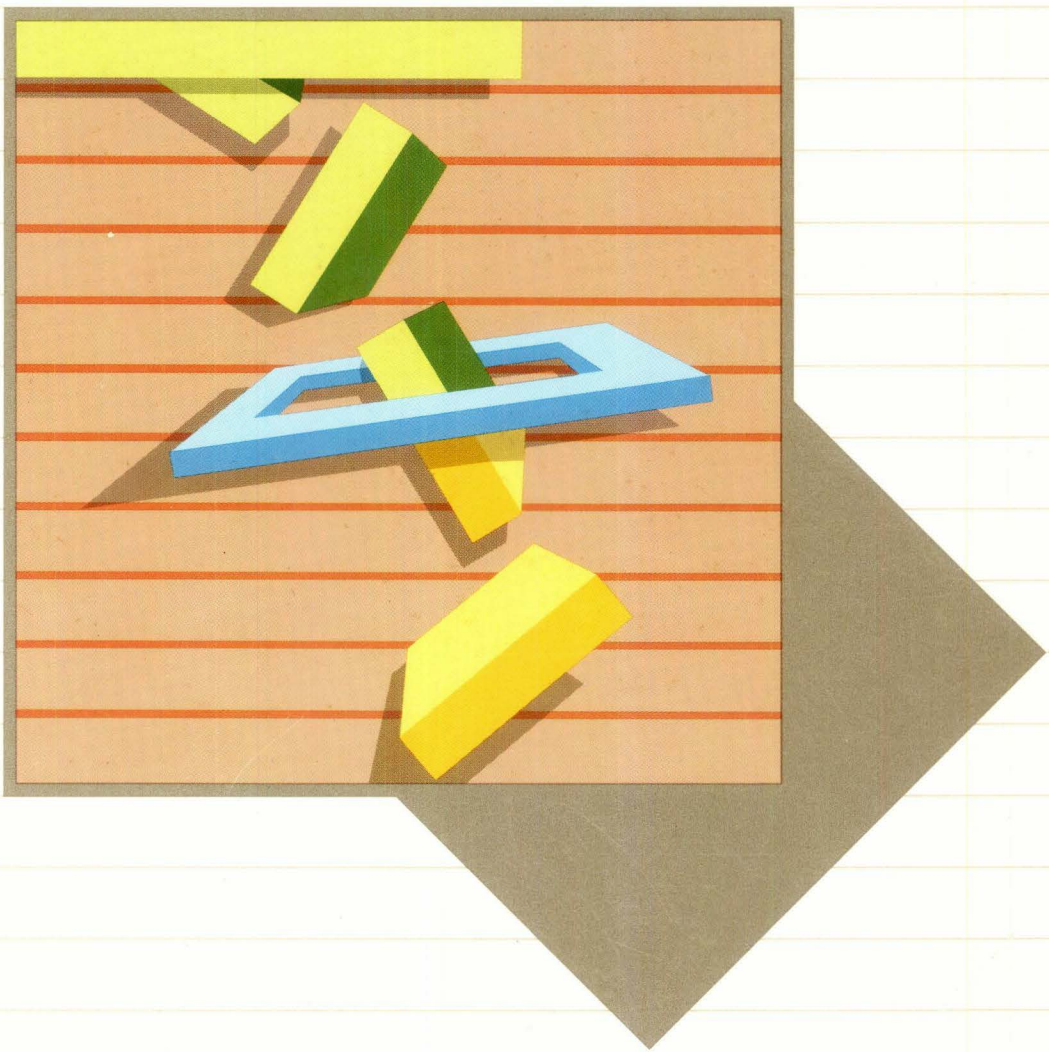
Apple *IIe*

Design Guidelines



Customer Satisfaction

If you discover physical defects in the manuals distributed with an Apple product or in the media on which a software product is distributed, Apple will replace the documentation or media at no charge to you during the 90-day period after you purchased the product.

In addition, if Apple releases a corrective update to a software product during the 90-day period after you purchased the software, Apple will replace the applicable diskettes and documentation with the revised version at no charge to you during the six months after the date of purchase.

In some countries the replacement period may be different; check with your authorized Apple dealer. Return any item to be replaced with proof of purchase to Apple or an authorized Apple dealer.

Limitation on Warranties and Liability

Even though Apple has tested the software described in this manual and reviewed its contents, neither Apple nor its software suppliers make any warranty or representation, either express or implied, with respect to this manual or to the software described in this manual, their quality, performance, merchantability, or fitness for any particular purpose. As a result, this software and manual are sold "as is", and you the purchaser are assuming the entire risk as to their quality and performance. In no event will Apple or its software suppliers be liable for direct, indirect, incidental, or consequential damages resulting from any defect in the software or manual, even if they have been advised of the possibility of such damages. In particular, they shall have no liability for any programs or data stored in or used with Apple products, including the costs of recovering or reproducing these programs or data. Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you.

Copyright

Product Revisions

Apple cannot guarantee that you will receive notice of a revision to the software described in this manual, even if you have returned a registration card received with the product. You should periodically check with your authorized Apple Dealer.

20525 Mariani Avenue
Cupertino, California 95014

Simultaneously published in the U.S.A and Canada.

Reorder Apple Product Number A2F2116

Part I Written by Joe Meyers of Apple PCS Publications Department

Part II Written by Bruce Tognazzini of Apple PCS Application Software Department

Contents

About This Guide

This guide is divided into two parts. Part I contains recommendations to software, firmware and hardware designers who want their products to work smoothly with the Apple IIe, as well as the Apple II and II Plus. These recommendations pertain to the interface between Apple II Series computers and the products that are to work with them.

Part II pertains to the interface between software products and their human users. The recommendations in this section of the guide apply to designers of software for Apple *III*s as well as Apple IIs. The user interface guidelines derive from the experience of countless Apple II and *III* users, as observed by more than a dozen computer and teaching professionals. These guidelines should make it easier for both programmers and users to create and benefit from the tools that Apple computers put at their disposal.

Part I
Product Design Guidelines

Introduction

This section contains guidelines for designers of software, peripheral card firmware, and hardware intended for use with the Apple IIe. Also refer to the sections of the Apple IIe Reference Manual (Apple Product Number A2L2005) pertinent to the product you are designing.

Software

Most of the software guidelines pertain to four new features of the Apple IIe:

1. The new RESET features and their implications.
2. The Apple IIe 80-column text cards and the need to check for the presence of such a card and turn it on and off at appropriate times.
3. The OPEN-APPLE and SOLID-APPLE keys and the fact that they are electrically connected to the pushbuttons of hand controls #0 and #1, respectively.
4. The new firmware, including enhanced Monitor program functions, expanded keyboard ROM map with lowercase characters, and twofold video display ROM map for primary and alternate character set display.

New RESET Features

The Apple IIe has all 64K of its memory in RAM. A reset now affects the contents of what used to be the "language card" area of main memory.

1. Do not require the use of the RESET key during program operation unless you are not concerned that the bank-switched RAM (former language card addresses) will be switched out.
2. Have BASIC or assembly language programs start up using the OPEN-APPLE CONTROL-RESET sequence rather than PR#s (slot s for the startup disk drive). This recommendation is related to the requirements of the new Apple 80-column text cards.

New Apple 80-Column Text Cards

These guidelines apply to BASIC and assembly language programs and their calls to Monitor service routines.

1. Have any greeting program's first action be to determine if an 80-column text card is in the system (see Peripheral Card Firmware section below for identification methods).

2. Make sure that an 80-column card is installed before attempting to turn it on; otherwise, unpredictable system conditions may result.
3. To turn on the Apple IIe 80-column firmware, use PR#3 or the equivalent. The Apple 80-column firmware is associated with slot 3 for compatibility with the Apple Pascal Operating System.
4. Do not use PR#0 to turn off the 80-column firmware. To turn it off, issue a CONTROL-U (NAK character; decimal code 21).
5. Never have a program issue a PR#0 or IN#0 while the 80-column firmware is active.
6. Before sending output to devices other than the video display, issue a CONTROL-L (Form Feed character, decimal code 12) to clear the screen, then a CONTROL-U to turn off the 80-column firmware.
7. If the 80-column firmware is active, look at location 49152 ($C000) directly to check for a keypress. If you use the BASIC GET command or the Monitor KEYIN routine, each ESC keypress will be executed before its modifying escape code can be retrieved.
8. If the 80-column firmware is active, a program should not attempt to display flashing lowercase characters; they are not available in the alternate character set.
9. If your software turns on the 80-column firmware, be sure it turns it off before ending.

The Pascal Operating System automatically checks for the presence of an 80-column text card in slot 3 or the AUX CONNECTOR slot, and turns on the 80-column firmware if such a card is present.

Hand Controls and the APPLE Keys

The new OPEN-APPLE key is connected to the pushbutton of hand control #0, and the new SOLID-APPLE key is connected to the pushbutton of hand control #1. Therefore, do not have a program check for the absence of hand controls by checking whether both pushbuttons have been pressed. Instead, have the program wait for a count of 512 (twice the normal count) and see if the hand control timer has timed out. If not, no hand controls are connected.

New Apple IIe Firmware

The Apple IIe Monitor has been carefully rewritten to maintain all the same entry points as those published in the original Apple II Reference Manual. (The same entry points are, of course, also listed in the Apple IIe Reference Manual.) At the same time, the Monitor screen-

handling routines have been changed to accomodate the requirements of 80-column display.

The keyboard ROM map now features lowercase characters as well as several characters not directly available on the Apple II and II Plus keyboard.

To adapt software to these new features, follow these guidelines:

1. Either avoid performing checksums on the resident firmware or be prepared to accept the checksum outcome of each model of Apple II that the software will run on.
2. Make sure that the program is designed to recognize lowercase characters or to convert them to uppercase as necessary.
3. Make sure the program reacts appropriately to the alternate single-quote character (decimal code 96) now on the keyboard, as well as the more commonly used single quote character (decimal code 39) that has always been present on Apple II keyboards.
4. Programs that compensated for the absence of certain characters (for example, \ or |) do not need to do so for the Apple IIe.
5. If a program uses Monitor input/output routines, it should not use the 80-column software switch at location 49165 ($C00D).
6. BASIC programs that use 80-column firmware should POKE location 36 with tab locations rather than attempting to do "comma tabbing."
7. Any program that uses 80-column firmware cannot also display flashing characters. Flashing characters are not available in the alternate character set, which is the set the 80-column firmware uses.

Peripheral Card Firmware

Any peripheral card that is to work on the Apple IIe should have firmware that takes into account the protocols of BASIC, Pascal 1.0, and Pascal 1.1. Pascal 1.1 protocols were purposely made flexible enough to meet the requirements of future versions of Pascal, extending the useful life of peripheral card firmware.

These protocols are not unique to the Apple IIe, but rather are published here to make it easier for peripheral firmware designers to find all requirements in one place.

BASIC Protocol

The BASIC protocol is very simple; it requires that three entry points be found at fixed locations for a card in slot s:

Address	Contains
$Cs00	initialization routine entry point
$Cs05	input routine entry point
$Cs07	output routine entry point

Pascal 1.0 Protocol

There are also three entry points for firmware cards under the Pascal 1.0 protocol:

Address	Contains
$C800	initialization routine entry point
$C84D	read routine entry point
$C9AA	write routine entry point

New peripheral cards can be "accepted" into the Pascal 1.0 system by using these entry points, as well as having the values $38 at location $Cs05 and $18 at $Cs07 (see Device ID discussion below).

Pascal 1.1 Protocol

Pascal 1.1 has a more flexible setup and supports more input/output functions than BASIC or Pascal 1.0. It makes indirect calls to the firmware in a (new) peripheral card through addresses in a branch table in the card's firmware. It also has facilities for uniquely identifying new peripheral I/O devices.

The I/O routine entry point branch table is located near the beginning of the $Cs00 address space (s being the slot number where the peripheral card is installed). This space was chosen instead of the $C800 space, since under BASIC protocol the $Cs00 space is required, while the $C800 space is optional.

The branch table locations that Pascal 1.1 uses are:

Address	Contains
$Cs0D	initialization routine offset (required)
$Cs0E	read routine offset (required)
#Cs0F	write routine offset (required)
$Cs10	status routine offset (required)
$Cs11	$00 if optional offsets follow; non-zero if not
$Cs12	control routine offset (optional)
$Cs13	interrupt handling routine offset (optional)

Notice that $Cs11 contains $00 only if the control and interrupt handling routines are supported by the firmware. Apple II Pascal 1.0 and 1.1 do not support control and interrupt requests, but such requests may be implemented in future versions of the Pascal BIOS and other future Apple II operating systems.

Here are the entry point addresses, and the contents of the 6502 registers on entry to and on exit from Pascal 1.1 I/O routines:

Addr.	Offset for	X Register	Y Register	A Register
$Cs0D	Initialization			
	On entry	$Cs	$s0	
	On exit	error code	(unchanged)	(unchanged)
$Cs0E	Read			
	On entry	$Cs	$s0	
	On exit	error code	(unchanged)	character read
$Cs0F	Write			
	On entry	$Cs	$s0	char. to write
	On exit	error code	(unchanged)	(unchanged)
	Status			$Cs10
	On entry	$Cs	$s0	request (0 or 1)
	On exit	error code	(changed)	(unchanged)

Notes: Request code 0 means, "Are you ready to accept output?" Request code 1 means, "Do you have input ready?" On exit, the reply to the status request is in the carry bit: carry clear means "No"; carry set means "Yes."

Pascal 1.1 uses four firmware bytes to identify the peripheral card. Both the identifying bytes and the branch table are near the beginning of the $Cs00 ROM space. The identifiers are listed in the following table.

Address	Value
$Cs05	$38 (standard BASIC requirement)
$Cs07	$18 (standard BASIC requirement)
$Cs0B	$01 (generic signature of firmware cards)
$Cs0C	$ci (device signature; see below)

The first digit, c, of the device signature byte identifies the device class.

Digit	Class
$0	reserved
$1	printer
$2	joystick or other X-Y input device
$3	serial or parallel I/O card
$4	modem
$5	sound or speech device
$6	clock
$7	mass storage device
$8	80-column card
$9	network or bus interface
$A	special purpose (none of the above)
$B-F	reserved for future expansion

The second digit, i, of the device signature byte is a unique identifier for the card, assigned by Apple Technical Support. For example, the Apple IIe 80-Column Text Card has a device signature of $88.

Although version 1.1 of Pascal ignores the device signature, applications programs can use them to identify specific devices.

Hardware

The Apple IIe Reference Manual specifies the overall physical and electrical characteristics of peripheral cards disigned for use with the Apple IIe computer. In addition to these requirements, some detailed guidelines apply:

1. To maintain a consistent installation procedure as well as avoid interference with adjacent cards, always design cards so their component sides face away from the power supply case.
2. Avoid designs that require connection to chip sockets on the main circuit board, as future revisions to the board may make such cards obsolete.
3. Do not require that a card be installed in slot 3 if its intended application can involve a text or video card in the AUX CONNECTOR slot.
4. Cards should not dissipate more than the amount of power specified in Chapter 7 of the Apple IIe Reference Manual.
5. Cables should use 9-pin or 25-pin "DB" style connectors. The four 19-pin openings (1 through 4) on the back panel are reserved for use with disk drives.
6. Internal cables should preferably connect to the keyboard end of the card. This gives the user more freedom in selecting a slot to install the card. It also alleviates strain and bending on the cable.
7. Cards that have firmware on them should be identifiable according to the protocols outlined in the Firmware section preceding this.

Part II
User Interface Guidelines

Introduction

The following guidelines and comments have been written for a diverse audience.

As a professional buyer and sellers of software, you can gain an insight into the elements that will make a program most useable to your customers. While this document is aimed primarily at program designers, you can pick up a "feel" for interface design. Is your difficulty in using a program because the designer failed to make it "friendly," or is it only because you lack specific experience in the subject of the program? Are the customers who will be buying the program from you well versed in that subject area? If not, they will have the same trouble you are having.

As a novice programmer, you will find not only specific information on how to implement certain guidelines, but a fair amount of philosophy of design that should be of help in areas of design not covered.

The expert program designer may skim past what has become the obvious—don't just give error numbers instead of meaningful error messages—to find standard layouts and key-function definitions for inputs, menus, command structures, and instruction pages.

You can explore additional BASIC basics of human interface design for the Apple in

Apple Backpack, Humanized Programming in BASIC, by Scot Kamins, Ph.D., published by Byte Books of McGraw-Hill.

There are two primary functions of a good human interface design: make the product easy to learn, and make it easy to use. We all know that our customers can learn to use our programs faster if they look and act like other programs with which the customer is already familiar.

When the Apple II and Apple *III* series computers first came on the market, software developers experimented with a wide variety of interface designs. Some were good, some were bad. All were somewhat hard to learn, because all were unique. As time went on, though, the natural personalities of the keyboards, displays, and computers led to a remarkable similarity of approach to certain basic problems of ease of use.

This document is drawn from applications programs written both within and without Apple. It further relies on human interface research projects inside Apple and standards developed by independent software developers.

We are not in any sense trying to dictate what program designers shall and shall not do within their own programs. Each program has its own needs; each guideline will have its own exceptions. Programs should not be judged by whether they adhere to each specific guideline presented in this manual; they should be judged by whether they are reasonable to learn, functional to use, and whether they get the job done.

What we are offering is a set of interface guidelines and standard key definitions to which we and many independent developers are committed, guidelines and definitions your customers will know and be comfortable with. We are releasing training material that will prepare your customers to use programs that work in the manner described in this document.

Good Human Interfaces: So Often Elusive

The human interface of a program is as vital to its success in the marketplace as its accuracy in performing its task. An otherwise well designed, powerful piece of software or hardware is nearly useless if it is poorly human engineered. As Dr. Frank Gilbreth, the father of time and motion study said: "It is cheaper and more productive to design machines to fit men rather than to force men to fit machines."

Human interface design should come into play from the very beginning. A good design is no mean task: expect to expend a great deal of design and programming effort toward a smooth interface. For most programs with a good human interface, the design of that interface consumes more design time, is more prone to bugs, and is harder to test than any other part.

Apple is in the process of making available a number of packages of routines that should make this task easier. Within the packages are the menu drivers and input routines described in this document. Exact publication times are being set as this document goes to press.

A Planning and Testing Methodology

Planning: the User Profile

In order to properly address the needs of the users, you must first know who they are and what their needs are. Software design should begin with a user-profile study. This study should cover the following three phases:

1. Select the target audience. Begin your human interface design by identifying your target audience. Are you writing for businesspeople or children? Will your audience consist of people relaxing at home or accountants under severe time constraints?

2. Ascertain the level and limitations of their pre-existing knowledge. You should have an understanding of how much the target users know about:

   A. using the Apple computer
   B. the general subject matter your program deals with.

3. Identify their needs. Once you have an understanding of the knowledge and limitations of the users, you can then figure out what types of information and level of support the the program will have to supply.

Figures 1 and 2 are mythical examples of two possible user-profiles for programs that fill the exact same function: tax planning. Even though the task performed, the formulas used, the raw data required are identical, the programs that would result from the two user-profiles might bear little external resemblance.

The "research" quoted in the examples is ficticious—do not start writing a tax planner based on it. (The "case histories" in this document are real; the samples of display and document designs are fictitious.)

Carrying out an early investigation such as the ones above requires a minimum of time and can save you man-months of work later on. The reports need not be works of art; it is only important that every member of the design team have a clear picture of who the audience for this product will be.

Make sure you consider all your users:

In a data-base program recently developed for a computer with large mass storage, no effort was spared in making every section of the program as "friendly" as possible. When a particular task proved somewhat difficult to learn or use, the task was reduced by picking up bits and pieces of it within other tasks. The program slowly drifted toward being consistently somewhat difficult to learn and use.

(Text continued after figures)

Figure 1

Big, Big Business Software Development Corporation

Houston, Texas
"Get the Big, Big Solution to your Little Old Problems"

Professional Tax Planner User Profile Study
July 17, 1983

User: CPA or Public Accountant

Anticipated knowledge of Apple computers: none. (The accountant may well have purchased the system just because of our program.)

Assumed knowledge of subject matter: Expert

Needs:

1. Staged learning curve. Must feel comfortable in a minimum time. Extended features can be picked up later.

2. Facility. Must be able to create and edit scenarios quickly.

3. Clear instructions and error messages. User may have never touched a computer before. Help should be aimed toward problems in the use of the system, rather than explanations of the difference between Short-term and Long-term capital gains.

4. Professional appearance. Accountants will be using this package not only to help their clients, but to impress them. The vocabulary used on the display and in printed reports should be serious and professional. It may contain accounting jargon in areas that will not cause confusion to clients. The accountant must be protected against embarrassing errors (and error messages); he may have a client sitting beside him.

5. Supplementary Features: accountants surveyed currently add or subtract amounts from the "accurate" figures produced by tax planners. Such items as a rough estimate of state tax liability may need to be figured into reports. Provide this facility.

6. Accountants are habitual users of adding machines: they may be expected to do all intermediate calculations on their own adder. No calculator function need be provided.

Figure 2

Aunt Treig's Software and Snowshoe Company
Petersberg, Alaska
"We'll never leave you out in the cold"

Personal Tax Planner User Profile Study
December 21, 1982

User: John Q. Middle to Upper-income Public

Anticipated knowledge of Apple computers: owner with some experience. (Research indicates that tax planning programs do not stimulate an initial computer purchase: people who already own the computer are buying the packages.)

Assumed knowledge of subject matter: None

Needs:

1. The prompting and documentation need to be tutorial: the user must be guided into finding the necessary information to enter into the program, carrying out the kind of explorations with the program that will be most beneficial, and then suggest where the user should go from here.

2. Clear content verification and error messages. "Unlikely" data should be confirmed by user. Help should be aimed toward problems in understanding the subject of taxes.

3. Appearance and use of accounting jargon. Non-professionals will be using this package. The vocabulary used on the display and in printed reports should be non-intimidating and not filled with accounting jargon.

4. User will probably use the program only a few times per year. There must be a minimum learning curve, even at the expense of reduced power and facility. A menu-driven format should certainly be considered as a first cut.

5. The user has to be asked for a lot of pre-computed figures: use an expression-evaluator input to allow them to add, subtract, multiply, and divide during input.

The designers had never considered who their audience was beyond their being "office workers," but when problems showed up during testing, they sat down and did a user profile. What they found was that there would be three separate users of the system:

1. The data-entry persons. These folk would be proficient typists who initially would be expected to enter a great deal of pre-existing information. They might be temporary help, or they might be people who normally performed a different job. Their needs were for an interface that is quick to learn and easy to use.
2. The decision makers. These people would be expected to draw information from the system, both by calling up data on the display and by generating reports. They could be expected to be habitual users of the system: they could handle a long but gentle learning curve that would give them progressively more power.
3. The key operators. These people are the ones who, in real life, read the manuals. They can be expected to spend some time with the system initially and can be expected to learn how to perform the more technical operation and maintenance tasks of the system.

Once the users of the system were identified, once their individual needs were identified, the designers were able to "unbalance" their equally-difficult interface, so that each user had a level of difficulty consistent with their skills and the amount of time they could spend learning the system.

Testing

Once the users have been profiled and a prototype built, it is time to begin testing.

Human interfaces are not made; they evolve. Software designers are simply too close to their product, their computer, and have put up with the most abysmal interfaces themselves for too many years to be able to outguess the naive user. Products must be repeatedly tested on "real people". ("Real people" means the target audience: as soon as you find yourself sitting in a meeting with other computerists, all announcing what users will or will not feel/think/do, you are in trouble. Build the prototype and find out.)

The job of the designer is to do her best to predict the response of the user; the job of the user is to do just the opposite.

Human interface testing is quite different from the kind of exhaustive "boundary condition" testing used to uncover bugs. You should begin testing as early as possible, using drafted friends, relatives, and new employees, to uncover the really big holes in your design. As you get closer to a finished product, try it out on larger groups drawn from the target population.

It is imperative that the designers actually watch people use the program. Do not just send off copies of the program and expect written responses. Get the users and the designers in a quiet room together.

Our testing method is as follows. We set up a room with five to six computer systems. We schedule two to three groups of five to six users at a time to try out the systems (often without their knowing that it is the software rather than the system that we are testing). We have two of the designers in the room. Any fewer, and they miss a lot of what is going on. Any more and the users feel as though there is always someone breathing down their necks.

The initial ground rules are that no questions will be answered, as by the time the formal testing begins, we can supply a draft of the manual. (Usually by the second group, some glaring defects in the interface have shown up, and we have to give them help getting past the stumbling blocks.)

Ninety-five percent of the stumbling blocks are found by watching the body language of the users. Watch for squinting eyes, hunched shoulders, shaking heads, and deep, heart-felt sighs. When a user hits a snag, he will assume it is "on account of he is not too bright": he will not report it; he will hide it. Make notes of each problem and where it occurred. Question the users at the end of the session to explore why the problems occurred. Do not make assumptions about why a user became confused. Ask him. You will often be surprised to learn what the user thought the program was doing at the time he got lost.

We have found that prepared questionnaires handed out at the end of a session are of little value: you will seldom predict the problem areas before testing, and users will lie to spare everyone's feelings. (If you had figured out the problem areas, you would have already fixed them.)

Generally, two or three groups on one occasion is more than sufficient: patterns will emerge almost immediately. You should have at least one more bank of testing after any major revision; as the next example shows, one often jumps out of the frying pan, into the fire.

The True Anecdote:

Herein follows a true anecdote that illustrates how difficult the most simple human interface issue can be, and why thorough testing on real people is so important. (If you dislike true anecdotes, please skip ahead to "Goals.")

As we tune in, the authors of APPLE PRESENTS...APPLE, both of whom pride themselves on clever interface design, have anguished for hours over difficult passages in their program. It was to turn out their guesses were quite accurate in said difficult passages. It was the simplest question of all that caused all the problems...

Problem:	in APPLE PRESENTS...APPLE, an Introduction to the Apple IIe Computer, the training program for teaching fundamentals of using the new Apple IIe computer, find out if the user is working with a color monitor.
User profiles:	new owner, customer in a computer store, or member of a class learning to use Apple computers.
Test user profiles:	customers in a computer store, non-computerists in a classroom environment, friends, and relatives.
First design: Prompt:	A color graphic would be displayed. "Are you using a color TV on the Apple?"
Anticipated problem:	Those who were using a monochrome monitor in a classroom or computer store situation wouldn't know whether the monitor was black and white or was color with the color turned off.
First attempt: Prompt: Failure rate:	A color graphic was displayed. "Is the picture above in color?" 25%
Reason:	As anticipated, but incorrectly overcome, those seeing black and white thought their color might be turned down. They didn't answer the question wrong; they turned around and asked one of the authors whether the monitor in question was color or not. A decision was made that the authors could not be shipped with each disk.

Second attempt:	A smaller graphic with large-letter words in their own vivid colors was substituted: GREEN BLUE ORANGE MAGENTA
Prompt:	"Are the words above in color?"
Failure rate:	color TV users: none black and white monitor users: none green-screen monitor users: 100%
Third attempt:	The graphic remained the same.
Prompt:	"Are the words above in more than one color?"
Failure rate:	color TV users: none black and white monitor users: 20% green-screen monitor users: 50%
Reasons:	the black and white monitor users who answered incorrectly admitted that they did so on purpose. (Our methods for wringing their confessions shall remain proprietary.) 50% of the green-screen folk considered that they were looking at both black and green — two colors — and answered the question accordingly.
Fourth attempt:	Same display of graphic and colored text
Prompt:	"Are the words above in several different colors?"
Failure rate:	color TV users: none black and white monitor users: 20% green-screen monitor users: 25%
Reasons:	By this time, the authors were prepared to supply everyone who bought an Apple with a free color monitor, just so we would not have to ask the question. It turns out that around 20% of the people were not really reading the question. They were responding to: "Are the words above, several different colors?"
Fifth attempt:	Same display of graphic and colored text
Prompt:	"Do the words above appear in several different colors?"
Failure rate:	none.

In case it appears the authors were simply dull fellows, be it known that this was a fully-interactive training program in excess of 100K, and this was the only interface issue that required more than one correction. It clearly exemplifies how even the most careful designers can totally miss when guessing at how users are going to respond.

Had the designers not tested the program, it is probable that dealers would not have used the program in their showrooms, as they would have wearied of telling potential customers that they were/were not using a color TV and that the APPLE PRESENTS... APPLE program was being very stupid to ask the question like that. (Potential customers, of course, wouldn't have fallen for such an explanation: they would have known it was the computer that asked the question, and everyone knows that one should always buy the computer that asks good questions.)

It is vital that programs and manuals be tested early and often with users from the target audience; this testing should be an integral part of any testing plan. This testing seems like a lot of extra effort, but in practice, it really isn't, beyond the mechanical difficulties of getting your equipment and test group together. (Computer stores, colleges, and shopping centers are often good random-testing locations.) The above testing cycles took only four days: the first two days were on-site, using new Apple employees. Only two days of testing required any set-up work at all, and the overall improvement to the product was clearly worth the effort.

Even if the interface had not changed at all, it would have been worth it just to be able to ward off all the self-proclaimed experts with their (day-after-going-to-production) comments of "Boy, I sure wouldn't have done it that way. A lot of people out there are gonna have trouble." What joy to turn to such people and announce with a clear conscience, "Well, we tried it out on 109 people, and they all sailed through with flying colors."

Goals

Simplicity

User interaction should be simple and easy to remember. Spend the necessary time to design a user interface that presents the best tradeoff between alternate design issues.

Once the user has become basically familiar with the human interface, if she guesses at an unknown response, she should be correct 95% of the time.

Simplicity is discussed in detail in the next section, General Program Structure.

> BASIC: When a package contains several programs on a diskette, the programs should always be selectable by the user via a menu display. The user should not have to RUN (or worse, BRUN) individual programs in immediate mode to get the package to function. Each program should end by causing a "menu" program to be run, which should provide the appropriate menu display. The menu program should be a simple program which displays a menu of all programs to which the user will be given direct access, and stores information on the environment in which it runs; for example, it can set any HIMEM: or LOMEM: required by a program on its menu. An example of this is INDEX on THE SHELL GAMES diskette.

Consistency

All programs written for a given computer should have as great a commonality as is practical. The purpose of these guidelines and standards is to achieve a level of consistency across all products designed to run on the Apple, a level that will make learning your product easier, but not be so rigid as to stifle your ability to create the specific human interface best suited to your particular application.

All programs produced by a given software house should perform the same function in the same way. The same key sequence must not do the opposite thing in different products (E=edit, E=eradicate). Many software houses have their own guidelines, guidelines from which we drew in preparing this document. These individual guidelines tend to outline in far greater detail the program "personality" that the software house wants to project. If you have not yet put together such a document, may we suggest you do so. It is a very effective way to eliminate those interface battles that tend to occur about three days before release to production—or three days after.

All software should be self-consistent: menu formats should be identical. If GET or READKEY is used for one input, it should be used for all inputs. If the LEFT-ARROW key deletes characters in one part of the program, it

should delete characters in all parts of the program. If you are working on a large project, be sure to spend enough time in team meetings being sure that everyone is on the same track—all too often the three or four sections of a program end up with an entirely different "feel." At the same time, avoid rigidity: human interfaces must be tested on real people. The agreed-upon interface at the beginning will undoubtedly need changing, once you try it out on real people.

Efficiency

The user should be able to perform the desired task in as little (perceived) time as possible, with the minimum (perceived) complexity. Match the program to the skill level of the user. If you are doing a pricing program for a shopkeeper, do not ask her what her historic elasticity of demand has been without letting her know what it is and giving her the tools to estimate it. (Also, the question may be unnecessary: the fact that you asked it in a similar program you wrote for a Fortune 500 company is no reason to ask it of a shopkeeper.)

Self-teaching

Often there is a trade-off between ease of learning and ease of use. Carefully balance your decisions: if the program is too difficult to learn, salespeople will not learn it and, thus, not sell it. If endless instructions and voluminous menus make it slow and cumbersome to use, people will get frustrated and tell their friends not to buy it.

You will find a number of guidelines devoted to overcoming this problem. Both syntactic and content help should be available at the point at which it is needed; designers are successfully doing that without encumbering the experienced user. See: Help and Menu. Many designers have sucessfully created a multi-tiered interface. See: Novice/expert modes.

Speediness

Actual speed of operations is important, but perceived speed is even more important. It may seem important to conserve keystrokes, but it is more important to conserve "brain strokes" and design the interface so that there is a natural flow. A more important goal is to reduce the amount of unproductive time, which is time spent deciding how to perform the desired task rather than time spent performing the task. This concern should permeate the entire design process.

React to user's input immediately. A user will interpret any delay of more than a few tenths of a second after he has pressed RETURN to mean

that either the program or the user has made an error. If you need to make a computation, first acknowledge that you have accepted the input.

In training or educational software, it is doubly important to react immediately to test questions. The greatest retention of knowledge occurs when response occurs either within one second or not until the end of the entire test. Apparently, waiting five to ten seconds for a correct/not correct judgement is so frustrating that people lose involvement with what is going on.

Minimum Strain on the User's Memory

Programs that are not used literally every single day will be forgotten. Users will not remember command words, the names of their files, nor the fact that you are accepting data not with RETURN, but with CTRL-V. (Violet was the name of your very first computer science teacher.)

Computers are notoriously good at remembering the above type of information. Share it with your user: make sure the information needed is available where and when needed.

Honesty

Do not lie to your users. Do not say, "File loaded" when the file is not loaded, only the name of the file has been "loaded," whatever that means.

General Program Structure

The contemporary microcomputer user may have no previous experience with a program. Therefore, a significant fraction of the programming effort must be dedicated to the creation of an intuitively

natural human interface. The program must, in the simplest way possible, anticipate the user's questions and needs and be prepared to answer and fill them the moment they arise.

There are two important principles being followed in the most successful human interfaces designs.

Keep It Simple

The external appearance of the program is as simple as possible. The user does not get lost within a maze of branches. (You may safely assume that the first-time user has not read the manual.)

The number of screens and menus is kept to a minimum. The ALF™ music editor and VisiCalc™ are excellent examples of this concept.

Displays are kept clean and simple. Questions posed are clear and free of ambiguity.

Fluidity: Movement within the program is easy and fluid. The structure is simple enough to allow the user to move from place to place without becoming confused.

Tools: The user is provided with the necessary tools to work with the program. For example, in a personal finance program, an input requesting annual rent should allow an answer such as 435.00 * 12 or 435.00 X 12, and not expect the user to work out the answer in his or her head. If a file name must be selected from the disk, those file names are either displayed or available for display.

Make It Familiar and Intuitive

Everything the program expects the user to do is either familiar or feels intuitively right. The user should feel comfortable within the program and the program should be supportive, responding to the user's best guess of the right thing to do at any one moment. Everything in this document really leads toward an intuitively correct program. But matters of intuition cannot be thoroughly dissected: the ultimate test lies with whether a new user can feel master of the program within a very short time, or whether he will simply flounder around, trying to figure out what the program wants and why.

Special Key Functions Are Consistent

If a user wishes to complete an input, she knows to always press the RETURN key. ESCAPE always allows the user to escape back whence she came.

Anticipation

The program anticipates as much as possible the needs and questions of the user and is prepared to handle them as they arise.

Intelligence

The program does not ask for unnecessary data, data which can be derived from information already at hand, or data already asked for and received before.

Confirmation

The program tries to prevent catastrophic errors. If the user commands that a 100K text file be deleted, the program should require cognizant confirmation:

Are you sure you want to destroy 5 days' work? Type DESTROY 5 DAYS' WORK to confirm.

If the user commands that a new file be saved under a name already being used for a 100K text file, the program will announce that saving the file under this name will result in the destruction of the original file, and then present a confirmation question similar to the one above if the user says to save the file under the duplicate name anyway.

Tree Structures

The tree structure of a program is designed to feel natural to a user, not the programmer. For example, one could design a program which will both create and play music. Saving created music and loading that music for later playing are highly similar programming tasks and can quite possibly be done using the same basic subroutines. But while it is structurally logical to share code between them, it is intuitively wrong to dump the two options adjacent to each other on a menu. Saving should be grouped with other music-creation options; loading with both creation (for editing), and playing.

Novice/Expert Modes

The first time a user runs a program he has quite different needs from the tenth time he uses it. In the beginning, he needs as much information presented as possible so that he can use the program with a minimum of learning. Later on, with a program used habitually, he wants speed and simplicity. He wants only information pertinent to the specific task being carried out, not a lot of instructions on how to delete an incorrect response.

Most large programs now have some sort of utility/configuration section. The configuration sections often enable the user to select date and time formats, color vs. black and white, and whether or not to have sound. In that

section, you can also enable the user to select a skill level. The rest of the program can then use the resulting flag, when set to expert, to simplify verbiage and perhaps enable more flexible branching within the program—branching that would serve to get the novice into trouble but gives the expert the added flexibility she needs.

The skill level selection could be more sophisticated, perhaps with more than two levels, perhaps based on the type of user. For example, a single tax planner program might better bridge the gap between accountant and Apple owner if the accountant could select, "Expert at taxes, Novice at Apple" and the Apple owner could select "Novice at taxes, Expert at Apple". (The possible combinations and permutations are truly boggling.)

Ending

The user is given a way out of the program. Even if your program is on a copy-protected disk and there really is no way out, give the user an End option and then tell him that he may now insert another disk and press RETURN, or whatever. Users feel positively trapped by programs with seemingly no end; they forget that the power switch solves all.

BASIC programs should also reset and clear Hires screens and revert to normal text condition upon a normal exit in order to prevent interference with other unrelated programs.

The programs should also reset other system parameters to customary settings.

The balance of this document presents more concrete guidelines for specific program areas.

Input

Two major languages native to the Apple II and Apple *III* series computers are BASIC and Pascal. Each has an input routine as part of the language. These input routines are used in many programs, and your customers have become familiar with them. However, they are not particularly well suited to most professional applications. As a result, in the past, each software engineer has created her own routine, usually a variation of one or the other "standard" input routine. The direction of this creative technology has been toward more sophisticated input schemes which allow insertion and deletion of characters. This proliferation of inputs, each with its attendant cursor and special keys, has left the poor user rather bewildered.

Enter The Three Cursors: New Apple owners are being trained to recognize and use three different inputs: BASIC's, Pascal's, and the new insert/delete input. They are being trained that they can tell what input they are faced with by the kind of cursor presented.

BASIC uses the blinking box cursor which overlays a character. Pascal uses the solid box cursor which overlays a character. The new input uses a blinking underline cursor which lies between characters.

Of all the standards being presented, this is the most important: The user should be able to tell the rules of the input from the kind of cursor being displayed. If everyone conforms to the use of the proper cursor for each personality of input (defined further below), Apple users will be relieved of a major source of frustration. It is really hard to concentrate on learning to do ellipsoid analysis of pork belly futures when you can't figure out what key to press to correct a typo.

If you need additional features, then keep the original cursor and enable the original features: make your input scheme a superset of the original. If you need to use an entirely different kind of input scheme, please select a different cursor and train your users to recognize it as yet another entity.

The (Apple II BASIC) Blinking-box Cursor

Computer	Move left	Move right	Insert	Delete	Accept	Cancel
Apple II's	LEFT ARROW	RIGHT ARROW	*	*	RETURN	CONTROL-X
Apple ///'s	LEFT ARROW	RIGHT ARROW	*	*	RETURN	CONTROL-X

* A user can both insert and delete within a BASIC input line using ESCAPE - cursor keys. As a practical matter, however, few other than experienced BASIC programmers can actually do so with any facility, as the screen ceases to reflect what the character-input buffer is actually holding.

The (Pascal) Solid-box Cursor

Computer	Move left or right	Insert	Delete-from-end	Accept	Cancel
Apple II's			LEFT ARROW	RETURN	CONTROL-X
Apple ///'s			LEFT ARROW	RETURN	CONTROL-X

Apple II Series

Keystroke	Editing Operation
Necessary:	
LEFT-ARROW	moves cursor left within input line.
RIGHT-ARROW	moves cursor right within input line.
CTRL-D	deletes character to the left of the cursor - Apple II & II+.
DELETE	deletes character to the left of the cursor - Apple IIe.
RETURN	accepts entire response, regardless of current cursor position.
CTRL-B	has no effect on a display with a single input. On a multiple- input display (see next section), CTRL-B accepts entire response, moving user back to previous input.
Useful if implementation language allows sufficient speed:	
CTRL-X	deletes all characters on the input line.
CTRL-Y	deletes all chars from present cursor position to end of line.
CTRL-R	recalls display of default response. If no default, then it acts the same as CTRL-X.
Optional:	
CTRL-P	Prints the contents of the display on the default printer.

Notes

Because this input is new to Apple II and Apple II+ users, it is particularly inportant that you expressly state on the display that CTRL-D is used to delete characters. (Apple IIe users have been trained how to use this input on the APPLE PRESENTS...APPLE diskette, but a reminder to use the DELETE key would be helpful.)

Typing any printing character will automatically insert that character into the input line at the current cursor position.

Pressing RETURN with the cursor anywhere within the input line will accept the entire input.

Default responses are displayed with the cursor at the end of the response.

RETURN	will accept that response.
LEFT-ARROW	will move cursor back into the default response, enabling the user to edit it.
RIGHT-ARROW	will signal that you wish to append material to the response.
CTRL-D	(Apple II & II+) will delete a single character from the end of a response and signal that you wish to edit the response.

DELETE	(Apple IIe) will delete a single character from the end of the response and signal that you wish to edit the response.

Pressing any other key will clear the default response and begin a new response in its place.

Apple III Series

Keystroke	Editing Operation
Necessary:	
LEFT-ARROW	moves cursor left within input line.
RIGHT-ARROW	moves cursor right within input line.
Either:	
CTRL-SPACE, CTRL-LEFT-ARROW, or DELETE	will delete the character to the left of the cursor.
Either:	
CTRL-RIGHT-ARROW or CTRL-DELETE	will delete the character to the right of the cursor.
RETURN	accepts entire response, regardless of current cursor position.
CTRL-B	has no effect on a display with a single input. On a multiple-input display (see next section), CTRL-RETURN accepts entire response, moving user back to previous input. Programs often use
CTRL-RETURN	in addition to CTRL-B for accept and move back: CTRL-RETURN is indistinguishable from CTRL-M, or CTRL-whatever-character-your-user-has-defined-to-be-in-M's-standard-keyboard-position. Please take this potential risk into account before enabling CTRL-RETURN as well as CTRL-B.
CTRL-E	deletes (erases) all characters on the input line.
CTRL-K	deletes all chars from present cursor position to end of line.
CTRL-U	(Un-do) recalls display of default response. If no default, then it acts the same as CTRL-E.
Optional:	
CTRL-P	Prints the contents of the display on the default printer.

Notes

Typing any printing character will automatically insert that character into the input line at the current cursor position.

Pressing RETURN with the cursor anywhere within the input line will accept the entire input.

Default responses are displayed with the cursor at the end of the response.

RETURN	will accept that response.
LEFT-ARROW	will move cursor back into the default response, enabling the user to edit it.
RIGHT-ARROW	will signal that you wish to append material to the response.
CTRL-SPACE or DELETE	will delete a single character from the end of a response and signal that you wish to edit the response.

Using the New Input on an Apple II or Apple III Series Computer

The program input statement asks the user for information by displaying a verbal prompt. Prompts should terminate in a colon (:) or greater-than sign (>) if a statement, a question-mark (?) if a question. The prompt is followed by 2 spaces on an 80-column display, 1 space on a 40-column display.

A default answer may be displayed, with the cursor following, in which no field length is denoted. If there is no default response offered, or the default is rejected by the user, the program can display a finite input field with a series of periods (standard character set) or "ghost" underlines (hi-res character set). The latter character is essentially a shortened underline with every other dot turned off.

The specification of the number of spaces between the prompt and the input field is quite important: users can become confused as to where their answer begins. If all programs adhere to one space with 40 column displays, 2 spaces with 80 column displays, the users will know whether they have inadvertently typed a leading space or not. (As a separate issue, leading and trailing spaces should be routinely stripped off, unless they are specifically needed.)

Keystroke errors are best trapped immediately: if you are accepting a decimal number, do not accept a letter such as "A" or "B".

Here is the example of the input which is taught on APPLE PRESENTS...APPLE for the Apple IIe:

What is a "drift"?

```
> A whole lot of cattle_
```

(Consider the underline to be blinking — the printer was not able to quite capture the effect.) The problem presented is to change the answer to read:

```
> A herd of cattle
```

To edit the response, the user first moves back to the end of the word "lot," using the LEFT ARROW. It looks like this:

```
> A whole lot of cattl_e

> A whole lot of catt_le

> A whole lot of cat_tle

> A whole lot of ca_ttle

> A whole lot of c_attle

> A whole lot of _cattle

> A whole lot of_ cattle

> A whole lot o_f cattle

> A whole lot _of cattle

> A whole lot_ of cattle
```

The user is next instructed to press the DELETE key several times, until the words "whole lot" have been deleted:

```
> A _ of cattle
```

Next, the user types the word "herd":

```
> A h_ of cattle

> A he_ of cattle

> A her_ of cattle

> A herd_ of cattle
```

Finally, the user can press RETURN to accept the entire response:

```
> A herd of cattle
```

Cursor Movement with no Action Taken

Sometimes programs such as word processors require pure cursor movement with no action taken. The standard keys in such cases are as follows:

Keys for up, down, left, and right motion:

Apple II and Apple II Plus:

```
          I=up
J=left          K=right
        M=down
```

These keys are often prefixed with an ESCAPE.

Apple IIe: the four arrow keys
Apple *III:* the four arrow keys

Keys for vertical, horizontal, and diagonal motion:

Apple II, Apple II Plus, and Apple IIe:

```
U=up,left    I=up    O=up,right
       J=left      K=right
N=down,left  M=down  ,=down,right
```

These keys are often prefixed with an ESCAPE.

Apple *III:*

Full cursor movement on the Apple *III* is done using the numeric keypad:

```
7=up,left     8=up     9=up,right
4=left                 6=right
1=down,left   2=down   3=down,right
```

Keyword Matching During Input: the Disambiguator

In the olden days of computers, one typed a command to a computer onto a punched card. One then walked down the hall to the computer room, left said card, and returned some hours later to find that the command was syntactically incorrect. Later, time-sharing changed all that. Now one could type in the command, then press a single key called

RETURN which would send the command down the hall. Soon (15 seconds or so later) the computer would announce the command was syntactically incorrect.

Many microcomputer programs still wait around until the user has made a thorough fool of himself and pressed RETURN before sending the results "down the hall" to the program unit which does keyword matching. This unnecessary waste of processing time and power is inherent in the built-in input routines of the languages which have been ported over to micros from time-share systems. Since we are generally supplanting those old routines with the new blinking-underline routine, which is accessible and can be taken apart, this waste need not go on.

Keyword matching is used in programs which are command-driven and programs with lists, such as file names, from which a user must select by typing in the name of the selection. Often, the human interface is rather sparse:

Command Word? _.....

File Name? _.....

Enter command and command-object: _.....

or, even more simply, (for the programmer)

_.....

Command-driven programs offer a speed and flexibility not generally attainable in a menu-driven program. They also typically offer a much steeper learning curve—so debilitating a learning curve that salespeople will often avoid selling a command-driven program because of the time and practice required for them to give even a rudimentary demonstration. Result? Lost sales.

In order to type a command or list selection without prompting, the user must learn what words he can type at any one point in the program. This learning problem can be overcome in a straightforward manner by displaying a list of all options then available. Typically, this can be done by creating a display similar to the standard menu format, with the center region devoted to the list of words.

Commodity Analyzer	Belly Processes	Pork Bellies
Commands	Inventory Types:	
display	complete bellies	
graph	partial bellies	
compute	dancing bellies	
buy		
sell		
eat		
delete		

Options: ESCAPE to leave OPEN-APPLE-? for help

As the available options change, so do the options displayed. Thus, the user knows at every point exactly what she can select. (The display of command words could be made optional through the novice/expert flag; variable lists of words, such as file names, should always be visible.)

A second, more subtle learning problem must be handled a different way: Typically, a command word system will allow abreviations: why make the user type in "display dancing bellies" when "di d" is all that is required to make the user's intentions clear? (Display and Delete both start with d. Once the i is added, display is the only possible answer. The only command-object that currently starts with a d is "dancing bellies." Thus, the user's meaning is not ambiguous.

Usually the user has either had to look up abbreviations of command words in the manual, or discover them by trial and error. Words from lists such as current file names have simply had to be typed out completely, with no abbreviations accepted. The Disambiguator changes all that.

The algorithm for figuring out at what point the user has typed enough so that an answer is unique (not ambiguous) is really quite simple: on each keystroke, the list of possible words is scanned for a match-up of as many letters have been typed so far. As soon as only one match can be found, the word has been found and can be completed by the program. In the above example, the sequence would look like this:

Type your instruction and press RETURN.
D_

No unique match is found (both delete and display match) so the input only echos the user's character.

Type your instruction and press RETURN.
DI_splay

User's response is no longer ambiguous: the program supplies the remaining characters in the opposite case from that the user is typing. (An alternate character set can be used when the environment permits, so that the user can type both uppercase and lowercase characters.)

Type your instruction and press RETURN.
DIS_play

The user has not noticed that the computer has made a match, which happens often with a touch-typist. There is no penalty: the new character is echoed and the program now supplies just that portion of the command word still remaining.

Type your instruction and press RETURN.
DISPLAY _

The user has noticed that the answer has been found and has pressed the terminating character, in this case a space. The program completes the word, using the case that the user has been typing and adds the space to the end.

Type your instruction and press RETURN.
DISPLAY D_ancing bellies

As soon as the "D" in "dancing bellies" has been typed, the remainder of the phrase becomes clear.

Type your instruction and press RETURN.
DISPLAY DANCING BELLIES

The user terminates the input with RETURN and the full answer is echoed and acted upon.

The Disambiguator input has been used in a number of programs over the last two years: it has proven to be quite successful. It lets people get used to abbreviations at their own speed, without their having to look up anything or get yelled at by the computer for using the wrong abbreviation, yet it accomplishes this with no penalty to the touch-typist who is just as happy pounding out the entire word. As a fringe benefit, it speeds up the program's response time: when the user presses RETURN, the program already knows what the instruction is and that it is legal.

Pascal programs handling lists of up to thirty words in any one context have been implemented with this routine in Pascal. Pascal programs with very long lists or any BASIC programs need the routine to be implemented in native code.

The Disambiguator algorithm is just an example of the kind of processing that can go on actively during the user's input to make the user's life a little easier. There is no longer any technical reason for programs to stand by while users flounder, waiting to pounce on them when they press RETURN.

"Press RETURN to continue"

The user can control the movement from one display to the next by pressing the RETURN key (or, optionally but consistently, SPACE bar). He is informed by a message such as, "Press the RETURN key to go on to the menu." on the bottom line of the screen. (Delay loops are difficult to judge as to the proper duration, and become somewhat insulting to the intelligence of the user.) The actual prompt message should give some indication as to what will happen next, rather than simply saying "Press RETURN to continue."

The educational software community has pretty much selected SPACE bar instead of RETURN to control movement: children were found to occasionally press RESET by accident on the older Apple II's and Apple II Plusses. Please be consistent in your choice of RETURN or SPACE bar, not only within a given program, but across your complete product line.

Do not tell the user to "press any key." On the Apple II series computers, you cannot read every key by itself: RESET, SHIFT, CONTROL. We have also found in testing that new users, in particular, panic when asked to press any key. Over 80% of them will turn around and say, "but what key should I press?" In questioning them about this response, we discovered that they are quite convinced that even though the prompt implied all keys were OK to press, some could be dangerous. Of course, they were quite right.

While you should not tell them to press any key, you may, in this specific case only, accept more than the key specified. Both RETURN and SPACE bar should be accepted, even though only one is prompted for: users grow used to using one or the other. You may optionally (and only in this specific case of using the key as a switch) want to accept most keys, so that a user striking out for SPACE bar and pressing V by accident will not be penalized. Do not accept ESCAPE instead of RETURN or SPACE bar.

Displays with several input statements:

- Movement from input to input is sequential: the user may move back and forth but not randomly skip around.
- Pressing the RETURN key automatically positions the user at the next input statement.
- Pressing CTRL-B automatically positions the user at the previous input statement. The prior response to the previous input will be displayed as that input's default.
- The last input on the display will normally ask if the user has completed all responses to her satisfaction.
- No input will be accepted without the user explicitly terminating it, usually with RETURN or CTRL-B. The fact that the user has used up all the spaces available in the field should not automatically move the user to the next question.

Errors

Error Trapping

In most situations, user inputs must be checked for validity. Account numbers, employee numbers, and dates are just a few examples of items that should be checked to see if the data requested is on file or plausible. Numeric inputs should be screened for values too small or too large, if extreme values are invalid or potentially damaging to the program. An error message line should be provided in a consistent location toward the bottom of the display.

Many types of errors can be circumvented through software design: If, in testing, you find users repeatedly making the same kind of errors, change the software.

Make your program insensitive to upper/lower case when no distinction is necessary. Be particularly aware on Apple II programs: the new Apple IIe can generate lower-case characters. (Make sure you only

transform characters: many of those obscure punctuation marks are often-used special characters on foreign keyboards.)

Spaces should never be significant. Users look upon a space as a lack of a character, not as a character. Strip leading and trailing spaces, and intervening spaces too, when practical. For example, when prompting the user for the name of an existing file, should the user respond "door bell", first look for a literal match, then strip spaces, so that you can match with the user's original, but now forgotten, "doorbell".

Likewise, do not refer to "the RETURN character", unless you are prepared to deliver an essay. Users steadfastly cling to their belief that RETURN is an action, not a character.

Do not make commands position-dependent. For example, do not set up a stream of "parameters" such that if the user wishes to change the fourth parameter, she must type three commas to signify acceptance of the first three "default" parameters. If the meaning of the above sentence is not immediately clear, you have gotten the point.

When a menu offers a set of choices, or the user is otherwise prompted to respond to a restricted set of options, then the program should recognize only the responses that are valid. Do not offer the user a menu of options, most of which cannot be used. If the user needs to select a file before deciding to Edit, Save, or Delete, let him know. Don't make him go down through a list, getting the same unenlightening message, "Option not currently valid."

Enable only those keys you have informed the user you are enabling. Do not prompt: "Press ESCAPE to end, RETURN to continue..." and fail to announce that SPACE bar will eliminate this afternoon's files. The classic negative example of this is an early Apple text editor with a verified replace option. According to the manual (no instructions were displayed), R meant replace this occurrence, SPACE bar signified do not replace this occurrence. The actual code was such that any character with an ASCII value of 82 (R) or above caused a replacement, and any character with an ASCII value less than 82 caused a skip. Therefore, "[" would replace, "8" would not, "^" would replace, "," would not. Confusion yet reigns over that one.

Having presented the rule, here is the exception: when using SPACE bar or RETURN as a switch ("Press RETURN to continue") you may want to accept all reasonable keys surrounding the intended target, so as to not penalize the user for poor aim.

Error Messages

Error messages should alert the user, identify the problem, and direct the user toward solutions. They should do so with a minimum of disruption: ring the bell once—the whole room doesn't need to know the user has yet again made a fool of himself.

Remove the error message as soon as the user takes proper action to correct the condition. Users will believe you: as long as the message is there, they will continue to correct the problem. It has been shown that attempting to correct a problem that has already been corrected will usually result in a brand new problem.

There are two classes of error messages that either intimidate or infuriate users.

First, the computer-gibberish special:

> Application error #1463

Error messages should not only provide information (in the user's native tongue, not computerese) as to what the error was, but should offer solutions as to what the user can do to correct the situation. A better message might be:

> You have not yet selected the name of the file you want to work from. Please type the name of one of the above files.

Second, the it's-easier-to-flag-an-error-than-correct-it error:

> Data entry error: no comma after Aardvark

Users soon catch on that if the computer/language/program (everything gets blamed) knows for a fact that there should be a comma after Aardvark, that the computer/language/program should supply said comma. Therefore, the computer/language/program is either really stupid or is lying.

Your application should have a designated area of the screen where error messages are displayed. The usual location has come to be lines 23 and 24 of the display, but whether you choose these lines or not, make the location consistent. Long help instructions may require a different "page". Preserve the contents of display as much as possible while providing help, and once help is terminated, restore the context completely.

Defaults

Please do not ever use the word default in a program designed for humans. Default is something the mortgage went into right before the evil banker stole the Widow Parson's house. There is an exhaustive list of substitutes (previous, automatic, standard, etc.) in the Appendix to How to Write a Manual.

Defaults should be declared, not assumed. Undeclared (not displayed) defaults such as pressing RETURN for Yes (or for No?) will cause confusion and anger.

You need not declare ESCAPE every time you enable it: ESCAPE always gets you out of where you are, to where you came from, without causing damage or confusion. As long as you adhere to that benign definition, you may feel free to slip in ESCAPE anywhere.

Displays

Inverse, Flash, Focus

Inverse. There are many recent model home color TVs and older black and white TV's that display inverse mode very poorly. Inverse mode can be used effectively to accent screen material, particularly on the limited 40 column screen of the Apple II and II Plus. It should be used creatively in business software where it is expected the user will have a quality monitor. However, for software aimed for the home market, avoid inverse mode unless the entire screen or several adjacent lines are simultaneously inversed.

Flashing mode should only be used to indicate imminent destruction of data or the program.

Focus. TV sets in particular, as well as many lower-priced monitors, have very poor focus in the extreme corners. Use the corners for familiar character groupings (such as words), rather than clusters of unrelated characters (such as numbers). The human mind can figure out that what looks like "8ASIC FUNCTIONS" is probably "BASIC FUNCTIONS", but will have less luck discerning that what looks like "1500.00" is really "7500.00", an error that could have far-reaching effects. Keep input lines away from the corners for the same reason: the user needs to be able to check each individual character typed.

Vocabulary

Jargon

Avoid computer jargon. A great deal of it has an unrelated emotional charge. (Abort, for example.) The appendix to How to Write a Manual has a comprehensive list of standard terms.

Keycap Names

Whenever possible, call keys by the names printed on them, written out in full. Three of the keys do have a standard abbreviation:

First choice	Second choice	Abbreviation (last choice)
⍰	OPEN-APPLE	OA
◆	SOLID-APPLE	SA
CONTROL	CONTROL	CTRL

The Apple II and Apple II Plus have CTRL printed on the CONTROL key and should therefore be prompted by "CTRL" as a first choice. (However, the Apple IIe has CONTROL spelled out on the CONTROL key.) Foreign-language keyboards have various special symbols for shift, caps lock, tab and return; use the "local" character in each "local" version where possible.

Abbreviations

Use abbreviations only where absolutely necessary or where an abbreviation is better understood than what it stands for, e.g., 8 PM.

Title Pages

Programs should begin with a clear announcement of the title of the program, authorship credits, and a copyright notice (if any).

Help

The user should not be faced with page after page of instructions: experience has proven that people simply will not read them. Rather, supply help as it is needed. One way of doing that is described in the section on menus.

When you try your program out on new users, be sensitive to the times they need fundamental help in using the features of the programs. For example, while you may have a program portion with detailed explanations on why ellipsoid analysis is so effective in figuring hog belly futures, your user may never get there: you may not have provided necessary help in how to enter preliminary data.

The standard help key on the Apple IIe and Apple *III* series computers is either OPEN-APPLE-? or SOLID-APPLE-? (The SHIFT should not be

required: therefore, also accept OPEN-APPLE-/ and SOLID-APPLE-/.)

The standard help key on the Apple II and Apple II Plus, where practical, is a question mark or slash, or else ESCAPE ? or ESCAPE /.

The normal location for help messages is the bottom of the screen. Whether you use this location or another, make your location consistent. When using a separate help screen, clearly title it as a "help" display, and make the transitions as smooth as possible. When help has been given, restore the screen to its original condition.

Menus

Many of the menu guidelines may appear somewhat arbitrary—they are. It appears only happy coincidence that so many developers are making menus so similar. As with many aspects of human interface standards, what is important is that one standard be set so the poor user can get used to one way of doing things. It is not nearly so important what that standard is.

A menu should display all choices legibly, using numbers for menu selection. Research has shown that people will learn a number on an often used menu as quickly as a letter, even though letters on the surface appear more friendly. ("Was that E for Edit or E for Erase?") Mnemonic letter schemes are a nightmare to translate, and well over half of computer users are not touch-typists. For these reasons and others, the vast majority of developers have settled on numbers.

If you do settle on letters, for your own reasons, be sure to be consistent: don't make the user use numbers for one section of a program and letters for another.

All sub-menus should allow the user to move to the next higher menu by pressing the ESCAPE key. The main menu should not have an ESCAPE key option, so that the user can feel confident about leaning on the ESCAPE key to get all the way back to the top level, without worrying about being bounced completely out of the program upon reaching the top level. Each menu should (redundantly) have a last option that will move the user back to the next highest level. The Main Menu's option should end the program.

The following is an example of a menu using the standard menu format:

Commodity Analyser **Futures Menu** **Select Future**

1. Pork Belly Futures
2. Corn Futures
3. Present Futures
4. Crystal Ball Futures
5. Return to the Main Menu

Type your Selection (1-5) and press RETURN: _..

Options: ESCAPE to leave OPEN-APPLE-? for help

The exact number of lines devoted to the three regions is not graven in stone: the real standard being striven for is that there be three regions with solid lines separating them, that these be devoted to titles, choices presented, and instructions. (The Apple II and Apple II Plus can not produce a solid line in text mode; use either their hyphens or their short-underline characters.

The title region can have up to three titles (usually two in forty-column mode). The middle title (or left title, if only 2) should be the name of the menu, and it should contain the word, "menu." Other displays you will use, such as data entry and information, may have a similar format: make sure your user is clearly aware of what he is being asked to do. Similarly, on information screens, do not number itemized lists, asterisk them: otherwise, about 25% of your users will try to type in a "selection".

You may use or not use the other titles as you see fit, but they should have a consistent meaning throughout a given application.

Note that a field length longer than one has been allowed for the number input. A field length of only one character will not give the user enough feedback as to how the input works. Users who have typed the wrong number will often panic, assuming the computer has somehow locked up. By giving them a few extra spaces, they can see from the action on the screen what is going on and deduce what to do about it. Since you will be stripping leading and trailing spaces (won't you?), this extra freedom afforded the user will not affect the program.

The instruction region doubles as the error message region.

One method you can use to enable the user to get descriptions of any option is to have them type the number of the option, followed by the help key (OPEN-APPLE-? on the Apple IIe and Apple *///*, ? on the Apple II and II Plus). The user is able to get extensive information only on those items of interest, without having to wade through masses of information that are not needed.

Routines for doing these menus will be available from Apple: first in Pascal for the Apple *///* in the Screen Manager and in BASIC for the Apple II series within the sample programs (Magic Menu, Disk Menu, etc.) supplied with the Apple IIe Applesoft Tutorial and Reference manuals package. The BASIC version is implemented with the help facility described above.

Keyword Displays

(The following is a repeat of an earlier section. The keyword display is repeated here to make the document useful for reference.)

In order to type a command or list selection without prompting, the user must learn what words he can type at any one point in the program. This learning problem can be overcome in a quite straightforward manner by displaying a list of all options then available. Typically, this can be done by creating a display similar to the standard menu format, with the center region devoted to the list of words:

Commodity Analyser **Belly Processes** **Pork Bellies**

Commands:

- display
- graph
- compute
- buy
- sell
- eat
- delete

Inventory Types:

- complete bellies
- partial bellies
- dancing bellies

Type your instruction and press RETURN: _

Options: ESCAPE to leave OPEN-APPLE-? for help

As the available options change, so do the options displayed. Thus, the user knows at every point exactly what she can select. (The display of command words could be made optional through the novice/expert flag; variable lists of words, such as file names, should always be visible.)

Keep Them Informed When You Are Away

When the computer will be either carrying out computations or accessing the disk for an extended period of time, a message should be left on the screen, instructing the user that the computer will return shortly (this is not the suggested message). Some periodic change in the screen, as a lengthening line of periods (hence periodic) should occur so the user knows the computer has not simply gone into an endless loop somewhere. When the computation period is over, clearly signal it: simply showing up with a blinking cursor over in the corner won't do after a brief, 20-minute pause.