

Alpha Draft Programmer's Introduction to the Cortland

Programmer's Introduction to the Cortland

Alpha Draft
Part No. 030-3122
July 31, 1986

Writer: William H. Harris
Apple Technical Publications



Contents

Figures and Tables x

Preface About This Manual

- Don't read this book unless... 1
- Roadmap: How this manual fits into the Cortland Technical Manual Suite 2
- What's in this manual 7
- Other materials you'll need 7
- Visual cues 8
- Language notation 8

Chapter 1 Deciding to Write Desktop Applications

- About desktop applications 10
- Gains and costs 10
- Differences from Apple II programming 11
- Differences from Macintosh programming 12
 - If you're porting over from the Mac 13

Chapter 2 Understanding the Cortland Programming Environment

- About native mode and this book 15
- Hardware/Firmware factors 15
 - Featuring the 65816 15
 - Working with the stack and direct page 16
 - Managing memory 18
 - The screen display 20
 - Summary: the native mode execution environment 21
- The Cortland toolbox 22
 - Calling the tools 22
 - The big five 22
 - Desktop interface tools 23
 - Math tools 24
 - Printer tools 24
 - Sound tools 24
 - Specialized tools 24

Chapter 3 Programming a Desktop Application

- About event-driven applications 27

An outline of an event-driven application	27
An example event-driven application	28
Step 1. Setting up the programming environment	28
Step 2. Defining the data structures and the data	29
Step 3. Starting up the Tool Locator	29
Step 4. Starting up the Memory Manager	30
Step 5. Starting up the Miscellaneous Tools	30
Step 6. Loading other tools	30
Step 7. Requesting direct page space for the tools	31
Step 8. Starting up QuickDraw II	32
Step 9. Starting up the Event Manager	33
Step 10. Starting up the Window Manager	34
Step 11. Starting up the Menu Manager	34
Step 12. Starting up other tool sets	34
Step 13. Setting up environment for main event loop	35
Step 14. Setting up the system menu bar	35
Step 15. Beginning the main event loop	36
Step 16. Handling application-specific events	37
Step 17. Shutting down the application	37
Preparing an event-driven algorithm	37
An event-driven application is...	38
Some practical hints	38

Chapter 4 Displaying in Color

The color possibilities	40
Drawing in color	42
Modifying the colors	43
Changing a color in a color table	43
Swapping whole color tables	44
Text colors	44

Chapter 5 Dealing With Files

About file handling - ProDOS 16	46
Creating files	46
Opening files	47
Reading and writing files	47
Closing and flushing files	48
Presenting the standard file interface	48

Chapter 6 Coding Static and Dynamic Segments

Introducing static and dynamic segments	50
Coding static segments	50
Coding dynamic segments	51

Chapter 7 Writing a Corland Desk Accessory

The Different Styles of Desk Accessories	53
Writing Classic Desk Accessories	53
Writing New Desk Accessories	54

Chapter 8 Writing Shell Applications

Shell Environments 57

Writing top-level shells 57

Writing programs to run under shells 57

Appendix A Assembly Language Source Code for the Event-Driven Example

Appendix B Pascal Source Code for the Event-Driven Example

Appendix C C Source Code for the Event-Driven Example

Appendix D Application Disk Files

Appendix E Pointers to Other Cortland Books



Preface



About This Manual

Don't read this book unless...

you are a programmer interested in developing new applications or desk accessories for the Cortland. This book is specifically designed to get you started writing source code which takes full advantage of the powerful capabilities of the Cortland.

Please don't expect this book to teach you about all the varieties of programming possible for the Cortland. In particular, the book does not tell you how to revise an existing Apple II application to take advantage of the new Cortland features. Applications can be written for the Cortland that also run on the other members of the Apple II family, but that kind of programming lies outside the scope of this book.

In this book, we wish to present a particular style of application known as a **desktop application**. Such applications present commands as options in pull-down menus, and material being worked on appears in rectangular areas of the screen called windows. The user can select commands or other material by using the mouse to move a pointer around the screen. In addition, such applications are **event-driven**; that is, the user causes an event (typically a mouse click or a keypress) which the application takes care of and then begins waiting for the next event.

We assume that you are either a programmer or very familiar with programming concepts, but we do not assume that you know C, Pascal, or 65816 assembly language. In fact, except in the appendixes which list the source for the example desktop program, we don't refer to any real programming language at all. We prefer to present the concepts in English, so that you can understand the overall structure of an event-driven program without concentrating too much at first on the details.

Besides the desktop applications, we also present some information on how to work with Desk Accessories and Shell Applications. Note that this book is not intended to be an exhaustive reference. There are many levels of understanding about programming on the Cortland; this book is on the first level, while other books in the Cortland suite provide the deeper levels. Those books are briefly described in the next section of this preface.

Roadmap: How This Manual Fits Into the Cortland Technical Manual Suite

The Cortland has many advanced features, making it more complex than earlier models of the Apple II. To describe it fully, Apple has produced a suite of technical manuals. Depending on the way you intend to use the Cortland, you may need to refer to a select few of the manuals, or you may need to refer to most of them.

The technical manuals are listed in Table 1. Figure A-1 is a diagram showing the relationships among the different manuals. In addition, Appendix E has some additional specific references for particular subjects application developers might be interested in.

Table 1
The Cortland Technical Manuals

Title	Subject
Technical Introduction to the Cortland	What the Cortland is
Cortland Hardware Reference	Machine internals—hardware
Cortland Firmware Reference	Machine internals—firmware
Programmer's Introduction to the Cortland	Concepts and a sample program
Cortland Toolbox Reference: Volume 1	How the tools work
Cortland Toolbox Reference: Volume 2	More Toolbox specifications
Cortland Programmer's Workshop	The development environment
Cortland Workshop Assembler Reference*	Using the assembler
Cortland Workshop C Reference*	Using C on the Cortland
Cortland Workshop Pascal Reference*	Using Pascal on the Cortland
ProDOS 8 Technical Reference	ProDOS for Apple II programs

Cortland ProDOS 16 Reference

ProDOS, Loader, and Finder for Cortland

Human Interface Guidelines

For all Apple computers

Apple Numerics Manual

Numerics for all Apple computers

*There is a Pocket Reference for each of these.

The Introductory manuals

These books are introductory manuals for developers, computer enthusiasts, and other Cortland owners who need technical information. As introductory manuals, their purpose is to help the technical reader understand the features of the Cortland, particularly the features that are different from other Apple computers. Having read the introductory manuals, the reader will refer to specific reference manuals for details about a particular aspect of the Cortland.

The Technical Introduction

The *Technical Introduction to the Cortland* is the first book in the suite of technical manuals about the Cortland. It describes all aspects of the Cortland, including its features and general design, the program environments, the Toolbox, and the development environment.

Where the *Cortland Owner's Guide* is an introduction from the point of view of the user, the *Technical Introduction* describes the Cortland from the point of view of the program. In other words, it describes the things the programmer has to consider while designing a program, such as the operating features the program uses and the environment in which the program runs.

The Programmer's Introduction

This book is the one that tells you how to begin writing programs that use the Cortland user interface (with windows, menus, and the mouse), and provides the concepts and guidelines you need. It is not a complete course in programming, only a starting point. It introduces the routines in the Cortland Toolbox and the program environment they run under. It includes a sample program that demonstrates how a program uses the Toolbox and deals with the operating system.

The machine reference manuals

There are two reference manuals for the machine itself: the *Cortland Hardware Reference* and the *Cortland Firmware Reference*. These books contain detailed specifications for people who want to know exactly what's inside the box.

The Hardware Reference manual

The *Cortland Hardware Reference* is required reading for hardware developers, and it will also be of interest to anyone else who wants to know how the machine works. It contains detailed information about the hardware, including the mechanical and electrical specifications of all connectors, both internal and external. It also describes much of the internal hardware, to provide a better understanding of the machine's features.

The Firmware Reference manual

The *Cortland Firmware Reference* describes the programs and subroutines that are stored in the machine's read-only memory (ROM), with two significant exceptions: Applesoft BASIC and the Toolbox, which have their own manuals. The Firmware Reference includes information about interrupt routines and low-level I/O subroutines for the serial ports, the disk port, and for the DeskTop Bus, which controls the keyboard and the mouse. The Firmware Reference also describes the Monitor, a low-level programming and debugging aid for the 65816.

The Toolbox manuals

Like the Macintosh, the Cortland has a built-in Toolbox. The *Cortland Toolbox Reference, Volume 1*, introduces concepts and terminology and tells how to use the tools. It also tells how to write and install your own tool set.

Of course, you don't have to use the Toolbox at all. If you only want to write simple programs that don't use the mouse, or windows, or menus, then you can get along without the Toolbox. However, if you are developing an application that uses the desktop interface, or if you want to use the Super Hi-Res graphics display, you'll find the Toolbox to be indispensable.

The Cortland Programming Languages

Apple is currently providing an assembler and compilers for C and Pascal. Other compilers can be used with the workshop, provided that they follow the standards defined in the *Cortland Programmer's Workshop Reference*.

There is a separate reference manual for each programming language on the Cortland. Each manual includes the specifications of the language and of the Cortland libraries for the language, and describes how to write, compile or assemble, and link a program in that language. The manuals for the languages Apple provides are the *Cortland Programmer's Workshop Assembler Reference*, the *Cortland Programmer's Workshop C Reference*, and the *Cortland Programmer's Workshop Pascal Reference*.

The Programmer's Workshop manual

The development environment on the Cortland is the Cortland Programmer's Workshop. The Workshop is a set of programs that enable developers to create and debug application programs on the Cortland. The *Cortland Programmer's Workshop Reference* includes information about the parts of the workshop that all developers will use, regardless which programming language they use: the Shell, the Editor, the LInker, the Debugger, and the utilities. The manual also tells how to run other programs, such as custom utilities and compilers, under the Workshop Shell.

The standard object-module format, defined in the *Programmer's Workshop Reference*, makes it possible to combine segments written in different languages into a single program. The workshop manual includes a sample program to show how this is done.

What About ProDOS

There are two operating systems that run on the Cortland: ProDOS 16 and ProDOS 8. Each operating system is described in its own manual: *ProDOS 8 Reference* and *ProDOS 16 Reference*. ProDOS 16 uses the full power of the Cortland and is not compatible with earlier Apple IIs. The ProDOS 16 manual includes information about the System Loader, which works closely with ProDOS 16. If you are writing programs for the Cortland, whether as an application programmer or as a system programmer, you are almost certain to need the ProDOS 16 Reference.

ProDOS 8, previously just called ProDOS, is compatible with the models of Apple II that use 8-bit CPUs. As a developer of Cortland programs, you need to use ProDOS 8 only if you are developing programs to run on 8-bit Apple II's as well as on the Cortland. We don't consider those kind of programs in this manual.

All-Apple manuals

In addition to the Cortland manuals mentioned above, there are two manuals that apply to all Apple computers: *Human Interface Guidelines* and *Apple Numerics Manual*. If you develop programs for any Apple computer, you should know about those manuals.

The *Human Interface Guidelines* describes Apple's standards for the human interface of programs that run on Apple computers. If you are writing an application for the Cortland, you should be familiar with the contents of this manual.

The *Apple Numerics Manual* is the reference for the Standard Apple Numeric Environment (SANE), a full implementation of the IEEE standard floating-point arithmetic. The functions of the Cortland's SANE tool set match those of the Macintosh SANE package and of the 6502 Assembly Language SANE software. If your application requires accurate arithmetic, you'll probably want to use the SANE routines in the Cortland. The Apple Numerics Manual is the comprehensive reference for the SANE numerics routines. A description of the version of the SANE routines for the 65816 is available through the Apple Programmer's and Developer's Association, administered by the A.P.P.L.E. cooperative in Renton, Washington.

What's In This Manual

This manual has been designed to introduce you to event-driven programming on the Cortland. To that end, Chapters 1 and 2 provide the background material you need to understand before you can begin to code event-driven applications.

Chapter 3, "Programming an Event-Driven Application", is the core of the book. In that chapter, we present a walkthrough of an example event-driven application. The housekeeping for such applications is so standard that you will be able to steal all of the sequence, and maybe most of the code, for your own applications.

Chapter 4, "Displaying in Color", is an introduction to the colorful capabilities of the Cortland. We introduce you to some of its simpler aspects, and briefly sketch out how you can begin to change the colors.

Chapter 5, "Dealing With Files", provides an overview of the ProDOS 16 file calls. You'll need to know about them if your application is going to read or write files from a disk.

Chapter 6, "Coding Static and Dynamic Segments", tells you the basic rules that apply to how programs can be segmented for the Cortland.

Chapter 7, "Writing a Cortland Desk Accessory", discusses the two different types of desk accessories and the basic process desk accessories must go through to execute safely.

Chapter 8, "Writing Shell Applications", indicates the special rules for writing both top-level and subordinate shell applications.

Appendixes A, B, and C provide the source code for the event-driven application in 65816 Assembly, Pascal, and C respectively. Appendix D lists the files that have to be on a disk to make it a system disk, and Appendix E adds some cross-references for specific topics to other books in the Cortland suite.

Other materials you'll need

In order to program event-driven applications, you'll also need at least one of the languages available for the Cortland.

Visual cues

New terms

Each new term introduced in this manual is printed first in **boldface**. That lets you know that the term has not been defined earlier, and also indicates that there is an entry for it in the glossary.

Notes and warnings

The following typographical conventions mark special messages to you:

❖ *Note:* Text set off in this manner presents sidelights or interesting points of information.

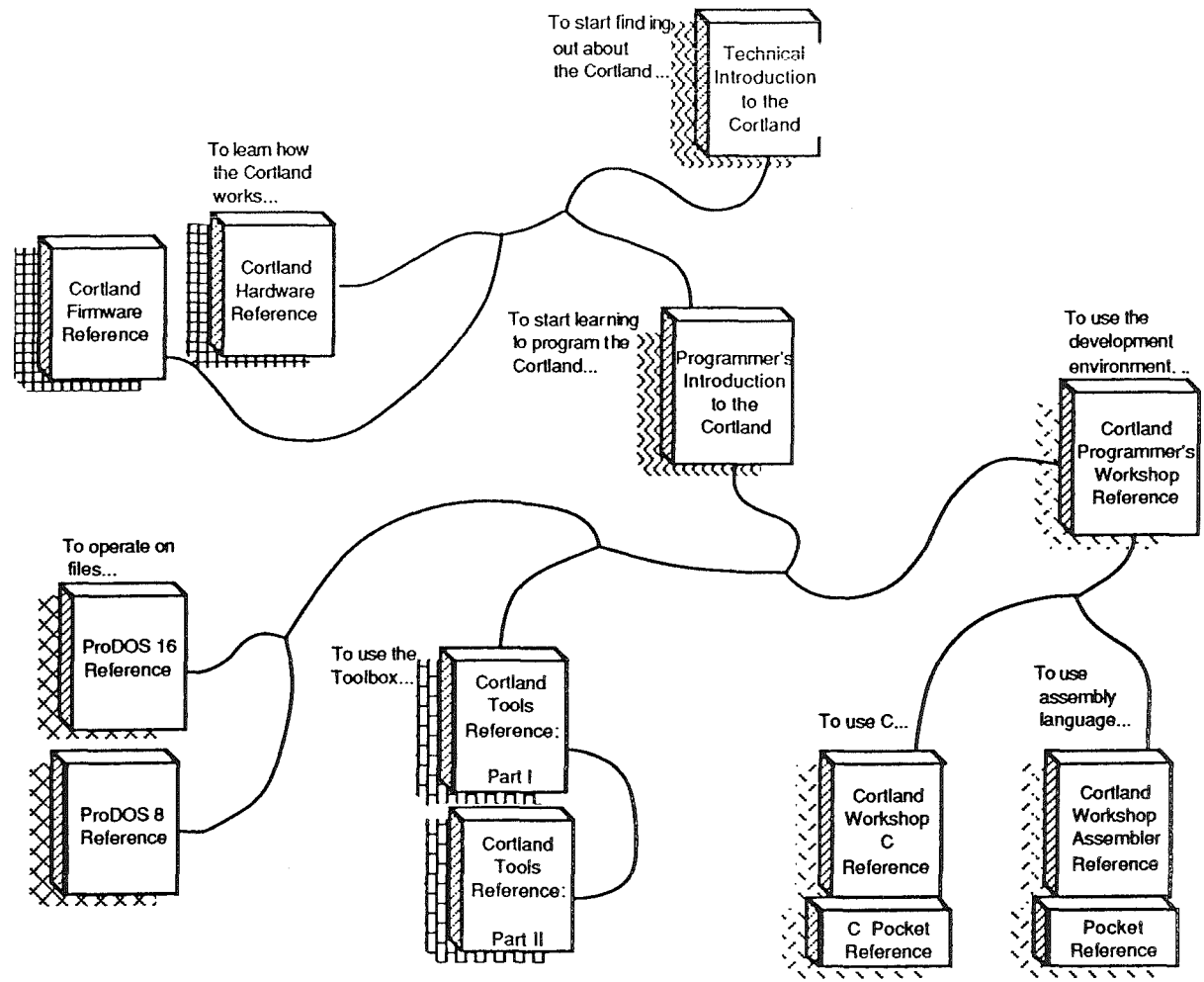
Important Text set off in this manner—with the word Important in the margin—presents important information or instructions.

Warning Messages like this indicate potential problems or disasters.

Language notation

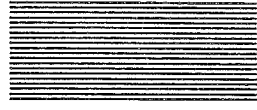
Since we don't use an actual computer language in this book, there isn't much special notation you need to know about. There's only a couple worth mentioning:

- Actual ProDOS commands are set off in all capital letters.
- (**Anything else???)

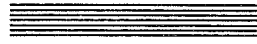


- Starting out on the Cortland
- Learning about the machine
- Operating systems
- Cortland Toolbox
- Programmer's Workshop





Chapter 1



**Deciding to Write Desktop
Applications**



About desktop applications

Desktop applications are those applications which adhere to the concepts presented in the *Apple Human Interface Guidelines*. Those guidelines establish a consistent user interface for all applications.

One of the most basic tenets of the guidelines is that users should feel they are controlling the program, rather than the program controlling them. On the Cortland, that guideline is embodied most fully in its support of **event-driven** applications. An event-driven application responds to the user's actions, such as a mouse click or a keypress, by taking some direct action. When that action is completed, the application waits for the next event, when it leaps into action again. This type of program is different (and we think better) than the "series-of-orderly-menus" programs, which assume that the user should do thus-and-so in such-and-such order.

The next section will help you decide whether or not you wish to write this type of application.

Gains and costs

The biggest reason to program desktop applications on the Cortland is the consistent interface such an application presents to the user. This means that users spend less time learning and more time using an application.

As usual, however, there's no such thing as a free lunch. Event-driven Cortland programs will not run on the Apple IIe and IIc, unless you write additional code which recognizes which machine it is running on and reacts accordingly.

Another cost is that you will have to learn event-driven programming, which is not necessarily easier than menu-driven programming. However, we think that this type of user interface is better for the user, and we prefer that the burden be placed on you as the programmer; after all, you're the professional. The disadvantage is minor when the ease of use and the compatibility between the Mac and Cortland interfaces are considered.

One final note: this is the programming style preferred for the Macintosh. The techniques you learn (although not the actual code, in most instances) will be directly applicable to that machine.

Differences from Apple II programming

❖ *Note:* This section is meant for experienced Apple II programmers only. If you're not familiar with the Apple II, skip this section and go on to the next one.

The Cortland has removed some of the limitations of the older 8-bit Apple II world.

In this book, we'll use **8-bit Apple II's** to refer to the Apple II, II+, IIe, and IIc.

The Cortland has more memory, 16-bit registers, a bigger stack and direct page, improved screen resolution, and advanced sound capabilities (for details, refer to Chapter 2 and to the *Cortland Technical Introduction*). The Cortland obviously represents a leap forward for the Apple II family. However, if you're an old Apple II hacker, the leap may also be in a slightly different direction than you anticipated. The syllogism can be constructed as follows:

1. The new capabilities of the Cortland are best accessed through desktop applications.
2. Desktop applications are designed to use the system software, rather than talking to the hardware directly.
3. Therefore, good Cortland Programmers don't write metal!

This departure from the old absolute address-oriented style of Apple II programming is not accidental. The software of the system has been designed so that it is self-defeating to use hardware entry points or absolute addresses. If you go ahead and use such things as hardware addresses, we won't be held responsible. We can almost guarantee that any "hard-coding" you attempt will not remain compatible with other versions of this machine in the future.

This policy has been deliberately adopted, in order to avoid the old snarl where changes and improvements could not be made because applications depended upon the actual locations. The current system is a lot more flexible, and we hope you'll soon get used to it.

The policy also extends to such things as memory management, where, instead of controlling memory directly, your application asks the Memory Manager for space. Once again, the flexibility and ease make up for the indirectness of the process. (See Chapter 2 under "Managing Memory" for details).

Similarly, the fixed-address global page of the 8-bit Apple II world has disappeared, to be replaced by ProDOS 16 calls.

Finally, note that the 65816 microprocessor adds some new addressing modes and instructions to the 6502 world.***Should I mention more language differences here???

Differences from Macintosh programming

❖ *Note:* This section is meant for experienced Macintosh programmers only. If you're not familiar with the Mac, skip this section and go on to the next chapter.

Beyond the obvious difference that the Cortland has a very different microprocessor than the Macintosh, resulting in different programming languages and capabilities, there is an essential similarity between the Cortland and the Mac: both use desktop, event-driven applications. Many of the toolbox routines are even named the same and function in the same way. The toolbox has been implemented differently on the Cortland, though, so watch out for differences when using the tools. In particular, be aware that the Cortland doesn't have a Resource Manager. The Cortland can load data segments (which could be constructed to perform the same functions as resources), but a standard format for maintaining the data segments is not provided.

You'll also have to examine the Cortland tool calls you make carefully. There are some tools that do not exist in the Mac (at least at the time of this writing). For example, the Window Manager call TaskMaster makes taking care of scrolling, zooming, etc. easier on the application. For another example, a window type exists on the Cortland with predefined scroll boxes. Such differences are typical; you can safely assume that a Cortland event-driven program will work in the same general fashion as a Mac event-driven program, but you can't assume that all the calls work in exactly the same way.

*****MPW-CPW comparisons appropriate here, or better left to CPW book?*****

There are also some specific things you should check for if you're trying to port a Mac application over; these are documented in the next section.

If you're porting over from the Mac...

*****is this section appropriate, or too specific? I just couldn't imagine where else "porters" would look for the information? Or won't that many be ported over???'*****

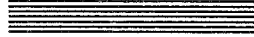
The steps to take if you're attempting to port over a Macintosh application are as follows:

- Convert resource files to data segments. Whenever independent (or shared) data blocks are needed, place them into a separate segment.
- Remove references to Mac resource tools and replace them with the appropriate non-resource from of the tool.
- If display dimensions exist as constants, change them. The Macintosh has an aspect ratio of x to x; the Cortland has a ratio of X to X.
- Replace File Manager calls with ProDOS 16 calls.

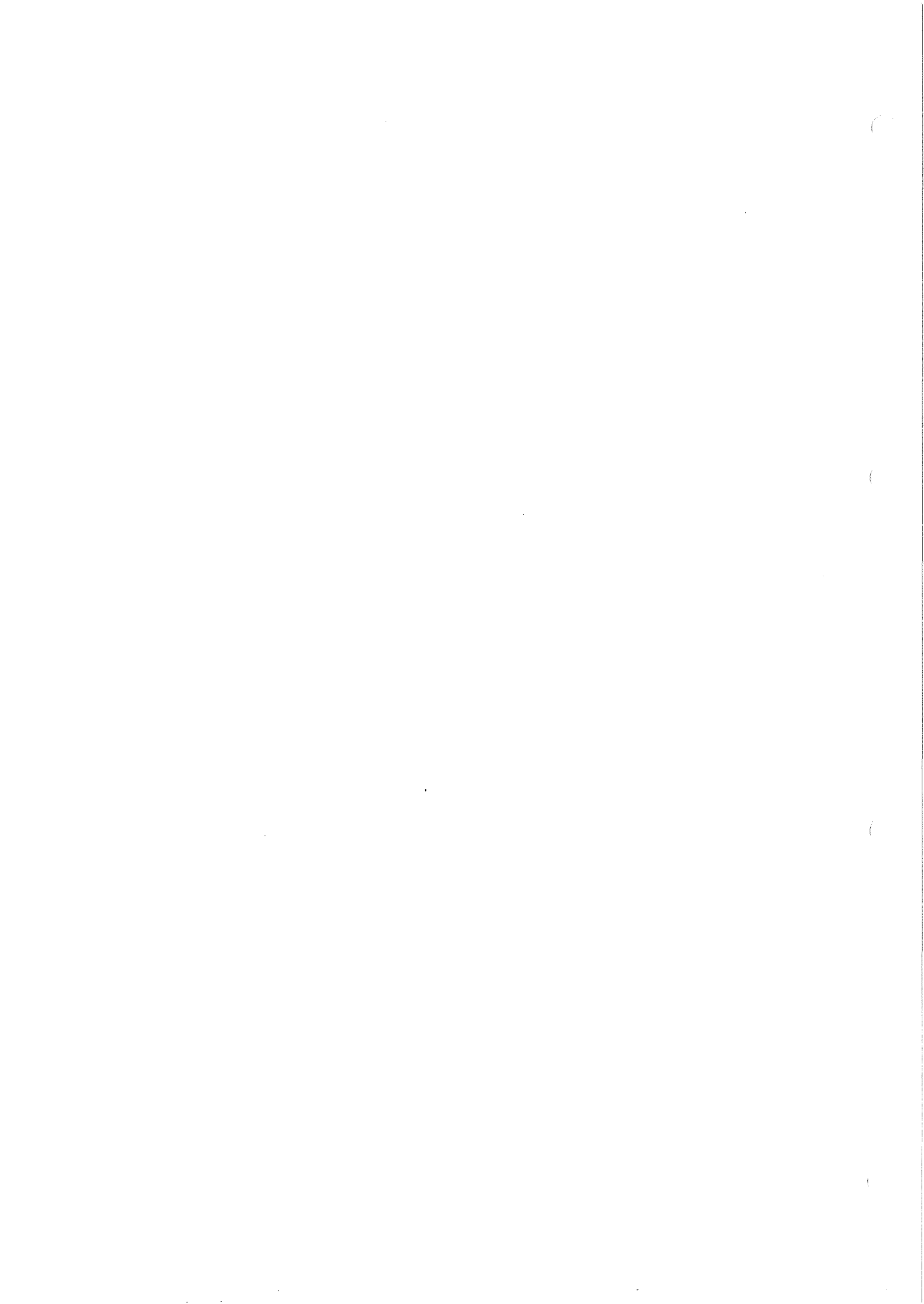
*****What else??? Is this list small, finite-but-huge, or infinite???'*****



Chapter 2



Understanding the Cortland Desktop Environment



About native mode and this book

The Cortland can operate as a full 16-bit microprocessor (in so-called **native mode**), or can emulate the 6502 of the 8-bit Apple II world, or can operate in several states in between. In this book, we consider only the native mode of the Cortland, since that is the only environment which supports desktop applications. In this chapter, we present the background you need in order to program in this environment.

Hardware/firmware factors

In this section, we discuss the increased power of the Cortland, and particularly the features you will have to consider when writing your program.

Featuring the 65816

From the programmer's point of view, the microprocessor is probably the most important piece of hardware. The Cortland uses a 65816 microprocessor, some of whose are summarized in the following table:

Table X-X Features of the 65816 Microprocessor

16-bit accumulator
16-bit X and Y registers
Relocatable direct page (16-bit Direct register)
Relocatable stack (16-bit Stack Pointer register)
8-bit Data Bank register
8-bit Program Bank register
Addressing modes which take advantage of the 16- and 24-bit possibilities offered by the above registers

Given these features of the 65816, you can see that the Cortland is truly a 16-bit machine when it is functioning in native mode. It is this "16-bitness" that provides many of the new capabilities of the Cortland, with a large, relocatable direct page and stack being among the most important.

Working with the stack and direct page

Because the 65816 microprocessor's stack-pointer register is 16 bits wide, the hardware stack may theoretically be located anywhere in bank \$00 of memory and be as much as 64K bytes deep. In practice, however, the stack is limited to about 32K bytes because of reserved memory areas in bank \$00.

The direct page is Cortland's equivalent to the 8-bit Apple II's zero page. The difference is that the direct page doesn't have to be page zero in memory. Like the stack, the direct page may be placed in any unused area of bank \$00. The 65816's Direct register is 16 bits wide, and all direct-page addresses are added as offsets to the contents of that register. At least in theory, then, the direct page can also be 64K bytes.

In practice, however, less space is available. First, only the lower 48K bytes of bank \$00 can be allocated; the rest is reserved for I/O and system software. Also, because more than one program can be active at a time, there may be more than one stack and more than one direct page in bank \$00. Furthermore, many applications may want to have parts of their code as well as their stacks and direct pages in bank \$00.

Important

Finally, the stack and the direct page grow towards each other in bank \$00. Direct-page addresses are offsets from the base of the allocated space, and the stack grows downward from the top of the space. Thus, the available space in bank \$00 must serve for both

ProDOS 16 provides no way to detect stack overflow or underflow, or the stack colliding with the direct page. Your program must be carefully designed and tested to make sure this can't happen.

Your program should therefore be as efficient as possible in its use of stack and direct-page space. The total size of both should probably not exceed about 4K bytes in most cases. Still, that means you can write programs that require stacks and direct pages much larger than the 256 bytes available for each on 8-bit Apple II's.

Allocating stack and direct page space

You can choose one of three ways to define the total amount of space for stack and direct page:

1. Assign the total amount of space for the direct page and stack when you assemble and compile your program. This is the most flexible method, since you don't have to change your source code at all and can test the various effects of changes by simply recompiling.
2. Allocate your own stack and direct-page space at run time.
3. Don't do anything, which gives the application a 1K total stack and direct page by default.

Assigning stack and direct page during compilation: You can define your program's stack and direct-page needs when you assemble or compile your program. The size you specify is the total amount of stack and direct-page space your program needs. The space is assigned as a page-aligned, locked, dynamic memory block of the appropriate size in bank \$00. When your program terminates with a QUIT call, the direct-page/stack block is purged along with the program's other allocated blocks. The stack and direct page are therefore not preserved between program starts; they must be reallocated each time the program is run.

Important There is no provision for extending or moving the direct-page/stack space after its initial allocation. Because bank \$00 is so heavily used, the space you request may be unavailable—the memory adjoining your stack is likely to be occupied by a locked segment. Make sure that the amount of space you specify at link time fills all your program's needs.

Allocating space at run time: Your program may allocate its own stack and direct-page space at run time. When ProDOS 16 transfers control to you, be sure to save the UserID value left in the accumulator before doing the following:

1. Using the starting or ending address left in the D or S register by ProDOS 16, make a FindHandle call to the Memory Manager, to get the memory handle of the automatically-provided direct-page/stack space. Then, using that handle, get rid of the space with a DisposeHandle call.
2. Allocate your own direct-page/stack space through the Memory Manager NewHandle call. Make sure that the allocated segment is purgeable, unmovable, and locked.
3. Place the appropriate values (beginning and end addresses of the segment) in the D and S registers.

Accepting the default stack and direct page: If the system finds no direct-page/stack specification at load time, it still returns the program's UserID and starting address to ProDOS 16, but it does not allocate a direct-page/stack space and returns zeros as the base address and size of the space. ProDOS 16 then calls the Memory Manager itself, and allocates a locked 1K direct-page/stack in bank \$00.

Once allocated, the default direct-page/stack is treated just it would be if it had been explicitly specified by the program.

Managing memory

(*What I need here, I think, is a definition and list of only those elements which affect the source code; that is, given the power of the Memory Manager, the programmer does not need to know details about the memory map or even what memory he is allowed to use. Is this correct???)***)**

The whole point of the memory management scheme on the Cortland is that you don't do it; that is, you as the programmer don't have to keep track of what memory is available, and don't have to worry about stomping on something already using some of memory. Managing memory on the Cortland has been largely lifted from you and given to the Memory Manager, who controls all of the memory available to your application.

You might think of the Memory Manager as a bookkeeper who signs out memory for your application to use, and protects that memory while your application is using it. When your application finishes with the memory, it is returned to the Memory Manager who can then sign it out again. *****place for a graphic of the Memory Manager as a bookkeeper - or would that be too cutesy or over-anthropomorphic???**

Given this approach, your application doesn't have to concern itself with the details about where to place code or data in memory. Instead, your application asks the Memory Manager for space, and the Memory Manager returns a **handle**, rather than a simple pointer, to that space. A handle points to a **master pointer** which points to the memory block. If a block is moved, the program can still find the block by examining the master pointer, which never moves. The following picture illustrates a handle and a master pointer.

(illustration from Memory Manager ERS)

The most visible way this method affects your source code is that you will sometimes need or want to use a pointer to the block rather than the handle. To do that, you need to **dereference** the handle by storing the current contents of the handle into a pointer.

When a handle has been dereferenced, the program must be sure the block doesn't move, which might occur if the Memory Manager is called. Your application specifies whether a particular block is movable or immovable. For the main segments of program code, it is often best to specify that the block be **fixed**, in which case the block will never move.

There are other restrictions on what parts of memory can be used; for example, desk accessories may not use special memory and code may not cross bank boundaries. For full details on all of the possible block attributes, see the Memory Manager chapter in Volume 1 of the *Cortland Toolbox Reference*.

For other program segments, you might only want to temporarily **lock** them down, so that they don't move while you're using them, but then **unlock** them so that the Memory Manager can make more efficient use of memory by moving blocks around. Similarly, data blocks should be locked and unlocked in the same fashion. This also brings up the concept of **purging**. When you unlock a block, and you don't think you'll need it for awhile, you should mark it purgeable so that the Memory Manager can throw out the contents of the block and free the space if it has to. If the space is not needed, the Memory Manager does not throw it out, thus leaving the original contents available to the application. If the block is purged, and the segment is requested, the Memory Manager has to ask the System Loader to reload the block.

The System Loader is the other memory-affecting entity you should know about. The System Loader has the capability to load segments while a program is running, which opens up some space- and efficiency saving options for applications.

The space-saving options are discussed more fully in Chapter 6 of this book.

Although you usually won't make calls directly to the System Loader, be aware that it functions most efficiently if you allow it to place programs wherever the Memory Manager tells it to. The upshot is that you shouldn't write code that is position-dependent. In particular, avoid the use of absolute origins and addresses, since those defeat the purpose of the relocating loader *****anything else to avoid???**. This doesn't mean that your code has to be relocatable once it's been placed, but simply that the code must not depend upon starting at a particular address.

See the *Cortland ProDOS 16 Reference* for complete details on the System Loader.

The screen display

The display modes new to the Cortland, called the **Super Hi-Res graphics modes**, are able to produce high-quality, high-resolution color graphics.

❖ *Apple II*: As usual, we won't talk about the display modes available on 8-bit Apple II's. Those are also available on the Cortland, but are outside the scope of this book.

There are two Super Hi-Res graphics modes on the Cortland. Both modes display 200 horizontal lines. **640 mode** can produce 640 vertical lines, while **320 mode** can produce 320 vertical lines. There are advantages to each mode; 640 mode obviously has higher resolution, while 320 mode is easier to work with in color.

You have to set the resolution in your program when you start up QuickDraw II, one of the Cortland tool sets. We'll discuss the Cortland tools in a moment. For now, you should also know that QuickDraw II determines the colors of the display. The default for text and graphics is black characters and lines on a white background. For information on changing the defaults, refer to Chapter 4 "Displaying in Color," in this book.

Summary: the native mode execution environment

ProDOS 16 automatically sets up the following environment for desktop, event-driven programs:

Table x.x The Native Mode Execution Environment

Operating system	ProDOS 16
Accumulator size	16 bits
Index register size	16 bits
Direct page address	any page in bank \$00
Stack address	any page from \$0800 to \$BF00 in bank \$00
Shadowing of I/O spaces	on
Shadowing of Text pages	on
Shadowing of Hi-Res graphics pages	off
Default display	Super Hi-Res
Available RAM	banks \$00 and \$01, expansion
RAM	parts of banks \$E0 and \$E1

The Cortland toolbox

In this section, we introduce the Cortland tools. The Cortland tools are collections of routines provided by Apple to make desktop programming easier. The tools support the standard desktop interface and provide you with building blocks to help you construct your application.

Calling the tools

How to access the tools depends, of course, upon the programming language from which you're calling them. However, there is at least one rule that makes using the tools relatively easy: from any language that supports the tools, you call an individual routine by invoking its name and providing the proper input parameters. Thus, the the tool calls are sort of a macro language in themselves.

See Volume 1 of *Cortland Toolbox Reference* for details on the calling conventions.

The big five

These tools provide the basic the framework upon which the other tools can build. All of these tools must be used in every event-driven application. The tools in this group are as follows:

- | | |
|---------------------------|--|
| Tool Locator | Provides the mechanism for dispatching tool calls. This tool allows you to get away with not knowing where in memory the tools reside; the Tool Locator knows and retrieves them when you make a tool call. Once you start the Tool Locator, its operation is automatic. |
| Memory Manager | Allocates all memory available to the application. When your application needs memory, you'll request it from the Memory Manager. |
| Miscellaneous Tool | Include mostly system-level routines that must be available. |
| QuickDraw II | Controls the graphics environment and draws simple objects. Other tools call QuickDraw II to draw such things as windows. |

Event Manager Traps events as they happen and passes them to the application.

Examples of their use are given in the next chapter.

Desktop interface tools

This group of tools controls the desktop interface. The Window and Menu Managers and Line Edit will almost always be used, in order to adhere to the Human Interface Guidelines; the other tools should be used if your application needs its feature (for example, the Dialog Manager will be needed if your application uses dialog boxes). Many of these tools are also needed to support New Desk Accessories.

The list of tools needed to support New Desk Accessories is given in Table 7.1 in Chapter 7 of this book.

The tool sets are as follows:

Window Manager Updates windows.

Menu Manager Controls and maintains the pull-down menus and the items in the menus.

Line Edit Presents text on the screen, and allows that text to be edited.

Control Manager Presents **controls**, which are objects on the screen that the user can manipulate with the mouse to cause instant action or change settings.

Dialog Manager Implements **dialog boxes**, which are to appear on the screen when an application needs more information to carry out a command.

Scrap Manager Supports the **desk scrap**, which allows data to be copied from one application to another (or from one spot to another within an application).

Desk Manager Enables applications to support desk accessories, which are "mini-applications" that can be run at the same time as another application.

Standard File Operations

Presents the standard user interface when a file is to be saved or opened.

Examples of Window Manager and Menu Manager calls are given in the next chapter.

Math tools

Integer Math Tool Set Supports mathematics routine with integers, long integers, and signed fractional numbers. Also converts integers, hex, and decimal numbers from one form to another.

SANE Supports the Standard Apple Numerics package, which allows IEEE standard extended-precision calculations.

Printer tools

High-Level Printer Driver
(*****Information not yet available.*****)

Low-Level Printer Driver
(*****Information not yet available.*****)

Sound tools

Sound Manager Supports the sound tool sets interfaces to the Cortland toolbox and provides the basic sound capabilities.

Note Synthesizer (*****to be provided*****)

(*****How many others of these should we list????*****)

Specialized tools


Apple Desktop Bus Tool Set
Controls Apple Desktop Bus activity.

Scheduler Prevents a tool call from crashing the system by asking for a temporarily unavailable system resource.


Text Tool Set

Provides an interface between Apple II character device drivers and applications running in native mode.

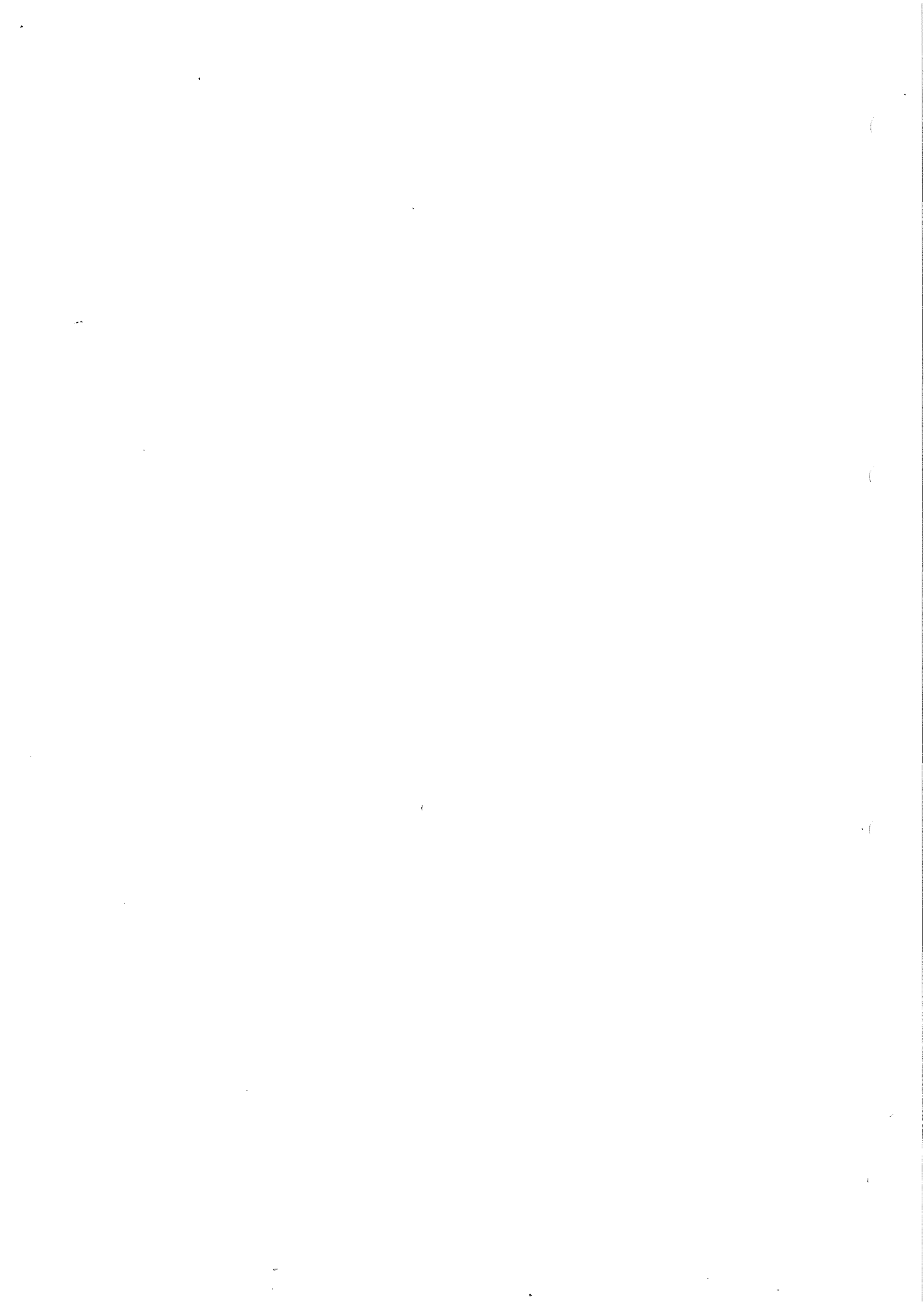




Chapter 3



**Programming an Event-Driven
Application**



About event-driven applications

If you want to program native-mode applications on the Cortland, you need to understand the concept and practice of **event-driven applications**. This type of application essentially waits for the user to do something (that is, cause an "event"), such as click the mouse or press a key on the keyboard. Such an application is organized around a main event loop that traps the event and reports its type. Conditional statements then determine what action to take based on what type of event happened.

This chapter explores a demonstration event-driven application developed by Apple. Since we can't tell which of the various languages you're using, we walk through the application without presenting the actual code.

The source code is reproduced in Appendixes A, B, and C if you wish to refer to it.

You also should have received the source code in the language package you purchased from Apple; feel free to steal the application and use it as a "skeleton" which you can then flesh out into your full-fledged application.

An outline of an event-driven application

This section simply lists the programming steps involved in writing an event-driven application. The list gives you an overview of the entire application. Each step is explained in more detail in the next section.

Note that the steps are introduced chronologically, and not listed in the order in which they might appear in source code. How modules are chained together depends upon the language you're using and your own predilections; we simply indicate the order in which things must or should happen.

(writer's note: should one more level of abstraction be provided in the following list; e.g. "Start up the basic tool sets needed by all applications")**

1. Depending upon the language, set up the programming environment.
2. Define the data structures and the data.
3. Start up the Tool Locator.
4. Start the Memory Manager.
5. Start up the Miscellaneous Tools.
6. Load any other desired tools.
7. Request zero page space from the Memory Manager.
8. Start up QuickDraw II.
9. Start up Event Manager.
10. Start up Window Manager.
11. Start up Menu Manager.
12. Start up any other tool sets needed.
13. Set up environment for main event loop.
14. Set up system menu bar.
15. Begin main event loop
16. Handle application-specific events.
17. Shut down application.

Note that, in some instances, the steps do not have to be taken in exactly the suggested order. We're presenting an order that works in the example application; this gives you a basis to begin experimenting with the source code to see what can be changed.

An example event-driven application

Step 1. Setting up the programming environment

If you're programming in assembly language, there's at least one thing you have to do before you start; you must turn on the 65816 instruction set and indicate under what name the output file will be stored with a KEEP instruction. There are other various options you may want, but we leave them up to you.

(Anything likely in this slot for C or Pascal???)**

Step 2. Defining the data structures and the data

Where and how you define your data structures and the actual data depends upon the language you're using. In any case, the data must be defined somewhere; we'll leave the housekeeping decisions for you. Remember, though, to place all text that is to be displayed in a local data area so that it can easily be changed for international markets. (*****can we make other recommendations across-the-board or specific recommendations for Assembler, C, and Pascal?*****)

The structures you define will, of course, depend upon your program. This example program defines the following:

- The graphics mode.
- The menu colors.
- The basic structure of an event record.
- The basic structure of a rectangle.
- The basic structure of a window.
- The coordinates for the windows.
- The event codes returned by the Event Manager.
- Various control structures.
- Zero page data.

Another option is to place the data structures in separate modules, which can then be retrieved through the appropriate language's USING or INCLUDE facility.

Step 3. Starting up the Tool Locator

As the name implies, the Tool Locator is the tool which does the work of finding all of the Cortland tools. It figures, then, that the application must start up the Tool Locator before it makes any other tool calls. The call is simple; it is simply

TLStartup

without any parameters. Once the Tool Locator is up and running, you can begin starting up the other tools you need.

Step 4. Starting up the Memory Manager

The "Memory Manager", for the Cortland, is the tool set which does the housekeeping of assigning and cleaning memory. Because all of the other tools ask the Memory Manager for any space they need, the Memory Manager must be active before all other tools. The call to start the Memory Manager is

MMStartup

without any input parameters. The call returns the User ID for this execution of the application, which other Managers and Tools need to reference in order to get memory space.

Step 5. Starting up the Miscellaneous Tools

The next tool you need to start is the collection of tool sets and managers known as the Miscellaneous Tools. Don't be misled by the name; this set of routines is crucial to the success of the event-driven application. Once again, the call to start up the tool is simply

MTStartup

Step 6. Loading other tools

Now that the Tool Locator, Memory Manager, and Miscellaneous Tools are in place, it's time to load all other tool sets your application will use. To simplify things, and to ensure that all RAM-based tools are loaded from disk, it is best to load all tools at this time. The reloading of the Tool Locator, Memory Manager, and Miscellaneous Tools doesn't hurt anything, and also gives you the opportunity to ensure that the correct minimum version of all of those tools is present. Loading all tool sets also saves you the trouble of determining which tool sets are in ROM and which in RAM.

The loading is accomplished by the LoadTools call, which needs as input a pointer to a tool table listing the total number of tool sets and the number of each tool set needed. The numbers of the tool sets are given in table 3-1:

Table 3-1
Tool Set Numbers

Tool set number	Tool set name
1	Tool Locator
2	Memory Manager
3	Miscellaneous Tools
4	QuickDraw II
5	Desk Manager
6	Event Manager
7	Scheduler
8	Sound Manager
9	Apple DeskTop Bus Tool Set
10	SANE
11	Integer Math Tool Set
12	Text Tool Set
13	Reserved for Apple Use
14	Window Manager
15	Menu Manager
16	Control Manager
17	Loader
18	High-Level Printer Driver
19	Low-Level Printer Driver
20	Line Edit
21	Dialog Manager
22	Scrap Manager

Important Any RAM-based tools must be located in the TOOLS subdirectory of the SYSTEM directory (for a complete list of the files necessary on an application disk, refer to Appendix D).

Step 7. Requesting direct page space for the tools

Some of the tools, particularly QuickDraw II and the Event Manager, require some **direct page** space. Your application has to reserve that space before it starts up any tools which use it.

Direct page is the Cortland's improvement on the 8-bit Apple II's "zero page", where the first page (256 bytes) of memory (starting at location \$0000 in bank \$00) was used extensively to save time and space. On the Cortland, the direct page can actually start at any location in bank \$00, and can be theoretically any size up to 64K bytes. However, the direct page still provides the same advantage of increasing performance, essentially by providing each manager with its own "zero page". See the discussion of direct page in Chapter 2.

To reserve the space, you call the Memory Manager routine `NewHandle`. `NewHandle` needs as inputs the following information:

- the size of the memory block to reserve
- the User ID of the program requesting the space (the User ID was provided by the Memory Manager in Step 4)
- the attributes of the block; that is, whether it is fixed, page aligned, fixed bank, etc.— for direct page, the block must be fixed, fixed bank, and locked
- the location where the block will begin

The call returns the **handle** of the direct page. As discussed in the Memory Manager section in Chapter 2 (and more fully discussed in the *Cortland Toolbox Reference* description of the Memory Manager), a handle is a pointer to a pointer.

That handle is used in the next steps to assign the zero page space for QuickDraw II and the Event Manager.

Step 8. Starting up QuickDraw II

QuickDraw II is the tool set responsible for manipulating graphics on the Cortland. Many of the other tool sets use QuickDraw calls to draw their graphics, particularly those controlling the desktop interface, such as the Menu and Window Managers. Therefore, QuickDraw II must be started up before those other tools.

First, dereference the handle and store the associated pointer (the block is locked, so the pointer is better). To start QuickDrawII, you call the `QDStartup` routine and provide the following inputs:

- the starting location for QuickDraw II's zero page (it needs two consecutive pages of direct page space)
- the Master Scan Line Byte, which controls the basic properties of the lines that will appear on the screen, such as the resolution and the color table for the scan line
- the maximum width in bytes of the largest pixel map that will be drawn (a zero equals the entire screen)
- the User ID of the program requesting the space (the User ID was provided by the Memory Manager in Step 4)

The basic structure of the tools is now in place; in fact, the information up to this point is so generic you may wish to place it in a single module. You could then either access that module from all of your applications or copy and modify it slightly when necessary. However, the next step takes you directly into the event-driven, desktop-interface world, where there is still more generic information that you might include into such a common module.

Step 9. Starting up the Event Manager

The Event Manager provides the basic support for event-driven applications by monitoring:

- the user's actions, such as those involving the mouse and keyboard
- the actions taken by other Managers, such as the Window and Control Managers

As discussed in the "Philosophy of event-driven applications" in this chapter, a typical event-driven application decides what to do next by asking the Event Manager for the next event and then responding appropriately to that event.

The Event Manager has to be started up before the rest of the desktop tools can be used. To start the manager, call the EMStartup routine and provide the following inputs:

- the starting location for the Event Manager' zero page (it needs one page of direct page space)
- the maximum number of events that the Event Queue can hold (zero uses default of 20; maximum is 3639)
- the borders for the mouse or cursor, called the clamp values
- the User ID of the application (the User ID was provided by the Memory Manager in Step 4)

Now that the Event Manager has been started, the rest of the desktop interface tools can be started, beginning with the Window Manager.

Step 10. Starting up the Window Manager

The Window Manager keeps track of the application's windows, and needs to be started for all event-driven applications which use the desktop interface. To start the Window Manager, call the `WindStartup` routine and provide the User ID of the application as input.

Step 11. Starting up the Menu Manager

The Menu Manager supports the menu bar at the top of the screen, an integral part of the desktop interface. To start the Menu Manager, call the `MenuStartup` routine and provide the following inputs:

- the User ID of the application (the User ID was provided by the Memory Manager in Step 4)
- the starting location for the Menu Manager's zero page (it needs one page of direct page space)

Up to this point, the information is almost entirely standard for all event-driven applications. Thus, the code could be kept in a common file and reused for any application.

Step 12. Starting up other tool sets

At this point, you now start up any other tool sets your application will need. There is no prescribed order, although it's nice conceptually to start with the tools your applications will use most often. Note that the remaining tool sets require different inputs when an application starts them up:

- a few, such as the Control Manager, require some direct page space and the User ID (as provided by the memory manager)
- others, like the Sound Manager, require only some direct page space
- still others, like the Integer Math Tool Set, do not require any inputs

What each tool set requires as input to the startup call is documented in the *Cortland Toolbox Reference: Volume 1 and Volume 2*. There is also a minimum number of tool sets required to support desk accessories.

The list of minimum tool sets needed for desk accessories is given in Table 7-1 in Chapter 7.

The example application starts up the Control Manager and the Integer Math Tools.

Step 13. Setting up environment for main event loop

The application is now almost ready to start its own work. If there are any other flags or pointers that need to be cleared, they should be done now (*****Jim's application clears the zero page. Anything else?*****)

Step 14. Setting up the system menu bar

First, it's a good idea to set up the system menu bar. Now is when the tool calls start to be really convenient, as they chain together to do many of the background tasks required for an event-driven application.

To set up the menu bar, start with the tool call `NewMenu`, providing the following input:

- the "normal" menu colors; that is, the colors of the menu bar and text when it is not being selected
- the "selected" menu colors; that is, the colors of the menu bar and text when it is being selected
- a pointer to the data for the menu

The data for the menu should contain the title for each menu and the items listed under each menu. For more information about menu strings, see to the Menu Manager chapter in Volume 1 of the *Cortland Toolbox Reference*.

The call returns a handle to the data for the menu, which can then be used by the tool call `InsertMenu` to place the data in the list of menus. `InsertMenu` needs the following input:

- the pointer to the menu string
- the place in the list of menus to insert the menu (zero to insert the string at the front of the list)

Your application can then calculate the appropriate height of the menu bar with a `FixMenuBar` call, which returns a height based on the tallest font being used. The menu bar can then actually be drawn with a `DrawMenuBar` call.

The example application then allocates windows and storage space for each window (*****what can we say about this?*****).

Step 15. Beginning the main event loop

You're now ready to have the application start its main work doing nothing—or rather, sitting around in the event loop waiting for an event to happen so that it can be handled. Most of the special personality of your application will be built in this module, so we're more limited in what we can prescribe for the module. We'll give you some general guidelines, though, and indicate some of the capabilities of the event loop by exploring the example application.

The first thing the application should check for is whether its time to quit. If it isn't, the application checks for the next event.

When an event does happen, the application gets the event and passes it to the Window Manager routine called `TaskMaster`. `TaskMaster` essentially filters out the events which affect the structure of windows, such as a click in the Zoom, Go Away, and Grow boxes. `TaskMaster` can automatically handle those events, so your application doesn't need to deal with them. To use `TaskMaster`, you provide the following inputs:

- the event mask, used to call the Event Manager routine `GetNextEvent`
- a pointer to the extended task event record that `TaskMaster` uses

If `TaskMaster` can't handle the event, it passes the event code back to the application, where the application must deal with it. For example, if the user selects a menu item, the application must find out which item was selected and take action based on the item. When the action is finished, the application returns to the main loop to wait for some other event to happen.

Step 16. Handling application-specific events

We've now reached the point where your application does its work. As an introduction to what kind of work an event-driven application can do, we present two of the choices from the menu of the example application.

The first example is the opening of the first QuickDraw window. To reach this window, the user has made a selection on the menu...**(**to what level of detail should we go in this section???)**

The second example is **(**hopefully, there will be time to include in the example program a module that changes the information in the content of the window?)**

Step 17. Shutting down the application

When it's time to quit the application, there is a series of steps which ensure a graceful exit. They are as follows:

1. Turn off all tool sets (except the three listed below) by using the ShutDown call from each tool set. This is another good place for a common subroutine.
2. Call the Miscellaneous Tool Set ShutDown routine (with no parameters).
3. Provide the User ID to the Memory Manager ShutDown call. Call the Tool Locator ShutDown routine (with no parameters).
4. Use the ProDOS QUIT call to leave the application. The most common situation will be to give the QUIT call without any parameters, which will shut the application down and return to the program in control "above" it (typically a finder or launcher of some sort). Other possibilities are available; refer to the ProDOS 16 manual for more information.

Preparing an event-driven algorithm

Now that you've seen an example event-driven application, it should be clear that the creative part of your programming task will be handling the various kinds of events you define in your application. In this section, we provide some background information and some practical hints to laying out this kind of application.

An event-driven application is...

When the Macintosh introduced event-driven programming, Mac programmers may have completed this section heading with the words "trouble", "impossible", or perhaps something unprintable. The programmers were not used to the chaotic world of reality where users actually interrupt one action to perform another. To ease you over that hurdle, we present in this section several alternative concepts that might help you learn to write event-driven applications.

None of the following concepts is the "correct" way to think about event-driven applications. All of them are merely attempts to get you to grasp the essence of this type of programming.

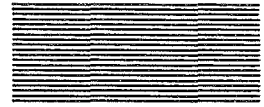
You might think of an event-driven application as:

- An endless loop waiting for an event to happen so that the loop can dispose of the event and go back to waiting.
- Interrupted by user action, so that every event is sort of a mini-interrupt.
- Causing a series of event filters until the end of the chain is found, at which point the kind of event remaining is precisely known.
- Anarchy, not hierarchy.

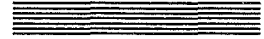
(Any other ideas???)**

Some practical hints

(Tips and tricks on how to analyze a budding application for its event-driven possibilities? Sort of a practical application of the philosophy stated at the beginning of the chapter? I would need a brainstorming session with experienced application programmers for this sort of thing**)**



Chapter 4



Displaying in Color

The color possibilities

The Cortland was designed to be a colorful character. The video display hardware has many color capabilities; as usual, though, we insulate you from the hardware by providing tool calls to manipulate the color. To understand the way QuickDraw manipulates it, you need to understand some background concepts.

Each physical horizontal line on the Cortland screen is called a **scan line**, and each scan line is controlled independently by a scan line control byte (an **SCB**). The SCB controls, among other things, the resolution and the **color table** for the line.

Each color table has a palette of sixteen colors associated with it. The colors in the palette are associated with a color number, which assigns the color's order in the table. The standard palette for the default color table (table zero) for 320 mode is illustrated in Table 4-1:

Table 4-1
Standard palette, table zero - 320 mode

Color number	Default color	Master color value	
0	Black	0 0 0	Opposite of White
1	Dark Gray	7 7 7	
2	Brown	8 4 1	
3	Purple	7 2 C	
4	Blue	0 0 F	
5	Dark Green	0 8 0	
6	Orange	F 7 0	
7	Red	D 0 0	
8	Flesh	F A 9	
9	Yellow	F F 0	
10	Green	0 E 0	
11	Light Blue	4 D F	
12	Lilac	D A F	
13	Periwinkle Blue	7 8 F	
14	Light Gray	C C C	
15	White	F F F	Opposite of Black

The "master color value" specifies the red-green-blue values that blend to make up the color. More information is given in the "Changing a Color in a Color Table" section below.

The standard palette was selected because of its flexibility and appearance; we recommend that you use it at first until you are used to the concept. You can then begin experimenting by resetting individual colors in the palette, or even replacing the color table with an entirely new one.

There is a total of 16 palettes available at any one time. Since the color table is controlled independently for each scan line, it seems that the table could be changed for each line, which could result in 256 colors on the screen at once (16 colors in a palette times 16 palettes). In theory, that's true, but in practice we recommend that you use only one color table for applications that use windows. Remember, the scan line controls a physical line on the screen, not the "relative" line in a window. If a window gets moved to another part of the screen, the effects on the color would be at worst unpredictable and at best difficult to control.

If the entire screen is under control (*****presumably such an application should not allow partial windows???*****), then more complicated things can be done with the SCB's and the color tables. You'll have to study the QuickDraw II chapter in Volume 2 of the *Cortland Toolbox Reference* for that information.

The color for 640 mode is more difficult to handle; in order to take advantage of 640 mode's special capabilities, you'll have to understand more about the video display than we wish to discuss in this book. We recommend that you experiment with color in 320 mode until you understand the principles. In this chapter, we deal only with 320 mode. For further information refer to the *Cortland Hardware Reference* and the QuickDraw II chapter in Volume 2 of the *Cortland Toolbox Reference*. (*****should we say more in this forum???? I realize it will have to be added to the QuickDraw discussion*****)

❖ *Note:* The Cortland colors default to black and white. Thus, you can easily use 640 mode for high-resolution drawing without even worrying about the color.

Drawing in color

If you specify the default color table when you fire up QuickDraw II and then simply use a QuickDraw II call to draw on the screen, it will draw black lines (or whatever) on a white background. To change the color of the line that is being drawn, or the screen background, you use the QuickDraw tool calls dealing with pen patterns.

For example, using the default color table, you could make the pen draw in the color red by calling the QuickDraw II routine `SetSolidPenPat` and specifying color number 7. Similarly, other QuickDraw calls can set the color of the background or (*****any other things worth mentioning???? *****)

Any of the colors in the current color table can be selected as the color to draw with or as the background

Modifying the colors

Changing a color in a color table

Probably the first thing you'll want to change about a color table is the substitution of one color for another. You might, for example, want to have four shades of green on the palette in order to paint a subtle natural landscape.

To change a color, you use the QuickDraw II call `SetColorEntry`, which sets the value of a specified color number in a specified color table. For example, assume that you want to change the color orange in the default table to a light green. You would specify to a `SetColorEntry` call that you wanted to change table zero (the number of the default table), color number 6 (the number of the color orange) to the new color value. Of course, you also have to specify the new color value, which is taken from a two-byte value as follows:

(picture from hardware reference)

Since higher hexadecimal numbers mean darker colors, and we wanted a light green, you would set the color value to "0C0", which changes color 6 in table zero to that color.

While we have modified table zero to contain a different color (and could continue to do so until table zero was completely changed), you can also use this method to create an entirely new color table—table 1 for instance. Of course, to do so would take sixteen `SetColorEntry` calls. There is a more efficient way, at least in terms of calls. You can also use the QuickDraw II call `SetColorTable` to set a whole table at once to the values stored at a certain range of bytes in the video buffer. However, that's a bit hardware-oriented for the purposes of our discussion, so we'll let you explore that technique on your own. **(**is this an appropriate place to end the discussion???)**

Refer to QuickDraw II in Volume 2 of the Corland Toolbox Reference for details.

Swapping whole color tables

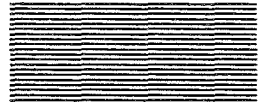
After you get used to setting individual colors in the default color table, you may want to create your own customized palette in an entirely new color table. Using the `SetColorEntry` or `SetColorTable` techniques described above, you can set the colors in the table to any of the 4096 possible colors. You then tell the Cortland to use the new table with the `GetColorTable` call, which retrieves the values from the color table; for example, you could swap out table zero with the values from table 1, etc.

You should also be aware that you could change the colors on the screen by building the new table and then using the Scan Line Control byte (SCB) to tell the line to use the new color table.


Text colors

You determine the color of text displaying on the screen by using the QuickDraw II calls `SetForeColor` and `SetBackColor`. Once again, you specify the color number to the call, which sets the foreground (the "characters") and the background colors according to the corresponding color in the current color table. (**Anything else to mention??**)





Chapter 5



Dealing With Files

For complete information on the ProDOS 16 calls which affect files, refer to the *Cortland ProDOS 16 Reference*.

About file handling - ProDOS 16

When your application needs to store or retrieve information from a disk or hard disk, you will need to deal with disk files. These are in the province of ProDOS 16, which, among other duties, is in control of all disk files. In this chapter, we discuss the basic processes your source code must go through in order to access disk files. We don't give you the details of how to make the ProDOS calls; instead, we introduce you to them and indicate what they do.

Creating Files

Your application places a file on a disk by using the ProDOS 16 CREATE call. When you create a file, you assign to it the following properties:

- A **pathname**. A ProDOS 16 pathname is a series of filenames, each preceded by a slash (/). This pathname is a unique path by which the file can be identified and accessed. The first filename in a pathname is the name of a volume directory. Successive filenames indicate the path, from the volume directory to the file, that ProDOS 16 must follow to find a particular file. The maximum length for a pathname is 64 characters, including slashes.

This pathname must place the file within an existing directory.

- An **access byte**. The value of this byte determines whether or not the file can be written to, read from, destroyed, or renamed.
- A **file type**. This byte indicates to other programs the type of information to be stored in the file. It does not affect, in any way, the contents of the file.
- A **storage type**. This byte determines the physical format of the file on the disk..

Opening Files

Before you can read information from or write information to a file, you must use the OPEN call to open the file for access. When you open a file you specify it by pathname. The pathname you give must indicate a previously created file; the file must be on a disk mounted in a disk drive.

The OPEN call returns a reference number and a buffer location to be used for transferring data to and from the file. All subsequent references to the open file must use its reference number. The file remains open until you use the CLOSE call.

When you open a file, some of the file's characteristics are placed into a region of memory called a **file control block**. Several of these characteristics—the location in memory of the file's buffer, a pointer to the end of the file (the **EOF**), and a pointer to the current position in the file (the file's **MARK**)—are accessible to system programs via ProDOS 16 calls, and may be changed while the file is open.

Be aware of the differences between a file on the disk and an open file in memory. Although some of the file's characteristics and some of its data may be in memory at any given time, the file itself still resides on the disk. This allows ProDOS 16 to manipulate files that are much larger than the computer's memory capacity. As a program writes to the file and changes its characteristics, new data and characteristics are written to the disk.

Reading and Writing Files

READ and WRITE calls to ProDOS 16 transfer data between memory and a file. For both calls, the system program must specify three things:

- The reference number of the file (assigned when the file was opened).
- The location in memory of a buffer that contains, or is to contain, the transferred data. Note that this cannot be the same buffer whose location was returned when the file was opened.
- The number of bytes to be transferred.

When the request has been carried out, ProDOS 16 passes back to the program the number of bytes that it actually transferred.

Closing and Flushing Files

When you finish reading from or writing to a file, you must use the CLOSE call to close the file. CLOSE writes any unwritten data from the file's I/O buffer to the file, and it updates the file's size in the directory, if necessary. Then it frees the buffer space for other uses and releases the file's reference number. To access the file again, you have to reopen it.

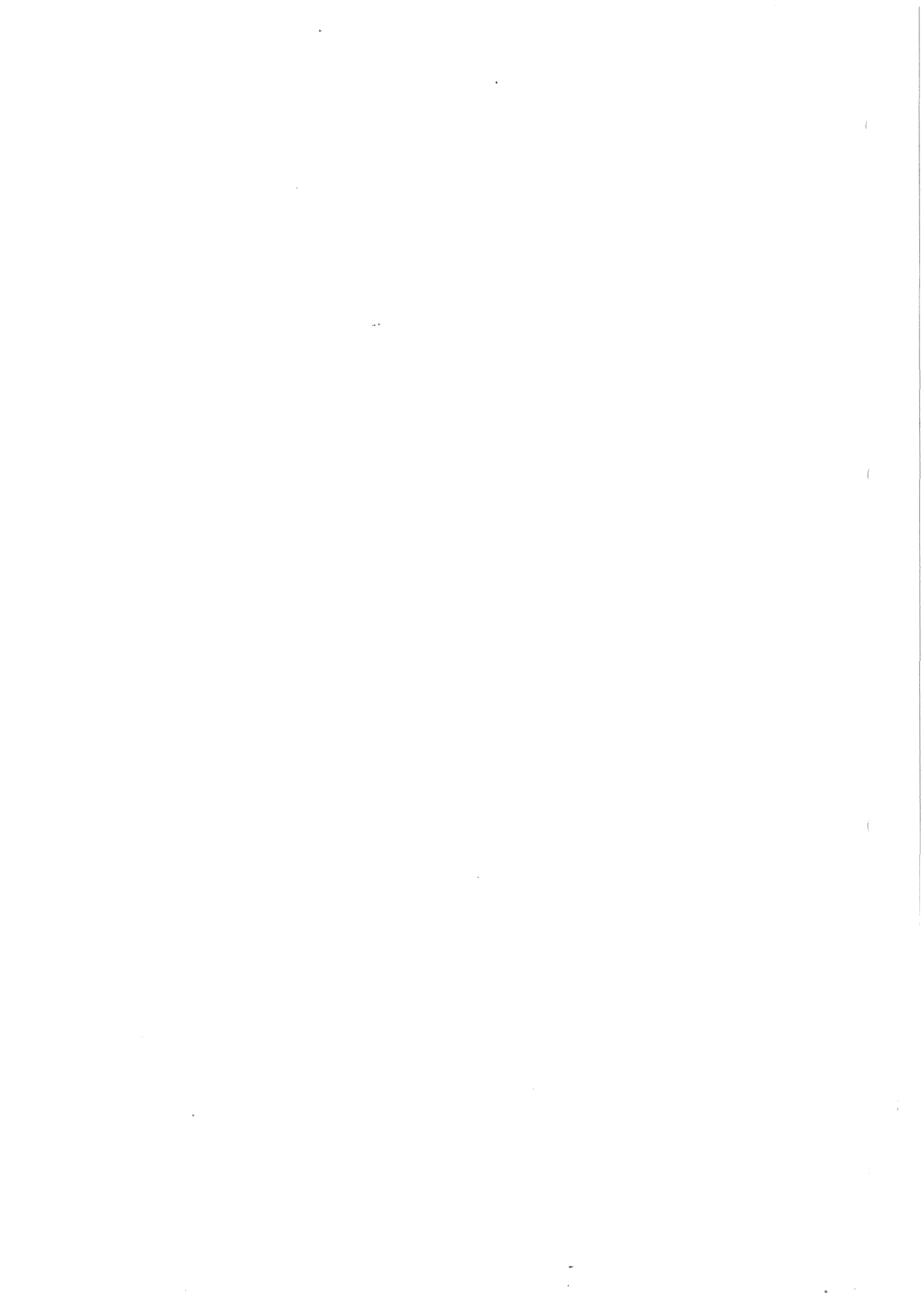
Information in the file's directory, such as the file's size, is normally updated only when the file is closed. If the user presses Control-Reset (typically halting the current program) while a file is open, data written to the file since it was opened could be lost, and the integrity of the disk could be damaged. This can be prevented by using the FLUSH call.


FLUSH, like CLOSE, writes any unwritten data from the file's I/O buffer to the file, and updates the file's size in the directory. However, it keeps the file's buffer space and reference number active, and allows continued access to the file. In other words, the file stays open. If the user presses Control-Reset while an open but flushed file is in memory, there is no loss of data and no damage to the disk.

Both the CLOSE and FLUSH calls can close or flush all files or specific groups of files.

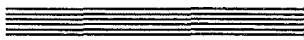
Presenting the standard file interface

The Standard File Operations tool set allows you to present the user with a standard way of dealing with files. The tool calls present a standard dialog box whenever a file is to be opened or saved. Your application needs to make only those tool calls to ensure the consistent user interface (*****details to be added later, when the File Operations tools settle down*****)





Appendix D



Application Disk Files

Table D-1 lists the application disk files and structure necessary to support a desktop, event-driven application.

Table D-1
Desktop Application Disk Files

Directory/File
PRODOS
SYSTEM
P16
LOADER
START
LIBS?
TOOLS/
(and all the RAM-based tool sets needed to support the application)
FONTS/
DESK.ACCS/
SYSTEM.SETUP/
TOOL.SETUP



Appendix E



Pointers to Other Cortland Books

In this appendix, we supply some references by specific topic to chapters in the other Cortland books. We hope this list will help you find specific topics quickly. (*****Writer's note: this section is intended as a catch-all for any topics that are too large or too complicated for the scope of this book. Please contribute!*****)

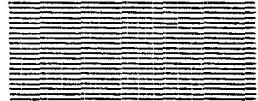
Managing devices: How to manage block devices can be found in the *ProDOS 16 Reference* in the chapter "Adding Routines to ProDOS 16". You can read and write blocks and name devices, but the information is normally only useful if you are writing such things as disk fixers. The file calls normally control input and output from and to the disk.

Character devices are not really relevant to the desktop environment, since that kind of input/output is normally handled by tool calls such as Line Edit, QuickDraw II, and the Event Manager.

Handling firmware interrupts: These can be found in the *Cortland Firmware Reference* in the chapter "System Interrupts".

Installing interrupt handlers: Look under the vector initialization calls in the "Miscellaneous Tools " chapter in Volume 1 of the *Cortland Toolbox Reference*.

Accessing the sound capabilities of the Cortland: Look under the "Sound Manager" chapter in Volume 2 of the *Cortland Toolbox Reference*. Some of the RAM-based sound tools are sold as a separate package and have their own book.



Chapter 6



Coding Static and Dynamic Segments

Introducing static and dynamic segments

A segment is simply a module of an application. Static segments are loaded at program execution time, and are not unloaded during execution; dynamic segments are loaded and unloaded during program execution as needed. The loading and unloading is transparent to the application.

This dynamic or static quality of a segment is assigned at link time. We're not going to tell you how to assign those qualities in this book; instead, we're going to give you the picture of how you'll have to write your code in order for it to execute properly as static or dynamic.

For complete information on linking segments, see the *Cortland Programmer's Workshop Reference*.

Coding static segments

Actually, the title of this section is slightly misleading; there's nothing extra to do to code a static segment, save the general requirement of not starting the segment at a particular address.

If a program is one static segment, the static segment will be loaded where the Memory Manager wants it (as long as you don't specify an origin at linking time). The program will simply remain there until it terminates. For small applications and desk accessories, one static segment is often all that's necessary. However, if your program is beginning to strain the limits of memory, or you wish to decrease the time it takes to load and run, you might want to use the Cortland's capability to load dynamic segments.

Coding dynamic segments

The System Loader for the Cortland has the capability to load segments of an application while the application is running. This allows you to decrease the initial load time of your application by loading only the "permanent" part of the application as static segments. Less-used code or data can be kept in dynamic segments which will be loaded when the program requests them.

Such dynamic segments must as usual not contain absolute addresses, so that the segment is relocatable. Most importantly, dynamic segments must be subroutines which are jumped to with a JSL or equivalent (*****what is the C and Pascal equivalent of a JSL???**).

Once a dynamic segment has been automatically loaded, it will stay in memory until your application unloads it. At that point, the segment is marked as purgeable. If the Memory Manager doesn't need more space before the next request for the segment, the segment won't need to be reloaded and can be accessed quickly.

If the Memory Manager does need more memory than it can obtain by compaction, it will purge the segment. The next time the segment is asked for, the System Loader will have to load it again.

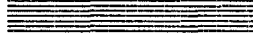
Automatically-loaded dynamic segments are usually used for data segments or for code segments that are accessed infrequently. (*****any more tips???**)

If your program wants to do a little more work and load dynamic segments manually, it can make direct calls to the System Loader. That process is out of the scope of this book, but is detailed in the System Loader chapters of the *ProDOS 16 Reference*. (*****or should we discuss it briefly here???**)

- ❖ *For Macintosh programmers:* The capability to manually load dynamic data segments can be extended to load Mac-like "resources" if desired.



Chapter 7



Writing a Cortland Desk Accessory

The different styles of desk accessories

A **desk accessory** is a "mini-application" than can run at the same time as another Cortland application. The Cortland supports two different kinds of desk accessories; **Classic Desk Accessories** (CDA's) and **New Desk Accessories** (NDA's).

Classic Desk Accessories are designed to execute in a non-desktop environment. In essence, the CDA interrupts the application and gets full control of the Cortland.

New Desk Accessories, on the other hand, are designed to execute in a desktop environment. As such, they operate in a window and are subject to the same rules as an event-driven application. They are not stand-alone applications, however, since they rely upon applications already having started up the Cortland tools.

Neither type of desk accessory has a lot of extra programming overhead apart from the actual task the accessory performs. Both types depend heavily for support upon the Cortland tool called the Desk Manager. In this chapter, we don't discuss the support the Desk Manager gives, but rather we concentrate on what form the Desk Accessory has to take.

For full details on the Desk Manager and its desk accessory support, see Volume 1 of the *Cortland Toolbox Reference*.

Writing Classic Desk Accessories

When a Classic Desk Accessory gets control from the Desk Manager, the processor is in full native mode. Since the Desk Manager has already saved the necessary parts of the old state, the CDA can concern itself solely with its own work.

The basic procedure of a CDA is to:

1. Initialize for new state. Remember, the Desk Manager has already saved the old state when the CDA gets control.

2. Do the actual work of the CDA. Like all Cortland applications, a CDA should ask the Memory Manager for any space that it needs. In addition, the CDA must not cut the stack back further back than it is when it gets control.
3. After the work of the accessory is finished, it must return to the Desk Manager with an RTL or its equivalent. The Desk Manager then automatically restores the old state and returns to the desk accessory menu.

In order for a CDA to be found by the Cortland, it must have a file type of \$B9 (assigned when linking) and be placed in the DESK.ACCS subdirectory of the SYSTEM directory. The accessory must start an identification section which specifies the name of the CDA and a pointer to the start of the code. The exact specifications for the identification section are listed under the Desk Manager chapter in the first volume of the *Cortland Toolbox Reference*. (**Or does anyone think we should list that structure here??***)

Writing New Desk Accessories

All New Desk Accessories are loaded from the disk at boot time. When a NDA gets control from the Desk Manager, the processor is in full native mode. The NDA can assume that the following tools shown in Table X.x. have already been loaded and initialized.

Table X-X
Tools available to New Desk Accessories

Tool name
Tool Locator
Memory Manager
Miscellaneous Tools
QuickDraw
Event Manager
Window Manager (**List not yet settled**)
Menu Manager
Line Edit
Control Manager
Dialog Manager
Scrap Manager
High- and Low-Level Printer Drivers

The basic task of an NDA is to:

1. Save important global values like the GrafPort (*****final list needed when available*****)
2. Monitor the events filtered through to it by TaskMaster, the Event Manager, and some other system events. Based upon the event it receives, the desk accessory must take action.

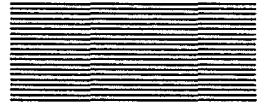
Note that a desk accessory cannot obtain any direct page space, but must use the stack. (*****Otherwise, does it ask the Memory Manager for other space***??? Steve Glass is thinking about this one***)**

3. When the accessory is closed, it must restore the global values and return to the Desk Manager with an RTL or its equivalent.


In order for an NDA to be found by the Cortland, it must have a file type of \$B8 (assigned when linking) and be placed in the DESK.ACCS subdirectory of the SYSTEM directory.

An NDA must start an identification section which specifies the pointers to the four routines, how often it gets run codes, what events it wants, and what the text is that appears as the menu item. The exact specifications for the identification section are listed under the Desk Manager chapter in Volume 1 of the *Cortland Toolbox Reference*. (*****Or does anyone think we should list that structure here?*****)





Chapter 8



Writing Shell Applications

Shell environments

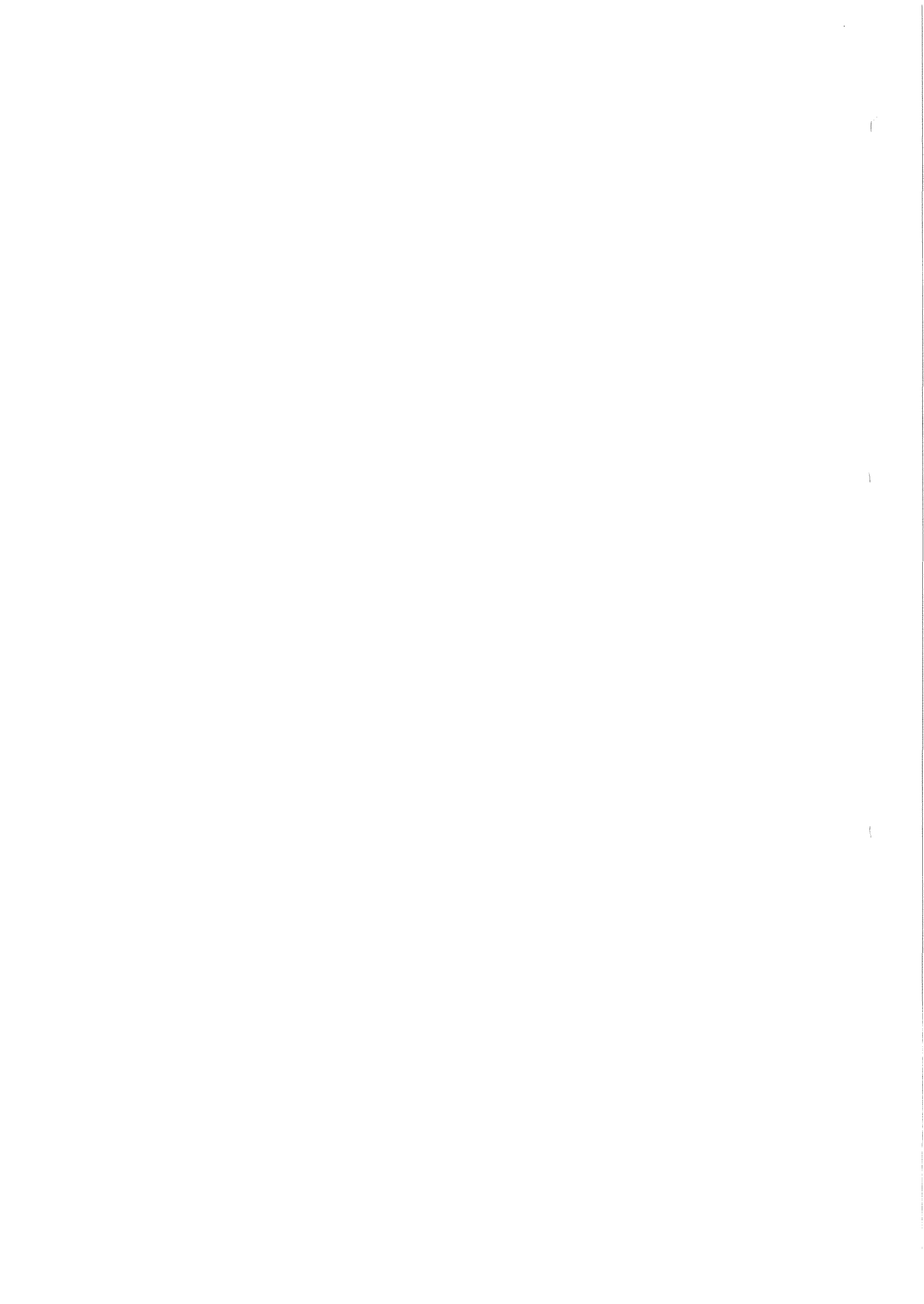
(I have put this chapter here as a conversation piece. Do we want such a chapter, or should it be folded in with the other issues in Appendix E??? Is there enough material to warrant its inclusion, or is it too complex or too uncommon for this book's level????)***

Writing top-level shells

(basic approach - if designed to run under a shell, the application can count on the resources of the shell being there. CPW example of such a shell.)

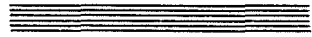
Writing programs to run under shells

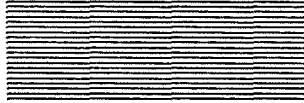
(as above, count on resources being there)



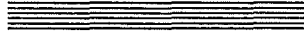


Appendixes





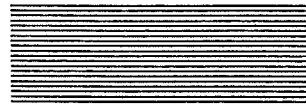
Appendix A



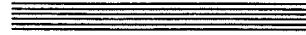
Assembly Language Source Code - Event-Driven Example

(to be supplied**)**



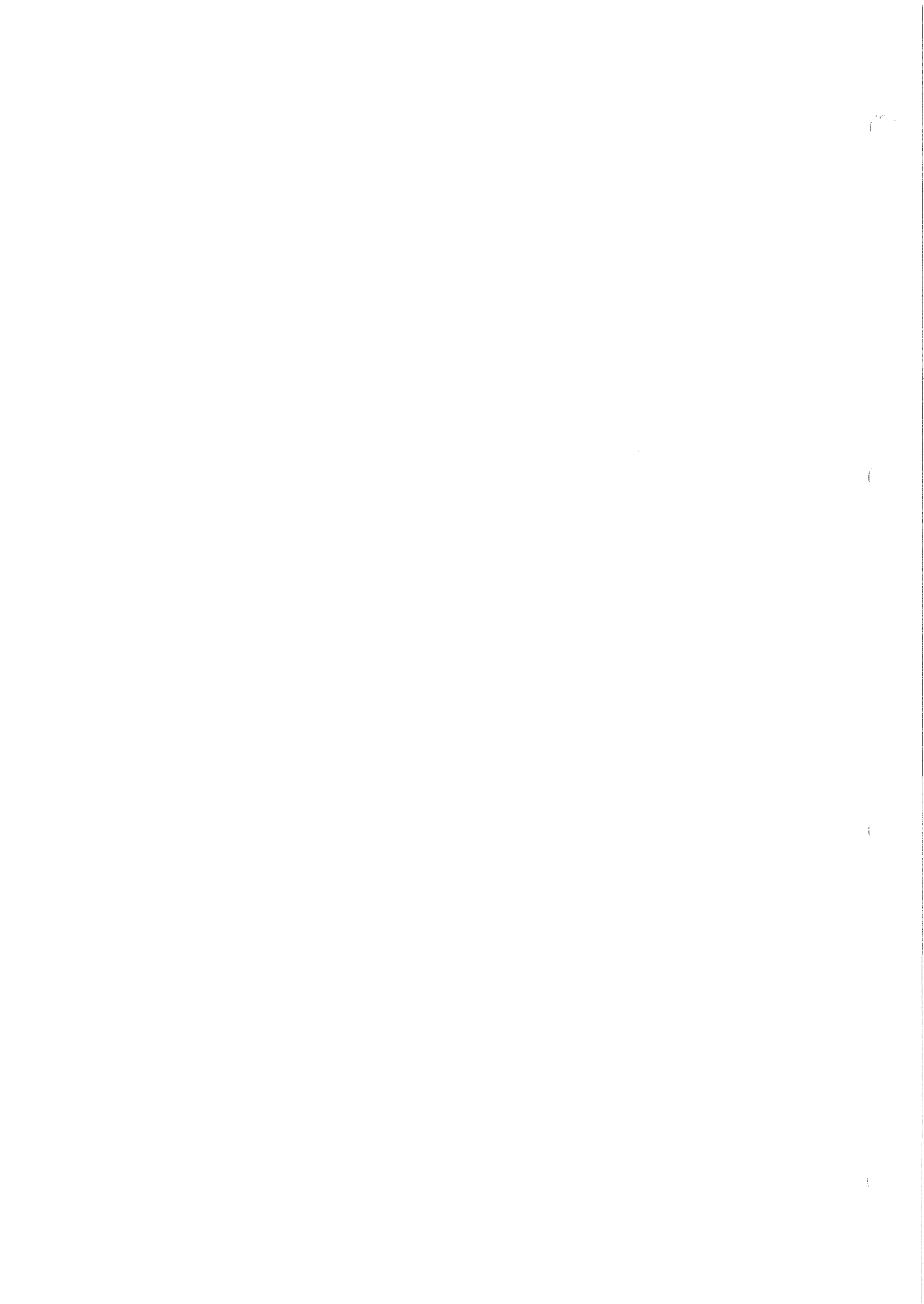


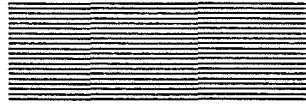
Appendix B



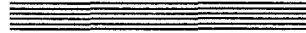
Pascal Language Source Code - Event-Driven Example

(to be supplied**)**





Appendix C



**C Language Source Code -
Event-Driven Example**

(to be supplied**)**

