow a circular path if the width and height are the same, however ovals created in 640 mode will have to have the width twice the height for the same effect.

CALL PE the BIG PEEK command

This command allows you to peek values greater than 256. You can peek 1,2,3 or 4 byte values (*values up to 4.29E +09*). This value can be peeked from any address in the IIgs... (*16,777,214*). The number of bytes and address can be specified as decimal, hex, binary, integer or floating point constants and/or variables. The returned value must be an FP variable.

This commands primary use is for getting Handles, pointers and words. Handles and pointers have 4 bytes and words have 2.

CALL PE,D,A,V

D
Number of bytes used (*1-4*) to represent the value.

A
A IIgs memory location (*0-16,777,214*).

V
The returned value.

CALL PO the BIG POKE command

This command allows you to poke values greater than 256. You can poke 1,2,3 or 4 byte values (*values up to 4.29E +09*). This value can be poked into any address in the IIgs... (*16,777,214*). The value and address can be specified as decimal, hex, binary, integer or floating point constants and/or variables.

This commands primary use is for setting Handles, pointers and words. Handles and pointers have 4 bytes and words have 2.

CALL PO,D,A,V

D
Number of bytes used (*1-4*) to represent the value.

A
A IIgs memory location (*0-16,777,214*).

V
The value you want to poke into the memory location.

CALL PL the PALETTE commands

There are 4 commands that control color palettes. A color palette is 16 color values in a table. Each value in this table represents a color composed of different intensities of RED, GREEN and BLUE. You can have up to 16 of these tables accessable at one time depending on how the SCB's are set.

To set table number 3, entry number 6 to pure RED:

```
CALL PL,0,3,6,15,0,0
```

To set table number 3, entry number 6 to pure GREEN:

```
CALL PL,0,3,6,0,15,0
```

To set table number 3, entry number 6 to pure BLUE:

```
CALL PL,0,3,6,0,0,15
```

CALL PL,0,(*tbl*),(*ent*),**R,G,B**	**SET COLOR:** Set the color specified by R,G,B in the table entry.
CALL PL,1,(*tbl*),(*ent*),**R,G,B**	**GET COLOR:** Gets the color specified by the table entry in R,G,B.
CALL PL,2,(*tbl*)	**SET STANDARD PALETTE:** Sets the table specified to the standard palette for this screen. mode.
CALL PL,3, (*tbl*),"*pathname*"	**LOAD PALETTE:** loads a palette specified by pathname.

(*tbl*)
A value 0 - 15 that represents the table number.

(*ent*)
A value 0 - 15 that represents the entry number.

R
A value 0 - 15 that represents the intensity of RED.

G
A value 0 - 15 that represents the intensity of GREEN.

B
A value 0 - 15 that represents the intensity of BLUE.

"*pathname*"
A ProDOS pathname for the palette to load.

CALL PL the PALETTE commands *(continued)*

CALL PN the PEN commands

There are **9** commands that control pens. Each graphics port has its own pen. This pen has several attributes that determine how it will behave. Any drawing, be it a line, rectangle, oval, text or whatever is done with a pen. For example let's say that you want to draw a framed rectangle using pattern number 7 and have the vertical lines of the rectangle 2 pixels wide and the horizontal 1 pixel high:

First set the pen size to 1 x 2:

CALL PN,2,1,2

Now make the pen visible... (*able to draw*)

CALL PN,4,7

And draw the framed rectangle:

CALL RE,0,X,Y,W,H

This pen will keep these attributes until you change them to something else. If you draw to another graphics port then the attributes of that ports pen will be active which may or may not have the same attributes as the previous ports pen.

All pens start up with defaults of pen size = 1 x 1 and pen mode = copy.

Note: Do not confuse a pen with a cursor, even if it looks like a little pen! Pens do not have a graphically visible counterpart like a cursor but are conceptual in nature. You only see the trail left by a pen and not the pen itself.

Pens have so many commands because they are at the heart of all graphic goings on in the toolbox and many variations are needed to create the effects seen in a IIgs application. The pen can draw in any one of 8 different modes as shown in **Figure 2.11**. It is best to experiment with the different modes to get a feel for how they work.

CALL PN,0	**HIDE PEN:** makes the pen invisible.
CALL PN,1	**SHOW PEN:** makes the pen visible.
CALL PN,2,H,V	**MOVE PEN:** moves the pen to the coordinates specified by V and H.
CALL PN,3,(*color #*)	**SET PEN TO COLOR:** sets a color from 0-15 of the current palette.
CALL PN,4,(*pattern #*)	**SET PEN TO PATTERN:** sets a pattern from 0-15 of the current pattern.
CALL PN,5,WI,HT	**SET PEN SIZE:** sets the pens width and height.

CALL PN the PEN commands *(continued)*

CALL PN,6,(*mode*) — **SET PEN MODE:** sets the pen to any of the modes described in **Figure 2.11.**

CALL PN,7 — **RESET PEN:** sets the pen to default attributes.

CALL PN,8,"*pathname*" — **LOAD PATTERN:** loads a set of 16 patterns.

H
Horizontal pen position in global coordinates.

V
Vertical pen position in global coordinates.

(*color #*)
A number 0-15 that selects the solid color to use.

(*pattern #*)
A number 0-15 that selects the pattern to use.

WI
Width of pen in pixels.

HI
Height of pen in pixels.

(*mode*)
a number 0-7 that selects the pen mode (see **Figure 2.11**).

(*pathname*)
a ProDOS pathname of the pattern to load.

CALL PN the PEN commands *(continued)*

0 = COPY 1 = notCOPY This is the typical drawing mode.

COPY		Pen 0	Pen 1
Destination	0	0	1
	1	0	1

notCOPY		Pen 0	Pen 1
Destination	0	1	0
	1	1	0

2 = OR 3 = notOR This mode is used for non-destructive overlays.

OR		Pen 0	Pen 1
Destination	0	0	1
	1	1	1

notOR		Pen 0	Pen 1
Destination	0	1	0
	1	1	1

4 = XOR 5 = notXOR This mode is used for cursor drawing and rubberbanding.

XOR		Pen 0	Pen 1
Destination	0	0	1
	1	1	0

notXOR		Pen 0	Pen 1
Destination	0	1	0
	1	0	1

6 = BIC 7 = notBIC This mode is used to erase (*turn off*) pixels.

BIC	0	Pen 0	Pen 1
Destination	0	0	1
	1	1	0

notBIC	0	Pen 0	Pen 1
Destination	0	0	0
	1	0	1

Figure 2.11 Pen modes

CALL PN the PEN commands *(continued)*

CALL PT the PORT commands

There are 5 commands that control ports. A port is analogous to a super hi-res screen, it differs from a Quickdraw II port in that it has 16 palettes and 200 SCB's in it. Remember that everything goes on in ports, you need to create them before you use them and you need to point to them before drawing in them. Port 0 is always the super hi-res display screen at $E1/2000 and is initialized when the Call Box BASIC interface is started-up. Simple applications may only need this port for all of their graphics.

CALL PT,0,N	**SET PORT:** set all the Quickdraw II action to the port specified by N. *(0-31)*
CALL PT,1,N	**CREATE PORT:** creates a port in the current screen mode. You assign the I.D. number N. *(1-31)*
CALL PT,2,N,"*pathname*"	**LOAD PORT:** load or create/load a port filling it with a super hi-res picture from disk specified by pathname. *(0-31)*
CALL PT,3 ***	Unused... Reserved
CALL PT,4,N1,X1,Y1,N2,X2,Y2, W,H	**PORT TO PORT:** copy a specified rectangle of pixels from one port to a specified location in another port.
CALL PT,5,N	**DISPLAY PORT:** copy the picture in the specified port to the viewable super hi-res screen port at $E1/2000. *(1-31)*
CALL PT,6,N,X,Y	**GOLBAL TO LOCAL:** converts global coordinates (*X,Y*) to local coordinates in port (*N*).
CALL PT,7,N,X,Y	**LOCAL TO GLOBAL:** converts local coordinates of port (*N*) to global coordinates.

N
Port I.D. number 0-31. Port 0 is always the super hi-res screen display at $E1/2000.

"*pathname*"
A ProDOS 8 pathname of the picture you wish to load. This picture must be filetype $C1 (*unpacked super hi-res picture format*).

N1
The source port I.D. number.

N2
The destination port I.D. number.

X1
The left edge of the source rectangle.

CALL PT the PORT commands *(continued)*

Y1
The top edge of the source rectangle.

X2
The left edge of the destination rectangle.

Y2
The top edge of the destination rectangle.

W
Width in pixels of the rectangle.

H
Height in pixels of the rectangle.

CALL QF Shutdown the Call Box BASIC interface

A-CB starts up the **Call Box BASIC driver** and CALL TL,xx starts up special tools but when your done playing with your toys you need to put them away! **CALL QF** does this in one stroke. This call shuts down the **Call Box BASIC driver**, all the tools, and any memory blocks that have been allocated for your program.

CALL QF

Your program would usually end after CALLing **QF** by exiting to BASIC, running another program, or issueing a "**BYE**" to a P16 application (*like HyperLaunch or the Finder*) if that's who launched you.

CALL QF Shutdown the Call Box BASIC interface *(continued)*

CALL RE the RECTANGLE commands

There are 5 commands that draw rectangles. The rectangles are drawn with the pen in the current port.

Note: Any command that draws something, draws to a graphics port. You must make sure that the port you want to draw in is the current port before drawing to it. Because you are drawing to a port and not necessarily the screen, the coordinates used to draw are in a coordinate system unique to the port you have selected. This coordinate system is called **LOCAL. GLOBAL** coordinates on the other hand refer to the screen coordinates... or the current cursor position. If you choose to draw to the screen then the **LOCAL** coordinates will equal the **GLOBAL** coordinates.

CALL RE,0,X,Y,W,H	**FRAME RECTANGLE:** draws a rectangle outline in the current pen color.
CALL RE,1,X,Y,W,H	**PAINT RECTANGLE:** draws a rectangle and fills it with the current pen color.
CALL RE,2,X,Y,W,H	**ERASE RECTANGLE:** draws a rectangle and fills it with color 0.
CALL RE,3,X,Y,W,H	**INVERT RECTANGLE:** draws a rectangle and inverts the pixels colors.
CALL RE,4,X,Y,W,H,P	**FILL RECTANGLE:** draws a rectangle and fills it with the current pen pattern.

X
Left edge of the rectangle in LOCAL coordinates.

Y
Top edge of the rectangle in LOCAL coordinates.

W
Width of the rectangle in pixels.

H
Height of the rectangle in pixels.

P
Pattern used to fill rectangle (*0-15*)

CALL RE the RECTANGLE commands *(continued)*

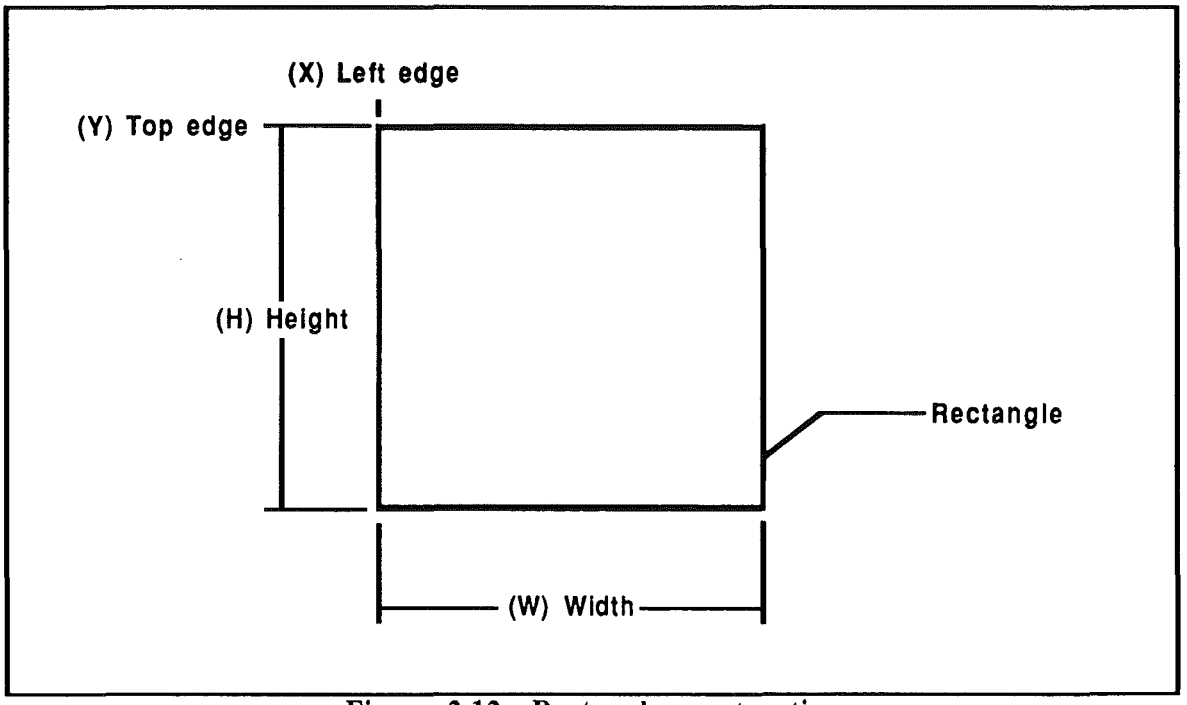


Figure 2.12 Rectangle construction

CALL RR the ROUNDED RECTANGLE commands

There are **5** commands that draw Rounded Rectangles. The Rounded Rectangles are drawn with the pen in the current port.

Note: Any command that draws something, draws to a graphics port. You must make sure that the port you want to draw in is the current port before drawing to it. Because you are drawing to a port and not necessarily the screen, the coordinates used to draw are in a coordinate system unique to the port you have selected. This coordinate system is called **LOCAL. GLOBAL** coordinates on the other hand refer to the screen coordinates... or the current cursor position. If you choose to draw to the screen then the **LOCAL** coordinates will equal the **GLOBAL** coordinates.

CALL RR,0,X,Y,W,H,OW,OH	**FRAME RRECTANGLE:** draws a rrectangle outline in the current pen color.
CALL RR,1,X,Y,W,H,OW,OH	**PAINT RRECTANGLE:** draws a rrectangle and fills it with the current pen color.
CALL RR,2,X,Y,W,H,OW,OH	**ERASE RRECTANGLE:** draws a rrectangle and fills it with color 0.
CALL RR,3,X,Y,W,H,OW,OH	**INVERT RRECTANGLE:** draws a rrectangle and inverts the pixels colors.
CALL RR,4,X,Y,W,H,OW,OH,P	**FILL RRECTANGLE:** draws a rrectangle and fills it with the current pen pattern.

X
Left edge of the rrectangle in LOCAL coordinates.

Y
Top edge of the rrectangle in LOCAL coordinates.

W
Width of the rrectangle in pixels.

H
Height of the rrectangle in pixels.

OW
Corner oval width in pixels

OH
Corner oval height in pixels

P
Pattern used to fill rrectangle (*0-15*)

CALL RR the ROUNDED RECTANGLE commands *(continued)*

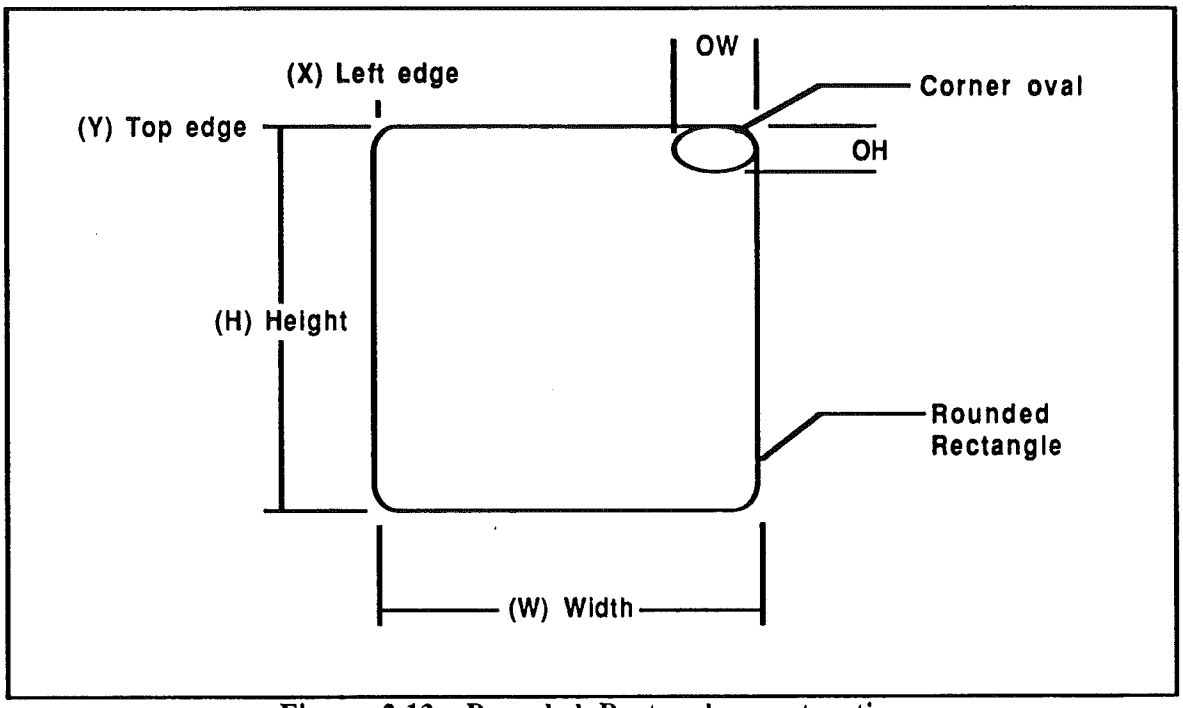


Figure 2.13 Rounded Rectangle construction

CALL SB the SCAN LINE CONTROL BYTE commands

There are 2 commands that control the Scan line Control Bytes (*SCB's*). A Scan line Control Byte describes the behavior of the pixels for each row of the screen. The screen has 200 rows of pixels that can be either 320 or 640 pixels wide depending on the mode setting.

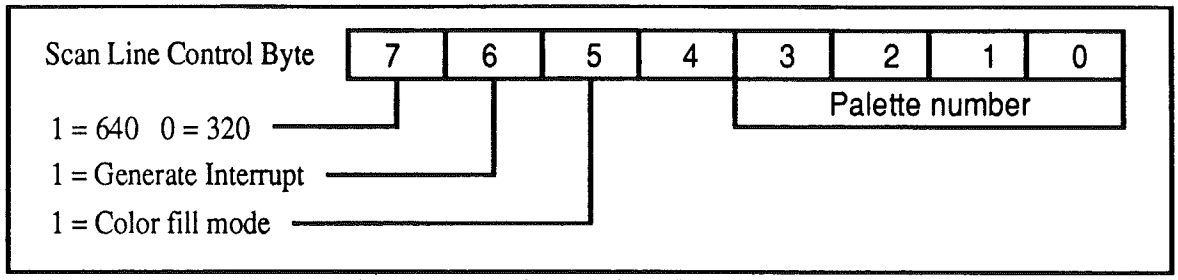


Figure 2.14 Scan Line Control Byte

Binary is a convienient method to specify scan line control bytes. To set all 200 SCB's to 640 mode graphics use the following program line:

```
CALL SB,0,!10000000,0,200
```

You will usually use palette 0 for all of your needs even though there are 16 palette spots avaliable. It should be noted that specifying a palette that has not been filled by a CALL PL will produce unpredictable results.

CALL SB,0,(*scb*)**,**(*start*)**,**(*end*) — **SET RANGE OF SCB's:** Sets any or all of the 200 scan line control bytes associated with the screen.

CALL SB,1,(*line*)**,R** — **GET ONE SCB VALUE:** Returns the value of a specified SCB.

(*scb*)
A value as described in **Figure 2.14**.

(*start*)
Top row of the range of SCB's to be set.

(*end*)
Bottom row of the range of SCB's to be set.

(*line*)
Line number of SCB to read.

R
Returned value... must be an Applesoft FP (real) variable.

CALL SC the SCREEN commands

There are 4 commands that control the super hi-res screen. This screen is graphics port #0 at $E1/2000. You can either enable or disable the screen, change it to a specified color or set its mode.

CALL SC works closely with CALL SB.. for example: If you want to set the screen to 640 mode graphics you must set the mode and set the scan line control bytes...

```
CALL SC,2,640 : CALL SB,0,!10000000,0,200
```

or to set the screen for 320 mode...

```
CALL SC,2,320 : CALL SB,0,!00000000,0,200
```

Note: SCB's are represented in binary, this is not necessary... only convienient. SCB's are actually bit flags and binary representation makes the bits easier to see.

CALL SC,0	**SCREEN OFF:** Changes the screen from super hi-res to text.
CALL SC,1	**SCREEN ON:** Changes the screen from text to super hi-res.
CALL SC,2,(*mode*)	**SCREEN MODE:** Sets the screen mode to either 320 or 640.
CALL SC,3,(*colorword*)	**SCREEN COLOR:** Clears the screen to the color specified in the colorword.

(*mode*)
Must be 320 or 640 only. All other values are invalid and will return an error.

(*colorword*)
A colorword is a value anywhere from 0 to 65535. There are only 16 solid colors in that range... all the rest are dithers. Specifying a colorword in hex is the easiest because the 16 values are in straight numeric order:

$0000 solid #0	$4444 solid #4	$8888 solid #8	$CCCC solid #12
$1111 solid #1	$5555 solid #5	$9999 solid #9	$DDDD solid #13
$2222 solid #3	$6666 solid #6	$AAAA solid #10	$EEEE solid #14
$3333 solid #4	$7777 solid #7	$BBBB solid #11	$FFFF solid #15

Figure 2.15 Solid colorwords

CALL TL the TOOL commands

There are 2 commands that control the starting up of supported tool sets. A tool set must be started up before it can be used. The TOOL command "CALL TL" provides an easy method for the BASIC programmer to startup required tool sets.

There are two options available with both the startup and shutdown calls: a) Startup a specific tool set, or b) Startup dependent tool sets in order (*the HighOrder call*).

Example: The HighOrder Startup call will startup all tool sets required by the one you specify. So if you wanted to use Dialog in your program, all tools needed by the Dialog Manager would be started up for you automatically; you don't need to make a call for each individual tool.

Note: It is necessary to make the CALL QF before exiting BASIC to any other system. (*CALL QF takes care of freeing up and disposing of all memory reserved for the BASIC Interface as well*).

CALL TL,0,*"toolname"*	**STARTUP ONE TOOL:** starts up the tool specified by toolname.
CALL TL,1,***	**Unused...**(*reserved*)
CALL TL,2,*"toolname"*	**HIGH ORDER STARTUP:** starts up all tools required by, up to and including, the toolset

"toolname"
One of the following strings describing which tool(s) to use in the call. The strings are case insensitive: they can be any combination of cases and still function right. However, if not spelled as shown below, you will receive a "**Tool Not Supported**" error. More tools will become available in a later release.

WINDOW	**DIALOG**
CONTROL	**SCRAP**
MENU	**LIST**
LINEEDIT	**DESK**

The toolname "DESK" will startup all currently supported tools and is the recommended command for desktop applications. Starting up fewer tools does not free up any memory. All tools remain memory resident throughout all CB BASIC operations.

CALL SC the SCREEN commands

There are 4 commands that control the super hi-res screen. This screen is graphics port #0 at $E1/2000. You can either enable or disable the screen, change it to a specified color or set its mode.

CALL SC works closely with CALL SB.. for example: If you want to set the screen to 640 mode graphics you must set the mode and set the scan line control bytes...

```
CALL SC,2,640 : CALL SB,0,!10000000,0,200
```

or to set the screen for 320 mode...

```
CALL SC,2,320 : CALL SB,0,!00000000,0,200
```

Note: SCB's are represented in binary, this is not necessary... only convienient. SCB's are actually bit flags and binary representation makes the bits easier to see.

CALL SC,0	**SCREEN OFF:** Changes the screen from super hi-res to text.
CALL SC,1	**SCREEN ON:** Changes the screen from text to super hi-res.
CALL SC,2,(*mode*)	**SCREEN MODE:** Sets the screen mode to either 320 or 640.
CALL SC,3,(*colorword*)	**SCREEN COLOR:** Clears the screen to the color specified in the colorword.

(*mode*)
Must be 320 or 640 only. All other values are invalid and will return an error.

(*colorword*)
A colorword is a value anywhere from 0 to 65535. There are only 16 solid colors in that range... all the rest are dithers. Specifying a colorword in hex is the easiest because the 16 values are in straight numeric order:

$0000 solid #0	$4444 solid #4	$8888 solid #8	$CCCC solid #12
$1111 solid #1	$5555 solid #5	$9999 solid #9	$DDDD solid #13
$2222 solid #3	$6666 solid #6	$AAAA solid #10	$EEEE solid #14
$3333 solid #4	$7777 solid #7	$BBBB solid #11	$FFFF solid #15

Figure 2.15 Solid colorwords

CALL SF the STANDARD FILE commands

Standard Files are high level tool functions which are dependent on other toolbox functions as well as GS/OS (*ProDOS16*) commands. These functions need to be initialized by using the "high level" startup command **CALL TL,2,"Desk"** (*see CALL TL in this manual for a complete description*). This call will startup all tools needed for desktop applications... the Standard File tools use most of them.

The Standard File tools are located in the file **SF** which must be in the SYSTEM/SETUP directory along with CB.INITb1. The file CB must be version 2.1b3 or greater.

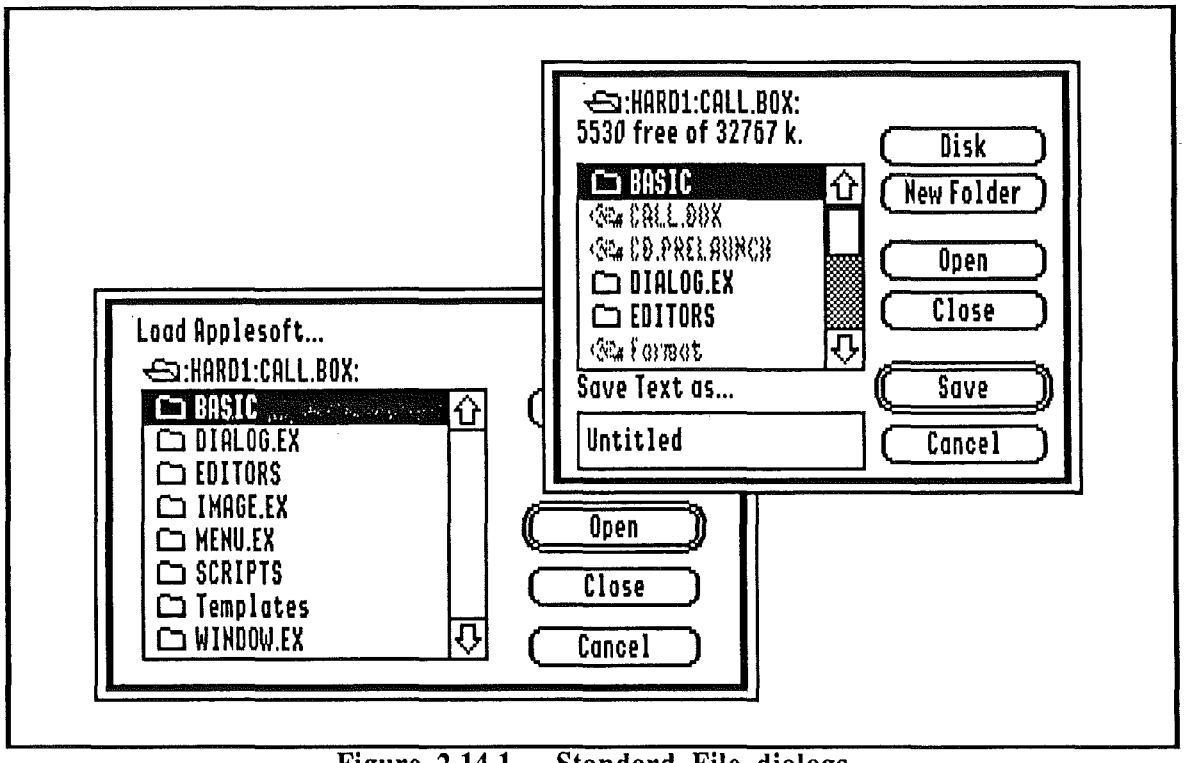


Figure 2.14.1 Standard File dialogs

Standard File dialogs are used to concatenate ProDOS pathnames using the point and click desktop method. The dialog has buttons with which you can OPEN or CLOSE files, Switch Volumes and Create folders. The buttons will change the contents of the list window within which you can select or double click individual filenames. When your selection process is finished Standard File returns you the Filename and Full Pathname selected. You then use this information to actually load and save the data using ProDOS or GS/OS commands.

The Standard File commands require you to supply the upper left hand coordinates for the box and a prompt string (*plus a default filename for save boxes*). When you are done with the box then the Standard File commands return a good flag, filename and full pathname using slashes as delimiters. The good flag will be 0 if you make a selection and will contain 1 if you have selected cancel.

CALL SF the STANDARD FILE commands

There are 2 commands that control Standard File boxes. You can either bring up a load or save Standard File Box .

This toolset is proprietary to Call Box BASIC and emulates the functions of the Apple IIgs Standard File Tools. This tool is installed at system initialization time and is not physically part of the file CB. If the files SF and CB.INITb1 are not in the SYSTEM/SYSTEM.SETUP subdirectory and/or the file CB (version 2.1b3 min.) is not being used then issueing these calls will cause the system to crash or hang. The New Folder function of the save standard file box is not functional in this beta release.

CALL SF,0,Y,X,"*prompt*",F,F$,P$
GetStdFile: This is the standard "LOAD" dialog box.

CALL SF,1,Y,X,"*prompt*","*DefName*",F,F$,P$
PutStdFile: This is the standard "SAVE" dialog box.

X
Horizontal position or left side.

Y
Vertical position or the top side.

"*prompt*"
A load or save message like... Load File.. or Save File as...

"*DefName*"
Default filename for the edit box in SAVE boxes.

F
Good Flag, 0 = F$ and P$ are valid, 1 = F$ and P$ are invalid

F$
Filename selected

P$
Full pathname selected (*delimited with slashes instead of colons*)

CALL TL the TOOL commands

There are 2 commands that control the starting up of supported tool sets. A tool set must be started up before it can be used. The TOOL command "CALL TL" provides an easy method for the BASIC programmer to startup required tool sets.

There are two options available with both the startup and shutdown calls: **a)** Startup a specific tool set, or **b)** Startup dependent tool sets in order (*the HighOrder call*).

Example: The HighOrder Startup call will startup all tool sets required by the one you specify. So if you wanted to use Dialog in your program, all tools needed by the Dialog Manager would be started up for you automatically; you don't need to make a call for each individual tool.

Note: It is necessary to make the CALL QF before exiting BASIC to any other system. (*CALL QF takes care of freeing up and disposing of all memory reserved for the BASIC Interface as well*).

CALL TL,0,"*toolname*"	**STARTUP ONE TOOL:** starts up the tool specified by toolname.
CALL TL,1,***	**Unused...**(*reserved*)
CALL TL,2,"*toolname*"	**HIGH ORDER STARTUP:** starts up all tools required by, up to and including, the toolset

"*toolname*"
One of the following strings describing which tool(s) to use in the call. The strings are case insensitive: they can be any combination of cases and still function right. However, if not spelled as shown below, you will receive a "**Tool Not Supported**" error. More tools will become available in a later release.

WINDOW	**DIALOG**
CONTROL	**SCRAP**
MENU	**LIST**
LINEEDIT	**DESK**

The toolname "DESK" will startup all currently supported tools and is the recommended command for desktop applications. Starting up fewer tools does not free up any memory. All tools remain memory resident throughout all CB BASIC operations.

CALL TL the TOOL commands *(continued)*

CALL TX the TEXT commands

There are **6** commands that control text. Font I.D. #0 is reserved for the system font (*Shaston 8*). Fonts #1-15 can be loaded in from disk and are the IIgs modified Mac font file type $C8. You can set the color for foreground and background plus set the font face for**normal, bold, underline, outline, shadow and italics.** The mode can be set the same as in the CALL PN commands.

The font is plotted at the current ports pen position... after a string is ploltted the horizontal pen position is advanced to the end of the plotted string.

CALL TX,0,F,*"string"*	**DRAW TEXT:** this will plot the string using font F.
CALL TX,1,FO,BK	**SET COLORS:** stes the foreground and background colors 0-15.
CALL TX,2,TF	**SET TEXT FACE:** sets the style that the font will appear in (*see* **Figure 2.14**).
CALL TX,3,MO	**SET FONT MODE:** sets the mode that the font will be plotted in (*see* **Figure 2.11**).
CALL TX,4,F,*"pathname"*	**LOAD FONT:** loads a type $C8 font from disk as I.D. F.
CALL TX,5,F,*"string"*,**A**	**GET TEXT LENGTH:** returns the width of the string in pixels using font F. A has the width.

F
Font number 0-15 (*font #0 is Shaston 8*).

FO
Foreground color of text (*actual text color*).

BK
Background color of a rectangle that encloses the text.

TF
Text face... this is a bit flag byte that enables the various text styles (*see* **Figure 2.14**).

MO
A number 0-7 that selects the drawing mode for the text (*see* **Figure 2.11**).

"string"
A text string of ASCII characters not to exceed 255 characters in length.

"pathname"
A ProDOS 8 pathname of the font you want to load. This font must be a filetype $C8. Fonts specified by filename only must be in the boot volumes **SYSTEM/FONTS** subdirectory.

CALL TX the TEXT commands *(continued)*

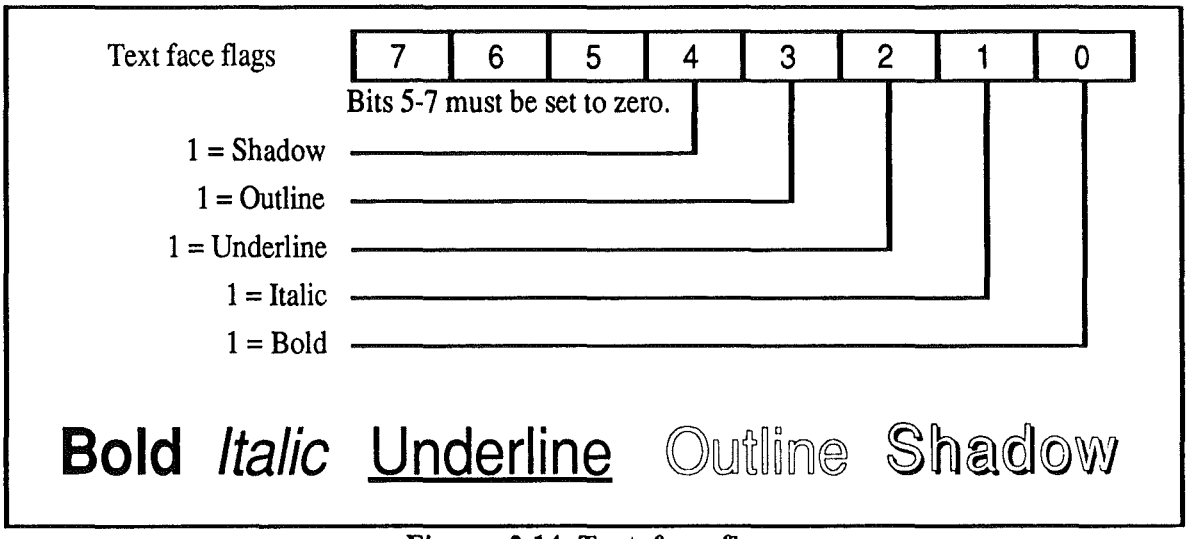


Figure 2.14 Text face flags

CALL WN the WINDOW commands

Windows are high level tool functions which are dependent on other toolbox functions as well as GS/OS (*ProDOS16*) commands. The menu functions need to be initialized by using the "high level" startup command **CALL TL,2,"Desk"** (*see CALL TL in this manual for a complete description*). This call will startup all tools needed for desktop applications... the Window Manager is one of them.

Windows are designated as "*entities*" in the Call Box BASIC driver and these entities are created using the Call Box TPS**Window Editor**. The output type "*object*" must be used for windows that are to be used by the BASIC driver.

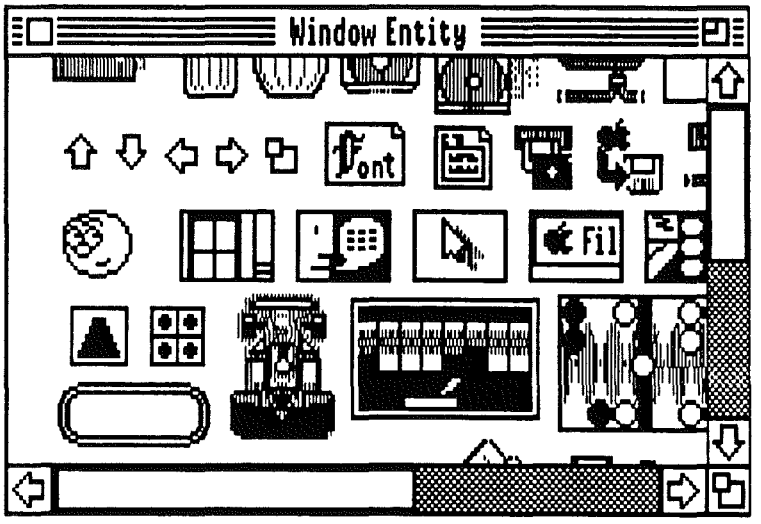


Figure 2.15 A Window entity

Operating a Window is automatic, you must load it and open it. Once a window is open you would probably draw something in it to display. To draw to a window you simply point to the windows port by using the port command **CALL PT,0,**(*entity number*) and then use drawing and text commands to create the windows contents. The TaskMaster call in your programs event loop is at the heart of window operations. TaskMaster will return task codes (*see* **Figure 2.8**) which will inform you if a particular part of a window was clicked in as well as other status information. If you want the "**system**" to handle the operation of the windows then you would take no action on the window type task codes and loop back to your TaskMaster call. Moving, growing-shrinking, Scrolling and Zooming the window will happen without any program lines on your part, and you usually only need to handle the go-away region task code (#22) in order to close the window.

CALL WN the WINDOW commands *(continued)*

Drawing to a Window

Each window has a contents region which displays the information contained in the window. This information is "**drawn**" to the window using any of the drawing commands included in the Call Box BASIC driver. Information can also be put in a window by using the "**port to port**" command (*such as super hi-res pictures*). When something is drawn to a window it will be drawn in "**local**" coordinates... the mouse position is in "**global**" coordinates, if you wish to draw into the window using the mouse/cursor like a pen then you must convert the mouse X and Y positions to local coordinates using the "**Global to Local**" command in the port commands.

Before drawing to a window you must direct all drawing action to the window using the command "**Set Port**" in the port commands. When you first create a window the port is set to the window just created so the Set Port command will not be necessary. Windows can be drawn to even when they are not visible, a subsequent use of the command "**Show window**" will display the window with the contents already drawn. These few simple rules cover the handleing of the windows contents... most other window functions are handled automatically by the system and need little or no attention.

Window Operation

After starting up and initializing Call Box BASIC load a Window entity created by the Call Box Window Editor.

```
10 CALL WN,0,N,"MyWindow"
```

Next you need to build the window which sets up the memory required by the window, this operation will not make the window visible.

```
12 CALL WN,1,N
```

Note: At this point you need to draw the initial window contents as described above. Pre-drawing the contents makes the windows contents appear quicker when the window is displayed.

Once your window is all set-up you display it by using the Show Window command:

```
14 CALL WN,4,N,1
```

The window will now be visible on the screen and will operate automatically. Several windows can be up on the screen at one time and may overlap each other. When TaskMaster in your programs Event loop shows that a window region (*refer to* **Figure 2.8**) has had the mouse clicked in it you can run routines based on which item is clicked in or simply ignore the indication and have the system handle the operation itself by looping back to the TaskMaster command. One value returned by TaskMaster must be handled by your program and this is the close box (*go away region*) #22. To close your window issue a Hide Window or if you want to dispose of the window the Close window command.

```
16 CALL WN,4,N,0  or  CALL WN,2,N
```

CALL WN the WINDOW commands *(continued)*

These are the minimum required window commands to operate a window. Other functions such as **Get/Set origin** and **Set Title**, add some custom functions to the repitorie of commands. There are 3 other commands that should be of value to you the programmer these are: **Refresh Desktop**, **Duplicate a Window** and **Get Window Pointer/Entity number**. Refresh Desktop should be issued prior to the use of windows, dialogs or menus... this command sets up the desktop environment and redraws the whole desktop. Sometimes during an applications execution some non desktop commands such as drawing directly to the desktop will "*trash*" the looks of the screen. Issueing a Refresh Desktop command will straighten things out. The Duplicate a Window command will allow you to use the same window template for several window entities, this will eliminate the need to load a separate window template for each window entity. The Get Window Pointer/Entity Number command will return any of three pointers and indicators necessary for window manipulation. An example of the use of this command is when the mouse is clicked in the top-most window out of several windows visible at one time. If you wish to close this window then you need to know the entity number of this window for the Close Window command. **TaskMaster** will return the Windows pointer in the "**D**" variable. This pointer is converted to the entity number as follows:

```
100 CALL WN,8,N,D,2
```

The windows entity number is returned in the variable "**N**".

There are **9** commands that control windows. Windows must be used only when the **desktop** is active.

CALL WN,0,N,*"pathname"*	**LOAD WINDOW:** this loads a window template from disk as entity number (*N*).
CALL WN,1,N	**BUILD WINDOW:** this creates the memory structure for the window (*N*).
CALL WN,2,N	**CLOSE WINDOW:** this closes the window specified by (*N*) and releases its memory.
CALL WN,3	**REFRESH DESKTOP:** this redraws the entire desktop.
CALL WN,4,N,V	**HIDE/SHOW WINDOW:** this will hide (*V=0*) or show (*V=1*) the window specified by (*N*).
CALL WN,5,N,X,Y,V	**GET/SET ORIGIN:** this will get (*V=0*) or set (*V=1*) the origin (*X*,*Y*) of the windows contents.
CALL WN,6,N,A$	**SET TITLE:** this will change the title (*A$*) of the window specified by (*N*).
CALL WN,7,N1,N2	**DUPLICATE A WINDOW:** this will copy the window entity (*N1*) to a new window entity (*N2*).
CALL WN,8,N,WP,V	**GET WINDOW POINTER/ENTITY NUMBER:** This will return the background port pointer (*V=0*) or the windows port pointer (*V=1*) or the entity number (*V=2*) of the window specified by (*WP*) in the variable (*N*).

CALL WN the WINDOW commands *(continued)*

"pathname"
A ProDOS 8 pathname of the window template to load.

N
Entity number for this window (*1-31*).

V
Code number for the type of operation requested.

X
The input and output horizontal positions

Y
The input and output vertical positions

A$
The string used to set the windows title.

N1
The source window entity number.

N2
The destination window entity number.

WP
The window ports pointer.

Apple IIgs System and Toolbox calls

This listing is a reference for all the system and toolbox calls that can be made by using the Call Box BASIC driver call **CALL LC** (*LongCall*). Only the call code numbers and the name are given in this appendix. Complete call descriptions can be found in the Apple IIgs Toolbox Reference Vols. 1,2 and 3 avaliable from A.P.D.A. (*Apple Programmers and Developers Association*) or Addison-Wesley publishing Co.

ProDOS 16 / GS/OS
0010 P16:OPEN
0012 P16:READ
0014 P16:CLOSE
0016 P16:SET_MARK
0017 P16:GET_MARK
0019 P16:GET_EOF
200E GS/OS:ExpandPath
200F GS/OS:GetSysPrefs
2010 GS/OS:Open
2012 GS/OS:Read
2014 GS/OS:Close
2016 GS/OS:SetMark
2017 GS/OS:GetMark
2019 GS/OS:GetEOF
201A GS/OS:SetLevel
201B GS/OS:GetLevel

Tool Locator
0101 TLBootInit
0201 TLStartUp
0301 TLShutDown
0401 TLVersion
0501 TLReset
0601 TLStatus
0901 GetTSPtr
0A01 SetTSPtr
0B01 GetFuncPtr
0C01 GetWAP
0D01 SetWAP
1101 TLMountVolume
1201 TLTextMountVolume
1301 SaveTextState
1401 RestoreTextState
1501 MessageCenter
1701 MessageByName

Memory Manager
0102 MMBootInit
0202 MMStartUp
0302 MMShutDown
0402 MMVersion
0502 MMReset
0602 MMStatus
0902 NewHandle
0A02 ReAllocHandle
0B02 RestoreHandle
1002 DisposeHandle
1102 DisposeAll
1202 PurgeHandle
1302 PurgeAll
1802 GetHandleSize
1902 SetHandleSize
1A02 FindHandle
1B02 FreeMem
1C02 MaxBlock
1D02 TotalMem
1E02 CheckHandle
1F02 CompactMem
2002 HLock
2102 HLockAll
2202 HUnlock
2302 HUnlockAll
2402 SetPurge
2502 SetPurgeAll
2802 PtrToHand
2902 HandToPtr
2A02 HandToHand
2B02 BlockMove
2F02 RealFreeMem

Misc. Tools
0103 MTBootInit
0203 MTStartUp
0303 MTShutDown
0403 MTVersion
0503 MTReset
0603 MTStatus
0903 WriteBRam
0A03 ReadBRam
0B03 WriteBParam
0C03 ReadBParam
0D03 ReadTimeHex
0E03 WriteTimeHex
0F03 ReadAsciiTime
1003 SetVector
1103 GetVector
1503 SysFailMg
1603 GetAddr
1703 ReadMouse
1803 InitMouse
1903 SetMouse
1A03 HomeMouse
1B03 ClearMouse
1C03 ClampMouse
1D03 GetMouseClamp
1E03 PosMouse
1F03 ServeMouse
2003 GetNewID
2103 DeleteID
2203 StatusID
2303 IntSource
2403 FWEntry
2503 GetTick
2603 PackBytes
2703 UnPackBytes
2803 Munge
2903 GetIRQEnable
2A03 SetAbsClamp
2B03 GetAbsClamp
2C03 SysBeep
3003 SetInterruptState
3103 GetInterruptState
3203 GetIntStateRecSize
3303 ReadMouse2
3403 GetCodeResConverter

QuickDraw II
0104 QDBootInit
0204 QDStartUp
0304 QDShutDown
0404 QDVersion
0504 QDReset
0604 QDStatus
0904 GetAddress
0A04 GrafOn
0B04 GrafOff
0C04 GetStandardSCB
0D04 InitColorTable
0E04 SetColorTable
0F04 GetColorTable
1004 SetColorEntry
1104 GetColorEntry
1204 SetSCB
1304 GetSCB
1404 SetAllSCBs
1504 ClearScreen
1604 SetMasterSCB
1704 GetMasterSCB
1804 OpenPort
1904 InitPort
1A04 ClosePort
1B04 SetPort
1C04 GetPort
1D04 SetPortLoc
1E04 GetPortLoc
1F04 SetPortRect
2004 GetPortRect
2104 SetPortSize
2204 MovePortTo
2304 SetOrigin
2404 SetClip
2504 GetClip
2604 ClipRect
2704 HidePen

2804 ShowPen
2904 GetPen
2A04 SetPenState
2B04 GetPenState
2C04 SetPenSize
2D04 GetPenSize
2E04 SetPenMode
2F04 GetPenMode
3004 SetPenPat
3104 GetPenPat
3204 SetPenMask
3304 GetPenMask
3404 SetBackPat
3504 GetBackPat
3604 PenNormal
3704 SetSolidPenPat
3804 SetSolidBackPat
3904 SolidPattern
3A04 MoveTo
3B04 Move
3C04 LineTo
3D04 Line
3E04 SetPicSave
3F04 GetPicSave
4004 SetRgnSave
4104 GetRgnSave
4204 SetPolySave
4304 GetPolySave
4404 SetGrafProcs
4504 GetGrafProcs
4604 SetUserField
4704 GetUserField
4804 SetSysField
4904 GetSysField
4A04 SetRect
4B04 OffsetRect
4C04 InsetRect
4D04 SectRect
4E04 UnionRect
4F04 PtInRect
5004 Pt2Rect
5104 EqualRect
5204 NotEmptyRect
5304 FrameRect
5404 PaintRect
5504 EraseRect
5604 InvertRect
5704 FillRect
5804 FrameOval
5904 PaintOval
5A04 EraseOval
5B04 InvertOval
5C04 FillOval
5D04 FrameRRect
5E04 PaintRRect
5F04 EraseRRect
6004 InvertRRect
6104 FillRRect
6204 FrameArc
6304 PaintArc
6404 EraseArc
6504 InvertArc
6604 FillArc
6704 NewRgn
6804 DisposeRgn
6904 CopyRgn
6A04 SetEmptyRgn
6B04 SetRectRgn
6C04 RectRgn
6D04 OpenRgn
6E04 CloseRgn
6F04 OffsetRgn
7004 InsetRgn
7104 SectRgn
7204 UnionRgn
7304 DiffRgn
7404 XorRgn
7504 PtInRgn
7604 RectInRgn
7704 EqualRgn
7804 EmptyRgn
7904 FrameRgn
7A04 PaintRgn
7B04 EraseRgn
7C04 InvertRgn
7D04 FillRgn
7E04 ScrollRect
7F04 PaintPixels
8004 AddPt
8104 SubPt
8204 SetPt
8304 EqualPt
8404 LocalToGlobal
8504 GlobalToLocal
8604 Random
8704 SetRandSeed
8804 GetPixel
8904 ScalePt
8A04 MapPt
8B04 MapRect
8C04 MapRgn
8D04 SetStdProcs
8E04 SetCursor
8F04 GetCursorAdr
9004 HideCursor
9104 ShowCursor
9204 ObscureCursor
9304 SetMouseLoc
9404 SetFont
9504 GetFont
9604 GetFontInfo
9704 GetFontGlobals
9804 SetFontFlags
9904 GetFontFlags
9A04 SetTextFace
9B04 GetTextFace
9C04 SetTextMode
9D04 GetTextMode
9E04 SetSpaceExtra
9F04 GetSpaceExtra
A004 SetForeColor
A104 GetForeColor
A204 SetBackColor
A304 GetBackColor
A404 DrawChar
A504 DrawString
A604 DrawCString
A704 DrawText
A804 CharWidth
A904 StringWidth
AA04 CStringWidth
AB04 TextWidth
AC04 CharBounds
AD04 StringBounds
AE04 CStringBounds
AF04 TextBounds
B004 SetArcRot
B104 GetArcRot
B204 SetSysFont
B304 GetSysFont
B404 SetVisRgn
B504 GetVisRgn
B604 SetIntUse
B704 OpenPicture
B804 PicComment
B904 ClosePicture
BA04 DrawPicture
BB04 KillPicture
BC04 FramePoly
BD04 PaintPoly
BE04 ErasePoly
BF04 InvertPoly
C004 FillPoly
C104 OpenPoly
C204 ClosePoly
C304 KillPoly
C404 OffsetPoly
C504 MapPoly
C604 SetClipHandle
C704 GetClipHandle
C804 SetVisHandle
C904 GetVisHandle
CA04 InitCursor
CB04 SetBufDims
CC04 ForceBufDims
CD04 SaveBufDims
CE04 RestoreBufDims
CF04 GetFGSize
D004 SetFontID
D104 GetFontID
D204 SetTextSize
D304 GetTextSize
D404 SetCharExtra
D504 GetCharExtra
D604 PPToPort
D704 InflateTextBuffer
D804 GetRomFont
D904 GetFontLore

Desk Manager
0105 DeskBootInit
0205 DeskStartUp
0305 DeskShutDown
0405 DeskVersion
0505 DeskReset
0605 DeskStatus
0905 SaveScrn
0A05 RestScrn
0B05 SaveAll
0C05 RestAll
1105 ChooseCDA
1305 SetDAStrPtr
1405 GetDAStrPtr
1505 OpenNDA
1605 CloseNDA
1705 SystemClick
1805 SystemEdit
1905 SystemTask
1A05 SystemEvent
1B05 GetNumNDAs
1C05 CloseNDAByWinPtr
1D05 CloseAllNDAs
1E05 FixAppleMenu
2105 RemoveCDA
2205 RemoveNDA

Event Manager
0106 EMBootInit
0206 EMStartUp
0306 EMShutDown
0406 EMVersion
0506 EMReset
0606 EMStatus
0906 DoWindows
0A06 GetNextEvent
0B06 EventAvail
0C06 GetMouse
0D06 Button
0E06 StillDown
0F06 WaitMouseUp
1006 TickCount
1106 GetDblTime
1206 GetCaretTime
1306 SetSwitch
1406 PostEvent
1506 FlushEvents
1806 SetEventMask
1906 FakeMouse
1A06 SetAutoKeyLimit
1B06 GetKeyTranslation
1C06 SetKeyTranslation

Scheduler
0107 SchBootInit
0207 SchStartUp
0307 SchShutDown
0407 SchVersion
0507 SchReset
0607 SchStatus
0907 SchAddTask
0A07 SchFlush

Sound Manager
0108 SoundBootInit
0208 SoundStartUp
0308 SoundShutDown
0408 SoundVersion
0508 SoundReset
0608 SoundStatus
0908 WriteRamBlock
0A08 ReadRamBlock
0B08 GetTableAddress
0C08 GetSoundVolume
0D08 SetSoundVolume
0E08 FFStartSound
0F08 FFStopSound
1008 FFSoundStatus
1108 FFGeneratorStatus
1208 SetSoundMIRQV
1308 SetUserSoundIRQV
1408 FFSoundDoneStatus
1508 FFSetUpSound
1608 FFStartPlaying
1708 SetDocReg
1808 ReadDocReg

Desktop Bus
0109 ADBBootInit
0209 ADBStartUp
0309 ADBShutDown
0409 ADBVersion
0509 ADBReset
0609 ADBStatus
0909 SendInfo
0A09 ReadKeyMicroData
0B09 ReadKeyMicroMemory
0D09 AsyncADBReceive
0E09 SyncADBReceive
0F09 AbsOn
1009 AbsOf
1109 RdAbs
1209 SetAbsScale
1309 GetAbsScale
1409 SRQPoll
1509 SRQRemove
1609 ClearSRQTable

SANE
010A SANEBootInit
020A SANEStartUp
030A SANEShutDown
040A SANEVersion
050A SANEReset
060A SANEStatus
090A FPNum
0A0A DecStrNum
0B0A ElemNum

Integer Math
010B IMBootInit
020B IMStartUp
030B IMShutDown
040B IMVersion
050B IMReset
060B IMStatus
090B Multiply
0A0B SDivide
0B0B UDivide
0C0B LongMul
0D0B LongDivide
0E0B FixRatio
0F0B FixMul
100B FracMul
110B FixDiv
120B FracDiv
130B FixRound
140B FracSqr
150B FracCos
160B FracSin
170B FixATan2
180B HiWord
190B LoWord
1A0B Long2Fix
1B0B Fix2Long
1C0B Fix2Frac
1D0B Frac2Fix
1E0B Fix2X
1F0B Frac2X
200B X2Fix
210B X2Frac
220B Int2Hex
230B Long2Hex
240B Hex2Int
250B Hex2Long
260B Int2Dec
270B Long2Dec
280B Dec2Int
290B Dec2Long
2A0B HexIt

Text Tools
010C TextBootInit
020C TextStartUp
030C TextShutDown
040C TextVersion
050C TextReset
060C TextStatus
090C SetInGlobals
0A0C SetOutGlobals
0B0C SetErrGlobals
0C0C GetInGlobals
0D0C GetOutGlobals
0E0C GetErrGlobals
0F0C SetInputDevice
100C SetOutputDevice
110C SetErrorDevice
120C GetInputDevice
130C GetOutputDevice
140C GetErrorDevice
150C InitTextDev
160C CtlTextDev
170C StatusTextDev
180C WriteChar
190C ErrWriteChar
1A0C WriteLine
1B0C ErrWriteLine
1C0C WriteString
1D0C ErrWriteString

1E0C TextWriteBlock
1F0C ErrWriteBlock
200C WriteCString
210C ErrWriteCString
220C ReadChar
230C TextReadBlock
240C ReadLine

Window Manager

010E WindBootInit
020E WindStartUp
030E WindShutDown
040E WindVersion
050E WindReset
060E WindStatus
090E NewWindow
0A0E CheckUpdate
0B0E CloseWindow
0C0E Desktop
0D0E SetWTitle
0E0E GetWTitle
0F0E SetFrameColor
100E GetFrameColor
110E SelectWindow
120E HideWindow
130E ShowWindow
140E SendBehind
150E FrontWindow
160E SetInfoDraw
170E FindWindow
180E TrackGoAway
190E MoveWindow
1A0E DragWindow
1B0E GrowWindow
1C0E SizeWindow
1D0E TaskMaster
1E0E BeginUpdate
1F0E EndUpdate
200E GetWMgrPor
210E PinRect
220E HiliteWindow
230E ShowHide
240E BringToFront
250E WindNewRes
260E TrackZoom
270E ZoomWindow
280E SetWRefCon
290E GetWRefCon
2A0E GetNextWindow
2B0E GetWKind
2C0E GetWFrame
2D0E SetWFrame
2E0E GetStructRgn
2F0E GetContentRgn
300E GetUpdateRgn
310E GetDefProc
320E SetDefProc
330E GetWControls
340E SetOrgnMask
350E GetInfoRefCon
360E SetInfoRefCon
370E GetZoomRect
380E SetZoomRect
390E RefreshDesktop
3A0E InvalRect
3B0E InvalRgn
3C0E ValidRec
3D0E ValidRgn
3E0E GetContentOrigin
3F0E SetContentOrigin
400E GetDataSize
410E SetDataSize
420E GetMaxGrow
430E SetMaxGrow
440E GetScroll
450E SetScroll
460E GetPage
470E SetPage
480E GetContentDraw
490E SetContentDraw
4A0E GetInfoDraw
4B0E SetSysWindow
4C0E GetSysWFlag
4D0E StartDrawing
4E0E SetWindowIcons
4F0E GetRectInfo
500E StartInfoDrawing
510E EndInfoDrawing
520E GetFirstWindow
530E WindDragRect
540E GetDragRectPtr
550E DrawInfoBar
560E WindowGlobal
580E GetWindowMgrGlobals
590E AlertWindow
5A0E StartFrameDrawing
5B0E EndFrameDrawing
5C0E ResizeWindow
5D0E TaskMasterContent
5E0E TaskMasterKey
5F0E TaskMasterDA
600E CompileText
620E ErrorWindow

Menu Manager

010F MenuBootInit
020F MenuStartUp
030F MenuShutDown
040F MenuVersion
050F MenuReset
060F MenuStatus
090F MenuKey
0A0F GetMenuBar
0B0F MenuRefresh
0C0F FlashMenuBar
0D0F InsertMenu
0E0F DeleteMenu
0F0F InsertMItem
100F DeleteMItem
110F GetSysBar
120F SetSysBar
130F FixMenuBar
140F CountMItems
150F NewMenuBar
160F GetMHandle
170F SetBarColors
180F GetBarColors
190F SetMTitleStar
1A0F GetMTitleStart
1B0F GetMenuMgrPort
1C0F CalcMenuSize
1D0F SetMTitleWidth
1E0F GetTitleWidth
1F0F SetMenuFlag
200F GetMenuFlag
210F SetMenuTitle
220F GetMenuTitle
230F MenuGlobal
240F SetMItem
250F GetMItem
260F SetMItemFlag
270F GetMItemFlag
280F SetMItemBlink
290F MenuNewRes
2A0F DrawMenuBar
2B0F MenuSelect
2C0F HiliteMenu
2D0F NewMenu
2E0F DisposeMenu
2F0F InitPalette
300F EnableMItem
310F DisableMItem
320F CheckMItem
330F SetMItemMark
340F GetMItemMark
350F SetMItemStyle
360F GetMItemStyle
370F SetMenuID
380F SetMItemID
390F SetMenuBar
3A0F SetMItemName
3B0F GetPopUpDefProc
3C0F PopUpMenuSelect
3D0F DrawPopUp
450F HideMenuBar
460F ShowMenuBar

Control Manager

0110 CtlBootInit
0210 CtlStartUp
0310 CtlShutDown
0410 CtlVersion
0510 CtlReset
0610 CtlStatus
0910 NewControl
0A10 DisposeControl
0B10 KillControls
0C10 SetCtlTitle
0D10 GetCtlTitle
0E10 HideControl
0F10 ShowControl
1010 DrawControls
1110 HiliteControl
1210 CtlNewRes
1310 FindControl
1410 TestControl
1510 TrackControl
1610 MoveControl
1710 DragControl
1810 SetCtlIcons
1910 SetCtlValue
1A10 GetCtlValue
1B10 SetCtlParams
1C10 GetCtlParams
1D10 DragRect
1E10 GrowSize
1F10 GetCtlDpage
2010 SetCtlAction
2110 GetCtlAction
2210 SetCtlRefCon
2310 GetCtlRefCon
2410 EraseControl
2510 DrawOneCtl
2610 FindTargetCtl
2710 MakeNextCtlTarget
2810 MakeThisCtlTarget
2910 SendEventToCtl
2A10 GetCtlID
2B10 SetCtlID
2C10 CallCtlDefProc
2D10 NotifyCtls
2E10 GetCtlMoreFlags
2F10 SetCtlMoreFlags
3010 GetCtlHandleFromID
3410 SetCtlParamPtr
3510 GetCtlParamPtr
3710 InvalCtls

QuickDraw Aux

0112 QDAuxBootInit
0212 QDAuxStartUp
0312 QDAuxShutDown
0412 QDAuxVersion
0512 QDAuxReset
0612 QDAuxStatus
0912 CopyPixels
0A12 WaitCursor
0B12 DrawIcon
0C12 SpecialRect

Line Edit

0114 LEBootInit
0214 LEStartUp
0314 LEShutDown
0414 LEVersion
0514 LEReset
0614 LEStatus
0914 LENew
0A14 LEDispose
0B14 LESetText
0C14 LEIdle
0D14 LEClick
0E14 LESetSelect
0F14 LEActivate
1014 LEDeactivate
1114 LEKey
1214 LECut
1314 LECopy
1414 LEPaste
1514 LEDelete
1614 LEInsert
1714 LEUpdate
1814 LETextBox
1914 LEFromScrap
1A14 LEToScrap
1B14 LEScrapHandle
1C14 LEGetScrapLen
1D14 LESetScrapLen
1E14 LESetHilite
1F14 LESetCaret
2114 LESetJust
2214 LEGetTextHand
2314 LEGetTextLen
2414 GetLEDefProc

Dialog Manager

0115 DialogBootInit
0215 DialogStartUp
0315 DialogShutDown
0415 DialogVersion
0515 DialogReset
0615 DialogStatus
0915 ErrorSound
0A15 NewModalDialog
0B15 NewModelessDialog
0C15 CloseDialog
0D15 NewDItem
0E15 RemoveDItem
0F15 ModalDialog
1015 IsDialogEvent
1115 DialogSelect
1215 DlgCut
1315 DlgCopy
1415 DlgPaste
1515 DlgDelete
1615 DrawDialog
1715 Alert
1815 StopAlert
1915 NoteAlert
1A15 CautionAlert
1B15 ParamTex
1C15 SetDAFont
1E15 GetControlDItem
1F15 GetITex
2015 SetIText
2115 SelIText
2215 HideDItem
2315 ShowDItem
2415 FindDItem
2515 UpdateDialog
2615 GetDItemType
2715 SetDItemType
2815 GetDItemBox
2915 SetDItemBox
2A15 GetFirstDItem
2B15 GetNextDItem
2E15 GetDItemValue
2F15 SetDItemValue
3215 GetNewModalDialog
3315 GetNewDItem
3415 GetAlertStage
3515 ResetAlertStage
3615 DefaultFilter
3715 GetDefButton
3815 SetDefButton
3915 DisableDItem
3A15 EnableDItem

Scrap Manager
0116 ScrapBootInit
0216 ScrapStartUp
0316 ScrapShutDown
0416 ScrapVersion
0516 ScrapReset
0616 ScrapStatus
0916 UnloadScrap
0A16 LoadScrap
0B16 ZeroScrap
0C16 PutScrap
0D16 GetScrap
0E16 GetScrapHandle
0F16 GetScrapSize
1016 GetScrapPath
1116 SetScrapPath
1216 GetScrapCount
1316 GetScrapState

Note Synthesizer
0119 NSBootInit
0219 NSStartUp
0319 NSShutDown
0419 NSVersion
0519 NSReset
0619 NSStatus
0919 AllocGen
0A19 DeallocGen
0B19 NoteOn
0C19 NoteOff
0D19 AllNotesOff
0E19 NSSetUpdateRate
0F19 NSSetUserUpdateRtn

Note Sequencer
011A SeqBootInit
021A SeqStartUp
031A SeqShutDown
041A SeqVersion
051A SeqReset
061A SeqStatus
091A SeqSetIncr
0A1A SeqClearIncr
0B1A SeqGetTimer
0C1A SeqGetLoc
0D1A SeqAllNotesOff
0E1A SeqSetTrkInfo
0F1A StartSeq
101A SeqStepSeq
111A StopSeq
121A SeqSetInstTable
131A SeqStartInts
141A SeqStopInts
151A StartSeqRel

List Manager
011C ListBootInit
021C ListStartUp
031C ListShutDown
041C ListVersion
051C ListReset
061C ListStatus
091C CreateList
0A1C SortList
0B1C NextMember
0C1C DrawMember
0D1C SelectMember
0E1C GetListDefProc
0F1C ResetMember
101C NewList
111C DrawMember2
121C NextMember2
131C ResetMember2
141C SelectMember2
151C SortList2
161C NewList2

A.C.E.
011D ACEBootInit
021D ACEStartUp
031D ACEShutDown
041D ACEVersion
051D ACEReset
061D ACEStatus
071D ACEInfo
091D ACECompress
0A1D ACEExpand
0B1D ACECompBegin
0C1D ACEExpBegin

MIDI
0120 MidiBootInit
0220 MidiStartUp
0320 MidiShutDown
0420 MidiVersion
0520 MidiReset
0620 MidiStatus
0920 MidiControl
0A20 MidiDevice
0B20 MidiClock
0C20 MidiInfo
0D20 MidiReadPacket
0E20 MidiWritePacket
0F20 MidiRecordSeq
1020 MidiStopRecord
1120 MidiPlaySeq
1220 MidiStopPlay
1320 MidiConvert

Video Overlay
0121 VDBootInit
0221 VDStartUp
0321 VDShutDown
0421 VDVersion
0521 VDReset
0621 VDStatus
0921 VDInStatus
0A21 VDInSetStd
0B21 VDInGetStd
0C21 VDInConvAdj
0D21 VDKeyControl
0E21 VDKeyStatus
0F21 VDKeySetKCol
1021 VDKeyGetKRCol
1121 VDKeyGetKGCol
1221 VDKeyGetKBCol
1321 VDKeySetKDiss
1421 VDKeyGetKDiss
1521 VDKeySetNKDiss
1621 VDKeyGetNKDiss
1721 VDOutSetStd
1821 VDOutGetStd
1921 VDOutControl
1A21 VDOutStatus
1B21 VDGetFeatures
1C21 VDInControl
1D21 VDGGControl
1E21 VDGGStatus

Text Edit
0122 TEBootInit
0222 TEStartUp
0322 TEShutDown
0422 TEVersion
0522 TEReset
0622 TEStatus
0922 TENew
0A22 TEKill
0B22 TESetText
0C22 TEGetTextID
0D22 TEGetTextInfo
0E22 TEIdle
0F22 TEActivate
1022 TEDeactivate
1122 TEClick
1222 TEUpdate
1322 TEPaintText
1422 TEKey
1622 TECut
1722 TECopy
1822 TEPaste
1922 TEClear
1A22 TEInsert
1B22 TEReplace
1C22 TEGetSelection
1D22 TESetSelection
1E22 TEGetSelectionStyle
1F22 TEStyleChange
2022 TEOffsetToPoint
2122 TEPointToOffset
2222 TEGetDefProc
2322 TEGetRuler
2422 TESetRuler
2522 TEScroll
2622 TEGetInternalProc
2722 TEGetLastError

The Editors

Version 1.0

Aug 15, 1989

NOTICE

So What Software reserves the right to make improvements in the product described in this manual at any time without notice.

So What Software
10221 Slater Ave.
Suite 103,
Fountain Valley CA.
92708

So What Software makes no warranties, either express or implied, with respect to this product, its quality, performance, merchantability or fitness for any particular purpose. The programs are provided "as is."

This software package was created using the following software and hardware products: Apple IIgs /W 1.5M & GS/OS V5.0, Applied Ingenuity 40M Inner Drive, Apple Laserwriter IINT, Apple 3.5 drives, Apple LocalTalk network, Applied Engineering TranswarpGS, Byte Works Orca/M assembler/linker, Claris AppleworksGS, Milliken Medeley, Baudville 816 Paint.

So What Software Product #M400-000

TABLE OF CONTENTS Page

PREFACE

The CALL-BOX editors produce "Templates" used with the Apple IIgs toolbox tools to control how a screen item looks and behaves. The Apple IIgs toolbox displays three types of information panels [Windows, Dialog boxes and Menus]. There are more types but only these are supported on this version of CALL-BOX. A separate editor is provided for each panel (window) type and are selectable from the Editors menu bar selection in the CALL-BOX main menu.

The Image Editor handles the creation of Icons, Pixel Images and Cursors. Images have no absolute address references contained in them. The filetypes, however, are output as if they did have these references so they can be linked into a program the same as any other object module.

Data created by these editors needs special "care and feeding" to operate efficiently from your application program. Sample code for each template is provided in the section of the manual for that editor to show you how to use them properly.

The editors can handle three filetypes for input and output so as to support several languages. Each editor handles ORCA/APW source code and Apple standard OMF2 relocatable code modules. These two styles encompass most popular programming languages currently used. The third filetype opens up these editors to any language present or future by the use of Resources. Resources were introduced with GS/OS V5.0 and are fully supported by the CALL-BOX editors.

More editors are under development and will be made available to registered owners when they are released. No firm schedule presently exists but you will be notified when they become available.. Be sure to send in your warranty card as this is the only method we have to insure that you are advised of updates.

The following pages describe the operation of the CALL-BOX editors. Each section will describe the particular editors functions with descriptions of the various filetypes and programming methods unique to each editors output.

CHAPTER 1 - THE WINDOW EDITOR

OVERVIEW

The CALL-BOX Window Editor creates templates for use by the Apple IIgs Window Manager. This editor can load either OMF2 object code or resources and can output APW/ORCA sourcecode, OMF2 object code and resources. The resource filetype is $1002 and uses a converter to get into memory. The standard programming procedure for resources is presented at the end of this chapter.

WINDOWS

A window is a presentation feature in which text or graphic information, can be displayed. Windows can be of any size and can be displayed on the screen singly or in groups depending on the application.

Windows allow an application to control more information than the screen can display at one time. The term Window is used because the user sees through the window into a larger area.

The Window Editor, can produce Document and Alert windows. The alert window frame is a double rectangle, the same as is used by the Dialog Manager to create a dialog box. The Document frame is a single outline. While the Alert frame is just a frame the document windows frame can have controls as described below.
(See Fig 1.1)

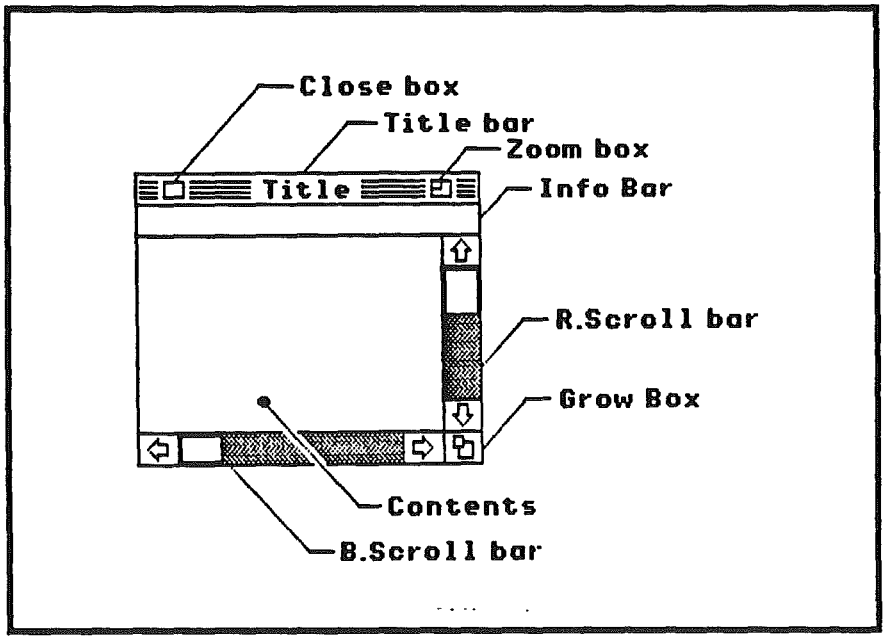


Figure 1.1 Typical Window

- Title bar — Holds the window's title. It may also hold close and zoom boxes, and can act as a drag region for moving the window.
- Close box — Used for closing the window.
- Zoom box — Selects the current or alternate sizes for the window.
- R.Scroll bar — Used to scroll the data in the window vertically.
- B.Scroll bar — Used to scroll the data in the window horizontally.
- Grow box — Used to change the size of the window.
- Info bar — Provides for an additional display line in the window.

The Window Manager's main function is to keep track of overlapping windows. You can draw in any window without running over onto the windows in front of it. You can move windows to different locations on the screen, change their planes (front to back order), or change their sizes without concern for how they overlap. The Window Manager keeps track of newly exposed areas and insures that they are properly re-drawn

EDITOR OPERATION

The Window Editor is a "Desk-top" type P16 application and follows the standard conventions for desk-top applications. Support for New Desk Accessories (NDA's) is provided via the apple selection in the menu bar and an Edit menu which become activated when a "system" window is up.

To best illustrate how to create a window template we will run through an editing session and create one from scratch.

> ☞ Let's use 640 mode... if you are just starting the editor this will be the screen mode. If you want to create a window in 320 mode select MODE-320 .

① Select EDIT-NEW WINDOW. The SETTINGS dialog box will appear. (See Fig 1.2)

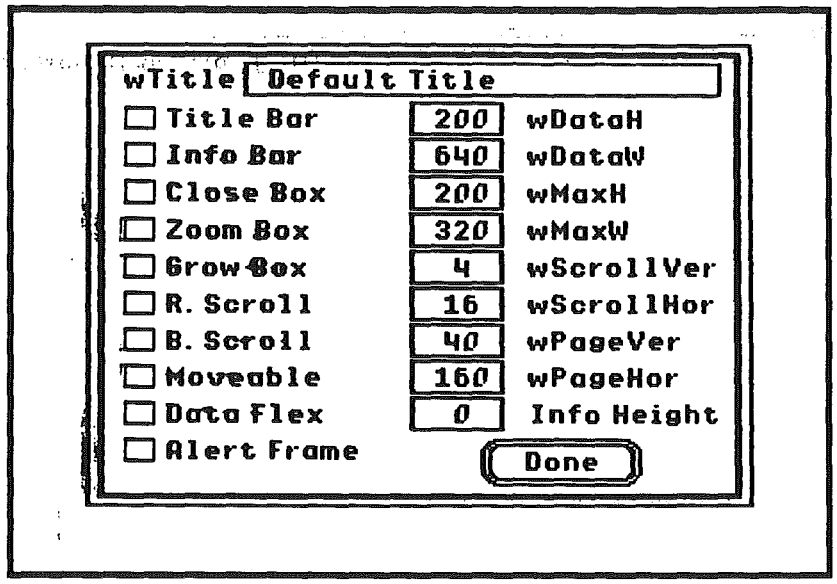


Figure 1.2 Settings Dialog Box

☞ We will create a title bar with the title of "Title" and will incorporate right and bottom scroll bars, grow box, close box, and a zoom box. We will also provide the ability to move the window around the screen by dragging the title bar.

② Click the following check boxes: Title Bar, Close Box, Zoom Box, Grow box, R.Scroll, B.Scroll, Moveable.

③ Triple-click the wTitle text "Default Title". and press the DELETE key. Type two spaces followed by Title followed by two spaces.

④ Either press the RETURN key or click on the DONE button.

Your newly created window will appear on the desktop.

⑤ Select EDIT-COLORS and the COLORS dialog box will appear. (See Fig 1.3)

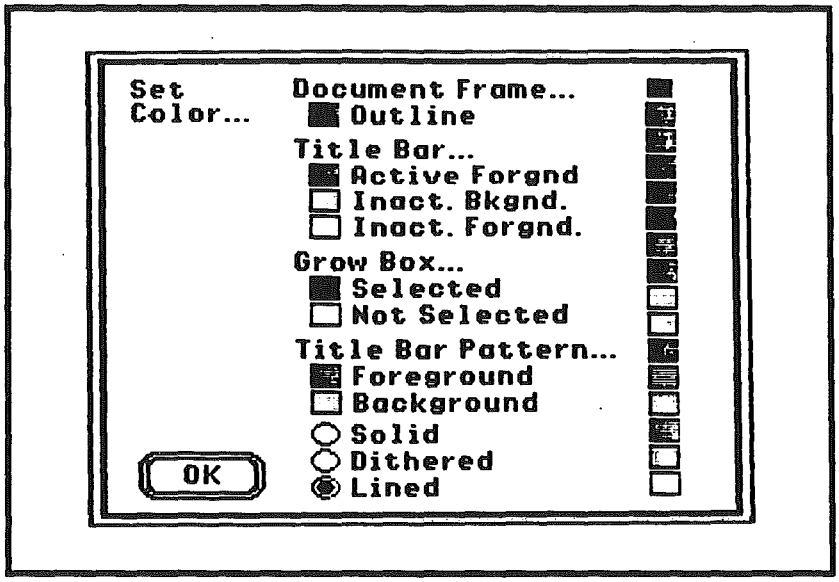


Figure 1.3 Colors Dialog Box

☞ Let's put some color into the window... color it to suit your taste.

⑥ To set colors click the color you want from the palette at the right side of the dialog box and then click the check box next to the item that you want to color. You can select the style for the title bar with the three radio buttons at the bottom of the dialog box.

⑦ Either press the RETURN key or select and click the OK button.

Your newly colored window will appear on the desktop at this time.

⑧ Select EDIT-RECTANGLES and your arrow cursor will change to a cross-hair, the window will be replaced by a rectangle on the desktop.

We will set both the normal and zoomed sizes of our window. In the right side of the menu bar you will see the word NORMAL, this indicates that the window you are seeing is the NORMAL window size. When the cursor is an arrow it can be used to operate the window just the way it will in your application, including the ability to select the ZOOMED or NORMAL sizes.

⑨ Move the cross-hair cursor to position the windows upper left corner. Drag the mouse to the lower right corner and then release the mouse button.

Your window will reappear, re-sized and positioned to fit the rectangle.

⑩ Click the Zoom box then repeat the process for the zoomed size.

This completes the creation of a window from scratch. As you can see by the dialog boxes that appeared during this editing session other items could have been selected. These items need special explanation.

Special Explanation

- Alert Frame — If the alert frame is selected then no other selections should be made. An Alert Frame has no controls.
- Info Bar — This selection must be accompanied with a height specified in Info Height, The height is usually 13. Info bars are drawn with a special procedure outlined in the Toolbox reference manual.
- wDataH wDataW — Are the height and width (in pixels) for the data area used with this window. A standard Super Hi-Res picture in 320 mode is 200 by 320 pixels. A window that holds this picture would set wDataH to 200 and wDataW to 320.
- wMaxH wMaxW — Set these the same as wDataH and wDataW.
- wScrollVer wScrollHor wPageVer wPageHor — Set the way the scroll bars behave when you select the arrows or the thumb.
- Data Flex — Provides the ability to grow and shrink the windows data area dynamically.

SAVE WINDOWS

When you have created a window template you will want to save it to disk so it can be inserted in your program code.

Select FILE-SAVE AS... and a save dialog box will appear as below.

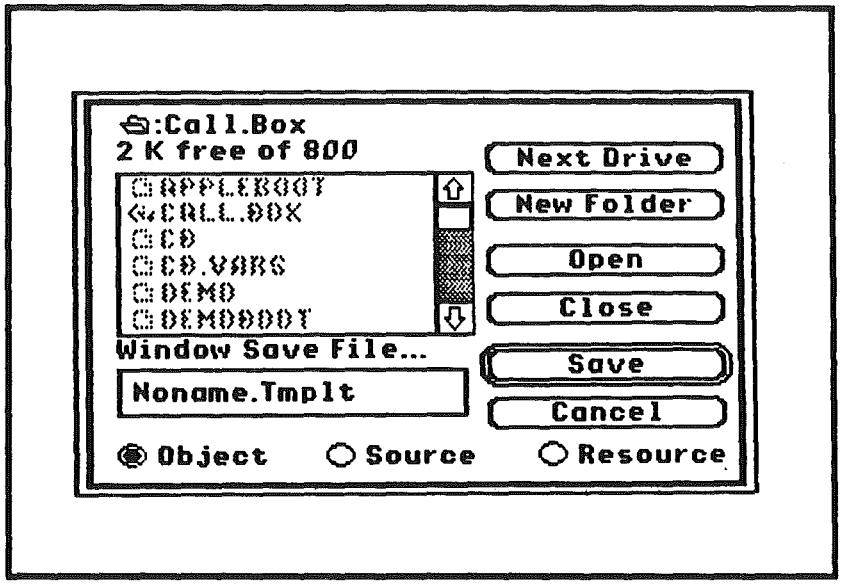


Figure 1.4 Save Dialog Box

This box has buttons to select the drive, create a new folder, open or close a folder, cancel this operation and save the file. There is also a box for typing in a filename and three radio buttons across the bottom of the dialog box. These three buttons select the type of output you will be saving.

- Object ($B1) Select this button to save the template as an OMF2 object file. This filetype can be loaded back in by the editor.
- Source ($B0) Select this button to save the template as an APW/ORCA source code file. You cannot reload this file back into the editor
- Resource (any filetype) Select this button to save the template to a resource fork of an extended ProDOS file.This filetype can be loaded back in by the editor.

Selecting either Object or Source will create or overwrite a file on disk. The operation is straight forward. Selecting Resource, however, will present some extra windows that control how resources are saved to disk. Resources are assigned types and I.D.'s, the type for a CALL-BOX window template is $1002 and is set by the editor. You only need set the I.D. for your resource. You can either rewrite an existing resource by double-clicking on its I.D. number or double-click the ---->New entry to save your resource with the next avaliable I.D. number.

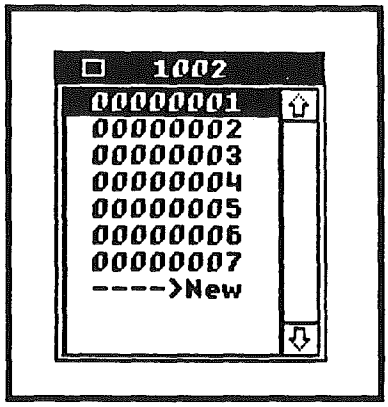


Figure 1.5 Save Resource I.D. Window

You can cancel the resource save operation by clicking on the close box in the title bar of the resource I.D. window. You can also edit the resource I.D. (renumber or delete) by first pressing and holding the OPTION key while double clicking the desired I.D. (See Fig 1.6). When renumbering resource I.D.'s be sure to use eight hex digits in the I.D. number (use leading zeros to pad small numbers). Failure to do so will cause unpredictable results and could ruin the resource fork of the ProDOS file.

If a resource fork does not exist for a given ProDOS file a dialog box will appear that gives you the option of creating one.

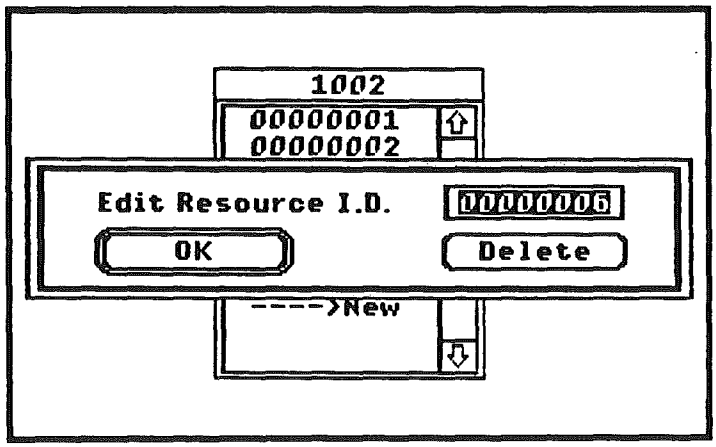


Figure 1.6 Edit Resource I.D. Dialog Box

LOAD WINDOWS

Once you have created windows and saved them to disk you may want to load them back into the editor for further editing.

Select FILE-OPEN... and a load dialog box will appearas shown.

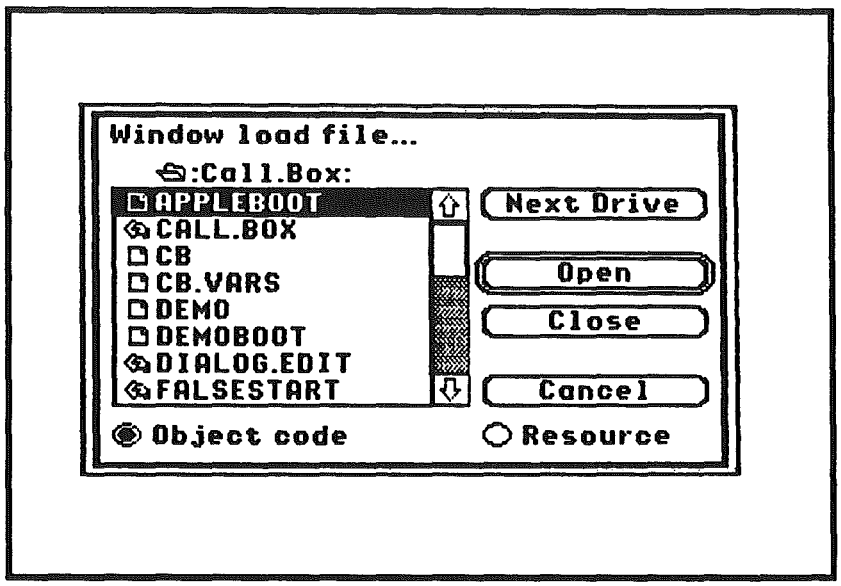


Figure 1.7 Load Dialog Box

This box has buttons to select the drive, open or close a folder, cancel the operation or open a file. There are two radio buttons at the bottom of the dialog box. These buttons select the type of input file you will be loading.

- Object ($B1) — Select this button to load an OMF2 window template file.
- Resource (any filetype) — Select this button to load the template from a resource fork of an extended ProDOS file.

Selecting Object will load a file from disk. The operation is straight forward.

Selecting Resource, however, will present some extra windows that control how resources are loaded into memory.

Resources are assigned by types and I.D.'s. The type for a CALL-BOX window template is $1002 and is set by the editor. The only thing you need to set is the I.D. for your resource. You can load a resource by double-clicking on the desired resource I.D. number. (See Fig 1.8)

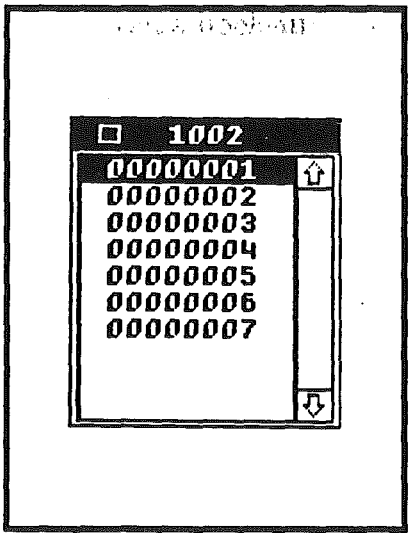


Figure 1.8 Load Resource I.D. Window

You can cancel the resource load operation by clicking the close box in the title bar of the resource I.D. window. You can also edit the resource I.D. (re-number or delete) by first pressing and holding the OPTION key while double-clicking the desired I.D. (See Fig 1.6) When re-numbering resource I.D.'s be sure to have 8 hex digits in the I.D. number window (use leading zeroes to pad small numbers).

Failure to do so will cause unpredictable results and could ruin the resource fork of the ProDOS file.If a resource fork does not exist for a given ProDOS file then no template will be loaded.

SOURCE CODE FILETYPE $B0

This code is for appending to APW/ORCA source code listings. A simple word processor can be used to edit the file, however the APW/ORCA assembler is needed to assemble the code into your application. Source code listings are easiest to hook-up to special processes, that templates might include, by allowing you to add symbolic references as required. (See Fig. 1.9)

```
WindowData      DATA

                dc i'WinEnd-WindowData'
                dc i'%1101110010000100'
                dc i4'WinTitle'
wRefCon         dc i4'0'
                dc i'0,0,200,640'
                dc i4'WinColor'
                dc i'0'
                dc i'0'
                dc i'200'
                dc i'640'
                dc i'200'
                dc i'640'
                dc i'4'
                dc i'5'
                dc i'40'
                dc i'160'
wInfoRefCon     dc i4'0'
                dc i'0'
wFrameDefProc   dc i4'0'
wInfoDefProc    dc i4'0'
wContDefProc    dc i4'0'
                dc i'39,10,160,210'
                dc i4'-1'
                dc i4'0'
WinEnd          anop

WinColor        dc h'00F0'
                dc h'F002'
                dc h'0F00'
                dc h'FF0F'

WinTitle        dc h'C',c'DefaultTitle  '

                END
```

Figure 1.9 Sample Source Code Listing

OBJECT CODE FILETYPE $B1

This type of code is for linking with the APW/ORCA linker. The code type can be used by any language that uses this linker. Object code can also be used by the loader call InitialLoad after you have changed the filetype to

$B5 (LoadFile). Use the disk utilities in the CALL-BOX shell to change the filetype of this file. (See Fig. 1.10)

```
DumpOBJ 1.1

Block count      : $00000001        1
Reserved space   : $00000000        0
Length           : $0000009E        158
Label length     : $0A              10
Number length    : $04              4
Version          : $01              1
Bank size        : $00000000        0
Kind             : $01              static data segment
Org              : $00000000        0
Alignment        : $00000000        0
Number sex       : $00              0
Language card    : $00              0
Segment number   : $0001            1
Segment entry    : $00000000        0
Disp to names    : $002C            44
Disp to body     : $0040            64
Load name        :
Segment name     : windowdata

000040 000000     | LCONST  ($F2) | 0000009E :
                  4E0080DF5800000000000000000220028
                  000000B4008024E000000000000000000
                  C8008002C800800204001002800A00000
                  000000000000000000000000000000000
                  000000000003200C8009600B801FFFFF
                  FFFFF0000000000000F0F02F000F0F0
                  0F2044656661756C74205469746C65200
                  000000000000000000000000000000000
                  000000000000000000000000000000000
                  000000000000000000000000000000000
                  0000000000000
0000E3 00009E  | cRELOC  ($F5) | 04 : 00 : 0004 : 0058
0000EA 00009E  | cRELOC  ($F5) | 04 : 00 : 0014 : 004E
0000F1 00009E  | END     ($00)
```

Figure 1.10 Sample Object Code Dump

RESOURCE FILETYPE (any)

Resources are stored in a resource fork of an extended ProDOS file. The exact filetype is not important and resources can be stored in any ProDOS file of any type.

Resources are defined with a 2 byte "type" number and a 4 byte "I.D." number. A type would be analogous to a window record, a pascal string, an icon etc... An I.D. number would identify which pascal string or which icon you are pointing to in a group of pascal strings or icons.

The type for a window template resource is $1002. The I.D.'s can be anywhere between 0 and 7FFFFFFF.

Window template resources are in OMF2 format and are loaded using the system converter.

USING SOURCE CODE

The source code created by this editor is a simple text file. It has a filetype of $B0 and is created in a form readily adaptable to source code listings created for APW or ORCA assemblers. You can use the filetype command in the APW/ORCA shell or the Disk Utilities function of the CALL-BOX shell to change the filetype.

Do not use periods (.) in the filename. This is commonplace in ProDOS, but periods are an illegal character in the assembler and will generate an error when assembled.

Window templates have pointers which are vital to making the window operate under your application. Two of these pointers are already installed (WinTitle and WinColor) but others must be put in before the window is opened. Each pointer contains a null (0) which tells the Window Manager to use internal default pointers which ignore the process calls. You must type in your own symbolic references where the directive and modifier : dc i4'0' exist in the template.

The pointers are:

wRefCon	application use
wInfoRefCon	value passed to wInfoDefProc
wFrameDefProc	window frame drawing routine
wInfoDefProc	Info bar drawing routine
wContDefProc	contents area drawing routine

The simplest way of hooking-up a CALL-BOX generated source code file to your applications source code is to use the COPY directive.

```
        .
        .
    (your code)
        .
        .
COPY Call-BoxWindow1   ;Your window template source file
        .
        .
    (your code)
        .
        .
```

Another way is to use the COPY function of the APW/ORCA editor (OpenApple-C) to put a copy of you window template source code in its SYSTEMP file. It can then be inserted into your source listing with an INSERT function (OpenApple-V).

Adapting this source code for other assemblers is up to you. We will support Apple prefered format APW or ORCA only.

USING OBJECT CODE

The object code created by the editor is a $B1 file. The file is in OMF2 format and is relocatable. The output is provided for Link-time integration or Library file use. To add an object module to a link add the filename to the link command in the APW or ORCA linker.

LINK myprogram mywindow1 mywindow2 (etc...etc).

Where mywindow1 and mywindow2 are the filenames of the object code files created with the CALL-BOX Window Editor.

This type of file can be used directly by your application similar to the way a library file is used. You must change the filetype of your object file to $B5 (Load File). The window template can then be loaded by the system loader using the InitialLoad call.

```
PushWord MyID          ;Applications I.D. number
PushLong #Pathname     ;Pointer to pathname buffer
PushWord #0            ;Spec. mem. flag (set to 0)
_InitialLoad
pla                    ;Size of Dir.pg/Stack buf.(N/A)
pla                    ;Addr of Dir.pg/Stack buf.(N/A)
PullLong WindPtr1      ;Pointer to the window template
                       ;in memory
pla                    ;Applications I.D. number
```

However you choose to use the object code template some hook-up is required for your window to properly operate in your application. Several pointers are contained in a window template that point to routines your application provides.
The pointers are:

wRefCon	application use
wInfoRefCon	value passed to wInfoDefProc
wFrameDefProc	window frame drawing routine
wInfoDefProc	Info bar drawing routine
wContDefProc	contents area drawing routine

Add the following equates to your applications source code:

```
_wRefCon          equ 8
_wInfoRefCon      equ 44
_wFrameDefProc    equ 50
_wInfoDefProc     equ 54
_wContDefProc     equ 58
_WinColor         equ 78
_WinTitle         equ 86
```

These equates are the offsets to pointers from the beginning of the window template. Use them to index into the window template or to hook the window up to your window procedures.

Let's hook-up a contents drawing routine to a window template...

```
PushLong WindPtr1           ;Put the windows pointer in
PullLong $0                 ;a direct page location
ldy #_wContDefProc          ;Load Y with the offset
lda #DrawContRoutine        ;Get the draw rout. (lo)
sta [$0],y                  ;Put it in the template
iny                         ;Advance to (hi)
iny
lda #DrawContRoutine+2      ;Get the draw rout. (hi)
sta [$0],y                  ;Put it in the template
```

Repeat this process for each reference you wish to link in. Once the template is hooked-up you can proceed with a _NewWindow call and get on with the business of being an application.

NOTE: This process also applies to window records that are stored as resources.

USING RESOURCES

All resources created by the CALL-BOX WYSIWYG editors are in OMF2 format and need to be "relocated" in memory. The Resource Manager call ResourceConverter is used to install these resources. For each resource your application is going to use you must "Log In" an appropriate OMF2 converter. To find an OMF2 converter use the Miscellaneous Tools call GetCodeResConverter. You need only make this call once.

```
PushLong #0                 ;Space for results
_GetCodeResConverter
PullLong ConverterPointer   ;Pointer to OMF2 converter
```

This call fetches a pointer to an internal OMF2 converter routine. You now need to "Log In" this converter for each resource type your application will be using (with the Resource Manager call ResourceConverter). This step is repeated for each type of relocatable resource your application will require.

```
PushLong ConverterPointer   ;OMF2 converter pointer
PushWord #$1002             ;Window Template type
PushWord #1                 ;Log In, Applic. conv. list
_ResourceConverter
bcs MemoryError
```

This sets up the resource manager to install and relocate these resources when they are called with OpenResource. You manipulate the resources from this point on. A typical sequence of events may be:

OPEN your resource file:

```
PushWord #0              ;Space for results
PushWord #0              ;Req. file access
PushLong #0              ;Res. header address
PushLong #PathName       ;Pointer to a class 1 pathname
_OpenResourceFile
PullWord FileID          ;Open resource file I.D.
```

And LOAD it into memory:

```
PushLong #0                  ;Space for results
PushWord #$1002              ;Requested Type
PushLong #1                  ;Requested I.D.
_LoadResource
PullLong ResourceHandle      ;Handle of resource in memory
```

At this point the resource is available to your application. When you are finished using the resource you can put it away with the call CloseResourceFile:

```
PushWord FileID
_CloseResourceFile
```

Be sure to "Log Out" your resource converter when your done by issueing a "Log Out" ResourceConverter call.

```
PushLong ConverterPointer    ;OMF2 converter pointer
PushWord #$1002              ;Window Template type
PushWord #0                  ;Log Out, Applic. conv. list
_ResourceConverter
bcs MemoryError
```

This covers the fundamental operation of resources in your application. There are several other functions you can perform with the Resource Manager but the previously outlined procedure should suffice for most of your CALL-BOX resource usage.

CALL-BOX Window Template resources are handled similar to object files are from within your application with the exception that the Resource Manager handles the loading and saving.

Reference: Universe ToolBox Update (Ch 21:Resource Manager 3/22/89)
Universe ToolBox Update (Ch 15:Miscellaneous Tools 3/22/89)

BASIC CONSIDERATIONS

The CALL-BOX BASIC Interface uses object code window templates. These templates are loaded into your Applesoft application as defined in the CALL-BOX BASIC Interface Manual. Windows under Applesoft are structured differently than in other languages and do not have the flexibility that other languages provide. While simpler to use from Applesoft, some functionality is lost.

Windows in Applesoft BASIC have two data areas. The additional area is a background pixel buffer where you do all your window drawing. This simplifies and standardizes the wContDefProc (contents drawing routine) and eliminates the need for the process to be programmed in BASIC. The wContDefProc is hard-wired as a _PPToPort call which uses this buffer as the source pixel image. Whatever you do with Quickdraw II in this background pixel buffer will automatically be reflected in the windows contents region.

Info Bars are not supported for windows used from Applesoft BASIC. The procedure would be too much of a hassle to be practical.

NOTE: Info bars if supported in future releases of the CALL-BOX BASIC interface, will cause the buffer overhead to double and increase the memory needed to support the window.

Custom Frame procedures are not supported for the same reason. No plans are contemplated to support this procedure in future releases.

Index of Chapter 1

CHAPTER 2 - THE DIALOG EDITOR

OVERVIEW

The CALL-BOX Dialog Editor creates templates for use by the Apple IIgs Dialog Manager. This editor can load either OMF2 object code or resources and can output APW/ORCA sourcecode, OMF2 object code and resources. The resource filetype is $1000 and uses a converter to get into memory. The standard programming procedure for resources is presented at the end of this chapter.

DIALOGS

A dialog is a presentation feature that appears when an application needs more information to carry out a command. A dialog box resembles a form on which the user checks boxes and fills in blanks.

The user supplies any necessary information in the dialog box; for example, by entering text or clicking a check box. The dialog box usually contains a button labeled OK to tell the application to accept the information provided and preform the command, and a button labeled CANCEL to cancel the command as though it had never been given.

Dialog boxes provide an alert window that displays items (Controls) for the user to select from. Several standard types are supported by this editor. (See Fig 2.11)

- Simple button — Causes an immediate or continuous action when the user clicks it with the mouse.
- Check box — Retain and display a setting, either checked(on) or not checked (off); clicking with the mouse reverses the setting.
- Radio button — Retain and display a setting. Grouped into a family in which only one button can be on at any time.
- Line edit item — Displays alphanumeric data and allows the user to edit the data from the keyboard.
- Static text — Displays text used for titles or messages that are not capable of being manipulated by the user.

- Icons, pixel images — For use as alert icons such as the STOP or CAUTION Icons.

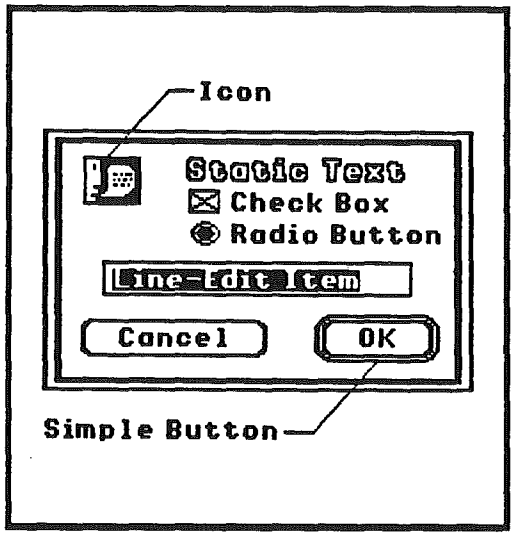


Figure 2.11 Typical Dialog Box

The Dialog Manager's main function is to present controls until a valid selection is made at which point the manager returns information to the user on what was selected or changed.

EDITOR OPERATION

The Dialog Editor is a "DeskTop" type P16 application and follows the standard conventions for desktop applications. Support for new desk accessories (NDA's) is provided via the apple selection in the menu bar and an Edit menu which become activated when a "system" window is up.

To best illustrate how to create a dialog template we will run through an editing session and create one from scratch.

> Let's make this dialog in 320 mode... if you are just starting the editor the screen mode will be 640. If you want to create a dialog in 320 mode select 320MODE from the GOODIES menu.

① Select FILE-NEW. A dialog box will appear.

② Select CONTROLS-DRAGGING, MODIFYING and when you move the cursor in the dialog box it will change to a hand for dragging things around.

Very Important!

This will be the normal cursor mode, the arrow is just for test operating your dialog box.

You are now ready to put some dialog items in your dialog box. Let's use one of each type to get some practice.

③ Select CONTROLS-STANDARD BUTTON followed by the other five types...CHECK BOX, RADIO BUTTON, ICON, LINE EDIT BOX and TEXT.

This wil put a mish-mosh of items in your dialog box one on top of the other.

④ Use the mouse and the hand cursor to drag the items off of each other and place them so that each one can be double-clicked.

You can grab and drag the right and bottom edge of the dialog box to either grow or shrink it and even grab it somewhere in the middle where it's free of an item and move the whole dialog box around.

To set-up a particular dialog item you must double-click the item in question. A dialog box editor will appear tailored for the particular item.

⑤ Double-click the Standard Button item in your dialog box. (See Fig 2.12)

A dialog box will appear. Change something and then select DONE to see the results. Repeat this several times to learn how to operate this part of the editor. A detailed explaination of these functions may introduce more doubt and uncertainty than if you work it out for yourself.

This learning technique applies to the following five steps and to life in general. Enough philosophy, fiddle around with the other items in the dialog box.

⑥ Double-click the Check Box . (See Fig 2.13)

⑦ Double-click the Radio button. (See Fig 2.14)

⑧ Double-click Icon (See Fig 2.15)

⑨ Double-click Text (See Fig 2.16)

⑩ Double-click Line Edit (See Fig 2.17)

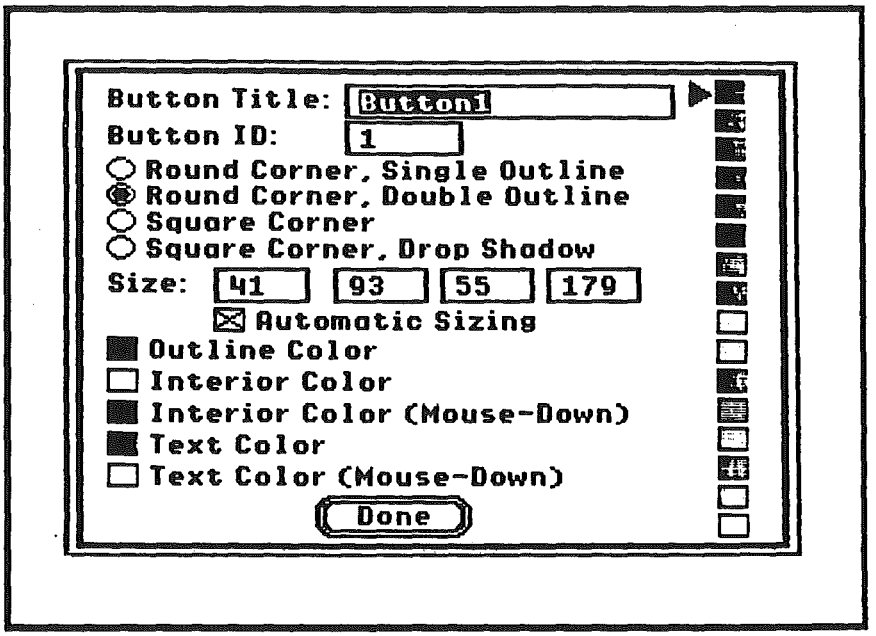


Figure 2.12 Standard Button Edit Dialog

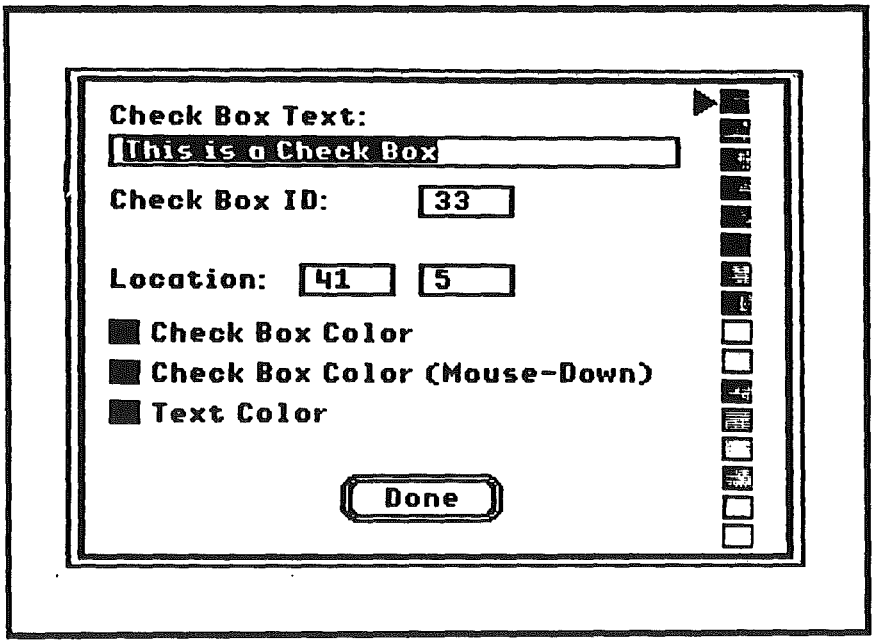


Figure 2.13 Check Box Edit Dialog

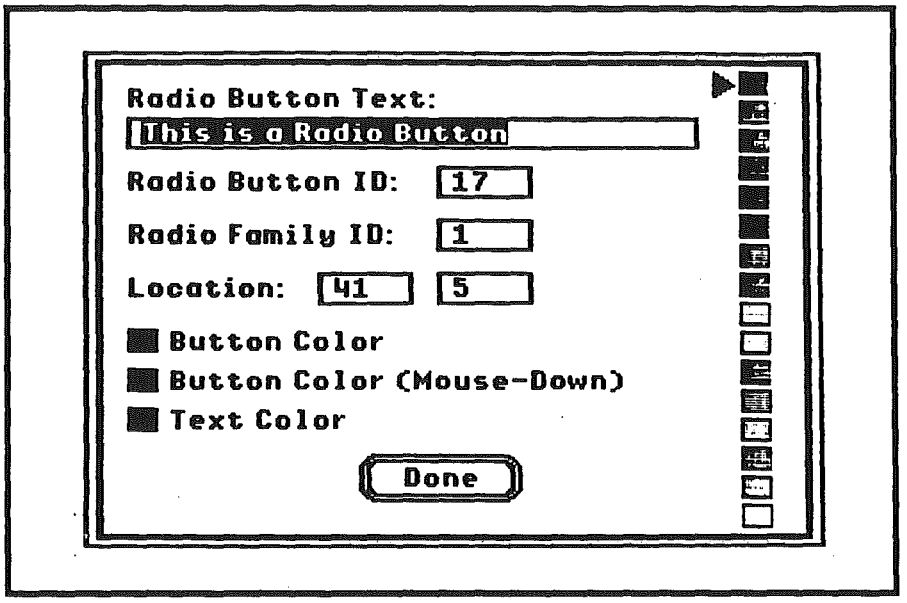


Figure 2.14 Radio Button Edit Dialog

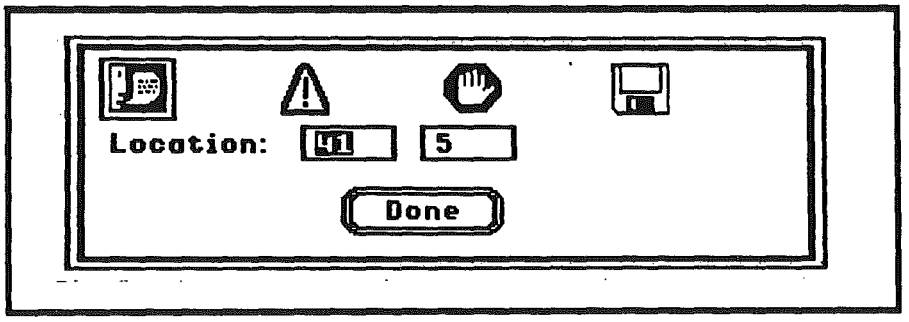


Figure 2.15 Icon Edit Dialog

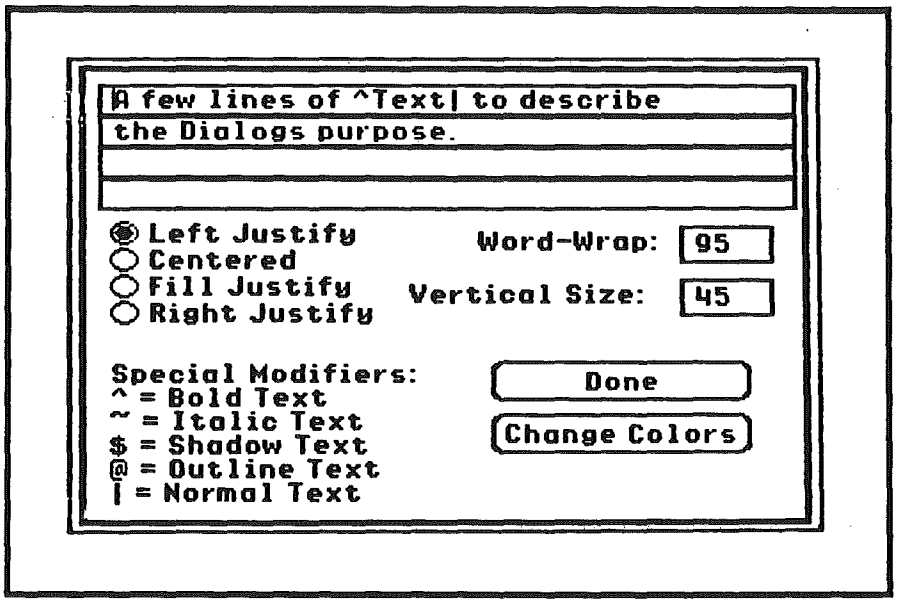


Figure 2.16 Text Edit Dialog

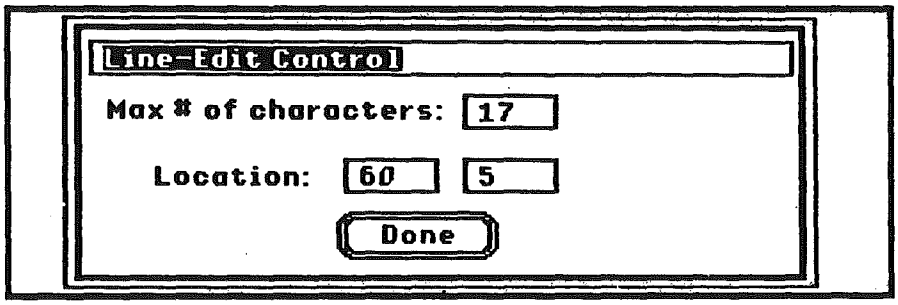


Figure 2.17 Line Edit Dialog Box

Several selections in the GOODIES menu will aid you when aligning and centering :

- Horizontal Grid — Enables or disables an invisible snap grid for item placement.
- Align Controls — Snaps items to the position of the invisible grid.
- Center Dialog — Moves the entire dialog box to the center of the screen.
- 2/3 Center — Moves the entire dialog box to center horizontally and two thirds of the way up from the bottom of the screen.

Fine adjustments can be made to an items placement by setting the coordinates numerically while each item is being edited.

To delete an item hold the OPTION key while double-clicking it and an option to delete it will come up.

This about finishes up creating your dialog box, you could save it at this point but let's cover some other things about this editor first.

Special Explanation

• The menu selection GOODIES-DISPLAY INFO will bring up a control info scroll window. This window represents a list of items in your dialog box that is kept internally in the editor.

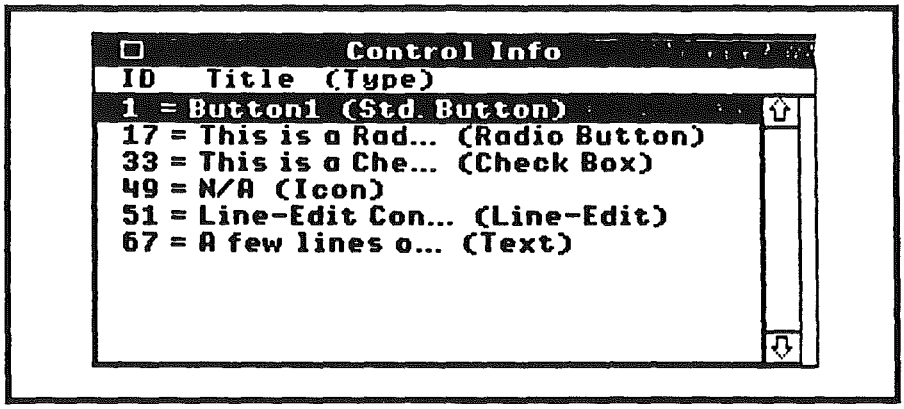


Figure 2.18 Info Window

Double-clicking an item in this window will bring up the edit dialog for that item the same as double-clicking the item itself.

- The menu selection GOODIES-PRINT INFO will dump this list to your printer. The list is very handy for subsequent identification of items and their I.D. numbers. The menu selection FILE-CHOOSE PRINTER will select the proper driver for your printer. Your printer driver must be in the SYSTEM/DRIVERS subdirectory of your boot volume.

- Do not be switch modes after a dialog box is started. Colors become strange and the rectangular limits often go askew.

- The menu selection CONTROLS-NORMAL RESPONSE will change the cursor to the system arrow and allow you to test your dialog box.

SAVE DIALOGS

Once you have created a dialog template you will want to save it to disk so it can be incorporated in your program code.

Select FILE-SAVE AS... A save dialog box will appear as shown.

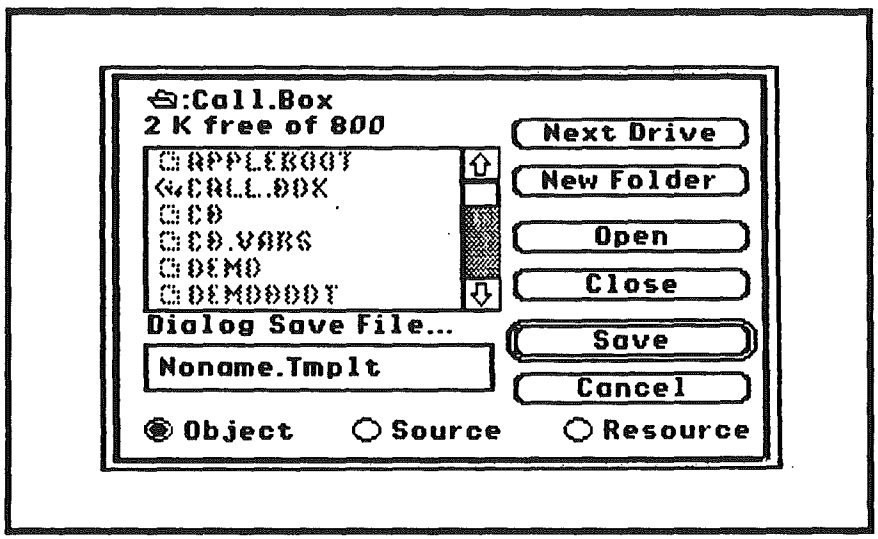


Figure 2.19 Save Dialog Box

This box has buttons to select the drive, create a new folder, open or close a folder, cancel the operation and save the file. There is also a box for typing in a filename and three radio buttons across the bottom of the dialog box. These three buttons select the type of output you will be saving.

- Object ($B1) — Select this button to save the template as an OMF2 object file. This filetype can reloaded by the editor.
- Source ($B0) — Select this button to save the template as an APW/ORCA source code file. This filetype can not be reloaded by the editor.
- Resource (any filetype) — Select this button to save the template to a resource fork of an extended ProDOS file. This filetype can be reloaded by the editor.

Selecting either Object or Source will create or overwrite a file on disk. The operation is straight forward. Selecting Resource, however, will present extra windows that control how resources are saved to disk.
Resources are assigned types and I.D.'s. The type for a CALL-BOX dialog template is $1000 and is set by the editor. You only need set the I.D. for your resource. You can either rewrite an existing resource by double-clicking on

its I.D. number or double-clicking the ---->New entry to save with the next avaliable I.D. number. (Fig 2.20)

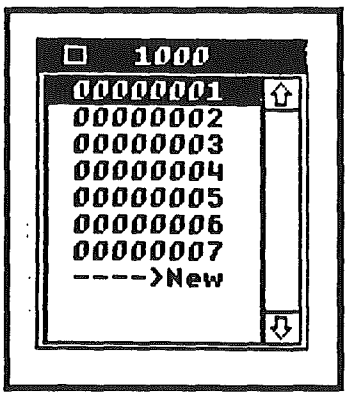


Figure 2.20 Save Resource I.D. Window

You can cancel the resource save operation by clicking the close box in the title bar of the resource I.D. window. You can also edit the resource I.D. (re-number or delete) by pressing and holding the OPTION key while double-clicking the desired I.D. (See Fig 2.21) When re-numbering resource I.D.'s be sure to use 8 hex digits in the I.D. number window (use leading zeroes to pad small numbers). Failure to do so will cause unpredictable results and could ruin the resource fork of the ProDOS file.

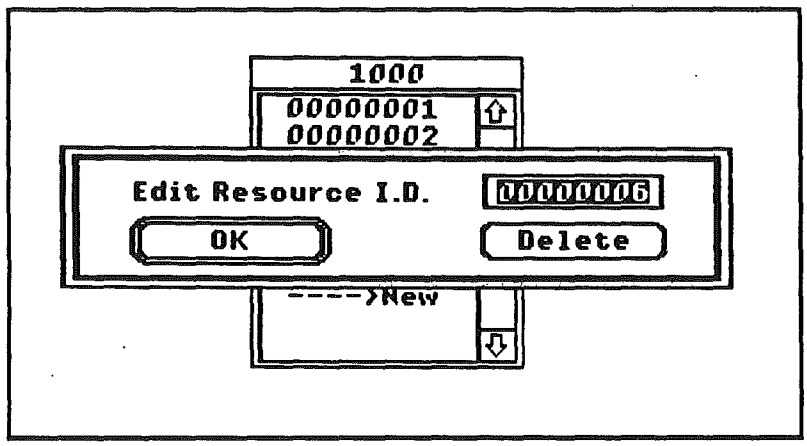


Figure 2.21 Edit Resource I.D. Dialog Box

If a resource fork does not exist for a given ProDOS file a dialog box will appear that gives you the opportunity to create one.

LOAD DIALOGS

Once you have created dialogs and saved them to disk you may want to load them to the editor for further editing.

Select FILE-OPEN... A load dialog box will appear. (See Fig 2.22)
This box has buttons to select the drive, open or close a folder, cancel this operation and open the file. There are 2 radio buttons at the bottom of the dialog box. These select the type of input you will be loading.

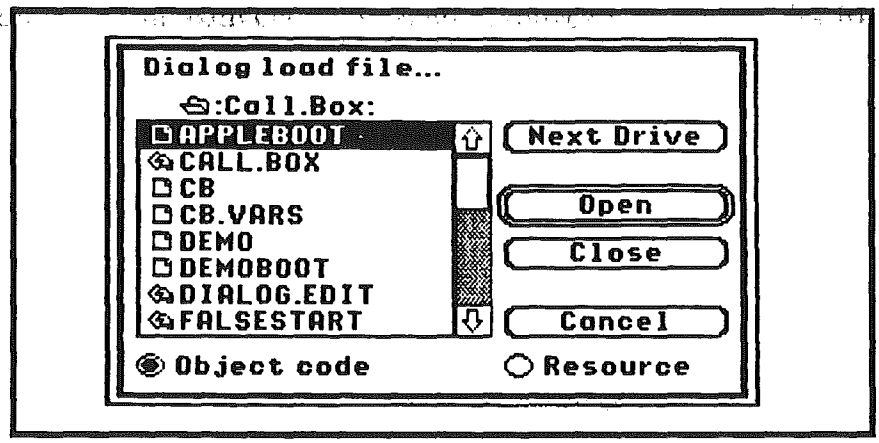


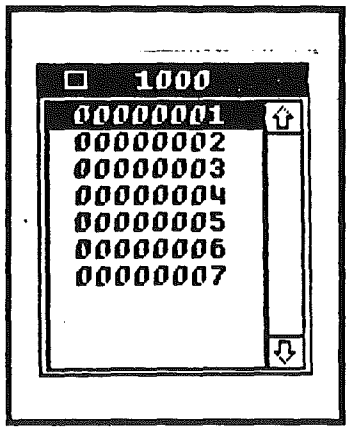
Figure 2.22 Load Dialog Box

- Object, ($B1) — Select this button to load an OMF2 type of dialog template file.

- Resource (any filetype) — Select this button to load the template from a resource fork of an extended ProDOS file.

Selecting Object will load a file from disk and the operation is straight forward. Selecting Resource, however, will present some extra windows that control how resources are loaded to memory.

Resources are assigned types and I.D.'s. The type for a CALL-BOX dialog template is set to $1000 by the editor. You set is the I.D. for your resource. You can load a resource by double-clicking on the desired I.D. number.



You can cancel the resource load operation by clicking the close box in the title bar of the resource I.D. window. You can also edit the resource I.D. (re-number or delete) by first pressing and holding the OPTION key while double-clicking the desired I.D. (See Fig 2.21) When re-numbering resource I.D.'s be sure to use 8 hex digits in the I.D. number window (use leading zeroes to pad small numbers). Failure to do so will cause unpredictable results and could ruin the resource fork of the ProDOS file.

Figure 2.23 Load Resource I.D. Window

If a resource fork does not exist for a given ProDOS file then no template will be loaded.

SOURCE CODE FILETYPE $B0

This type of code is for appending to APW/ORCA source code listings. A simple word processor is adequate for editing this file.

```
MyDialog    DATA
            dc i'52,27,140,192'
            dc i'$FFFF'
            dc i4'0'
            dc i4'Item1'
            dc i4'Item18'
            dc i4'Item33'
            dc i4'0'
Item1       dc i'1'
            dc i'66,109,80,156'
            dc i'$000A'
            dc i4'BTitle1'
            dc i'0'
            dc i'0'
            dc i4'BColor1'
BTitle1     str 'OK'
BColor1     dc i'$0000'
            dc i'$00F0'
            dc i'$0000'
            dc i'$00F0'
            dc i'$000F'
Item18      dc i'18'
            dc i'29,54,38,160'
            dc i'$000C'
            dc i4'RTitle18'
            dc i'1'
            dc i'$0001'
            dc i4'RColor18'
RTitle18    str 'Radio Button'
RColor18    dc i'$0000'
            dc i'$00F0'
            dc i'$00F0'
            dc i'$00F0'
Item33      dc i'33'
            dc i'17,53,26,138'
            dc i'$000B'
            dc i4'CTitle33'
            dc i'1'
            dc i'0'
            dc i4'CColor33'
CTitle33    str 'Check Box'
CColor33    dc i'$0000'
            dc i'$00F0'
            dc i'$00F0'
            dc i'$00F0'
            END
```

Figure 2.24 Sample source code listing

OBJECT CODE FILETYPE $B1

This type of code is for linking with the APW/ORCA linker. The code type can be used by any language that uses this linker. Object code can also be used by the loader call InitialLoad after you have changed the filetype to $B5 (LoadFile). Use the disk utilities in the CALL-BOX shell to change the filetype. (See Fig 2.25)

RESOURCE FILETYPE (any)

Resources are stored in a resource fork of an extended ProDOS file. The exact filetype is not important and in fact resources can be stored in any ProDOS file of any type.

Resources are refered to by a 2 byte "type" number and a 4 byte "I.D." number. A type would be analogous to a window record, a pascal string, an icon etc... An I.D. number would identify which pascal string or which icon you are pointing to in a group of pascal strings or icons.

The type for a dialog template resource is $1000. The I.D.'s can be from 0 to 7FFFFFFF.

Dialog template resources are in OMF2 format and are loaded using the system converter.

```
DumpOBJ 1.1

Block count          :$00000001          1
Reserved space       :$00000000          0
Length               :$000000B3          179
Label length         :$0A                10
Number length        :$04                4
Version              :$01                1
Bank size            :$00000000          0
Kind                 :$01                static data segment
Org                  :$00000000          0
Alignment            :$00000000          0
Number sex           :$00                0
Language card        :$00                0
Segment number       :$0001              1
Segment entry        :$00000000          0
Disp to names        :$002C              44
Disp to body         :$0040              64
Load name            :
Segment name         :

000040 000000        | LCONST  ($F2) |  000000B3 :
                                       2D00BB00AA00C401FFFF000000001E0000
                                       00480000007F0000000000000001002800
                                       5A003600B0000A0000000000000010001
                                       0001000742757474 6F6E310000F0000000
                                       F0000F0011004F000500580088000C0000
                                       0000000100010000000000165468697320
                                       6973206120526164696F20427574746F6E
                                       0000F000F000F000210030000 5003900A4
                                       000B0000000000010000000000000001354
                                       6869732069732061204368656 36B20426F
                                       780000F000F000F000
0000F8 0000B3        | cRELOC  ($F5)      | 04 : 00 : 000E : 001E
0000FF 0000B3        | cRELOC  ($F5)      | 04 : 00 : 0012 : 0048
000106 0000B3        | cRELOC  ($F5)      | 04 : 00 : 0016 : 007F
00010D 0000B3        | cRELOC  ($F5)      | 04 : 00 : 002A : 0036
000114 0000B3        | cRELOC  ($F5)      | 04 : 00 : 0032 : 003E
00011B 0000B3        | cRELOC  ($F5)      | 04 : 00 : 0054 : 0060
000122 0000B3        | cRELOC  ($F5)      | 04 : 00 : 005C : 0077
000129 0000B3        | cRELOC  ($F5)      | 04 : 00 : 008B : 0097
000130 0000B3        | cRELOC  ($F5)      | 04 : 00 : 0093 : 00AB
000137 0000B3        | END     ($00)
```

Figure 2.25 Sample Object Code Dump

USING SOURCE CODE

The source code created by this editor is a simple text file. It has a filetype of $B0 and is created in a form readily adaptable to source code listings created for APW or ORCA assemblers. You can use the filetype command in the APW/ORCA shell or the Disk Utilities function of the CALL-BOX shell to change the filetype.

Do not use periods (.) in the filename. This is commonplace in ProDOS, but periods are an illegal character in the assembler and will generate an error when assembled.

Dialog templates have a pointer table to access the various items they contain. Under normal operation you would not access these records directly but rather with tool calls designed specifically for that purpose. Sometimes, however, you need to set default conditions that may not be set in the loaded dialog template. This is where you break the rules and access the items directly.

The process is simple: Index to the table item you want to work on and use the address found there as the address to the dialogs item. At this point you need to index to the specific piece of data and make your change.

No other special considerations need to be made to use these dialog templates.

The simplest way of hooking-up a CALL-BOX generated source code file to your applications source code is to use the COPY directive.

```
        .
        .
    (your code)
        .
        .
  COPY CallBoxDialog1   ;Your dialog template source file
        .
        .
    (your code)
        .
        .
```

Another way is to use the COPY function of the APW/ORCA editor (OpenApple-C) to put a copy of your window template source code in its SYSTEMP file, which can then be inserted into your source listing with an INSERT function (OpenApple-V).

Adapting this source code for other assemblers is up to you. We will support Apple prefered format APW or ORCA only.

USING OBJECT CODE

The object code created by the dialog editor is a $B1 file. This type of file is in OMF2 format and is relocatable. This form of output is provided for Link-time integration or Library file use. To add an object module to a link add the filename to the link command in the APW or ORCA linker.

LINK myprogram mydialog1 mydialog2 (etc...etc)

Where mydialog1 and mydialog2 are the filenames of the object code files created with the CALL-BOX Dialog Editor.

This type of file can be used directly by your application similar to the way a library file is used. You must change the filetype of your object file to $B5 (Load File). The dialog template can then be loaded by the system loader using the InitialLoad call.

```
PushWord MyID          ;Applications I.D. number
PushLong #Pathname     ;Pointer to pathname buffer
PushWord #0            ;Spec. mem. flag (set to 0)
_InitialLoad
pla                    ;Size of Dir.pg/Stack buf.(N/A)
pla                    ;Addr of Dir.pg/Stack buf.(N/A)
PullLong DlogPtr1      ;Pointer to the dialog template
                       ;in memory
pla                    ;Applications I.D. number
```

This is all that is required to install this template into your program. Use standard dialog box operating procedures as outlined in the Toolbox reference manuals.

NOTE: This process applies to dialog records that are stored as resources as well.

USING RESOURCES

Resources created by the CALL-BOX WYSIWYG editors are in OMF2 format and must be "relocated" in memory. The Resource Manager call ResourceConverter is used to install the resources. For each type of resource your application is going to use you must "Log In" an OMF2 converter. To find an OMF2 converter use the Miscellaneous Tools call GetCodeResConverter. You need only make this call once.

```
PushLong #0                ;Space for results
_GetCodeResConverter
PullLong ConverterPointer  ;Pointer to OMF2 converter
```

This call fetches a pointer to an internal OMF2 converter routine. You now need to "Log In" this converter for each resource type using the Resource Manager call ResourceConverter. This step is repeated for each different type of relocatable resource your application will need.

```
PushLong ConverterPointer     ;OMF2 converter pointer
PushWord #$1000               ;Dialog Template type
PushWord #1                   ;Log In, Applic. conv. list
_ResourceConverter
bcs MemoryError
```

This sets up the resource manager to install and relocate these resources when they are called with OpenResource. You can now manipulate the resources from this point on. A typical sequence of events from this point may be:

OPEN your resource file:

```
PushWord #0                   ;Space for results
PushWord #0                   ;Req. file access
PushLong #0                   ;Res. header address
PushLong #PathName            ;Pointer to a class 1 pathname
_OpenResourceFile
PullWord FileID               ;Open resource file I.D.
```

And LOAD it :

```
PushLong #0                   ;Space for results
PushWord #$1000               ;Requested Type
PushLong #1                   ;Requested I.D.
_LoadResource
PullLong ResourceHandle       ;Handle of resource in memory
```

At this point the resource is avaliable to your application. When you are done using this resource you can put it away with the Resource Manager call CloseResourceFile:

```
PushWord FileID
_CloseResourceFile
```

Be sure to "Log Out" your resource converter when your finished by issuing a Log Out ResourceConverter call.

```
PushLong ConverterPointer     ;OMF2 converter pointer
PushWord #$1000               ;Dialog Template type
PushWord #0                   ;Log Out, Applic. conv. list
_ResourceConverter
bcs MemoryError
```

This covers the fundamental operation of resources in your application. There are several other functions you can perform with the Resource Manager but the previously outlined procedure will suffice for most of your CALL-BOX resource usage.

CALL-BOX Dialog Template resources are handled the same as object files are in your application except that the Resource Manager handles the loading and saving.

Reference: Universe ToolBox Update (Ch 21:Resource Manager 3/22/89)
Universe ToolBox Update (Ch 15:Miscellaneous Tools 3/22/89)

BASIC CONSIDERATIONS

The CALL-BOX BASIC Interface uses object code dialog templates. These templates are loaded into your Applesoft application as defined in the CALL-BOX BASIC Interface Manual. Dialogs under Applesoft are structured the same as if under a P16 application and need no special care or feeding.

Direct template access as presented in USING OBJECT CODE is possible but difficult from the CALL-BOX BASIC interface. The procedure is the same but all indexing and addressing must be done with LONG PEEK and LONG POKE, calls. Fortunately these commands are capable of specifying the values in either decimal, hex, or binary and can handle WORD and LONG values.

Index of Chapter 2

CHAPTER 3 - MENU EDITOR

CHAPTER 3 - MENU EDITOR

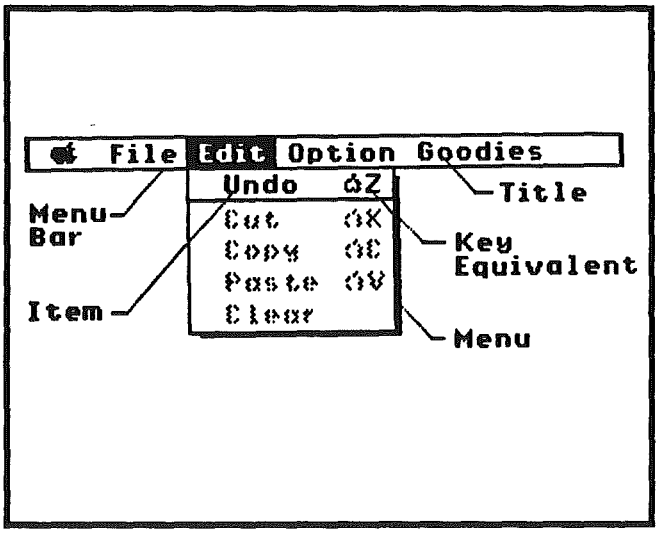
OVERVIEW

The CALL BOX Menu Editor creates templates for use by the Apple IIgs Menu Manager. This editor can load either OMF2 object code or resources and can output APW/ORCA sourcecode, OMF2 object code and resources. The resource filetype is $1001 and requires a converter to load into memory. The standard programming procedure for resources is presented at the end of this chapter.

ABOUT MENUS

A menu provides a means of displaying choices or options avaliable to the user without having to remember selections,words or special keys.

Menus appear as a panel usually located at the top of the screen,. Each menu has a title and a pull-down menu associated with it. The menu contains a series of selections or "Items" available to the user.
(See Fig. 3.26)



An I.D. number is associated with each title and item. The I.D. number is used by the menu manager to inform you when a title or item is selected.

Figure 3.26
Typical Menu Bar and Menu

To use the menu bar the user selects a menu bar title. The title bar is then hi-lited and a pull-down menu appears directly below the title.. Keeping the mouse button pressed and moving the cursor up and down the menu, hilites items as you move. When you hilite the menu item you want to select simply release the mouse button and the menu manager will return the I.D. number for the selected item.

Menus can be customized to a certain extent with this editor, some of the features are:

- Key Equivalents, An alternate way of selecting menu items from the keyboard.
- Text styles You can select bold, underline or italisized for the items text.
- Check Character You can add check marks, diamonds, open and closed-apples.
- Underlines and dividing lines, add visual divisions for groups of like items in menus.
- Disabling Lets you select only applicable items.

EDITOR OPERATION

The Menu Editor is a "Desk-top" type P16 application and follows the standard conventions for desk-top applications. Support for New Desk Accessories (NDA's) is provided via the apple selection in the menu bar and an Edit menu which activates when a "system" window is up.

To best illustrate how to create a Menu template let's run through an editing session and create one from scratch.

> ☞ Let's make this menu in 640 mode. The menu editor functions in 640 mode. You can test in either mode, but the actual editing process is in 640 .

①. Select FILE-NEW and the BUILD MENU window will appear. (See Fig 3.27)

The menu bar has the Apple menu already installed, so let's put an ABOUT item in the menu...

② Click the NEW ITEM button and a new item will appear in the scroll window.

③. Type the word ABOUT.... (add three periods just for style). Press RETURN when you are done.

This is how you build a menu, you can add dividing lines, delete or replace items and titles by hiliting them in the scroll window and clicking the desired button.

Once you have your menu selections in place you can try your menu by selecting the menu bar selection GOODIES-TRYIT 320 or GOODIES-TRYIT 640. Press the OPTION key to return to the build window.

There is an additional button present in the build window called ATTRIBUTES. This permits each menu items appearance and mode of operation to be set to reflect a particular status.

4. Select the ATTRIBUTES button in the build window (See Fig. 28)

This window allows you to

set the style of the items text,
select enabled or disabled (dimmed)
add a narrow dividing line
add one of 4 item markers and keypress equivalents.

Set the attributes for your menu items so that the menu bar is structured the way you want it at initialization.

This completes the creation of a menu template there is one more feature which is covered in the next section.

SPECIAL EXPLANATION

- The selection FILE-PRINT INFO will print you a list of the menu items cross-referenced by their I.D. numbers. this list is very helpful when integrating your menu into your application.

- The editor edits in 640 mode only! A menu for 320 mode or 640 mode can be created with the editor. To see the menu in different modes use GOODIES-TRYIT 320 and GOODIES-TRYIT 640.

- The menu selection GOODIES-DISPLAY INFO will bring up a Control Info scroll window. This window represents a list of the items in your menu box that is kept internally in this editor.(See Fig. 2.18)

  Double-clicking an item in this window will bring up the item edit menu for that item the same as double-clicking the item itself.

This window allows you to

set the style of the items text,
select enabled or disabled (dimmed)
add a narrow dividing line
add one of 4 item markers and keypress equivalents.

Set the attributes for your menu items so that the menu bar is structured the way you want it at initialization.

This completes the creation of a menu template there is one more feature which is covered in the next section.

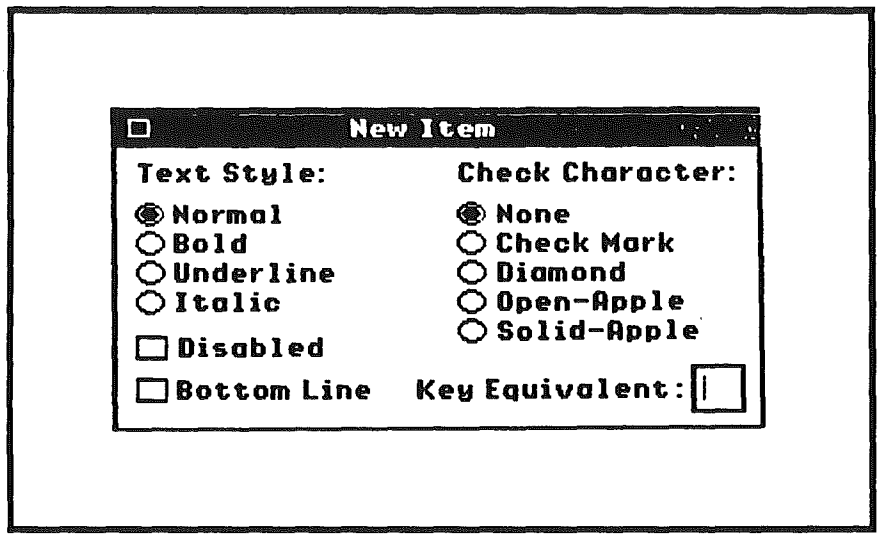


Figure 3.28 Attributes Window

Special Explanation

- The selection FILE-PRINT INFO will print you a list of the menu items cross-referenced by their I.D. numbers. this list is very helpful when integrating your menu into your application.

- The editor edits in 640 mode only! A menu for 320 mode or 640 mode can be created with the editor. To see the menu in different modes use GOODIES-TRYIT 320 and GOODIES-TRYIT 640.

SAVE MENUS

Once you have created a menu template you will want to save it to disk so it can be incorporated into your program code.

Select FILE-SAVE AS... and a save dialog box will appear.

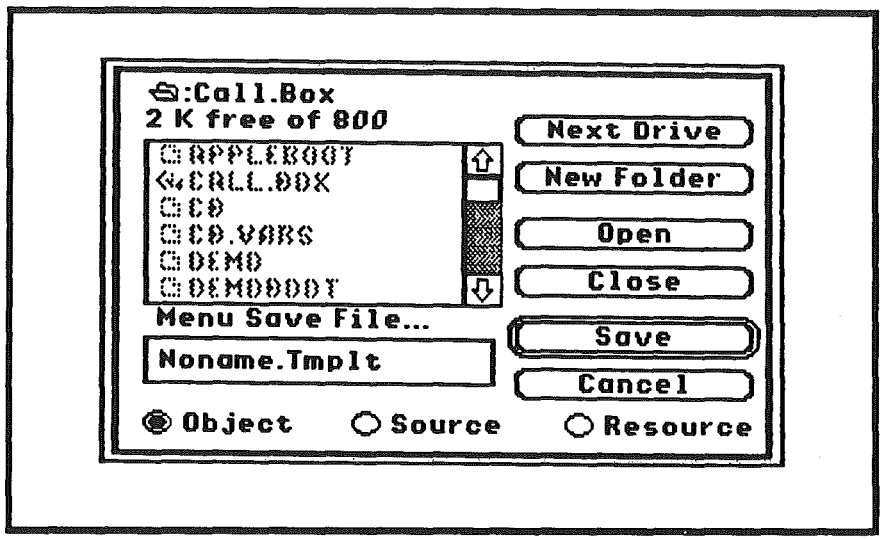


Figure 3.29 Save Dialog Box

This box has buttons to select the drive, create a new folder, open or close a folder, cancel this operation and save the file. There is also a box for typing in a filename and 3 radio buttons across the bottom of the menu box. These three buttons select the type of output you will be saving.

• Object ($B1)	Select this button to save the template as an OMF2 object file. This filetype can be loaded back in by the editor.
* Source ($B0)	Select this button to save the template as an APW/ORCA source code file. This filetype can not be loaded back into the editor.
• Resource (any filetype)	Select this button to save the template to a resource fork of an extended ProDOS file. This filetype can be loaded back in by the editor.

Selecting either Object or Source will create or overwrite a file on a disk and the operation is pretty straight foreward. Selecting Resource however will present some extra windows that control how resources are saved to disk. Resources come in types and I.D.'s, the type for a CALL BOX menu template is $1000 and is hard-set by the editor... the only thing you need to set is the I.D. for your resource. You can either rewrite an existing resource by double-clicking on its I.D. number if one exists or double-clicking the ---->New entry to save your resource as the next avaliable I.D. number. (See Fig 3.30).

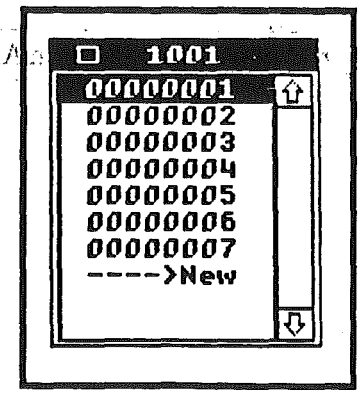


Figure 3.30 Save Resource I.D. Window

You can cancel the resource save operation by clicking the close box in the title bar of the resource I.D. window. You can also edit the resource I.D. (re-number or delete) by first pressing and holding the OPTION key while double-clicking the desired I.D. (See Fig 3.31).

When re-numbering resource I.D.'s be sure to have 8 hex digits in the I.D. number window (use leading zeroes to pad small numbers). Failure to do so will cause unpredictable results and could ruin the resource fork of the ProDOS file.

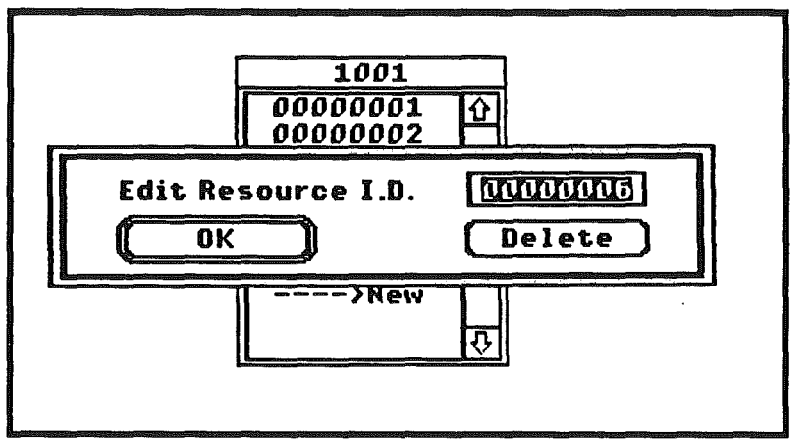


Figure 3.31 Edit Resource I.D. Dialog Box

If a resource fork does not exist for a given ProDOS file a dialog box will appear that gives you the option of creating one.

Once you have created menus and saved them to disk you may want to load them back into this editor for further editing.

Select FILE-OPEN... and a load dialog box will appear.

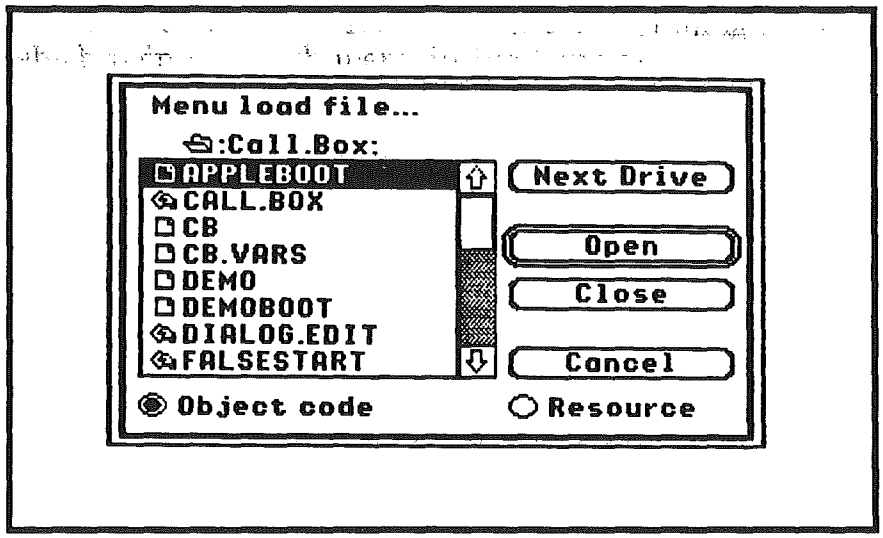


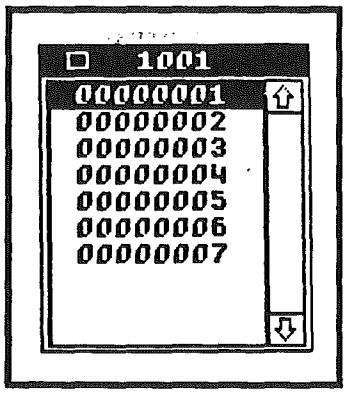
Figure 3.32 Load Dialog Box

This box has buttons to select the drive, open or close a folder, cancel this operation and open the file. There are radio buttons to select the type of input you will be loading.

- Object ($B1) — Select this button to load an OMF2 type of menu template file.
- Resource (any filetype) — Select this button to load the template from a resource fork of an extended ProDOS file.

Selecting Object will load a file from disk and the operation is pretty straight foreward. Selecting Resource however will present some extra windows that control how resources are loaded into memory.

Resources come in types and I.D.'s, the type for a CALL BOX menu template is $1001 and is hard-set by the editor... the only thing you need to set is the I.D. for your resource. You can load a resource by double-clicking on the desired resource I.D. number. (See Fig 3.33)



You can cancel the resource load operation by clicking the close box in the title bar of the resource I.D. window. You can also edit the resource I.D. (re-number or delete) by first pressing and holding the OPTION key while double-clicking the desired I.D. (See Fig 3.31) When re-numbering resource I.D.'s be sure to use 8 hex digits in the I.D. number window (use leading zeroes to pad small numbers). Failure to do so will cause unpredictable results and could ruin the resource fork of the ProDOS file.

Figure 3.33 Load Resource I.D. Window

If a resource fork does not exist for a given ProDOS file then no template will be loaded.

SOURCE CODE FILETYPE $B0

This type of code is for appending to APW/ORCA source code listings. A simple word processor is adequate for editing the file.

The source code for a menu bar and menus indicates the different menus it contains with symbolic references, object code and resources on the other hand have a dialog template style pointer table added to the beginning of the menu template.

```
MyMenu    DATA

Menu1     anop
          DC C'$$@\N1X',H'0D'
          DC C'>> About... \N256',H'0D'
          DC C'.'
Menu2      anop
          DC C'$$ File  \N2',H'0D'
          DC C'>> Open... \N257',H'0D'
          DC C'>> Close... \N258',H'0D'
          DC C'>> Quit \N259*Qq',H'0D'
          DC C'.'

Menu3     anop
          DC C'$$ Edit  \N3',H'0D'
          DC C'>> Undo \N260V*Zz',H'0D'
          DC C'>> Cut \N261*Xx',H'0D'
          DC C'>> Copy \N262*Cc',H'0D'
          DC C'>> Paste \N263*Vv',H'0D'
          DC C'>> Clear \N264',H'0D'
          DC C'.'

          END
```

Figure. 3.34 Sample Source Code Listing

OBJECT CODE FILETYPE $B1

This type of code is for linking with the APW/ORCA linker. This code type can be used by any language that uses this linker. Object code can also be used by the loader call InitialLoad after you have changed the filetype to $B5 (LoadFile). Use the disk utilities in the CALL BOX shell to change the filetype of this file. (See Fig. 3.35)

```
DumpOBJ 1.1

Block count          :$00000001          1
Reserved space       :$00000000          0
Length               :$000000D6          214
Label length         : $0A               10
Number length        :$04                4
Version              :$01                1
Bank size            :$00000000          0
Kind                 :$01      static data segment
Org                  :$00000000          0
Alignment            :$00000000          0
Number sex           :$00                0
Language card        :$00                0
 Segment number      :$0001              1
Segment entry        :$00000000          0
Disp to names        :$002C              44
Disp to body         :$0040              64
Load name      :
Segment name         :

000040 000000 | LCONST  ($F2) | 000000D6 :
                                740000002D000000100000000000000024
                                244C5C4E31580D3E3E2041626F75742E2E
                                2E205C4800012A00200D2E24242C204669
                                6C6520205C4802000D3E3E204F70656E2E
                                2E2E205C4801012A00200D3E3E20436C6F
                                73652E2E2E205C4802012A00200D3E3E20
                                51756974205C4803012A51710D2E242420
                                204564697420205C4803000D3E3E20556E
                                646F205C480401562A5A7A0D3E3E204375
                                74205C4805012A58780D3E3E20436F7079
                                205C4806012A43630D3E3E205061737465
                                205C4807012A56760D3E3E20436C656172
                                205C4808012A00200D2E
00011B 0000D6 | cRELOC  ($F5) | 04 : 00 : 0000 : 0074
000122 0000D6 | cRELOC  ($F5) | 04 : 00 : 0004 : 002D
000129 0000D6 | cRELOC  ($F5) | 04 : 00 : 0008 : 0010
000130 0000D6 | END    ($00)
```

Figure. 3.35 Sample Object Code Dump

RESOURCE FILETYPE (any)

Resources are stored in a resource fork of an extended ProDOS file. The exact filetype is not important and in fact resources can be stored in any ProDOS file of any type.

Resources are referred to by a two byte "type" number and a 4 byte "I.D." number. A type would be analogous to a window record, a pascal string, an icon etc... An I.D. number would identify which pascal string or which icon you are pointing to in a group of pascal strings or icons.

The type for a menu template resource is $1001. The I.D.'s can be anywhere from 0 to 7FFFFFFF.

Menu template resources are in OMF2 format and are loaded using the system converter.

USING SOURCE CODE

The source code created by this editor is a simple "TEXT" file. It has a filetype of $B0 and is created in a form readily adaptable to source code listings created for APW or ORCA assemblers. You can use the filetype command in the APW/ORCA shell or the Disk Utilities function of the CALL BOX shell to change this files filetype.

Each source code file created by this editor needs to have a filename that has no (.) peroids in it. This is commonplace in ProDOS, but peroids are an illegal character in the assembler and will generate an error when assembled.

The simplest way of hooking-up a CALL BOX generated source code file to your applications source code is to use the COPY directive.

```
.
.
(your code)
.
.
 COPY CallBoxmenu1   ;Your menu template source file
.
.
(your code)
.
.
```

Another way is to use the COPY function of the APW/ORCA editor (OpenApple-C) to put a copy of you window template source code in its SYSTEMP file which can then be inserted into your source listing with an INSERT function (OpenApple-V).

Adapting this source code for other assemblers is up to you. We will support Apple prefered format like APW or ORCA only on this editor.

USING OBJECT CODE

The object code created by this editor is a filetype $B1 file. This type of file is in OMF2 format and is relocatable. This form of output is provided for Link-time integration or Library file like use. To add an object module to a link add the filename to the link command in the APW or ORCA linker.

LINK myprogram mymenu1 mymenu2 (etc...etc).

Where mymenu1 and mymenu2 are the filenames of the object code files created with the CALL BOX menu Editor.

This type of file can be used directly by your application like a library file is used. You must change the filetype of your object file to $B5 (Load File) and then the menu template can be loaded by the system loader using the InitialLoad call.

```
PushWord MyID           ;Applications I.D. number
PushLong #Pathname      ;Pointer to pathname buffer
PushWord #0             ;Spec. mem. flag (set to 0)
_InitialLoad
pla                     ;Size of Dir.pg/Stack buf.(N/A)
pla                     ;Addr of Dir.pg/Stack buf.(N/A)
PullLong MenuPtr1       ;Pointer to the menu template
;                       ;in memory
pla                     ;Applications I.D. number
```

This is all that is required to install this template into your program. Inserting the menu in the system is slightly different than usual.

Menu templates have a pointer table at the beginning of them used to access the various menus they contain. _InsertMenu needs the address of the particular menu record to build the menu in the systems memory. You would usually point to a menu record and then call _InsertMenu for each menu your menu bar will contain. When this menu template is in OMF2 form you will not know where a particular menu record is at, the only thing you will know for certain is where the menu template begins. Fortunately

we have added an address table at the beginning of the menu template which points to each menu in the template.

To insert this type of menu bar use the following algorithmn:

```
          ldy #0
          PushLong #TmpltAddress      ;Get the tmplt addr. in z-page
          PullLong $0

Again     lda [$0],y                  ;Fetch the table address
          sta ThisMenu
          iny
          iny
          lda [$0],y
          sta ThisMenu+2
          ora ThisMenu                ;Is it null?
          beq AllDone                 ;If so then done inserting

          phy                         ;Preserve the index
          PushLong #ThisMenu          ;Push the handle
          PushWord #0                 ;and the flag
          _InsertMenu                 ;Insert this menu
          ply                         ;Restore index
          iny                         ;Advance to the next pointer
          iny
          bra Again                   ;Loop back

AllDone   rts                         ;EXIT!!!
```

NOTE: This process applies to menu templates that are stored as resources as well

USING RESOURCES

All resources created by the CALL BOX WYSIWYG EDITORS are in OMF2 format and need to be "relocated" into memory. The Resource Manager call ResourceConverter is used to install these resources in memory. For each type of resource your application is going to use you must "Log In" an OMF2 converter for that type. To find an OMF2 converter use the Miscellaneous Tools call GetCodeResConverter. You need only make this call once.

```
PushLong #0                 ;Space for results
_GetCodeResConverter
PullLong ConverterPointer   ;Pointer to OMF2 converter
```

This call fetches a pointer to an internal OMF2 converter routine. You now need to "Log In" this converter for each resource type your application will be using with the Resource Manager call ResourceConverter. This step would be repeated for each different type of relocatable resource your application will need.

```
PushLong ConverterPointer   ;OMF2 converter pointer
PushWord #$1001             ;menu Template type
PushWord #1                 ;Log In, Applic. conv. list
_ResourceConverter
bcs MemoryError
```

This sets up the resource manager to install and relocate these resources when they are called with OpenResource. You can now OPEN, LOAD, UPDATE or whatever to the resources from this point on. A typical sequence of events from this point may be:
OPEN your resource file:

```
PushWord #0             ;Space for results
PushWord #0             ;Req. file access
PushLong #0             ;Res. header address
PushLong #PathName      ;Pointer to a class 1 pathname
_OpenResourceFile
PullWord FileID         ;Open resource file I.D.
```

And LOAD it into memory:

```
PushLong #0                 ;Space for results
PushWord #$1001             ;Requested Type
PushLong #1                 ;Requested I.D.
_LoadResource
PullLong ResourceHandle     ;Handle of resource in memory
```

At this point the resource is avaliable to your application. When you are done using this resource you can put it away with the Resource Manager call CloseResourceFile:

```
PushWord FileID
_CloseResourceFile
```

Be sure to "Log Out" your resource converter when your done by issuing a Log Out type ResourceConverter call.

```
        PushLong ConverterPointer    ;OMF2 converter pointer
        PushWord #$1001              ;menu Template type
        PushWord #0                  ;Log Out, Applic. conv. list
        _ResourceConverter
        bcs MemoryError
```

This covers the fundamental operation of resources in your application. There are several other functions you can perform with the Resource Manager but the previously outlined procedure will suffice for most of your CALL BOX resource useage.

CALL BOX menu Template resources are handled the same as object files are from within your application except that the Resource Manager handles the loading and saveing.

Reference: Universe ToolBox Update (Ch 21:Resource Manager 3/22/89)
Universe ToolBox Update (Ch 15:Miscellaneous Tools 3/22/89)

BASIC CONSIDERATIONS

The CALL BOX BASIC Interface uses object code menu templates. These templates are loaded into your Applesoft application with syntax as defined in the CALL BOX BASIC Interface Manual. Menus under Applesoft need no special care and feeding.

Index of Chapter 3

CHAPTER 4 - THE IMAGE EDITOR

CHAPTER 4 - THE IMAGE EDITOR

ABOUT IMAGES

Images are pictures... this is very obvious but what is not obvious is the various types that the Apple IIgs uses.

Most people are familiar with PIC images.made by commercial "paint programs". PIC images can be in either 320 or 640 mode. [320 x 200] or [640 x 200] pixels. 320 mode PIC images can use 16 different solid colors while 640 mode PIC images can use only 4. You will see 640 mode images that appear to have more than 4 colors but don't be fooled. There are only four possible solid colors. When you see a 640 mode picture that appears to have more than four colors it is using dithered colors which are two dissimilar 640 mode pixels next to each other. The combination will appear as a third color to the eye. These images can be said to be 320 mode in a 640 mode framework.

This can be confusing. It is not too important that understand the details ... (but it is nice if you do!) The image editor handles these things automatically and provides you with 16 colors in both modes.

The Apple IIgs uses other images based on the above. This editor can create, Pixel Images, Icons and cursors.

A Pixel Image is a PIC Image that is less than 320 x 200 or 640 x 200 pixels in size... kind of a mini-PIC.. Pixel images are always rectangular.

Icons are similar to pixel images except that they contain a mask. A mask is an image the same size as the main pixel image that signifies which pixels will show up and which ones will not. This allows you to create images that are not rectangular. They can be of any shape, and even have holes in them like a doughnut.

The last type of image is a Cursor. Cursors are similar to an Icon but are handled differently by the system. The ARROW is an example of a cursor image. Cursors have an attribute that other images do not have, this is a "Hot Spot". A Hot Spot is that pixel out of the image that represents the cursors actual X and Y position. The tip of the ARROW cursor is its hot spot.

The image editor makes it very easy to load one type of image and save it as another type. The main power of this editor, however, is in enabling the user to take a standard "Paint Program" type picture and capture a portion of it as either a pixel image, Icon or cursor.

EDITOR OPERATION

To use this editor you must put an image in the "capture window", by loading in any of the seven input types previously described using the menu bar selection File/Open. or by selecting File/New320 or File/New640. If you are using a Filetype $C1 picture as an image source the editor will switch to the mode (640 or 320) that the picture was created in. You will have to use File/New320 or File/New640 to set the mode prior to loading the pixel image, icon or cursor types.

Once you have an image (even if its a blank screen for creating an image from scratch) you must select Edit/Capture and using the full window crosshair cursor click-drag-click a rectangle around the image you want to process. The image will slowly invert and an indicator bar will appear reporting the progress of the magnify operation taking place.

This magnify operation creates three windows containing the image just captured in a magnified form called "Image". The mask image (always completely black after capture) in a magnified form called a "Mask" and the captured image as an icon called an "Icon". The Image and Mask windows are editable with the "Pencil" cursor which is visible when the cursor position is in either window when it is the top-most or active window. The menu bar selection "Color" is now active and contains a color palette used to set the pencil color. You can now use the pencil to draw to the magnified image or mask.

Three functions help in drawing to the magnified images. Edit/Fill Image will change every pixel in the image window to the current color of the pencil. Edit/Fill Mask will set all the pixels in the mask window to black or white.(on or off) using color #15 as white and any other color as black. Edit/Image to Mask will set all pixels that correspond to colored pixels in the image window to black (on) in the mask window and set all corresponding white (color #15) pixels to white (off).

If you need to reframe your image you can select Edit/Re-Capture which clears out the capture window and plots your current magnified image in the upper left region of the capture window. You then need to select Edit/Capture and capture your image again only this time at new rectangular coordinates which will in turn create the Image, Mask and Icon windows again. You can continue editing if needed.

If you are creating a cursor you will need to set the "Hot Spot" with the menu selection Edit/Set Hot Spot. The hot spot is represented by a black rectangle initially around the upper left pixel of the magnified image. Simply select Edit/Set Hot Spot and click on the magnified pixel you want for the hot spot. The rectangle will move to surround the selected pixel.

When everything is just the way you want it select File/Save as.to preserve your handy-work. Be sure to set the radio buttons at the bottom of the save window for the type and style.

This outlines the major features and procedures implemented in using this editor, we will now discuss some of the finer points of image editing.

Special Explanations

• All images, either 320 or 640 mode can be treated like 320 mode images. Most good looking 640 mode images are actually in dithered colors which is simply two pixels of different colors side by side that appear like one larger pixel in a color that is the mix of the two colors. This way you can get sixteen apparently different solid colors in 640 mode that only has a palette of four colors (read about mini-palettes in the IIgs toolbox reference under Quickdraw II). If you are using dithering, and you probably will be, then you are actually handling a 320 mode image because one apparent pixel is composed of two smaller (640 mode) pixels.

• You can capture a 320 mode image, save it as a pixel image, switch modes using File/New640 and re-load it for editing in the new mode, the reverse is also true. Naturally the colors will be off but this is not important. A Pixel image, Icon or cursor responds to the current system palette your program is using, not necessarily the one that you are currently editing in. You are just producing a byte pattern that represents what color numbers to use and where they are to be used.

• 640 mode images sometimes exhibit a color shift which depends on where the viewable image has its left edge (on an even or odd numbered position). This is due to the mysteries of Dithered colors and poses no problems to your finished image. The image in the Icon window is a true representation of the colors in your finished image, the colors in the Image window may or may not be accurate in 640 mode depending on how the horizontal scroll bar is set.

• Cursors have a unique mode sensitivity problem which deals with the hot spot. If a cursor is created in 320 mode and then used in 640 mode the cursor will appear the same in both modes. The difference will be in the actual position of the hot spot. To best illustrate this anomaly an example is needed:

• Create a cursor in 320 mode that is a framed rectangle 5 x 5 pixels. Set the hot spot at 3,3 (Right in the middle of the framed rectangle). Use this cursor in 640 mode and the hot spot will seem to shift to the left of the rectangle. The hot spot numbers will still be 3,3 but the rectangle will no longer be 5 x 5! A 320 mode 5 x 5 rectangle becomes

a 640 mode 5 x 10 rectangle to maintain the same appearance. To rectify this situation you should use separate cursors for 320 and 640 modes. You could also directly rewrite the hot spot numbers based on the current mode your program is in. If your cursors hot spot is at a horizontal position of 0 or 1 don't worry. The mode shift is half the distance from the original hot spot to the left edge of the cursor.(if your position is 0 or 1 the shift is so slight as to not be noticeable).

• As if cursors did not cause enough problems there is yet another thing you have to look out for. Add one extra word (4 magnified pixels) to the right side of your cursor image and fill them with zeroes (black in the image and white in the mask). The way the Apple handles cursors necessitates this. I won't get into it here. Failure to add this word will cause you equipment to behave strangely. Check it out for yourself its actually quite interesting to see.

• The last thing we want to tell you about cursors is that you need a black (color #0) as the background of the image window and white (color #15) to draw your image. The colors are reversed in the image window when creating cursors.

• This editor is memory hungry and should be used with all the free memory you can summon up (at least 3 or 4 banks). The maximum size an image can be is directly linked to how much free memory is available. Images in excess of 75 pixels square should be avoided, (images that large plot slowly). Secondly you will run out of memory in the editor. You can go as large as memory will allow. We recommend not exceeding 75 by 75 which should be sufficiently large for most applications.

• You can close all of the windows except the Icon window by clicking in the close box of the windows frame. This will make the windows disappear. You can make the image and mask windows re-appear by selecting Edit/Image Edit or Edit/Mask Edit. The capture window does not have a selection to make it re-appear. You will have to re-load an image or use File/New320 or File/New640 to make the capture window appear.

SAVE IMAGES

Once you have created an image you will want to save it to disk so it can be incorporated into your program code.

Select FILE-SAVE AS... and a save dialog box will appear. (See Fig 4.36)

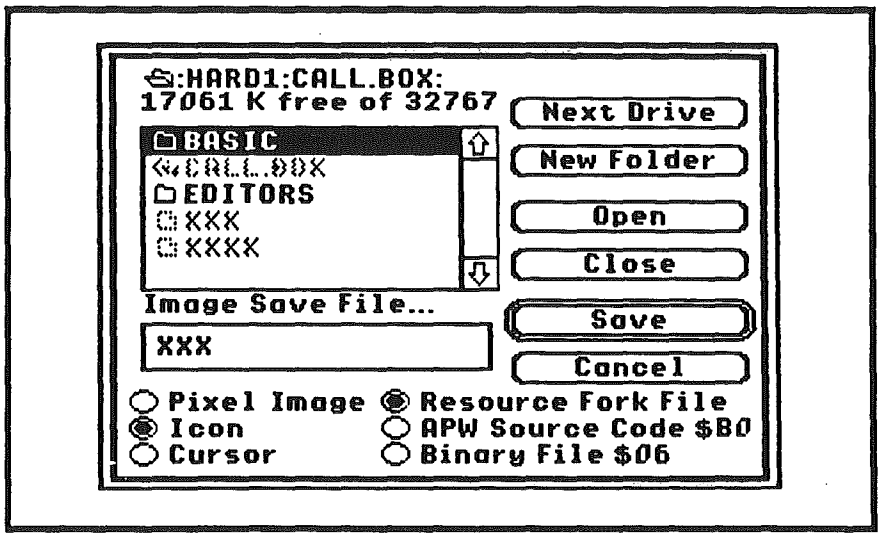


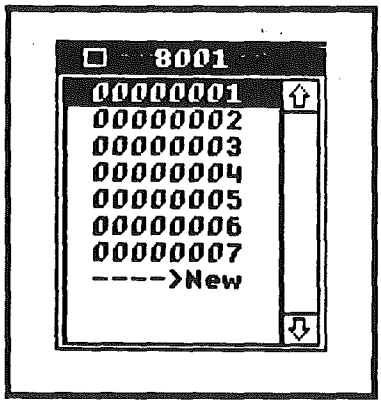
Figure 4.36 Save Dialog Box

This box has buttons to select the drive, create a new folder, open or close a folder, cancel the operation and save the file. There is also a box for typing in a filename and 6 radio buttons across the bottom of the dialog box. These six buttons select the type of output you will be saving.

Three buttons on the left side of the dialog box select the style of of image you can save (Pixel image, icon or cursor). The three buttons on the right side select the filetype as Resource, Source or Binary.

Images do not need to be relocated in memory because they do not have absolute address references. This is why a binary form is provided instead of object code. Both source and binary saves are fairly straight forward and need no real explanation. Selecting Resource however will present some extra windows that control how resources are saved to disk.

Resources are identified by types and I.D.'s. The type for CALL BOX images are $1003 for pixel images, $8001 for icons and $1004 for cursors. These types are set by the editor, you need to set the I.D. for your resource. You can either rewrite an existing resource by double-clicking on its I.D. number or double-clicking
the ---->New entry to save your resource as the next available I.D. number. (See Fig 4.37)



You can cancel the resource save operation by clicking the close box in the title bar of the resource I.D. window. You can also edit the resource I.D. (re-number or delete) by pressing and holding he OPTION key while double-clicking the desired I.D. (See Fig 4.38). When re-numbering resource I.D.'s be sure to use 8 hex digits (use leading zeroes to pad small numbers). Failure to do so will cause unpredictable results and could ruin the resource fork of the ProDOS file.

Figure 4.37 Save Resource I.D. window

If a resource fork does not exist for a given ProDOS file a dialog box will appear giving you the option of creating one.

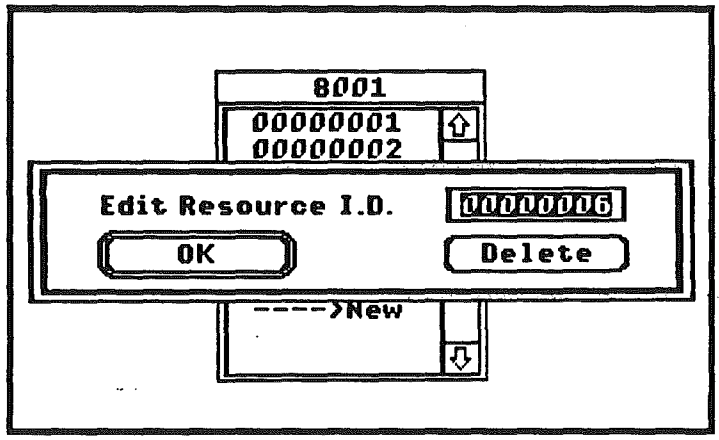


Figure 4.38 Edit Resource I.D. Dialog Box

LOAD IMAGES

Once you have created images and saved them to disk you may want to load them back into this editor for further editing.

Select FILE-OPEN... and a load dialog box will appear.

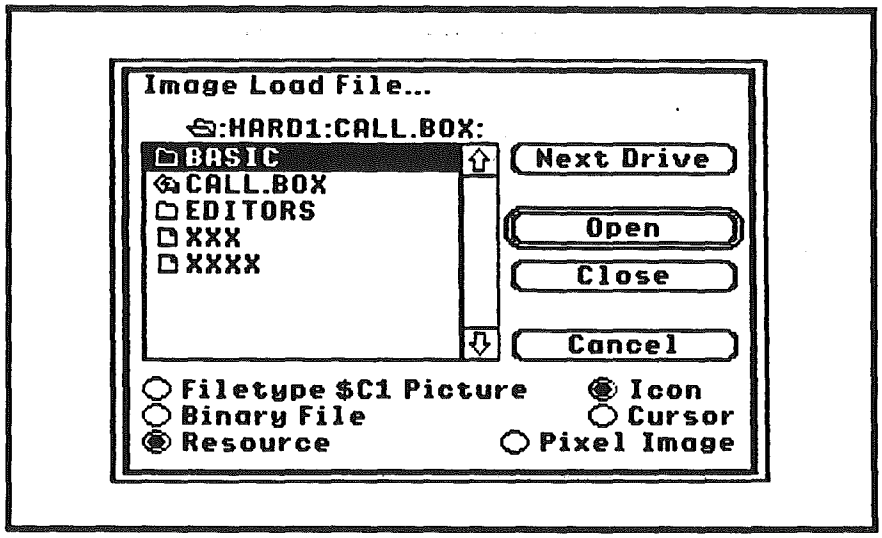


Figure 4.39 Load Dialog Box

This box has buttons to select the drive, open or close a folder, cancel this operation and open the file. There are six radio buttons across the bottom of the dialog box. These six buttons select the type of input you will be loading.

When loading images you must be very careful to have these buttons set properly for the type of input you will be loading. All filetypes will appear selectable in the scroll window and no special filtering is provided due to the fact that resources can be contained in any filetype. Loading the wrong type of file can result in a crash! You will just have to be careful.

Selecting Resource will present some extra windows that control how resources are loaded into memory.

Resources are assigned by types and I.D.'s. The type for CALL BOX images are $1003 for pixel images, $8001 for icons and $1004 for cursors. These types are set by the editor.you set just the I.D. for your resource. You can either rewrite an existing resource by double-clicking on its I.D. number or double-clicking the ---->New entry to save your resource as the next available I.D. number. (See Fig 40)

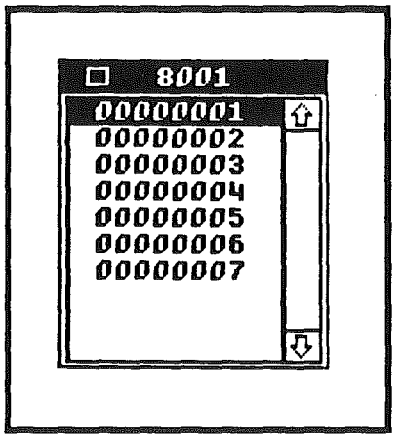


Figure 40 Load Resource I.D. Window

You can cancel the resource save operation by clicking the close box in the title bar of the resource I.D. window. You can also edit the resource I.D. (re-number or delete) by using the OPTION key while double-clicking the desired I.D. (See Fig 38) When re-numbering resource I.D.'s be sure to use 8 hex digits in the I.D. number (use leading zeroes to pad small numbers). Failure to do so will cause unpredictable results and could ruin the resource fork of the ProDOS file.

If a resource fork does not exist for a given ProDOS file a dialog box will appear giving you the option of creating one.

SOURCE CODE FILETYPE $B0

This type of code is for appending to APW/ORCA source code listings. A simple word processor is adequate for editing this file.

The three styles of image output are presented here using the same captured image.

```
Pixelimage DATA

           dc h'1100'
           dc h'1400'


           dc h'FFFFFFF44444FFFFFFFF'
           dc h'FFFF444EEE66444FFFFF'
           dc h'FF444EE444EE6664FFFF'
           dc h'F4666446664EEE664FFF'
           dc h'46446664466EEEEE64FF'
           dc h'44FF4E4FF466EEEE64FF'
           dc h'4FF114FF1146EEEEE64F'
           dc h'4FFFF4FFFF46EEEEE64F'
           dc h'44FF444FF466EEEEE64F'
           dc h'4644EEE446EEEEEEE64F'
           dc h'4EE46E646EEEEEEEE64F'
           dc h'F4E644466EE666EE64FF'
           dc h'F4EE6666EEEE466E64FF'
           dc h'FF4EEEEEE444E6E64FFF'
           dc h'FFF4666EEEE66664FFFF'
           dc h'FFFF44466666444FFFFF'
           dc h'FFFFFFF44444FFFFFFFF'

           END
```

Figure. 4.41 Sample Pixel Image Source Code Listing

```
icon        DATA

            dc h'0100'
            dc h'AA00'
            dc h'1100'
            dc h'1400'

            dc h'FFFFFFF44444FFFFFFFF'
            dc h'FFFF444EEE66444FFFFF'
            dc h'FF444EE444EE6664FFFF'
            dc h'F4666446664EEE664FFF'
            dc h'46446664466EEEEE64FF'
            dc h'44FF4E4FF466EEEE64FF'
            dc h'4FF114FF1146EEEEE64F'
            dc h'4FFFF4FFFF46EEEEE64F'
            dc h'44FF444FF466EEEEE64F'
            dc h'4644EEE446EEEEEEE64F'
            dc h'4EE46E646EEEEEEEE64F'
            dc h'F4E644466EE666EE64FF'
            dc h'F4EE6666EEEE466E64FF'
            dc h'FF4EEEEEE444E6E64FFF'
            dc h'FFF4666EEEE66664FFFF'
            dc h'FFFF44466666444FFFFF'
            dc h'FFFFFFF44444FFFFFFFF'

            dc h'0000000FFFFF00000000'
            dc h'0000FFFFFFFFFFF00000'
            dc h'00FFFFFFFFFFFFFF0000'
            dc h'0FFFFFFFFFFFFFFFF000'
            dc h'FFFFFFFFFFFFFFFFFF00'
            dc h'FF00FFF00FFFFFFFFF00'
            dc h'F00FFF00FFFFFFFFFFF0'
            dc h'F0000F0000FFFFFFFFF0'
            dc h'FF00FFF00FFFFFFFFFF0'
            dc h'FFFFFFFFFFFFFFFFFFF0'
            dc h'FFFFFFFFFFFFFFFFFFF0'
            dc h'0FFFFFFFFFFFFFFFFF00'
            dc h'0FFFFFFFFFFFFFFFFF00'
            dc h'00FFFFFFFFFFFFFFF000'
            dc h'000FFFFFFFFFFFFF0000'
            dc h'0000FFFFFFFFFFF00000'
            dc h'0000000FFFFF00000000'

            END
```

Figure. 4.42 Sample icon source code listing

```
cursor      DATA

            dc h'1100'
            dc h'0500'

            dc h'FFFFFFF44444FFFFFFFF'
            dc h'FFFF444EEE66444FFFFF'
            dc h'FF444EE444EE6664FFFF'
            dc h'F4666446664EEE664FFF'
            dc h'46446664466EEEEE64FF'
            dc h'44FF4E4FF466EEEE64FF'
            dc h'4FF114FF1146EEEEE64F'
            dc h'4FFFF4FFFF46EEEEE64F'
            dc h'44FF444FF466EEEEE64F'
            dc h'4644EEE446EEEEEEE64F'
            dc h'4EE46E646EEEEEEEE64F'
            dc h'F4E644466EE666EE64FF'
            dc h'F4EE6666EEEE466E64FF'
            dc h'FF4EEEEEE444E6E64FFF'
            dc h'FFF4666EEEE66664FFFF'
            dc h'FFFF44466666444FFFFF'
            dc h'FFFFFFF44444FFFFFFFF'

            dc h'0000000FFFFF00000000'
            dc h'0000FFFFFFFFFF00000'
            dc h'00FFFFFFFFFFFFFF0000'
            dc h'0FFFFFFFFFFFFFFF000'
            dc h'FFFFFFFFFFFFFFFFF00'
            dc h'FF00FFF00FFFFFFFFF00'
            dc h'F00FFF00FFFFFFFFFF0'
            dc h'F0000F0000FFFFFFFF0'
            dc h'FF00FFF00FFFFFFFFF0'
            dc h'FFFFFFFFFFFFFFFFFF0'
            dc h'FFFFFFFFFFFFFFFFFF0'
            dc h'0FFFFFFFFFFFFFFFF00'
            dc h'0FFFFFFFFFFFFFFFF00'
            dc h'00FFFFFFFFFFFFFF000'
            dc h'000FFFFFFFFFFFF0000'
            dc h'0000FFFFFFFFFF00000'
            dc h'0000000FFFFF00000000'

            dc h'0000'
            dc h'0000'

            END
```

Figure. 4.43 Sample cursor source code listing

BINARY FILETYPE $06

A binary filetype is the actual bytes that make up the image. These bytes can best be illustrated by Figs. 4.41,4.42 and 4.43. This filetype can be used by any language and is the most fundamental filetype in computing. Binary files need no relocation or special handling and can be loaded in the computer by standard P8, P16 or Applesoft commands.

RESOURCE FILETYPE (any)

Resources are stored in a resource fork of an extended ProDOS file. The exact filetype is not important and in fact resources can be stored in any ProDOS file of any type.

Resources are refered to by a two byte "type" number and a four byte "I.D." number. A type would be analogous to a window record, a pascal string, an icon etc... An I.D. number would identify which pascal string or which icon you are pointing to in a group of pascal strings or icons.

The type for image resources are $1003, $8001, $1004. The I.D.'s can be anywhere from 0 to 7FFFFFFF.

Image resources are in binary form and require no system converter to load to your IIgs.

USING SOURCE CODE

The source code created by this editor is a simple text file. It has a filetype of $B0 and is created in a form readily adaptable to source code listings created for APW or ORCA assemblers. You can use the filetype command in the APW/ORCA shell or the Disk Utilities function of the CALL BOX shell to change this files filetype.

Each source code file created by this editor needs to have a filename that has no (.) periods in it. This is commonplace in ProDOS, but periods are an illegal character in the assembler and will generate an error when assembled.

The simplest way of hooking-up a CALL BOX generated source code file to your applications source code is to use the COPY directive.

```
.
.
(your code)
.
.

 COPY CallBoxImage   ;Your Image source file
.
.
(your code)
.
.
```

Another way is to use the COPY function of the APW/ORCA editor (OpenApple-C) to put a copy of your image source code in its SYSTEMP file which can then be inserted into your source listing with an INSERT function (OpenApple-V).

Adapting this source code for other assemblers is up to you. We will support Apple prefered format like APW or ORCA only on this editor.

USING BINARY CODE

The binary code created by this editor is non-relocatable code and is very to install in your program. The binary image will be a separate disk file and can be loaded via
P16, P8 or even Applesoft BASIC. There are no special handling considerations for binary images and a sample load might go something like this:

```
; We will use GS/OS class 1 calls for this
; Get the files length so you can allocate a spot for it
; We will assume that you have set-up the parameter tables already

 GETFILEINFOGS GETFILEBlock
 ldx #$24                ;Offset to the EOF
 lda GETFILEBlock,x
 sta temp                ;Save a copy of the EOF
 lda GETFILEBlock+2,x
 sta temp+2
```

```
; Allocate a block to put the image in

        PushLong #0                 ;Space
         PushLong temp              ;Length
        PushWord MyID               ;user I.D.
        PushWord #0                 ;Attributes (none)
        PushLong #0                 ;Location (anywhere)
        _NewHandle
        CopyLong ImageHandle        ;Fetch the handle

        PullLong $0                 ;Deference it for a pointer
        ldy #2
        lda [$0]
        sta READBlock+4             ;Put it in the READ parameter
        lda [$0],y                  ;block
        sta READBlock+6


; OPEN-READ-CLOSE the image file

OPENGS OPENBlock                    ;Open up the image file
        lda OPENRefNum              ;Pass the reference numbers
        sta READRefNum
        sta CLOSERefNum
        READGS READBlock            ;Read the file into the block
        CLOSEGS CLOSEBlock          ;Close the file
```

This will put any binary image in a legal memory block for use by your program.

The images address can now be passed to the appropriate routine for plotting into a window or the super hi-res screen.

USING RESOURCES

Using the resource form of an image is quite similar to the binary form except that the P16 or P8 calls are replaced by calls to the resource manager.

OPEN your resource file:

```
        PushWord #0                 ;Space for results
        PushWord #0                 ;Req. file access
        PushLong #0                 ;Res. header address
        PushLong #PathName          ;Pointer to a class 1 pathname
        _OpenResourceFile
        PullWord FileID             ;Open resource file I.D.
```

And LOAD it into memory:

```
PushLong #0                     ;Space for results
PushWord #$8001                 ;Requested Type (icon)
PushLong #1                     ;Requested I.D.
_LoadResource
CopyLong ResourceHandle         ;Handle of resource in memory
PullLong $0                     ;Deference it for a pointer
ldy #2
lda [$0]
sta ResourcePointer
lda [$0],y
sta ResourcePointer+2
```

At this point the resource is available to your application. When you are done using this resource you can put it away with the Resource Manager call CloseResourceFile:

```
PushWord FileID
_CloseResourceFile
```

The images address can now be passed to the appropriate routine for plotting into a window or the super hi-res screen.

BASIC CONSIDERATIONS

The CALL BOX BASIC Interface uses binary code images. These images are loaded into your Applesoft application with syntax as defined in the CALL BOX BASIC Interface Manual. Images under Applesoft need no special care and feeding.

Index of Chapter 4

So What Software Notice #0

Call Box
SOFTWARE NOTICE Contents

April 1,1990

This notice is the index for all of the Call Box software notices. Current disk = sampler.1

So What Software Notice #1

Program Revision: CB
Version: 2.0.1 **April 1,1990**
Prior Version: 2.0 **January 15,1990**

This revision fixes 3 bugs found in the software: (see sampler #1)

---

1. The port commands for local to global and global to local were not referencing the windows port rec. This caused the wrong coordinates to be returned.

2. The long poke command would not poke values to any address other than bank 0. Attempts to poke addresses above bank zero would result in trashing some bytes somewhere in bank zero.

3. The SoDOS (*GS/OS Emulator*) Class1 OpenGS call would only return the first byte of the auxtype. This would cause unpredictable results for software using the codes stored in the auxtype fields of a ProDOS file.

So What Software Notice #2

New Program: AMP
Version: 1.0 **April 1,1990**

This program is an Ampersand interpreter for the program CB. and is supported by the Call Box BASIC program AMPER.EDIT found on C.B.P.A. sampler #1.

This program is installed after installing the file CB. You can now issue Ampersand (&) commands (*Edit these commands with the Call Box BASIC program AMPER.EDIT*). This program will use a page of bank zero memory just as most tools will. This page will be managed automatically by the program. It will be disposed of by the issue of the CBShutdown command (*CALL QF*).

Using ampersand commands is advantageous in the fact that you do not have to **RESTORE CB.VARS** because variables are not used to identify functional families in ampersand programming. The disadvantage in using ampersand commands is that they usually take up more code space than calls and if you heed the aforementioned advantage you will not be able to locate the BASIC Driver global page. If you use ampersand style commands (*preferable for foreign language users*) it is strongly advised that you still RESTORE CB.VARS because many "advanced" functions of the BASIC driver depend on some of the addresses it contains.

So What Software Notice #3

New Program: MERGE.EDIT
Version: X.0.1 **April 1,1990**

This program is only 2 functions of a future un-announced program. It is provided however to aid in the use of templates for those of you who do not have Applesoft merge programs.

Merge.Edit will merge two Applesoft programs into one or will remove previously merged code sections. This program is filetype S16 and should be placed in the EDITORS subdirectory so it can be selected from the Editors menu in the launching shell.

When you enter Merge.Edit you will be presented with an info window displaying stats on two Applesoft programs, one called Applesoft and the other called Applesoft (aux). You can select Display from the Functions menu and two scroll windows will fill the screen. This is an alternate display mode for this editor and should only be used when speed is not a factor to you... it is very slow! (*This will be fixed eventually, for right now this is experimental code*).

Use the File menu selection Load Applesoft to load your target program (*the program that is to be written to*). Next load the program that is to be appended to this file by using the File menu selection Load Applesoft (aux). Now select the Functions menu selection Merge to write the aux program into the main program. You can now select Save Applesoft or even Save Text which saves a text file of the Applesoft program code. You can repetitively load and merge several Applesoft aux program code segments without indicent. **NOTE:** Once a program has been merged the aux program copy in memory can not be used again, it must be reloaded if you want to remove the merged code for example.

So What Software Notice #4

New Library: ADVANCED FUNCTION TEMPLATES
Version: 1.0 **April 1,1990**

This library contains merge-able program segments which give your Call Box BASIC programs instant access to routines that are not in the supported command set.

The following are breif descriptions of each templates functions and use. These templates follow the Call Box BASIC Standard for Line Numbering and are fully REMed. Refer to the Tech Notes for the exact use of the routines contained in these templates.

DESKTOP.TMPLT

This template is the minimum required code necessry for a desktop application. It includes Desktop Initialization, entity loading thermometer, Event loop, Menu distributor, Close topmost window, No operation and Quit.routines and is used as the starting template in creating a new Call Box BASIC desktop application.

MEMORY.TMPLT

This template provides you with 4 important memory allocation / de-allocation routines. This template is needed by most of the other templates and should rarely be omitted. You can Allocate a block of memory and deallocate it or Allocate a direct page and deallocate it. **IMPORTANT PRIMAL FUNCTIONS!**

GSOS.TMPLT

This template gives you the ability to use SoDOS (*GS/OS emulator*) to issue Class 0 and Class 1 GS/OS commands. These commands are vital for things like sound or OBJ/EXEC file loading. You get A fully automated GSOS Class 1 file load, GSExpandPath, GSOpen, GSRead, GSClose, GSOS Error Handler plus error messages.

SOUND.TMPLT

This template gives you the ability to load and play sound files in either of the ACE compressed modes or normal uncompressed form. You get A fully automated sound file loader/uncompressor, Play sound and Play sound exclusive. Startup and shutdown code is included as well.

LONG.STRT.TMPLT

This template changes the first line in the DESKTOP.TMPLT to use the CB.STARTUP program which is used for long Call Box BASIC program code segments that can not to do a -CB from within themselves.

REM.TMPLT

This template is used to either zap the REM statements out of your program to conserve memory space or to put them back in to deliniate a printout. This template effects the REM statements in the Call Box Advanced Function Templates only.

So What Software Notice #5

New Program: AMPER.EDIT
Version: 1.0 **April 1,1990**

This program is used to edit the ampersand text for the program AMP.

This program is written in Call Box BASIC and is used to edit the text of the Ampersand commands found in the program AMP. This program must be run in the same directory as the file AMP is located. Each ampersand command is 8 characters maximum and all are edited by either pointing and clicking on them in the dialog box or step through them using the tab and left and right arrow keys. Caution should be observed when editing ampersand commands so that no command is created that contains character combinations that duplicate any Applesoft "tokens".

So What Software Notice #6

New Program: CB.STARTUP
Version: 1.0 **April 1,1990**

This program is used by the code supplied in LONG.STRT.TMPLT.

This program simply sets the screen to 80 column text, starts-up CB and RESTOREs the file CB.VARS. The trick to this program is that it CHAINs back into its calling program which then continues operation without ever having to start-up CB from within itself. This allows the calling program to be as large as there is memory for...(approx 28 to 30K). The calling program must use the program line presented in LONG.STRT.TMPLT and then this program will be evolked automatically. it should be in the same directory as CB, CB.VARS and your calling program, other configurations will necessitate that you alter the paths to suit.

This program should not be run directly! It should only be called from another Applesoft program using the LONG.STRT.TMPLT code line.

So What Software Notice #0

Call Box
SOFTWARE NOTICE Contents

July 1,1990

This notice is the index for all of the Call Box software notices. Current disk = sampler.2

So What Software Notice #1

Program Revision: CB
Version: 2.1b3 **July 1,1990**
Prior Version: 2.0.1 **April 1,1990**

This revision fixes **1** bug found in the software and adds **1** new function: (see sampler #2)

V2.0.1 April 1990

1. The port commands for local to global and global to local were not referencing the windows port rec. This caused the wrong coordinates to be returned.

2. The long poke command would not poke values to any address other than bank 0. Attempts to poke addresses above bank zero would result in trashing some bytes somewhere in bank zero.

3. The SoDOS (*GS/OS Emulator*) Class1 OpenGS call would only return the first byte of the auxtype. This would cause unpredictable results for software using the codes stored in the auxtype fields of a ProDOS file.

V2.1b3 July 1990

1. The Dialog command to return text would malfunction when 2 or more text items were fetched. The program was not updating an internal pointer.

2. CALL SF has been added to this driver. This call creates and operates LOAD and SAVE dialogs which are identical to STANDARD FILE TOOL boxes in GS/OS V5.0.2. CB V2.1b3 requires the file SF and CB.INITb1 to be present in the boot volumes SYSTEM/SYSTEM.SETUP subdirectory. The old CB.INIT can be discarded. To make the SF functions available in BASIC add the following statements directly after you issue a RESTORE CB.VARS...

```
GS = AY + 3 : SF = GS + 3
```

So What Software Notice #7

Program Revision: WINDOW.EDIT
Version: 1.1b3 **July 1,1990**

This revision fixes several bugs and adds 1 new feature.

This revision adds a dialog that lets you set the rectangles that describe the windows normal and zoomed rectangles. This selection is called NUMERIC RECTS. The color table has been fixed so that Alert windows come up looking right. This change affects the colors dialog and some re-familiarization will be needed to use this dialog properly. Several internal and obscure errors have been corrected. These errors altho not noticable by the user caused imbalances in the operating system that could cause hangs or crashes in special circumstances.

Call Box - So What Software 10221 Slater Ave. Suite 103 Fountain Valley, CA. 92708

So What Software Notice #8

New Program: ACE EDIT
Version: 1.0 **July 1,1990**

This program is an ACE (Audio Compression and Expansion) editor named ACErs.

This program is titled ACErs by Joe Jaworski and is used to create compressed sound files from uncompressed ones or to create uncompressed sound files from compressed ones. There are facilities to set the playback speed and volume as well as to preview your sound files.

Sound files created (sampled) with any of the popular digitizing hardware/software products can be edited with ACE EDIT and the size limit of the sound file is proportional to the amount of memory your computer has. This program has operating instructions included under the colored apple menu as well as a description of a proposed compression standard for these files.

Sound Files found on the GEnie BBS as well as other Apple specific boards are usually in forms compatible with this editor and you can directly download them for immediate use.

I would like to take this opportunity to thank Joe Jaworski for donating this program for your use. He has created some of the Call Box WYSIWYG Editors, HyperLaunch and numerous other programs and is one of the major movers and shakers of the Apple IIgs community aside from being one of the most intelligent programmers I know.

So What Software Notice #9

New Program: PATTERN.EDIT
Version: 1.0 **July 1,1990**

This program is a Pattern Editor which is writter in Call Box BASIC.

This program edits patterns for use by the Call Box BASIC driver. You can load or save patterns and edit or create patterns using point and click type of editing. The program is self evident as to how it works and uses some advanced techniques in Call Box BASIC programming. This program should be listed out and used as a tutorial and example code for you programming work. The controls used in this editor are "home brewed" and not under the direction of the Control Manager.

So What Technical Notes #0

Call Box
TECHNICAL NOTES Contents

April 1,1990

This technical note is the index for all of the Call Box technical notes.*R* = Revised *** = New

So What Technical Notes #1

Call Box
Tool Loading using CB.Tool.List

Written by: William Stephens January 15,1990

This technical note describes the loading of system tools using the special files named CB.TOOL.LIST, START and CB.PreLaunch in the Call Box TPS V2.0.

Tools are loaded under GS/OS V5.0.2 in the Call Box TPS environment. These tools are subsequently re-started by the Call Box BASIC Driver under ProDOS 8 control. A method for loading just the files you need is provided by the use of the file CB.TOOL.LIST. This file is placed in the SYSTEM/TOOLS subdirectory of your boot volume and is a standard text type file which you can edit with any word processor or text editing software. Instructions for editing the file is provided in the file itself which also covers the startup order for system tools.

The file CB.TOOL.LIST is read by the programs CB.PreLaunch and the special START program for use on bootable Call Box BASIC disks. These files startup and then shutdown the specified tools making them memory resident... Call Box BASIC can then re-hook these tools into the system and make them available to your Call Box BASIC programs.

The files CB.PreLaunch and the special START are located in the CB.Init subdirectory of the Launching Shell disk. This subdirectory contains other special files which you should not have to directly manipulate. These files are copied and used by installation scripts which are executed from within the Installer program provided on the Launching Shell disk..

Some of the tools specified in the file CB.TOOL.LIST are not currently supported in V2.0, these tools are noted as such and should not be installed or operated under Call Box BASIC. Failure to heed these warnings will cause crashes and hangs galore! The entries in the CB.TOOL.LIST file have the following format:

The tool table is terminated with two ASCII zeroes...00 at the beginning of the last line of the table. The format of the rest of this file is up to you and can contain comments, notes or whatever without restriction as long as the very beginning lines are the tool table.

Further Reference

Call Box BASIC Manual V2.0

So What Technical Notes #2

Call Box
Allocating Your Own Memory

Written by: William Stephens
Revised by: William Stephens

January 15,1990
February 15,1990

This technical note describes how to use **CALL LC** (*Long Call*) to allocate memory blocks for your own use in the higher banks of the Apple IIgs memory range.

Some programs need data buffers for special data needed by a particular programming applications. These buffers (*blocks*) can be allocated by using a Long Call to the Memory Managers New Handle function. This function needs a user I.D. to allocate blocks with and this I.D. is present in the Call Box Global Page. Use the following statement to get the user I.D. number:

```
ID = 6144 + PEEK(PO + 180)
```

This statement creates a special user I.D. in the form of $18xx, where xx is the assigned user I.D. for the Call Box Basic Driver. Once you have a user I.D. you can then allocate a block of memory. For example let's allocate a block of memory that is $1000 bytes long, can reside anywhere in memory and is locked:

```
CALL LC,_0,_$1000,ID,$0000,_0\$0902\_H
IF H = 0 THEN (memory allocation error handler)
```

This statement will allocate the block of memory and return a HANDLE for the block allocated. This handle is important to remember for de-allocating the block later on so keep this variable. **H** however needs to be DEFERENCED (*De-Referenced*) to derive a pointer to the memory block which is what you need to address the block of memory from your application. Use the following statement to deference the handle:

```
CALL PE,4,H,P
```

Now **H** will contain the handle of the memory block and **P** will contain its pointer.

The techniques outlined here apply to many toolbox functions which can be accessed by CALL LC. Some routines will require a user I.D. and some will require handles or pointers... by using these statements you will be able to derive the right kind of data for your tool calls.

Further Reference

Call Box BASIC Manual V2.0
Apple IIgs Toolbox Reference: Volume(s) 1,2 and 3

So What Technical Notes #2

Call Box
Allocating Your Own Memory

Written by: William Stephens
Revised by: William Stephens

January 15,1990
February 15,1990

This technical note describes how to use **CALL LC** (*Long Call*) to allocate memory blocks for your own use in the higher banks of the Apple IIgs memory range.

Some programs need data buffers for special data needed by a particular programming applications. These buffers (*blocks*) can be allocated by using a Long Call to the Memory Managers New Handle function. This function needs a user I.D. to allocate blocks with and this I.D. is present in the Call Box Global Page. Use the following statement to get the user I.D. number:

```
ID = 6144 + PEEK(PO + 180)
```

This statement creates a special user I.D. in the form of $18xx, where xx is the assigned user I.D. for the Call Box Basic Driver. Once you have a user I.D. you can then allocate a block of memory. For example let's allocate a block of memory that is $1000 bytes long, can reside anywhere in memory and is locked:

```
CALL LC,_0,_$1000,ID,$0000,_0\$0902\_H
IF H = 0 THEN (memory allocation error handler)
```

This statement will allocate the block of memory and return a HANDLE for the block allocated. This handle is important to remember for de-allocating the block later on so keep this variable. **H** however needs to be DEFERENCED (*De-Referenced*) to derive a pointer to the memory block which is what you need to address the block of memory from your application. Use the following statement to deference the handle:

```
CALL PE,4,H,P
```

Now **H** will contain the handle of the memory block and **P** will contain its pointer.

The techniques outlined here apply to many toolbox functions which can be accessed by CALL LC. Some routines will require a user I.D. and some will require handles or pointers... by using these statements you will be able to derive the right kind of data for your tool calls.

Further Reference

Call Box BASIC Manual V2.0
Apple IIgs Toolbox Reference: Volume(s) 1,2 and 3

So What Technical Notes #3

Call Box
The Call Box BASIC Global Page

Written by: Eric Joham February 5,1990

This technical note describes the Call Box BASIC driver GLOBAL PAGE. This page of memory contains important addresses for advanced programming of the Call Box BASIC driver.

Here is a more detailed description of the Call Box BASIC Driver Global Page. Reserved areas are not described in detail and should not be used as they may contain important information needed by the Driver.

$xx00 ⋮ $xx6B	(+0)	BASIC Driver entry point vectors. ***Must not be modified!*** All vectors are set to the following: **jsr $xx6C** where $xx is the most significant byte of the Global Page address.
$xx6C ⋮ $xx81	(+108)	Routine vector interpreter. Determines which set of calls are to be executed and jumps to high bank subroutine distributor. ***RESERVED!***
$xx82 ⋮ $xx8D	(+130)	Firmware entry point for native mode routines. ***RESERVED!***
$xx8E	(+142)	Active Flag: Call Box is active if bit 15 is set.
$xx90 ⋮ $xxB3	(+144)	Reserved direct page locations. ***MUST not be used or modified!***
$xxB4	(+180)	User ID: Can be used to obtain memory but must not be changed.
$xxB6 ⋮ $xxD1	(+182)	Reserved direct page locations. ***MUST not be used or modified!***
$xxD2	(+210)	Stack Size: holds number of pages reserved for native mode stack.
$xxD4 ⋮ $xxEF	(+212)	Reserved direct page locations. ***Must not be used or modified!***
$xxF0	(+240)	User Buffer Address: Holds location of user buffer.
$xxF2	(+242)	User Buffer Length: Holds length of user buffer.
$xxF4	(+244)	Direct Page Size: holds number of pages used by tools.
$xxF6-$xxFF	(+246)	Remaining locations are reserved.

There may be times when you want to directly access the individual Call Box commands such as when installing an Ampersand interpreter. The entry points are in the BASIC Driver entry point vectors at the beginning of this direct page. The following list identifies these entry points. It should be noted that non-Applesoft language access to these vectors is difficult at best because the calls get their input information from the Applesoft program listing pointed to by the Applesoft line parser located at $B1 in the 0/0 page and the output is usually returned in Applesoft variables.

+00	CALL PO	Big Poke command
+03	CALL PE	Big Peek command
+06	CALL QF	Shutdown Call Box BASIC command
+09	CALL SC	Screen commands
+12	CALL PL	Palette commands
+15	CALL SB	Scanline Control Byte commands
+18	CALL PN	Pen commands
+21	CALL LN	Line command
+24	CALL RE	Rectangle command
+27	CALL OV	Oval command
+30	CALL RR	Rounded Rectangle commands
+33	CALL AR	Arc commands
+36	CALL EV	Event commands
+39	CALL CU	Cursor commands
+42	CALL TX	Text commands
+45	CALL PT	Port commands
+48	CALL WN	Window commands
+51	CALL ME	Menu commands
+54	CALL DI	Dialog commands
+57	CALL TL	Tool commands
+60	CALL LC	Long Call command
+63	CALL AY	Super Array commands

Further Reference

Call Box BASIC Manual V2.0

So What Technical Notes #4

Call Box
Allocating Direct Pages

Written by: William Stephens February 9,1990

This technical note describes how to allocate direct pages needed by some tools and for your own applications use.

Direct pages are 256 bytes long and reside in bank zero of your IIgs. These pages are page aligned (starting on page boundaries ie: 0, 256, 512, 768...). Some tools require a direct page for their own use and some require more than one yet others require none... refer to the Tool Box Reference Manuals for the exact information on the tool you wish to use. Usually in Call Box BASIC you will not need to worry about this but when you wish to use tools not directly supported by the BASIC driver then you must take matters into your own hands and allocate your own memory.

Allocating direct pages under Call Box is quite simple but requires a machine code patch to call ProDOS 8 for some space. This patch can be installed by using the BigPOKE command.

As an example let's put the patch in a rarely used area of the input buffer at $2C0 (704).

```
20 CALL PO,4,704,$F52001A9 : CALL PO,2,708,$60BE
```

The first call installs 4 bytes and the second call installs another 2 bytes for a total of 6 bytes. The 01 part of the hexidecimal number in the first call is the number of pages you wish to allocate... if this is changed to 03 for example then 3 direct pages will be allocated. To actually do the allocation you need to call this patch.

```
CALL 704
```

The next thing you need to do is inform Call Box about this allocation by incrementing the direct page size by the amount of direct pages you have allocated.

```
A = PEEK(PO+244) : A = A + 1 : POKE PO+244,A
```

The last thing you need to do is pass the address of your global page back to your program so you have a record of where it is. This is accomplished by multiplying the current page count by 256 and subtracting it from the Call Box BASIC global page address:

```
A = PO - (A*256)
```

Direct page de-allocation should be performed by Call QF and should not be attempted on your own. You really have to know what you are doing to make this type of thing happen.

Further Reference

Call Box BASIC Manual V2.0

So What Technical Notes #5

Call Box
Finding a Ports Pixel Image

Written by: William Stephens February 15,1990

This technical note describes how to find the address to a grafports pixel image. This address is needed for direct accessing of the pixel image.

Grafports, unlike other entities do not have a call to return the handle or pointer to their images. The need will arise from time to time when you need to know this address for some reason like fetching color palettes or SCB's from the grafports pixel image. A grafport in Call Box BASIC differs from the traditional grafport in that the pixel image also includes all the SCB's and all 16 color palettes. When the port is displayed only the SCB's and the majority color palette is used, leaving the other 15 color palettes in the ports pixel image and not in the display pixel image at $E1/2000. To access these other palettes you need to know their address. To derive the address of a ports pixel image use the following procedure: (N = *Ports Entity Number* A = *Ports Pixel Image Address*)

```
A = (( PEEK(PO + 120)* 65536)+ 256) + (N * 4)
CALL PE,4,A,A : CALL PE,4,A,A : A = A + 2 : CALL PE,4,A,A
```

This will find the address of the grafports pixel image and put the results in A. If you want to access the SCB's then add **32,000** to the value A, if you want to access the color tables then add **32,256** to the value A.

Example:

Let's copy all of the color palettes associated with a Call Box Grafport to the display grafport. First you must run the above procedure and then use the Quickdraw II call **SetColorTable ($0E04)** in a FOR - NEXT loop which sequentially reads palettes 0 through 15:

```
A = A + 32256 : FOR N = 0 TO 15 : CALL LC,N,_A\$0E04\
A = A + 32 : NEXT
```

This is just a single example to give you the "*feel*" of how this technique works. It is handy to know about grafport record structure plus handles and deferencing to fully appreciate the power and flexibility of this type of procedure. Unfortunately... these things may be hard to understand for the uninitiated. At the very least this tech note will allow you to use this technique without having to fully understand it... for now that is.

Further Reference

Apple IIgs Toolbox Reference Vol 1,2 and 3
Call Box BASIC Manual V2.0

So What Technical Notes #6

Call Box
Using GS/OS Calls

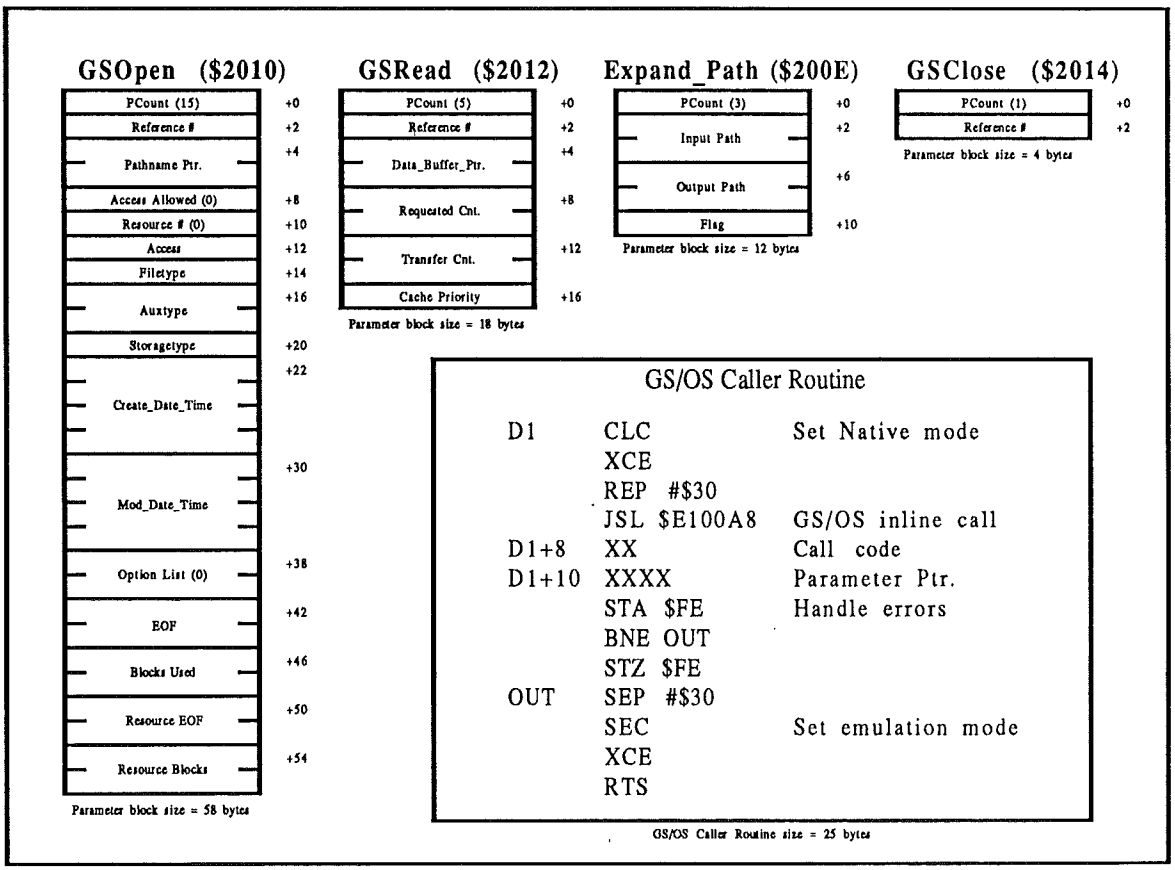
Written by: William Stephens

February 15,1990

This technical note describes how to use GS/OS Class 1 calls to load a disk file into an allocated block of memory.

Loading and managing your own blocks of data is vital for any sophisticated program and is necessary for things such as sound files for the Sound Manager and A.C.E. The process requires several steps and necessitates the set-up of a direct page for all the parameter blocks and pathname buffer. The scenario goes something like this... Allocate a direct page, Create the GS/OS Caller, Create the Pathname, OPEN the file, ALLOCATE a block for it, READ the file into the block followed by CLOSE the file. When you are done with the files data you DE-ALLOCATE the block of memory, and then DE-ALLOCATE the direct page and then quit. This is how it's done in Assembly language or for that matter from any language.

The following diagram shows the layout of the needed GS/OS parameter blocks:



The first step of this process is to allocate and then setup the direct page work area. Allocate 1 direct page as outlined in tech note 4:

```
CALL PO,4,704,$F52001A9 : CALL PO,708,$60BE : CALL 704 :
A = PEEK(PO + 244) : A = A + 1 : POKE PO + 244, A :
D1 = PO - (A * 256):
FOR I = 0 TO 255: POKE D1 + I,0: NEXT
```

D1 equals the starting address of this direct page. Let's layout the direct page as follows... starting at address D1 put the GS/OS Caller Routine which takes up 25 bytes. Next is the GSOpen parameter block which is 58 bytes long followed by the GSRead block (18 bytes) the Expand_Path block (12 bytes) and the GSClose block (4 bytes) finally let's use the last 128 bytes of the direct page for the pathname buffers. Give a variable to each of these locations to simplify the code.

```
D2 = D1 + 25 : D3 = D2 + 58 : D4 = D3 + 18 : D5 = D4 + 12 :
D6 = D1 + 128 : D7 = D1+190 : D8 = D1 + 188
```

Now D1 = GS/OS Caller Routine, D2 = GSOpen parameter block, D3 = GSRead parameter block, D4 = Expand_Path block, D5 = GSClose parameter block, D6 = Input buffer, D7 = Output Path and D8 = Output Path Buff.
Its time to setup this page by first writing in the GS/OS Caller routine:

```
CALL PO,4,D1,$30C2FB18: CALL PO,4,4 + D1,$E100A822:
CALL PO,4,14 + D1,$02B0FE85: CALL PO,4,18 + D1,$30E2FE64:
CALL PO,3,22 + D1,$60FB38
```

Next put the PCounts and buffer size in.

```
CALL PO,2,D2,15: CALL PO,2,D3,5: CALL PO,2,D4,3 : CALL PO,2,D5,1 :
CALL PO,2,D8,66
```

It's now time to put the pathname in the pathname buffer. For this example let's assume that the pathname is contained in the string variable A$. :

```
L1 = LEN(A$): FOR I = 1 TO L1:
A = ASC( MID$( A$,I,1)): POKE D6 + 1 + I,A: NEXT:
CALL PO,2,D6,L1
```

Use Expand_Path to create a full pathname from the partial in A$.

```
CALL PO,4,2 + D4,D6 : CALL PO,4,6 + D4,D8 : CALL PO,4,10 + D1,D4 :
CALL PO,2,8 + D1,$200E : CALL D1
IF PEEK(254) < > 0 THEN (GS/OS ERROR MESSAGE ROUTINE)
```

It's now time to make a GSOpen call:

```
CALL PO,4,4 + D2,D7 : CALL PO,4,10 + D1,D2: CALL PO,2,8 + D1,$2010:
CALL D1
IF PEEK(254) < > 0 THEN (GS/OS ERROR MESSAGE ROUTINE)
```

If this call is successful then all of the data specified in the GSOpen parameter block will be filled in. If this call fails for some reason then you need to handle the errors yourself, if the volume specified in the pathname is not online then a dialog box will appear prompting you to insert the proper volume before proceeding.

Pass the reference numbers to the other parameter blocks from the GSOpen block.:

```
CALL PE,2,2 + D2,A: CALL PO,2,2 + D3,A: CALL PO,2,2 + D5,A
```

The GSOpen call will return several pieces of data, one of them in particular is of intrest to you and that is the EOF. EOF (End Of File) is the length of the file you just opened, it's found at the 42nd byte of the GSOpen parameter block. You must now use the EOF to allocate a block of memory for the file to reside in. Use the procedure outlined in tech note 2 to allocate this block of memory, use the EOF for the 2nd (length) parameter in the call.:

```
CALL PE,4,42 + D2,L: ID = 6144 + PEEK(PO + 180)
CALL LC,_0,_L,ID,$0000,_0\$0902\_H
IF H = 0 THEN (MEMORY ERROR MESSAGE ROUTINE)
CALL PE,4,H,A
```

A will hold the address of the beginning of the allocated block and H will hold its handle... keep both of these for later use.

Now we can read the file into the block, but first we must setup the GSRead parameter block. Put the address (A) and the length (L) into the parameter block.:

```
CALL PO,4,4 + D3,A: CALL PO,4,8 + D3,L
```

Read the file into the block."

```
CALL PO,4,10 + D1,D3: CALL PO,2,8 + D1,$2012: CALL D1
IF PEEK(254) < > 0 THEN (GS/OS ERROR MESSAGE ROUTINE)
```

All that is needed now is to close the file.:

```
CALL PO,4,10 + D1,D5: CALL PO,2,8 + D1,$2014: CALL D1
```

This puts the file in the block and everything is ready for use. REMEMBER... A holds the address and H holds the handle. Do not loose these values because they are vital for locating and de-allocating this memory block when your program is ready to shutdown.

Example Code:

The following pages present actual Applesoft program lines which you can copy into your programs. There are 3 memory allocation routines, 5 GS/OS read routine and 3 GS/OS setup/error routines. The variables ID,I,A,L,L1,H,P,D1,D2,D3,D4,D5,D6,D7 and D8 are used by these routines. H = Memory block handle L = Block size P = Memory block address ID = special user I.D. number D1 = Direct page address.

Allocate a block of memory: (in = L, out = H,P,ID)
De-Allocate all special blocks: (in = ID, out =)
Allocate and clear a Direct Page: (in =, out = D1)
Load a file into memory GS/OS Class 1: (in =A$,D1,D2,D3,D4,D5,D6,D7,D8, out = H,P,L,ID)
GSExpand_Path: (in = A$,D1,D4,D6,D8 out =)
GSOpen: (in = D1,D2,D5,D7,{full pathname} out =)
GSRead: (in = D1,D3,P,L out =)
GSClose: (in = D1,D5 out =)
Set-Up the GS/OS Dir. Page and Error Messages: (in =, out = D1,D2,D3,D4,D5,D6,D7,D8)

Use routine **56000** once at the beginning of your program and simply put a full or partial pathname in **A$** and **GOSUB 50000** to load it into memory and get the Handle, Pointer and Length returned to you. An error handler is included as well using text file **GSOSERROR.T**.

```
44999  REM
                    Allocate a block of memory

45000 ID = 6144 +  PEEK (PO + 180): CALL LC,_0,_L,ID,$0000,_0\$0902\_H: IF H = 0 THEN
       55030
45010  CALL PE,4,H,P: RETURN
45019  REM
                     De-Allocate all special Blocks

45020  CALL LC,ID\$1102\: RETURN
45029  REM
                    Allocate and Clear a Direct Page

45030  CALL PO,4,704,$F52001A9: CALL PO,2,708,$60BE: CALL 704:A =  PEEK (PO +  244):A
         = A + 1:POKE PO + 244,A:D1 = PO - (A * 256) :  FOR I = 0 TO  255: POKE D1 + I,0: NEXT
        : RETURN
49999  REM
                    Load a file into memory GS/OS Class 1

50000  GOSUB 50100: GOSUB 50200: CALL PE,4,42 + D2,L: GOSUB 45000: GOSUB 50300:
        GOTO 50400
50099  REM
                  GSExpand_Path

50100 L1 =  LEN (A$): FOR I = 1 TO L1:A =  ASC ( MID$ (A$,I,1)): POKE D6 + 1 + I,A: NEXT :
         CALL PO,2,D6,L1: CALL PO,4,2 + D4,D6: CALL PO,4,6 + D4,D8
50110  CALL PO,4,10 + D1,D4: CALL PO,2,8 + D1,$200E: CALL D1: IF  PEEK (254)<  > 0 THEN
       55000
50120  RETURN
50199  REM
             GSOpen

50200  CALL PO,4,4 + D2,D7: CALL PO,4,10 + D1,D2: CALL PO,2,8 + D1,$2010: CALL D1: IF
         PEEK (254) <  > 0 THEN 55000
50210  CALL PE,2,2 + D2,A: CALL PO,2,2 + D3,A: CALL PO,2,2 + D5,A: RETURN
50299  REM
             GSRead

50300  CALL PO,4,4 + D3,P: CALL PO,4,8 + D3,L: CALL PO,4,10 + D1,D3: CALL PO,2,8 +
         D1,$2012: CALL D1: IF  PEEK (254) <  > 0 THEN 55000
50310  RETURN
50399  REM
             GSClose

50400  CALL PO,4,10 + D1,D5: CALL PO,2,8 + D1,$2014: CALL D1: RETURN
```

```
54999  REM
                GSOS Error Handler

55000  CALL SC,0: HOME : FOR I = 0 TO 21: CALL AY,2,ER,[I],A: IF A =  PEEK (254) THEN
       55020
55010  NEXT
55020   CALL SC,0: HOME : PRINT "GS/OS Error...";: CALL AY,2,ER$,[X],A$: GOSUB 45020:
          CALL  - 1052: CALL QF: END
55030   POP  : CALL SC,0: PRINT "Memory Allocation Error...": GOSUB 45020: CALL QF: END
55999  REM
                  Setup the GSOS Dir. Page and Error Messages

56000  GOSUB 45030:D2 = D1 + 25:D3 = D2 + 58:D4 = D3 + 18:D5 = D4 + 12:D6 = D1 + 128:D7 =
         D1 + 190:D8 = D1 + 188
56010  CALL PO,4,D1,$30C2FB18: CALL PO,4,4 + D1,$E100A822: CALL PO,4,14 +
          D1,$02B0FE85: CALL PO,4,18 + D1,$30E2FE64: CALL PO,3,22 + D1,$60FB38: CALL
            PO,2,D2,15: CALL PO,2,D3,5: CALL PO,2,D4,3: CALL PO,2,D5,1: CALL PO,2,D8,66
56020   CALL AY,1,ER,[21]: CALL AY,1,ER$,[21]: PRINT  CHR$ (4);"OPEN GSOSERROR.T" :
          PRINT CHR$ (4);"READ GSOSERROR.T"
56030  FOR I = 0 TO 21: INPUT A: CALL AY,3,ER,[I],A: NEXT : FOR I = 0 TO 21: INPUT A$:
          CALL AY,3,ER$,[I],A$: NEXT : PRINT  CHR$ (4);"CLOSE": A = FRE (0) : RETURN
```

The Call Box Standard for Line Numbering

To keep a desktop Applesoft program "tidy" we recommend that certain line number ranges be used for certain functions. This will allow you to cut and paste your Applesoft (*or at least cursor trace*) programs using standard templates... let's face it, toolbox programming requires a lot of information to be passed to and from the routines and if you already typed it out, why do it again! A growing library of templates is being generated for tricky Call Box procedures which are available to you through C.B.P.A. and will follow these guidelines, This tech note is one of them. The standard goes something like this...

0 - 198 Call Box initialization, Entity loading and program initialization.
200 - 298 Main Event loop stuff
300 - 398 Menu distributor stuff
400 - 498 Quit stuff
500 - 39998 (*Undefined*)
40000 - 44998 Sound stuff
45000 - 49998 Memory management stuff (*as shown in this tech note*)
50000 - 54998 GS/OS stuff (*as shown in this tech note*)
55000 - END Error message handling and environment initialization

Deviations on this scheme is purely up to you, we just hope that you have a good re-numbering program available because we will adhere to this standard and put out our tech notes and sample code disks using it.

This is the file **GSOSERROR.T**. Type this in on any word processor or text entry software and save it to disk as a type **TXT** file.

```
16
39
40
43
46
64
67
68
69
70
72
74
75
76
78
79
80
82
83
88
90
99
Device Not Found
I/O Error
No Device Connected
Disk is Write Protected
Disk Switched
Invalid Pathname Syntax
Invalid Reference Number
Path Not Found
Volume Not Found
File Not Found
Volume Full
Version Error
Unsupported Storage Type
EOF Encountered
Access Not Allowed
Buffer Too Small
File Is Open
Unsupported Volume Type
Invalid Parameter Value
Not A Block Device
Block number out of range
File Does Not Contain Resource Fork
~
```

Further Reference

GS/OS Reference
Apple IIgs Toolbox Reference: Volume(s) 1,2 and 3
Call Box BASIC Manual V2.0
Tech Notes 2 and 4

So What Technical Notes #7

Call Box
Setting up a Special Edit Menu

Written by: William Stephens February 16,1990

This technical note describes how to modify a menu entity to respond with Special Edit Menu features activated by System Windows.

A Special Edit Menu is a menu which contains the menu items UNDO, CUT, COPY, PASTE, CLEAR and CLOSE. These items have I.D. numbers of 250 through 255 and are activated whenever a system window is the topmost window on the desktop. Most New Desk Accessories (NDA's) use these items because NDA's are in system windows.

The Call Box Menu Editor is capable of including these items by name but not by I.D. number. I.D. numbers are automatically assigned in this editor starting at 256 and are in sequential order, so the special edit menu items in your menu would have I.D. numbers larger than 256. You can use Long Call (*CALL LC*) to make **SetMItemID ($380F)** calls to change these items I.D.'s to the range 250 - 255 which would setup the Special Edit Menu.

Example:
Make a simple menu entity using the Call Box Menu Editor with the following items:

Apple		(enabled)
	About	(enabled - underlined)
File		(enabled)
	Close	(disabled - underlined)
	Quit	(enabled - Key = Q)
Edit		(enabled)
	Undo	(disabled - underlined - key = Z)
	Cut	(disabled - Key = X)
	Copy	(disabled - Key = C)
	Paste	(disabled - Key = V)
	Clear	(disabled)

About = 256, Close = 257, Quit = 258, Undo = 259, Cut = 260, Copy = 261, Paste = 262 and Clear = 263. After you load and display this menu from within your program use the following statements to convert the menu item I.D.'s to the Special Edit Menu item I.D.'s.

```
CALL LC,255,257\$380F\
FOR I = 0 TO 4: CALL LC,250 + I,259 + I\$380F\: NEXT
```

Putting the Call Box BASIC Drivers Menu command Check Menu (CALL ME,2,N) in your programs event loop will detect the presence or absence of a system window and enable or disable these items automatically.

Further Reference

Apple IIgs Toolbox Reference: Volume(s) 1,2 and 3
Call Box BASIC Manual V 2.0

So What Technical Notes #8

Call Box
Directory Structures

Written by: William Stephens February 18,1990

This technical note describes the layout of subdirectories for disk and hard disk setups using the Call Box TPS or BASIC Driver.

The following descriptions should be used as a guide for the layout of Call Box environments. You will notice that the environment for 3.5 inch disks is a subset of the environment for hard disks.

Hard Disk Subdirectory levels

Volume (Root)	Level 1	Level 2	Level 3	Level 4	Level 5
ROOT	CALL.BOX	BASIC	ENTITY	WINDOW	
				DIALOG	
				MENU	
				IMAGE	
				SOUND	
				CONTROL	
		SCRIPTS			
		WINDOW.EX			
		MENU.EX			
		DIALOG.EX			
		IMAGE.EX			
		EDITORS			
		TEMPLATES			

3.5 inch disk Subdirectory levels

Volume (Root)	Level 1	Level 2	Level 3	Level 4	Level 5
CB.BASIC	ENTITY	WINDOW			
		DIALOG			
		MENU			
		IMAGE			
		SOUND			
		CONTROL			

Further Reference

So What Technical Notes #9

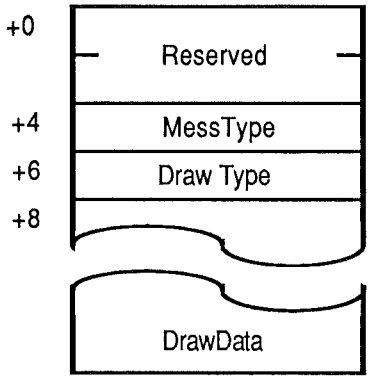
Call Box
Custom Desktops

Written by: William Stephens February 22,1990

This technical note describes how to make desktops containing patterns or pictures instead of the standard desktop colors.

You can set the desktop color by using the screen color call CALL SC,2,$???? after calling CALL TL,2,"DESK" and this is quite straight forward and simple.. but if you want some pattern or even a picture as your desktop you need to use a different process. This process requires you to use the Tool Locator call **MessageCenter** before calling CALL TL,2,"DESK".

The Window Manager which handles desktop drawing looks for a desktop message before it creates one. If a desktop message is present then the Window Manager uses the information in this message to draw the desktop, if no message is found then it uses the standard default colors (*usually Periwinkle blue in 640 mode or light blue in 320 mode*). The Toolbox Reference Manual describes this message as follows:



Long	Any Value... make it 0
Word	2 is the Desk Message
Word	0 = Pattern 1 = Picture
Data	32 bytes of pattern data or 32000 bytes of picture data.

Making a Message...

To make a pattern message put yourself in plain old Applesoft BASIC and type CALL -151, this puts you in the monitor. Type in the following line:

```
1000: 00 00 00 00 02 00 00 00
```

Next you either type in 32 bytes of data which is a pattern or if you have some saved to disk BLOAD them at $1008. Now issue the following command:

```
BSAVE PATTERN.MESS,A$1000,L$28
```

This saves the message to disk as the file PATTERN.MESS.

Making a picture message is very similar, but you first need to take a filetype **$C1** picture and change its filetype to **BIN** (*Use the File Utilities in the Call Box Launching Shell*) and then BLOAD it at $1008. Now from the monitor type in the following line:

```
1000: 00 00 00 00 02 00 01 00
```

Next type in this line:

```
BSAVE PICTURE.MESS,A$1000,L$7D08
```

This is all there is to it!

Using the Message...

After you startup the Call Box BASIC Driver (-CB) and load in its variables (RESTORE CB.VARS) you need to load and then pass the message. Use the GS/OS load code presented in tech note 6 to load in your message:

```
GOSUB 56000 : A$ = "(pattern or picture).MESS" : GOSUB 50000
```

Next you pass the message handle to the message center:

```
CALL LC,1,2,_H\$1501\
```

Now when you issue the CALL TL,2,"DESK" your picture or pattern will fill the desktop. This condition will persist until you shutdown your IIgs or you issue a delete message command...

```
CALL LC,3,2,_H\$1501\
```

As I said, this condition persists even if you "BYE" out of Applesoft into a GS/OS Launching program... If this launcher is a desktop application like Hyperlaunch or the Finder the desktop will still be in the style you put it in with these commands.

Further Reference

Apple IIgs Toolbox Reference: Volume(s) 1,2 and 3

So What Technical Notes #10

Call Box
Using Sound in your BASIC Applications

Written by: William Stephens February 24,1990

This technical note describes how to add Sound Manager and ACE (Audio Compression and Expansion) tool calls to your Call Box BASIC Applications.

Care and feeding

Edit the CB.TOOL.LIST (*see tech note #1*) file in the SYSTEM/TOOLS subdirectory of your boot volume. Load or (import) the file into your word processor or text edit software and copy the lines for Sound Manager and ACE and insert them just below the Dialog Manager in the tool list... save the CB.TOOL.LIST file back to the subdirectory. (*make sure it's still has filetype TXT*)

If you are running Call Box BASIC on disk then the file SYSTEM/START will install the ACE tools, if your on a hard drive and are not using HyperLaunch V3.0.2 as the launching program then you will have to launch the file CB.PreLaunch prior to launching into Call Box BASIC (usually once ACE gets put in it stays in... however, this is not guaranteed, when the system reverts to GS/OS again this tool is marked as purgeable and a subsequent applications memory allocation process may wipe the tools code out! Going back to Call Box BASIC and using ACE again will probably do something screwy... or destructive!!! be careful, if your not sure launch the file CB.PreLaunch again. This is not a concern on a bootable 3.5 inch disk but you must copy the tool TOOL.029 to your disks SYSTEM/TOOLS subdirectory, it is not put in by the default initialization. You can also alter your disk initialization installer script to include this tool as well.

Programming with sound

The sound calls are very simple, but the 2.0 version of SoDOS is limited to 256 block files. (No trees as of yet) This is where ACE comes in handy as well as saving disk space for those of you who do not have the benefit of a hard drive.

Both Sound Manager and ACE require a direct page, (*that's 2 pages*) and waveforms should never be loaded in bank zero for obvious space reasons so you will need GSOS calls to load in your waveforms (**use tech note #6**). When your program does its environment initialization you should do **2 "Allocate and Clear a Direct Page"** gosub's to get these pages, save the returned variable **D1** as**SO** for the first page and**AC** for the second. Next you should**GOSUB 56000** in the GS/OS code to setup the GS/OS call block. Now it's time to start these guys up:

```
CALL LC,SO\$0208\: REM  SoundStartup
CALL LC,AC\$021D\: REM  Startup ACE
```

The next thing you need to do is allocate a block of memory for your sound parameter block ... for this explanation we will use only 1. Each parameter block is 18 bytes long, to allocate a memory block for it gosub "Allocate a block of memory" and zero it out :

```
L = 18 : GOSUB 45000 : H1 = H: P1 = P :
FOR A = 0 TO 17 : CALL PO,1,0 + A + P1,0 : NEXT
```

Now you need to load in your sound file. Put the files pathname in the variable A$ then GOSUB "Load a file into memory GS/OS Class 1"

A$ = "{*sound file pathname*}" : GOSUB 50000 : H2 = H : P2 = P : L2 = L

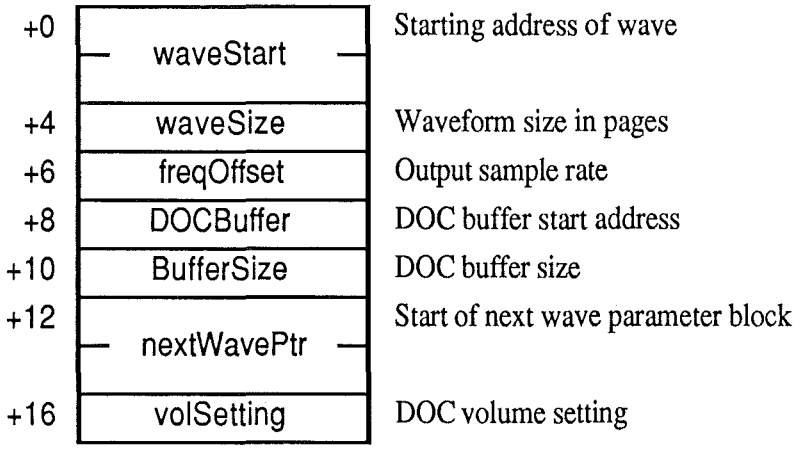
Even with the file closed you still have the open file information in the GS/OS caller block, so check the Aux type for the freqOffset. This is not a recommended practice by Apple Computer Inc. but is the way many waveforms are annotated. The shareware program ACERS by Joe Jaworski uses this type of annotation.

CALL PE,2,16 + D2,PB : IF PB > 32767 THEN PB = PB - 32768

You can now check the filetype and see if it's an ACEd file or not: (filetype $CD = ACEd sound file).

CALL PE,2,14 + D2,A : IF A = 205 THEN (*goto ACE routine*)

The last piece of overhead you must take care of before playing your sound is the setting up the sound parameter block. The toolbox reference specifies this parameter block as follows:



You need only supply the waveStart (**P**), the wavesize, the freqOffset (**PS**), and the volSetting (**0 - 255**). All other parameters should be zero.

CALL PO,4,P2,P : CALL PO,2,4 + P1,0 + L2/256 : CALL PO,2,6 + P1,PS : CALL PO,2,16 + P1,255

Any time after this you can play your sound by issuing the following line:

CALL LC,$0401,_P1\$0E08\: REM FFStartSound

Refer to the Apple IIgs Tool Box reference manual for the flags and other subtle nuances of the Sound Manager routines. Your sound file will play while your Applesoft program continue to execute. If you want program execution to stop while the sound is playing add the following lines after the **FFStartSound** line:

```
CALL LC,0,4\$1408\F: REM  FFSoundDoneStatus
IF F = 0 THEN {GOTO the FFSoundDoneStatus line}
CALL LC,16\$0F08\: REM  FFStopSound
```

The ACE Routine

ACE has 2 compression styles expressed as ratios... 8:4 or 8:3. 8:4 means that the compressed waveform stored as a file is 1/2 the size of the same waveform uncompressed, and the 8:3 means that the stored file is 3/8 the size of its uncompressed size.

When you load in a compressed sound file you need to get its size and allocate another block either 2 (*the reciprocal of .5 or 1/2*) times the size of the compressed file for 8:4 or 2.6667 (*the reciprocal of .375 or 3/8*) times the compressed size for 8:3. You can usually tell if a file is in 8:4 or 8:3 style by checking the files AuxType and see if it's greater than 32767, If it is then the file is usually compressed in 8:3 style, if it's not then it's 8:4. Allocate the expanded sound file memory block as follows:

```
L = 2 * L : F = 1 : CALL PE,2,16 + D2,A : IF A > 32767 THEN L = (L / 2) *
2.6667 : F = 2
GOSUB 45000 : H3 = H : P3 = P : L3 = L : L = L3 / 512
```

It's now time to expand the sound file in the first memory block to the new memory block (*this can take a while depending on the size of the original sound file*):

```
CALL LC,_H2,_0,_H3,_0,L,F\$0A1D\ : REM ACEExpand
```

Now dispose of the first memory block:

```
CALL LC,_H2\$1002\ : REM DisposeHandle
```

And setup the sound parameter block:

```
CALL PO,4,P1,P3 : CALL PO,2,4 + P1,0 + L3/256 : CALL PO,2,6 + P1,A :
CALLPO,2,16 + P1,255
```

Finally goto the **FFStartSound** line to play the sound file.

Remember your Memory...

YOU MUST BE VIGILANT! When you use these procedures you must dispose of the H2 or H3 blocks accordingly before you reuse these routines from within the same application. Failure to do this will cause the IIgs to accumulate uncompressed sound files in memory which takes up "Lots" of room!!! Sometimes this is exactly what you want to do. Remember to keep a record of all the allocated blocks handles and pointers for disposal or access later on because H2 and H3 are overwritten each time a sound file is loaded.

Example Code:

This page presents actual Applesoft program lines which you can copy into your programs. There are 3 Sound Manager and ACE routines, The load sound routine needs the GS/OS code presented in tech note #6. The variables ID,D1,D2,DT,SO,AC,A,PB,F,H,H1,H2,H2,P,P1,P2,P3,L,L1,L2 and L3 are used by these routines. H-H3 = Memory block handle L-L3 = Block size P-P3 = Memory block address ID = special user I.D. number D1 = Direct page address.

Load/Uncompress sound file: (in = A$, out = H,H2,H3,P,P2,P3,L,L2,L3,ID)
Play Sound: (in = , out =)
Play Sound exclusive: (in = , out =)

```
39999  REM
                    Load/uncompress sound file

40000  GOSUB 50000:H2 = H:P2 = P:L2 = L
40010  CALL PE,2,16 + D2,PB: IF PB > 32767 THEN PB = PB - 32768
40020  CALL PE,2,14 + D2,A: IF A = 205 THEN 40040
40030  CALL PO,4,P1,P: CALL PO,2,4 + P1,0 + L2 / 256: CALL PO,2,6 + P1,PB: CALL
       PO,2,16 + P1,255: RETURN
40040 L = 2 * L:F = 1: CALL PE,2,16 + D2,PB: IF PB > 32767 THEN L = (L / 2) *
      2.6667:F = 2:PB = PB - 32768
40050  GOSUB 45000:H3 = H:P3 = P:L3 = L:L = L3 / 512
40060    CALL LC,_H2,_0,_H3,_0,L,F\$0A1D\: CALL LC,_H2\$1002\: CALL PO,4,P1,P3:
        CALL PO,2,4 + P1,0 + L3 / 256: CALL PO,2,6 + P1,PB: CALL PO,2,16 + P1,255
     : RETURN
40099  REM
              Play sound

40100  CALL LC,0,4\$1408\F: IF F = 0 THEN 40100
40110   CALL LC,16\$0F08\: CALL LC,$0401,_P1\$0E08\: RETURN
40199  REM
                  Play sound (exclusive)

40200    CALL LC,$0401,_P1\$0E08\
40210  CALL LC,0,4\$1408\F: IF F = 0 THEN 40210
40220   CALL LC,16\$0F08\: RETURN
56099  REM
                Sound / ACE setup

56100 DT = D1: GOSUB 45030:SO = D1: GOSUB 45030:AC = D1:D1 = DT: CALL LC,SO\
          $0208\: CALL LC,AC\$021D\
56110 L = 18: GOSUB 45000:H1 = H:P1 = P: FOR A = 0 TO 17: CALL PO,1,0 + A + P1,
        0: NEXT : RETURN
```

Further Reference

Apple IIgs Toolbox Reference: Volume(s) 1,2 and 3
Call Box BASIC Manual V2.0
Tech Notes 2, 4 and 6

So What Technical Notes #11

Call Box
The Call Box Standard for Line Numbering

Written by: William Stephens March 15,1990

This technical note outlines the Applesoft Line numbering standard for use in conjunction with Call Box BASIC.

With the introduction of standard code templates it has become necessary to assign ranges of line numbers for special uses. There is still a vast range of undefined numbers for your unique program code. More templates will become available in the future so adherence to this standard should be observed to be compatible "across the board". Using program templates you can construct a very sophistocated and complex program without ever typing in a line of code... well, maybe one or two.

0 - 198 Call Box Initialization, Entity Loading, and general Program Initialization
200 - 298 Main Event Loop
300 - 398 Menu distributor
400 - 498 Quit handler

500 - 34998 (undefined) User assignable line numbers

35000 - 39998 (reserved for future expansion)
40000 - 44998 Sound Manager / ACE handler
45000 - 49998 Memory Management routines
50000 - 54998 GS/OS / OMF2 routines
55000 - END Error Message and Environment Initialization

The ????99 line number should always be reserved for a REM statement, so ranges should go from ????00 to ????98. REM statements should take the form of:

```
1999 REM
                This is a REM statement

2000 (the first line of your routine)
```

Getting a REM statement to appear this way requires you to type the line number then the letters REM followed by a space and then a control J (Down Arrow) and then your statement followed by another control J and the press return. This gives you a well space REM statement with a minimum of characters used.

The first 10 lines (line 0 thru 9) should be reserved for a title statement. This is illustrated in any of the Call Box BASIC templates. You will notice that line 9 of these titles is actually an END statement followed by the REM. This is just a reminder to you that templates are incomplete code segments and should never be run directly. When you make programs using templates these lines will always be present after merging, before you use the program created by this process be sure to delete lines 0 thru 9 or the program will not run.

So What Technical Notes #12

Call Box
Recommended Reference Documentation

Written by: William Stephens March 15,1990

This technical note lists recommended reference documentation needed to get a full understanding of the Apple IIgs and its operating systems.

The following are monthly publications that are Apple II specific. There are more and possibly better known publications than these but you will find that you do not get the depth and level of the information provided by these ones.

A2 Central
P.O. Box 11250
Overland Park, KS 66207
(913)469-6502

8/16
(Ariel Publishing, Inc.)
P.O. Box 398
Pateros, WA 98846
(509)923-2249

Computist
33821 E. Orville Rd.
Eatonville, WA 98328
(206)474-5750

The following are Apple IIgs reference manuals that describe the thousands of functions and features of the Apple IIgs and its operating system... a must for serious programming.

Manual Name	A2 Central	A.P.D.A.
Applesoft Programmers Reference	AW-021	A2Z2022
Beneath Apple ProDOS	QS-001	-----
Apple IIgs Firmware Reference	AW-022	A2G0054
Apple IIgs Hardware Reference	AW-002	A2G0055
Apple IIgs Toolbox Reference #1	AW-019	A2G0057
Apple IIgs Toolbox Reference #2	AW-006	A2G0058
Apple IIgs Toolbox Reference #3	-----	A0229LL/A
GS/OS Reference #1	-----	A2F2037
GS/OS Reference #2	-----	A0008LL/A

A2 Central
P.O. Box 11250
Overland Park, KS 66207
(913)469-6502

A.P.D.A.
20525 Mariani Avenue, M/S 33G
Cupertino, CA 95014-6299
1(800)282-2732

This support documentation is vital to understanding the Apple IIgs... at first glance it seems like a lot but once you have been through it it will seem slightly inadequate. This is the nature of the beast... when you are dealing with systems as complex as the ones associated with the Apple IIgs it is virtually impossible to document all the possible ways of turning it, however.. these references will take you most if not all of the way there.

Further Reference

So What Technical Notes #13

Call Box
Standard Program Templates

Written by: William Stephens March 20,1990

This technical note describes the Standard Call Box BASIC Templates which are used to construct Call Box BASIC applications.

The templates outlined here should be inserted into your program code using the MERGE.EDIT program provided on sampler #1. This merge program will overwrite existing lines which gives it an update capability... Other merge programs may not work this way and this type of action is needed for proper template integration.

Desktop.Tmplt

```
0    REM    ======================================
1   REM   BASIC CODE TEMPLATE
2    REM    ------------------------------------
3   REM    Desktop environment
4    REM    ------------------------------------
5   REM   Reference:
6    REM    ------------------------------------
7   REM   So What Software        V1.0   25-Jan-90
8    REM    ======================================
9   END : REM

10   PRINT   CHR$ (4);"PR#3": HOME : PRINT   CHR$ (4);"-CB": PRINT   CHR$ (4);"RESTORE
     CB.VARS"
20   CALL TL,2,"DESK": CALL SC,2,640: CALL SC,1: CALL WN,3: CALL CU,1: CALL CU,3:
     GOSUB 56000
39  REM
              Load in the Entities...

40    CALL  WN,0,3,"Entity:Window:Load.Window": CALL  WN,1,3:  CALL  PT,0,3:  CALL  TX,1,0,15:
        CALL PN,2,5,17: CALL TX,0,0,"Loading Entities ": CALL WN,4,3,1
41   GOSUB 42: GOTO 50
42   CALL PT,0,3: CALL TX,0,0,". ": CALL WN,4,3,1: RETURN
50    CALL  ME,0,1,"Entity:Menu:M.Master": CALL  ME,1,1:  GOSUB  42
51    CALL  DI,0,2,"Entity:Dialog:DemoQuit.D":  GOSUB  42
99  REM
            Set-up Entities

100   CALL WN,2,3: REM   Close the Loading window
140   CALL CU,2: REM   change to arrow cursor
199  REM
            Event Loop

200    CALL  EV,@,X,Y,B,M,K,T,C,D: CALL  ME,2,1:  REM    check  for  an  event
210   IF C = 17 THEN 300: REM   Menu item hit!
220   IF C = 22 THEN 290: REM   Close box hit!
288   GOTO 200
290   CALL WN,8,N,D,2: CALL WN,2,N: GOTO 200: REM Close the top window
```

Desktop.Tmplt (*continued*)

```
299  REM
                Menu Distributor

300 M1 =  INT (D / 65536):M2 = (D - M1 * 65536) - 255
310   ON  M2  GOSUB  330,400
320   CALL  ME,7,1,M1,0:  GOTO  200:  REM    Un-hilite  the  bar  item
330   RETURN : REM   Do Nothing !!
399  REM
          Quit

400    CALL  DI,1,2:  CALL  DI,2,2,I
410   IF  I  =  1  THEN  440
420   IF  I  =  2  THEN  480
430   CALL  DI,3,2:  RETURN
440  REM
479   POP  :  CALL  DI,3,2:  CALL  QF:  PRINT    CHR$  (4);"BYE"
480   REM
498   POP  :  CALL  DI,3,2:  CALL  QF:  END
55998   REM
                    *** Environment Initialization ***

56000   REM
59998   RETURN
```

Memory.Tmplt

```
0     REM     ======================================
1   REM   BASIC  CODE  TEMPLATE
2     REM     --------------------------------------
3     REM     Memory  allocation/de-allocation
4     REM     --------------------------------------
5   REM   Reference:  Tech  Note(s)  #2,#4  and  #6
6     REM     --------------------------------------
7   REM   So  What  Software          V1.0   15-Feb-90
8     REM     ======================================
9   END : REM

405    GOSUB  45020
485   GOSUB  45020
44999   REM
                  Allocate a block of memory

45000  ID  =  6144  +    PEEK  (PO  +  180):  CALL  LC,_0,_L,ID,$8000,_0\$0902\_H:  IF  H  =  0  THEN
         55030
45010    CALL  PE,4,H,P:  RETURN
45019   REM
                   De-Allocate all special blocks

45020    CALL  LC,ID\$1102\:  RETURN
```

Memory.Tmplt (*continued*)

```
45029  REM
                  Allocate and Clear a Direct Page

45030   CALL  PO,4,704,$F52001A9:  CALL  PO,2,708,$60BE:  CALL  704:A  =   PEEK  (PO  +  244):A
          = A + 1: POKE PO + 244,A:D1 = PO - (A * 256): FOR I = 0 TO 255: POKE D1 + I,0: NEXT
        : RETURN
55029  REM
              Memory Error

55030   POP  :  CALL  SC,0:  PRINT  "Memory  Allocation  Error...":  GOSUB  45020:  CALL  QF:  END
```

GSOS.Tmplt

```
0    REM    ======================================
1   REM   BASIC  CODE  TEMPLATE
2    REM    --------------------------------------
3   REM   GS/OS  Calls
4    REM    --------------------------------------
5   REM   Reference:  Tech  Note  #6
6    REM    --------------------------------------
7   REM   So  What  Software        V1.0   24-Feb-90
8    REM    ======================================
9   END : REM

49999   REM    Load  a  file  into  memory  GS/OS  Class  1

50000   GOSUB  50100:  GOSUB  50200:  CALL  PE,4,42  +  D2,L:  GOSUB  45000:  GOSUB  50300:
         GOTO 50400
50099  REM
               GSExpand_Path

50100 L1 =   LEN  (A$):  FOR  I  =  1  TO  L1:A  =   ASC  (  MID$  (A$,I,1)):  POKE  D6  +  1  +  I,A:  NEXT  :
          CALL  PO,2,D6,L1:  CALL  PO,4,2  +  D4,D6:  CALL  PO,4,6  +  D4,D8
50110   CALL  PO,4,10  +  D1,D4:  CALL  PO,2,8  +  D1,$200E:  CALL  D1:  IF   PEEK  (254)  <   >  0  THEN
        55000
50120   RETURN
50199  REM
              GSOpen

50200   CALL  PO,4,4  +  D2,D7:  CALL  PO,4,10  +  D1,D2:  CALL  PO,2,8  +  D1,$2010:  CALL  D1:  IF
         PEEK (254) <  > 0 THEN 55000
50210   CALL  PE,2,2  +  D2,A:  CALL  PO,2,2  +  D3,A:  CALL  PO,2,2  +  D5,A:  RETURN
50299  REM
              GSRead

50300   CALL  PO,4,4  +  D3,P:  CALL  PO,4,8  +  D3,L:  CALL  PO,4,10  +  D1,D3:  CALL  PO,2,8  +
          D1,$2012:  CALL  D1:  IF   PEEK  (254)  <   >  0  THEN  55000
50310   RETURN
```

GSOS.Tmplt (*continued*)

```
50399  REM
              GSClose

50400   CALL PO,4,10 + D1,D5: CALL PO,2,8 + D1,$2014: CALL D1: RETURN
54999  REM
               GSOS Error Handler

55000   CALL SC,0: HOME : FOR I = 0 TO 21: CALL AY,2,ER,[I],A: IF A =  PEEK (254) THEN
         55020
55010   NEXT
55020    CALL SC,0: HOME : PRINT "GS/OS Error...";: CALL AY,2,ER$,[I],A$: GOSUB 45020:
           CALL - 1052: CALL QF: END
55999  REM
                    Setup the GSOS Dir. Page and Error Messages

56000   GOSUB 45030:D2 = D1 + 25:D3 = D2 + 58:D4 = D3 + 18:D5 = D4 + 12:D6 = D1 + 128:D7 =
          D1 + 190:D8 = D1 + 188
56010    CALL PO,4,D1,$30C2FB18: CALL PO,4,4 + D1,$E100A822: CALL PO,4,14 +
                D1,$02B0FE85: CALL PO,4,18 + D1,$30E2FE64: CALL PO,3,22 + D1,$60FB38: CALL
                 PO,2,D2,15: CALL PO,2,D3,5: CALL PO,2,D4,3: CALL PO,2,D5,1: CALL PO,2,D8,66
56020    CALL AY,1,ER,[21]: CALL AY,1,ER$,[21]: PRINT  CHR$ (4);"OPEN GSOSERROR.T":
           PRINT  CHR$ (4);"READ GSOSERROR.T"
56030   FOR I = 0 TO 21: INPUT A: CALL AY,3,ER,[I],A: NEXT : FOR I = 0 TO 21: INPUT A$:
          CALL AY,3,ER$,[I],A$: NEXT : PRINT  CHR$ (4);"CLOSE":A =  FRE (0)
```

Sound.Tmplt

```
0    REM    ======================================
1   REM   BASIC CODE TEMPLATE
2    REM    --------------------------------------
3   REM   Sound / ACE Calls
4    REM    --------------------------------------
5   REM   Reference: Tech Note #10
6    REM    --------------------------------------
7   REM   So What Software        V1.0   14-Mar-90
8    REM    ======================================
9   END : REM

410  CALL LC\$031D : CALL LC\$0308\
490  CALL LC\$031D : CALL LC\$0308
39999  REM
                    Load/uncompress sound file

40000   GOSUB 50000:H2 = H:P2 = P:L2 = L
40010   CALL PE,2,16 + D2,PB: IF PB > 32767 THEN PB = PB - 32768
40020   CALL PE,2,14 + D2,A: IF A = 205 THEN 40040
40030   CALL PO,4,P1,P: CALL PO,2,4 + P1,0 + L2 / 256: CALL PO,2,6 + P1,PB: CALL
          PO,2,16 + P1,255: RETURN
40040 L = 2 * L:F = 1: CALL PE,2,16 + D2,PB: IF PB > 32767 THEN L = (L / 2) *
          2.6667:F = 2:PB = PB - 32768
40050   GOSUB 45000:H3 = H:P3 = P:L3 = L:L = L3 / 512
```

Sound.Tmplt (*continued*)

```
40060     CALL  LC,_H2,_0,_H3,_0,L,F\$0A1D\:  CALL  LC,_H2\$1002\:  CALL  PO,4,P1,P3:
              CALL PO,2,4 + P1,0 + L3 / 256: CALL PO,2,6 + P1,PB: CALL PO,2,16 + P1,255
          : RETURN
40099  REM
                Play sound

40100   CALL  LC,0,4\$1408\F:  IF  F  =  0  THEN  40100
40110    CALL  LC,16\$0F08\:  CALL  LC,$0401,_P1\$0E08\:  RETURN
40199  REM
                  Play sound (exclusive)

40200     CALL  LC,$0401,_P1\$0E08\
40210   CALL  LC,0,4\$1408\F:  IF  F  =  0  THEN  40210
40220    CALL  LC,16\$0F08\:  RETURN
56099  REM
                 Sound / ACE setup

56100  DT  =  D1:  GOSUB  45030:SO  =  D1:  GOSUB  45030:AC  =  D1:D1  =  DT:  CALL  LC,SO\
             $0208\:  CALL LC,AC\$021D\
56110  L  =  18:  GOSUB 45000:H1  =  H:P1  =  P:  FOR  A  =  0  TO  17:  CALL  PO,1,0  +  A  +  P1,
          0: NEXT
```

Long.Strt.Tmplt

```
0     REM     ====================================
1    REM    BASIC  CODE  TEMPLATE
2     REM     ------------------------------------
3    REM     Long  Program  Initialization
4     REM     ------------------------------------
5    REM    Reference:
6     REM     ------------------------------------
7   REM   So  What  Software          V1.0   12-Feb-90
8     REM     ====================================
9   END  :  REM

10    FN  $  =  "__Program  Name__":  PRINT     CHR$  (4);"CHAIN  CB.STARTUP"
```

Further Reference

Call Box BASIC Manual V2.0
Tech Note 2, 3,4,6 and 10

So What Technical Notes #0

Call Box
TECHNICAL NOTES Contents

Julyl 1,1990

This technical note is the index for all of the Call Box technical notes.*R* = Revised *** = New

So What Technical Notes #14

Call Box
Controls in Windows

Written by: William Stephens May 15,1990

This technical note describes how to create windows that have controls in them. This procedure gives you more flexibility than using just the Dialog Manager.

The Dialog Manager is a very convienient tool to use for user interaction but is limited in the types of controls it can present. You can take matters into your own hands and be your own Dialog Manager by using a Window that contains controls that you manage yourself. There is a little more work involved in managing your own controls than there is in operating the Dialog Manager, but the increased functionality of dialogs created this way far outweighs the added difficulty.

Virtually every call you will make with this type of window will be of the Long Call variety. Before we get into the exact procedure we should first cover the command syntax:

NewControl
This call adds a control parameter list to the specified window record and returns the handle for the control created. This handle is used to find the control later on and should be stored by your program for each control created. If zero is returned then there was an error in creating the control.

```
CALL LC,_0,_W,_R,_T,F,V,P1,P2,_C,_0,_0\$0910\_CH
```

FindControl
This call returns the handle of which control was hit (*if any*) based on the global mouse coordinates and the window pointer. You must supply a 4 byte buffer for the results.

```
CALL LC,0,_CH,X,Y,_W\$1310\P
```

TrackControl
This call acts like the Event Manager and tracks the mouse action you take after button down on the control. If you depress the mouse on a control and then without releasing, move the pointer off the control this routine will signify that no control has been selected.

```
CALL LC,0,X,Y,_0,_CH\$1510\P
```

DrawControls
This call draws all the controls in the specified window. This routine is usually used in update event loops and in wContDefProc's.

```
CALL LC,_W\$1010\
```

GetCtlValue
This call gets the value of the selected control.

```
CALL LC,0,_CH\$1A10\V
```

InvalRect
This call tells the Window Manager that a rectangle has changed and must be updated.

```
CALL LC,_RE\$3A0E\
```

HiliteControl
This call hilites or un-hilites the specified control. This call is used to blink or dim buttons and things like that.
```
CALL LC,H,_CH\$1110\
```

SetCtlValue
This call sets the value of the selected control.
```
CALL LC,V,_CH\$1910\V
```

SetContentDraw
This call sets the specified window with a new wContDefProc (*window contents drawing procedure*).
```
CALL LC,_P,_W\$490E\
```

BeginUpdate
This call sets up the Window Manager to handle update events.
```
CALL LC,_W\$1E0E\
```

EndUpdate
This call ends the update session.
```
CALL LC,_W\$1F0E\
```

_W
Windows pointer, Get from CALL WN,8...

_R
Pointer to controls enclosing rectangle

_T
Pointer to text string, Pascal 0

F
Control Flags (see fig. 14.1)

V
Controls Value.

P1
Data size for scroll bar, if not scroll then 0.

P2
View size for scroll bar, if not scroll then 0.

H
Hilite Flag 0 = none, 1-253 = part code, 255 = inactive

_C
Standard Control TypeValues:
$00000000 (0) = Button
$02000000 (33554432) = Check Box
$04000000 (67108864) = Radio Button
$06000000 (100663296) = Scroll Bar
$08000000 (134217728) = Size Box

_CH
Controls handle.

X
Horizontal mouse coordinate

Y
Vertical mouse coordinate

_P
Contents Draw pointer usually 0

_RE
Window rectangle pointer

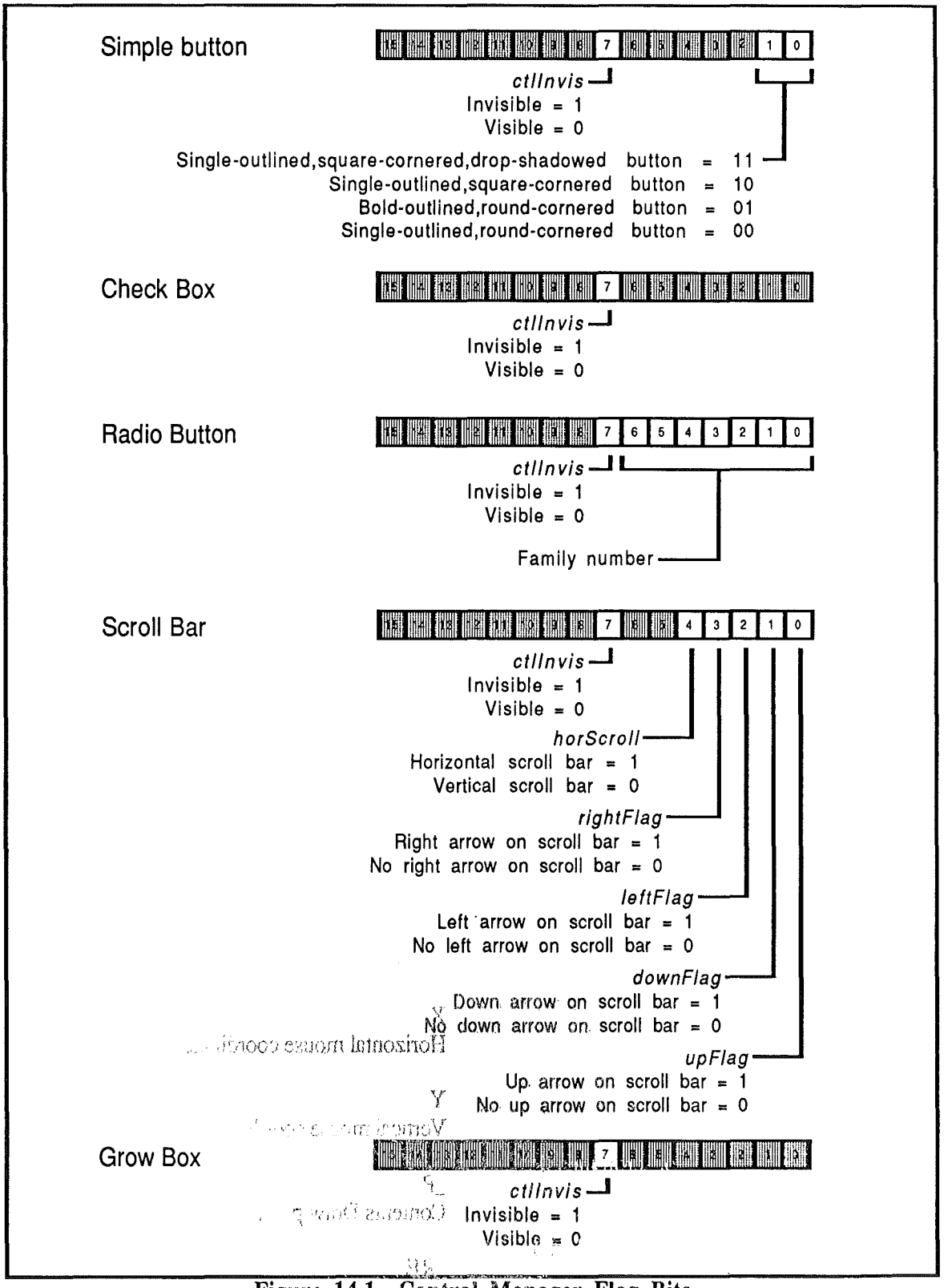


Figure 14.1 Control Manager Flag Bits

This procedure is shown in the program "control.demo" on the C.B.P.A.2 Sampler disk. This program was created by starting off using the "Desktop.Tmplt" program found on C.B.P.A.1 and then control operating code was added.

The first thing you need is a Window for your controls to be drawn in. Use the Call Box Window Editor to create a window with the following specifications...

Window type: Alert
Normal Rectangle: 25,150,175,550
Zoom Rectangle: 0,0,0,0
wDataH and wMaxH: 150
wDataW and wMaxW: 400

Save this window away to disk as an object file named "ctrl.window". Now load up the program template called "Desktop.Tmplt" and save it to disk as the program "my.ctrl.demo". All of the following steps will involve adding program lines to the "my.ctrl.demo"program... Let's roll up our sleeves and make a control window now!

Next we need is a work area for some of the control parameters. In this example I will use a "Direct Page" (*see tech note 4*) because I will not need over 256 bytes of space and both poke/peek and LongPoke/LongPeek commands work there. For larger control records you can use a larger block of memory as allocated by the Memory Manager in tech note 2. You may, however wish to use several direct pages instead because even though using direct pages decreases the available Applesoft code memory, both kinds of peeks and pokes will work. Memory Manager memory blocks can only be LongPeeked or LongPoked unless they are located in bank zero... which is impossible because Call Box BASIC and the SYSTEM own all of bank zero and nothing is left down there to allocate except for direct page requests made to ProDOS8 and Call Box BASIC.

Type in line 56010 which allocates a direct page for your use and assigns the variable UP to point to it.

```
56010  CALL PO,4,704,$F52001A9: CALL PO,2,708,$60BE: CALL704:A =
       PEEK(PO + 244): A = A + 1:POKE PO + 244,A:UP = PO - (A*256)
```

You now need to install the control wContDefProc. This is a native mode machine code routine which is specified as follows:

```
wContDefProc    pea   $0000      ;Push on the Window
                pea   $0000      ;Pointer
                ldx   #$1010
                jsl   $E10000    ;DrawControls
                rtl              ;Exit
```

Type in line 56020 to put this code in memory.

```
56020  DATA 244,0,0,244,0,0,162,16,16,34,0,0,225,107: FOR X = 0 TO 13:
       READ A: POKE UP + X,A: NEXT : DX = 16
```

The next thing to do is to put in the control rectangles and the control text if any. These two items need to be poked into your workspace because the control manager handles these items by their pointers and not the actual data themselves. This is a common type of referencing in tools and this is why you usually need some kind of workspace to use tools effectively. In this example we will use all 5 types of standard controls, namely: Simple Button, Check Box, Radio Button, Scroll Bar and Grow Box. I have divided the 256 byte work area into 16- 16 byte long sections where each section handles either a rectangle or a Pascal type 0 string. The first 16 bytes of this area contains the wContDefProc you just entered with line 56020. The first rectangle will be located 16 bytes into the area and its corresponding string will reside 128 bytes from the beginning of the rectangle. The second will be 32 bytes in and 160 bytes respectively and so on. This scheme is easy to index and is written by the control poker code at 55000.

Let's enter the control poker code first:

```
55000    REM
                Control Poker

55010    READ A: CALL PO,2,0 + DX + UP,A: READ A: CALL PO,2,2 + DX
         + UP,A: READ A: CALL PO,2,4 + DX + UP,A: READ A: CALL
         PO,2,6 + DX + UP,A: READ A$:DX = DX + 128: GOSUB 55900:
         DX = DX - 112: RETURN
55900    L1 = LEN (A$): FOR I = 1 TO L1:A = ASC (MID$ (A$,I,1)): POKE
         DX + UP + I,A: NEXT : POKE DX + UP,L1: RETURN
```

Use the following lines to poke the controls rectangles and text:

```
56029    REM
                Poke in the controls rectangles and text

56030    DATA 46,240,60,300,Button: GOSUB 55000
56032    DATA 68,240,82,400,Check Box: GOSUB 55000
56034    DATA 86,240,100,380,Radio Button: GOSUB 55000
56036    DATA 106,240,120,380,.: GOSUB 55000
56038    DATA 129,240,143,268,.: GOSUB 55000
```

This completes the Environmental Initialization for your program. Basically, you have created and initialized your direct page work area and installed the support data for the 5 controls. There is a bit more initialization left to do but some other things must happen first. You will notice that line 20 of your program ends with a GOSUB 56000 which runs all the code you just typed in. Add one more statement to the end of this line. This statement puts a pointer to the Call Box BASIC TaskRecord in the variable VR. This pointer is needed for Update event detection later on.

```
VR = ( PEEK (PO + 120) * 65536) + 593
```

Next you need to load your window in as entity number 4:

```
52      CALL WN,0,4,"entity:window:ctrl.window": CALL WN,1,4:
        GOSUB 42
```

Now that the window is in we can complete the control initialization. The following line derives the windows pointer and then patches this pointer into the control wContDefProc located in your direct page workspace:

```
110     CALL WN,8,4,WP,1: CALL PO,2,1 + UP,0 - INT (WP / 65536):
        CALL PO,2,4 + UP,0 + WP - INT (WP / 65536) * 65536
```

It's now time to issue the NewControl calls which add the controls to the window:

```
112     CALL LC,_0,_WP,_16 + UP,_16 + 128 + UP,!0000000000000011
        ,0,0,0,$00000000,_0,_0\$0910\C1
114     CALL LC,_0,_WP,_32 + UP,_32 + 128 + UP,!0000000000000000
        ,0,0,0,$02000000,_0,_0\$0910\C2
116     CALL LC,_0,_WP,_48 + UP,_48 + 128 + UP,!0000000000000000
        ,0,0,0,$04000000,_0,_0\$0910\C3
118     CALL LC,_0,_WP,_64 + UP,_64 + 128 + UP,!0000000000011100
        ,0,1,10,$06000000,_0,_0\$0910\C4
120     CALL LC,_0,_WP,_80 + UP,_80 + 128 + UP,!0000000000000000
        ,0,0,0,$08000000,_0,_0\$0910\C5
```

the final step in getting this window up is to hookup the wContDefProc, show the window and draw the controls:

```
180     CALL LC,_UP,_WP\$490E\: CALL WN,4,4,1: CALL LC,_WP\$1010\
```

If you run your program at this point you will see your window with the 5 controls drawn in it. These controls will not work as of yet but they will at least be there! The following discussion is about how to operate the controls that you have created. Operating the controls is similar to operating a dialog box except that you use TaskMaster to get your events rather than a proprietary dialog manager command like ModalDialog. You want to respond to Window Content Hit events (*code 19*) in order to find, track and respond to control hits.

Add the following line to detect a hit in the contents region of your window:

```
240     IF C = 19 THEN 500: REM Contents Hit!
```

This line will route any mouse click in your window to the routine at 500. The first thing you need to do at 500 is to check and see if a control was hit and if it was, which control it is. The FindControl command does this task and returns the controls handle. The next step is to fetch this handle and then TrackControl which checks if the mouse button was released in the same control. If it is then this can be considered a valid control hit which you can now process. Type in the following lines to Find and then Track your control:

```
500     CALL LC,0,_250+UP,X,Y,_WP\$1310\P: REM FindControl
510     IF P = 0 THEN 200
515     CALL PE,4,250 + UP,H: REM Fetch the control handle
520     CALL LC,0,X,Y,_0,_H\$1510\P: REM TrackControl
525     IF P = 0 THEN 200
```

Now we are in the home stretch... all that is left to handle is comparing the returned control handle to the handles returned from the NewControl calls and then taking some action based on which control was hit. Type in the following lines to compare the handles:

```
530     IF H = C1 THEN 598: REM Button Hit!
535     IF H = C2 THEN 600: REM Check Box Hit!
540     IF H = C3 THEN 600: REM Radio Button Hit!
545     IF H = C4 THEN 620: REM Scroll Bar Hit!
550     IF H = C5 THEN 598: REM Grow Box Hit!
```

As you can see some of the control handlers are the same... as a matter of fact I am only using 3 handlers for 5 controls. These handlers are analogous to some routine that you want to run in response to a control hit. Buttons and the Grow Box have no radical function except to remain at value 0 until you click in it when the value changes to 1... trapping the value of a button is kind of meaningless because just the fact that you detected a hit in the control is enough justification to act upon the hit. For our purposes we will use a "Do Nothing" routine for these controls:

```
598     GOTO 200
```

Check boxes and Radio Buttons have a more complicated life. These controls display and retain a status (*either checked and unchecked or selected and unselected*). You handle these controls just the same way you do in Dialogs, the only thing that changes is the Set and Get commands... the methodology is identical. In this example we will use a "value toggle" routine for both:

```
600     CALL LC,0,_H\$1A10\V: CALL LC,1 - V,_H\$1910\: GOTO 200
```

This leaves us with scroll bars which are more complicated still. Scroll Bars actually have 5 parts to them... 2 arrows, 1 thumb and 2 page regions (*the greyed areas*). Each of these parts has a part code. (see fig. 14.2)

0	No Part	11	Editable Line
1	Reserved	12	User Item
2	Simple Button	13	Long Static Text
3	Check Box	14	Icon
4	Radio Button	15-31	Reserved (internal)
5	Up Arrow	32-127	Reserved (application)
6	Down Arrow	128	reserved (internal)
7	Page Up	129	Thumb
8	Page Down	130-159	Reserved (internal)
9	Static Text	160-253	Reserved (application)
10	Size Box	254-255	Reserved (internal)

Figure 14.2 Control Manager Part Codes

In this example we will handle the arrow and page regions the same to simplify this example. Each part code usually responds to a different handler. The plan here is to check which part of the Scroll Bar was hit and then alter the value of the Scroll Bar to suit... a click on the left arrow or page will cause the control value to decrease by 1 and a click on the right arrow or page will increment it..

Type in the following lines to install this simple scroll bar handler:

```
620     IF P = 6 OR P = 8 THEN 650: REM Part Code = Rt.Ar. or Rt.Pg.
630     IF P = 5 OR P = 7 THEN 660: REM Part Code = Lt.Ar. or Lt.Pg.
640     GOTO 200
650     CALL LC,0,_H\$1A10\V: IF V = 9 THEN 200
651     V = V + 1: CALL LC,V,_H\$1910\: GOTO 200
660     CALL LC,0,_H\$1A10\V: IF V = 0 THEN 200
661     V = V - 1: CALL LC,V,_H\$1910\: GOTO 200
```

Now that you think you are done it is time to inform you that there is one more detail to take care of... that is Update Events! If something changes in your window the Window Manager will fix up the window to appear just right automatically, this is one of the benifits of using the desktop environment. As you may have noticed you have not entered any commands to draw the controls except initially in the setup code. You will not have to handle this task directly... the Window Manager handles this for you through the wContDefProc and the Update code. The Update code is very similar to the wContDefProc except that 3 new commands are needed. These commands are InvalRect, BeginUpdate and EndUpdate. You will have to install 3 more lines of code to handle Update events. One line installs a rectangle for the whole window, one line detects the Update Event and the last is the update event handler: Type in the following lines to complete your Control Window:

```
56040   DATA 0,0,150,400,.: GOSUB 55000

230     CALL PE,2,VR,V: IF V = 6 THEN 295: REM Update Event

295     CALL LC,_96 + UP\$3A0E\: CALL LC,_WP\$1E0E\: CALL
        LC,_WP\$1010\: CALL LC,_WP\$1F0E\: GOTO 200
```

This is about it for this tech note. I have given you the basics and you can take it from here altho proceeding without the Apple IIgs Toolbox Reference Manuals is kind of like buying a new car and telling the salesman to leave the drive shaft out! Everything appears to be there but the car will not move on its own. The Toolbox Reference's present much more information than is economically possible in a forum like this.

Some of the benifits in putting controls in windows is that you can also put other things in there as well... like Line Edit Items, List Controls, Icons, Quickdraw artwork and virtually any graphic device possible with the Apple IIgs Toolbox. The handling of these other items may possibly be outlined in future tech notes but don't count on it... these tech notes take a lot of time and effort to produce and there is more important fish to fry!

Further Reference

Call Box BASIC Manual V2.1
Apple IIgs Toolbox Reference Vol. 1,2 and 3
Tech Note 2 and 4

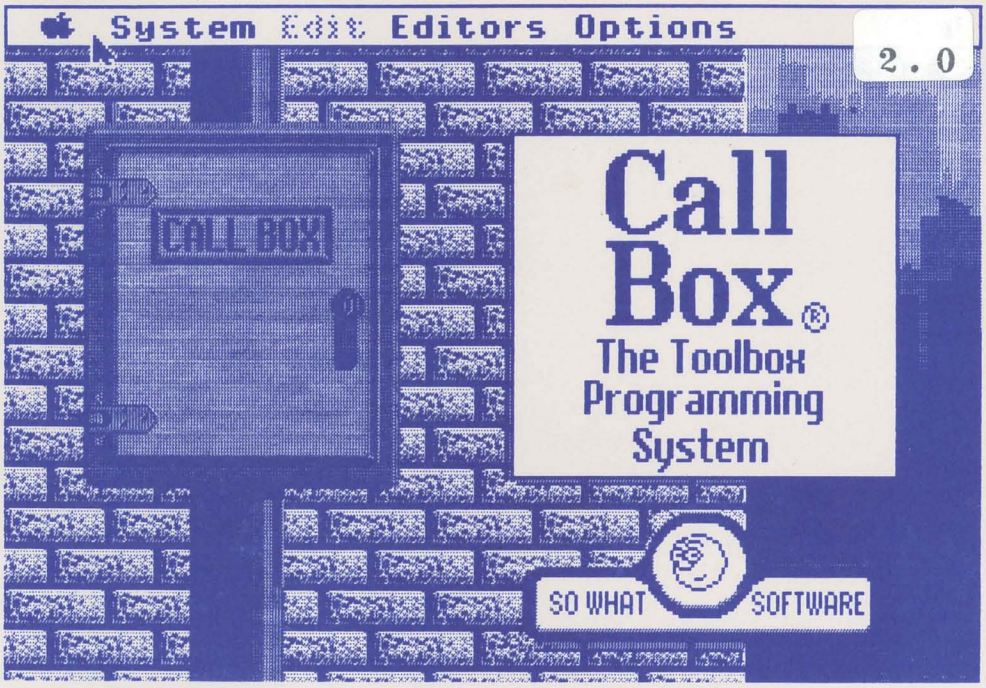
Call Box

About CALL BOX

Welcome to the wonderful world of **WYSIWYG.** (**W**hat **Y**ou **S**ee **I**s **W**hat **Y**ou **G**et.)

Create programs in assembly, C, pascal or even Applesoft BASIC using the powers of **CALL BOX** in a fraction of the time needed before.

CALL BOX is a programming system with several facets designed to maximize your programming ease and pleasure. Most popular languages are supported by the Editors included in this system which have been designed to produce "difficult to make" items needed in applications that use the Apple IIgs toolbox. You can create ICONS, PIXEL IMAGES, CURSORS, WINDOW PARAMETER LISTS, DIALOG TEMPLATES, and MENU TEMPLATES using the 4 WYSIWYG editors supplied on the **CALL BOX** disk. These editors support APW/ORCA Source code, OMF2 object files and Resources to allow you great flexibility when creating a program.



The **CALL BOX** BASIC Interface gives the ProDOS 8/Applesoft programmer access to the Apple IIgs toolbox functions once reserved for the C or assembly language programmer. Over 24 new CALLS and dozens of sub-CALLS make Applesoft BASIC a potent competitor in the program development arena. The functions of Quickdraw II, Event/Taskmaster, Quickdraw II aux., Window Manager, Control Manager, Dialog Manager, Menu Manager, and Line Edit Tools are available for Applesoft BASIC with a parametered CALL. Any toolbox tool can be operated through a generic long-CALL plus Applesoft enhancements like Long peek, Long poke, and Super Array are included to increase Applesoft's ability to deal with Apple IIgs.

CALL BOX starts up to a launching shell which has menu selections for the WYSIWYG Editors, Demo/tutorial, file utilities, and system routers. Your own application (such as your development shell) can be programmed into a user system router for easy flipping back and forth from your program editor to the **CALL BOX** system. The file utilities allow you to RENAME, DELETE, Set ACCESS bits, Set filetype and Set Auxtype for any file online currently. Another utility is provided for the Applesoft user which allows you to change the default variables in the CB.VARS file.

The Demo/tutorial shows the Applesoft BASIC programmer how to use the many functions of the **CALL BOX** BASIC Interface by demonstration and example. Applesoft BASIC never looked so good and now operating under GS/OS V5.0 even faster!

Fully GS/OS compatible (System Disk 5.0), not copy protected.

SO WHAT SOFTWARE
10221 Slater Ave. Suite 103 Fountain Valley,Ca.92708