WHAT 90% OF NEW OWNER'S NEED TO KNOW FIRST!

As a new proud owner of the ultimate copy-deprotection device, your first desire, undoubtedly, is to immediately backup one of your important programs. The new software enclosed (Revision 3.1 and up) allows the neophite to do this auomatically. (PLEASE NOTE: a language card or 16k Ram card is needed for this process.) The rest of this large manual is for the other 10% of the owners, those who are experts in machine language, and want to do sophisticated processing with this incredible card. For the majority of you who will probably never want to take the time to develop the expertise to "pack programs", this section is dedicated.

This is really a lot of fun, so relax and enjoy yourself as we step you smoothly through the procedures.

1) With the Apple turned OFF, insert your CRACK-SHOT card into any slot within the Apple. Make sure both flip switches are up. Leave the cover off your Apple, as you will need to get at those flip switches later.

2) ALWAYS, ALWAYS put a write protect tab on your original disk

3) Boot up your protected program.

4) After the program is fully loaded (that is, past the title page, introductions and "press any keys" stuff), then flip the rear switch on the card.

5) A message will appear across the top of the screen: "COPYING- HIT ANY KEY". At this point take out your protected original from the disk drive, and insert a blank disk. The blank does not have to be initalized.

6) Press any key, and the disk drive will whirl, and your program will be saved out onto diskette. See, that just took a few moments and you already have a copy! Now to run the copy easily.

7) Now turn off your Apple for about 10 seconds, and flip back up the rear switch on the CRACK-SHOT card.

8) Make a copy of your CRACK-SHOT software disk with Copya off your Apple System Master. Then insert and boot this copied disk. After the inital messages, type "run menu" at the prompt.

9) Choose Option 8.

10) Choose Option 4, which creates an initalized disk with the proper booting information on it for you. If you only have one disk drive, choose the reconfigure option first on this same menu.

11) Insert blank diskette into drive 2, and follow directions.

12) Now, back at the same menu, choose Option 1, which converts a CRACK-SHOT file (the one you made a few minutes ago from your protected master) into a binary file in position 1. Utilizing this technique, two programs can be stored on the same disk.

13) You will be prompted for a file name to call this new binary file. Usually you use the name of the program itself, but to avoid confusion, you might want to name it "Red" or "D4A7".

14) Now for something a little tricky. The program shows you a list of screen setups and ask you to choose one. These refer to the type of screen used by the program. For most hi-res programs, choose number 4- "Hires2".

15) The disk drives will now turn on, and the binary file with the appropriate startup file will be created on drive 2. Turn off the Apple when it is down, and reboot the new disk.

16) Select file 1, and the program will boot and off you go!

When you are ready to put the second program on the same disk, do all of the above steps again except for section 10, and use the disk you already have in place of the blank disk in section 11. Choose Option 2 in section 12, and when finished, you will now have a full disk. No other programs can be added to this one disk. Be sure to label the disk. It certainly is easy to forget what is on each one.

POSSIBLE TROUBLE SPOTS:

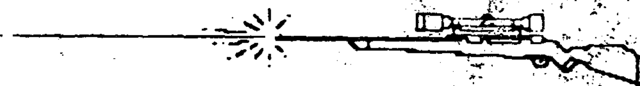
1) The most frequent spot is choosing the right screen for section #14. If your program comes up with a hash hi-res screen, then go back and redo sections 12-16. This time choose Option 2- Hires1. If that does not work use Option 5- Hires1 + Text. Of course for all text or lo-res programs choose the appropriate options.

2) If the procedure bombs out at section 6, then you will need to remove some or all of the extra cards that might be residing in your Apple. The D.C. Hayes Micromodem, Z-80 card, Videx 80-column card, and others can sometimes play havoc with the proper operation of the system. For a few programs, even the 16k ram card will need to be taken out until after section 6, then replaced during section 7 while the Apple is OFF. Never attempt to take out or put in any card while the Apple is on!

3) There are occasions when the copy appears to be made properly, but no matter what is chosen for section 14, the program does not run, or bombs shortly after starting. This is usually caused by the same cards mentioned already in previous section 2, and is remedied by the same procedure given.

There, that's all there is to it! Look Ma, NO PARMS!. If you have any other questions, you might well benefit from our 24-hour, 7 day- a- week Hot Line which is available to all CRACK-SHOT registered owners at a small additional fee. For further information call Pirate's Harbor by modem. Have fun and good luck!

CRACK-SHOT™



COPYRIGHT (C) 1982
PIRATES HARBOR
and
T.R.A.S.H. SYSTEMS

PIRATES HARBOR
P.O. Box 8928
Boston, MA 02114
617-738-5051 MODEM

INDEX

1.0 INTRODUCTION 03

2.0 HOW TO MAKE A COPY.....................05

3.0 HOW TO RUN A COPY......................07

4.0 MEMORY SETUP OF HARDWARE...............13

5.0 EDIT UTILITY...........................15

6.0 PACKER UTILITY.........................19

7.0 COMMAND FILE CREATE....................39

8.0 EXTRA USES FOR CRACK-SHOT CARD.........41

APPENDIX

A. HEX NUMBER SYSTEM......................45

B. CRACK-SHOT TRACK REFERENCE.............47

C. ASSEMBLY LANGUAGE REFERENCE............49

D. TIPS ON COPYING AND PACKING............55

E. PACKING EXAMPLES.......................57

F. TROUBLE SHOOTING.......................61

G. SCHEMATIC..............................63

H. ADDITIONAL INFORMATION.................65

I. PARAMETERS.............................67

The CRACK-SHOT card is guaranteed for 30 days after purchase. All parts and labor are under warranty. If a problem develops in hardware return the card to PIRATES HARBOR for repair.

CRACK-SHOT COPY has been designed as a gaming tool, programmers aid, and as a device to assist you in obtaining legal archival copies of your programs.

CRACK-SHOT should NOT be used for illegal purposes.

PIRATES HARBOR take no responsibility for any actions taken by users of this card.

1.0 INTRODUCTION

CRACK-SHOT is a program backup system designed for quick dependable archival backup of 'TOTAL LOAD' programs on the APPLE II. A single drive system is the only requirement.

The term 'TOTAL LOAD' is used to describe all those programs that, once booted from the original program disk (protected), never need to access that disk again. If the program needs to access a disk under normal DOS format it is still possible. This term covers a majority of games and many hobby and business programs. DOS disks can be formatted or created using the normal DOS 3.3 or 3.2. If a program is protected on a master disk but once running it uses normal DOS then this can be copied using CRACK-SHOT. If the program uses a nonstandard DOS but will initialize the disks for you then this program is also viable for a CRACK-SHOT copy.

CRACK-SHOT does not copy the original disk, rather it copies a program in memory at any time. To execute the CRACK-SHOT copy of a program you need the CRACK-SHOT card or supplied restart utilities. Due to the mechanism of storage the new copied program on disk is in a very simple nonstandard format. This in itself is no major problem to run from DOS 3.3 except for the fact that to make CRACK-SHOT totally effective you need extra scratchpad memory. The extra memory is so that *NO* memory is changed at time of copy. Only three bytes are changed by CRACK-SHOT at time of copy. This makes the CRACK-SHOT system very dependable. CRACK-SHOT copies all 48k ram in 15 seconds. Also copied to disk is the necessary information for restart. To reload and execute the program you need scratchpad memory and the program to reload and restart. The CRACK-SHOT card contains both or a language card can be used to restart.

Utilities are included to transfer your CRACK-SHOT disk copies to smaller files on DOS 3.3 compatible disks. This allows compaction of your copies and the ability to run copies from DOS 3.3 without the CRACK-SHOT card.

This effectively bypasses all the bizarre protection schemes used on formatting disks. The original disk boots up and will execute because it is the original bought copy. Once the program is in memory it can be copied. The current market programs lock out the CTRL-C key and change the reset key. Both of these will not affect CRACK-SHOT operation.

APPLE IS A REGISTERED TRADEMARK OF APPLE COMPUTERS

A nice benefit of CRACK-SHOT is for game players. CRACK-SHOT performs a nice service for players who reach a high level of expertise in a game. No longer do you have to sit through the initial stages of the game which tend to be too slow or boring. With CRACK-SHOT you can save the current state of the game at **ANY** time. This means if you have reached the 10th level you may save it at that level. Then boot that copy and you start at the level you saved it at. Generally the more interesting and faster playing levels may take 5-10 minutes of play to reach. Now you can enter at any level. With the PACKER utilities you can create binary file copies of these higher levels.

Besides being a program backup system CRACK-SHOT provides the user with several nice hardware considerations. The APPLE has an empty 2k block of memory at $C800 TO $CFFF for expansion use. This has now been filled up by the CRACK-SHOT system. For more information on the memory setup see section 3.0.

2.0 HOW TO MAKE A COPY

There are two switches on the CRACK-SHOT card. One switch is called the copy switch and the other is the transparency switch(more later). The copy switch will be used mostly. It is the switch you press to obtain a copy of a program that is running in memory. The copy switch is the switch closest to the red LED. The transparency switch is the switch closest to the user when the card is plugged in.

To make a copy of a program insert CRACK-SHOT in a slot with the copy switch up. (power off). Try the first time with the transparency switch up for enabled. The disk I/O card must be in slot six. To use CRACK-SHOT to copy a program slot zero must be empty (with CRACK-SHOT in any other slot but 6). An alternative to nothing in slot zero is to have CRACK-SHOT in slot zero, either will work. Turn on your APPLE and boot the program you want to copy. When you are ready to copy simply flip the copy switch on CRACK-SHOT down, the led will light and you will receive a warning on the top of the screen that a copy is about to be made. Have a blank disk in the drive at this time. This is a warning because the disk in drive one slot six (required slot&drive) is about to be written on. If the disk is write protected the program will stop. When you hit any key the program starts and 15 seconds later you have a copy.

When the computer gives you the prompt 'COPY MADE' the system will seem to hang. This is due to the mother board roms being locked out. You have two options to regain control. One is to turn power off, then on. The other is to toggle the switch another time or two until the led turns off. The mother board roms are then enabled, if the system does not come back with a prompt hit reset and it will possibly then respond. Possibly because now the memory is setup as at time of copy. If the copied program has set the break reset vectors then there is no telling what will happen. Remember when you hit reset on the APPLE locations $3FE and $3FF are used as a vector to a reset routine. APPLESOFT DOS 3.3 is probably not resident and memory is full of something???.

The transparency switch is to make the card invisible. Some programs will not boot up or start if they see an unknown card in one of the interface slots. The transparency switch is used to hide the presence of the card during bootup. When the program is running put the transparency switch up. When it is down the card is not visible to the computer. The normal position of the switch will be up for enabled. If you run into programs that refuse to boot up with CRACK-SHOT in the computer then use the switch.

INTENTIONALLY LEFT BLANK

3.0 HOW TO RUN A COPY

To run your copied program on a CRACK-SHOT disk you have several options.

A) Execute with the CRACK-SHOT card.

B) Execute with language card program

C) Pack the CRACK-SHOT copy to smaller DOS 3.3 file
and use BRUN from DOS

D) Convert to 48k DOS 3.3 file and run with
language card program

A. Execute with CRACK-SHOT card

To run the copied program you type 'CALL 52224' in basic or 'CC00G' in the monitor. Make sure the transparency switch is up for enabled.

CRACK-SHOT will ask you if you want to execute the program when it is reloaded. If you hit 'N' CRACK-SHOT will load the program and return control to you in the monitor. The memory setup at that time is described at the end of this section. If you hit 'Y' or any key except 'N' CRACK-SHOT will reload the program and execute or restart the program for you.

There is one more menu you see before program execution. This asks you what screen system to start the program in. One minor problem with CRACK-SHOT is that at the time of program capture CRACK-SHOT has no means of determining what screen is currently being viewed. You could be in text page one,two or hires1 etc..... the menu asks which screen to start program execution with. If you make an error simply restart the system and choose a different screen. Most games use hires one or two. Business programs usually use text one but, like some bit copiers, text page two is sometimes used. You might make a note on the copy disk once you know which screen the program uses.

One note on loading. CRACK-SHOT uses an extremely simple write format to disk but it does do checksum computations to detect load errors. Every track is checked. After ten checksum errors on a track the system will stop and tell you the track. If you get repeated checksum errors the problem area can be one of three things; A drive needs adjusting or maintenenc$, or A disk is physically bad. The third possible error is a bad copy to begin with. The first thing to try after repeated errors is the easiest, which is to recopy the program and try to execute that copy.

CRACK-SHOT writes out the copied program to disk when the switch is flipped. The data written to disk is verified at that time to minimize copy errors and make the restart more

reliable.

B) EXECUTE WITH LANGUAGE CARD PROGRAM

On the supplied utilities disk is a program (option 6 of menu) that will load into a language card and execute a CRACK-SHOT copy disk. The language card program is the same as in A above only it loads into the language card and the CRACK-SHOT card is not needed.

C) PACK CRACK-SHOT COPY DISK TO DOS 3.3 BINARY FILE

The PACKER program is described in full. See section 6.0 of this document.

D) CONVERSION OF CRACK-SHOT FORMAT TO DOS 3.3

On the supplied disk there are two programs used in converting the CRACK-SHOT copy disk to a DOS 3.3 16 sector format. The first program does the actual conversion of the CRC$-SHOT disk into a DOS disk. The second program is used to execute the DOS disk.

The CRACK-SHOT disk contains 48k of copied memory. The conversion program will copy all 48k of the CRACK-SHOT disk to an initialized DOS 3.3 disk. There is only room for two copied programs per side of a DOS 3.3 disk. To execute the copied program use the second program supplied.

You cannot use normal DOS commands to execute the copied program because it is 48k long. If you want to use normal DOS commands then use the PACKER program and condense the file down to a maximum of $91 hex.

To execute the program in DOS 3.3 format you need a language card. Option 9 on the CRACK-SHOT master disk will load a binary program into the language card, that program will then run your copied program.

The benefit comes in that you don't need the CRACK-SHOT card in the system, and the DOS disk is copyable by COPYA or any brute force copy method for normal DOS.

CONVERSION PROGRAM OPERATION

Option 8 on the CRACK-SHOT master disk is the conversion program. Run that option and you will receive instructions. The CRACK-SHOT format disk that you wish to convert goes in drive one. An initialized disk goes into drive 2.

The disk in drive 2 should not contain any information besides converted CRACK-SHOT files. The directory is not accessed normally. A CRACK-SHOT DOS 3.3 disk contains 1 or 2 programs stored in 48k long binary files. Only two files can fit on one side. You can think of the files as 1 and 2. Names are stored on the disk for directory information. The disk can

be cataloged by normal DOS and the files stored are shown as locked binary files of length 5 sectors. This is not their true length though.

When operating the CONVERSION program you can hit 'C to show what files are on that disk, if any. To convert put your CRACK-SHOT disk in drive 1, put an initialized disk in drive 2. You can store the copy in either position 1 or 2, you select. Hit 'C to see if another file is already in one ot those positions.

When you hit 1 or 2 the CONVERSION program will ask you for the file name to store on the converted disk. Then the program will prompt you for the screen to display on startup. This screen info is stored with the file and on startup the program will automatically show this screen. The conversion then takes place.

Again, the files stored can be seen by the catalog command from normal DOS but cannot be accessed except by option 9 (described next).

RUN CONVERTED CRACK-SHOT DISK.

This is option 9 on the CRACK-SHOT master disk. Once you have converted a CRACK-SHOT disk to DOS 3.3 format you can execute the disk with this program.

When you run this option the program will ask you which file 1 or 2 to execute. Hitting a 'C will show the files stored on that disk by name. Hitting a 1 or 2 will execute that file. The hires screen will come on during loading but after the program is all in the correct screen will appear and the program will begin.

These converted disks can be copied by COPYA or any brute force copy method that copies tracks $0-$22.

To make a complete stand alone disk transfer files LCBIN and EXLCBIN over to a disk **AFTER** you have converted 1 file. The disk can still hold small files. Make a HELLO program that has one line.....-=>10 print (ctrld)"EXEC EXLCBIN". Then when you boot that disk it will immediately ask you what file to execute ..1 or 2.

The data is stored on tracks :

$03-$0F FOR FILE 1
$11 DIRECTORY
$14-$20 FOR FILE 2

You can load a program into memory but not execute with the CRACK-SHOT card or the language card read (option 6 on utility disk).

The following register storage locations are for the CRACK-SHOT card. If the language card is used instead of $CBxx the data is at $DBxx.

If you choose not to execute the program but only to load the memory will be set up as follows.

PAGES	CONTENTS
\$00-\$01	TRASH, COULD BE ANYTHING
\$02-\$C0	COPIED AND RELOADED PROGRAM
\$C8	PAGE 0 OF COPIED PROGRAM
\$C9	PAGE 1 OF COPIED PROGRAM
\$CA	PAGE 4 OF COPIED PROGRAM

MEMORY	STORED FROM COPIED PROGRAM
\$CBFF	ACCUM A AT TIME OF NMI
\$CBFE	X REG AT TIME OF NMI
\$CBFD	Y REG AT TIME OF COPY
\$CBFC	STACK POINTER AT TIME OF COPY
\$CBFB	PROCESSOR STATUS AT TIME OF COPY
\$CBF9	PROGRAM COUNTER LOW AT TIME OF COPY
\$CBFA	PROGRAM COUNTER HIGH AT TIME OF COPY

This data is useful for program analysis. Please note though that if you choose *NOT* to execute the program at reload the action of reentry to the monitor may destroy some vital locations used by the copied program. As shown above pages 0,1,4 are up in the ram above \$C800. But the monitor does use screen memory pages 5,6,7 and some of page 2 and 3. To adequately save all and have unchanged access would require more ram then available. The utilities will selectively load sections of memory for analysis.

Advanced users can access the CRACK-SHOT data disk. The data is written to disk 10 pages/track from page 02 to C0 then C8 to CC. On each track there are 256 sync bytes written to disk first then the 10 pages are written in two page blocks. Each original memory page is made into two disk pages. Each original memory byte is made into two sequential disk bytes. The first disk byte is the original memory byte 'ORED' with \$AA. The next disk byte is the original memory byte shifted right one and 'ORED' with \$AA. This is similar to the encoding method used in address markers of normal DOS. In between each original memory page one byte of \$FF is written out to allow time for computation check of addresses. At the end of each track is a checksum calculated from the original 10 pages. The single byte checksum is stored as two disk bytes described above.

There are utilities included that operate under DOS3.3. The first is an editor to allow editing of data on a CRACK-SHOT disk. Several track at a time will be brought in and editing allowed before rewrite. The other utility accesses the CRACK-SHOT data disk to compact and store the copied program as a binary file on a DOS disk. This allows storage of several programs on one disk to condense the number of disks a user has.

QUICK COPY PROCEDURE

With APPLE power off insert CRACK-SHOT with switch up in any slot (make sure zero is empty or CRACK-SHOT is in slot zero). Turn on power and boot the program you wish to backup. Take out the original disk and put in a blank disk. When you have the program to a desirable point(usually the intro menu or game menu) flip the copy switch on CRACK-SHOT. CRACK-SHOT will give you a warning. If you wish to abort, turn off the APPLE. When you hit a key the copy will take place in only 15 seconds. Label the disk.

QUICK RUN PROCEDURE

CRACK-SHOT can be in any slot except six where the disk controller is. To run your copy you can be in basic or in the monitor. In basic type 'CALL 52224' or in the monitor type 'CC00G' and the CRACK-SHOT run program will start. It will ask you EXECUTE OR NOT? type space(default yes). It will ask you what screen to display. Again games usually use hires 1 or 2. You can always restart if you choose the wrong screen. After that the system will execute and you will see the program start.

If you turn power off count to 4 before turning on the power again. This allows a capacitor time to discharge. This time varies from card to card, it is anywhere between 2 - 5 seconds.

Alternate execution methods are covered on page 4.

*****NOTES********
****
****

It is important to stress again the placement of CRACK-SHOT. For making a copy of a program the CRACK-SHOT card can be in any slot. The main requirement is that no RAM or ROM card be present at time of copy. CRACK-SHOT could go in slot zero. The disk I/O card has to be in slot six. For running copied programs CRACK-SHOT can go in any slot. Slot zero can have any card. The disk I/O card again must be in slot six.

There is a capacitor used during powerup on the CRACK-SHOT card. If power is turned off the capacitor must be allowed to discharge before turning power back on. The required time is between 2 and 5 seconds. If you cycle power count before turning on again.

When the program to copy is interrupted (by copy switch) there are certain code sequences the processor will hang when the program copy is restarted. The solution to this is to interrupt at a slightly different time. This is a rare occurance but should be recognized.

The CRACK-SHOT system has been tested with many different programs. As the state of the art of programming evolves there may be programs not copyable by the current system. The software of CRACK-SHOT is socketed in eprom and can be updated as changes happen. The utility editor will allow some changes.

4.0 MEMORY SETUP OF HARDWARE

CRACK-SHOT consists of 1k ram using two 2114 chips of 1k by 4 bits. It also has a 2716 eprom with 2k by 8 bits eprom used for program storage.

1. 1k ram interfaced at $C800
2. 2716 eprom interfaced
   a) lower 1k at $CC00 to $CFFF
   b) upper 1k at $FC00 to $FFFF
      only active when NMI occurs

Program developers can write software to operate using the CRACK-SHOT card. The eprom is socketed.

In normal mode(led off) the lower 1k of the eprom is visible. The upper 1k of the eprom is only visible when a NMI interrupt is generated on CRACK-SHOT(led on) by flipping the switch on the card. The card locks out the motherboard rom and enables the upper 1k of the 2716 to the addresses of $FC00 TO $FFFF.

INTENTIONALLY LEFT BLANK

5.0 EDIT UTILITY

This utility will allow the user access to the CRACK-SHOT copy disk for editing and analysis purposes. The user may read in a section of memory from disk ,disassemble the buffer ,edit and write the buffer to disk. The last page of this documentation is a cross listing of original memory at time of copy versus what track the memory is stored on.

The editor is on a DOS3.3 disk.

When you run the program you will see a menu of options.

1. READ TRACKS TO BUFFER
2. DISASSEMBLE BUFFER
3. EXIT TO MONITOR
4. WRITE BUFFER BACK OUT
5. EXIT TO BASIC

1. READ TRACKS TO BUFFER

You may read in up to 13 decimal tracks at a time. The buffer starts at $1000 hex and goes up to $9100 hex if the maximum of 13 decimal tracks are read. On the right of the screen is the current status of the buffer. This status is updated when you read into the buffer. It is displayed in the main menu and the read/write sections. When you select option 1 the screen shows you all possible 13 hex tracks and the pages contained on each track. As mentioned before this page is duplicated at the back of this documentation. On entry to read section the program will prompt you for the starting and ending track numbers (enter in hex as displayed). Remember no more than 13 decimal (0D hex) tracks at a time.

Note that track 13 hex only has 4 pages. The restart data is on page $CB00, the fourth page of track 13. See CRACK-SHOT documentation for more on placement of the restart data.

When a track or series of tracks are read in remember they are not placed back to their copied location. They are placed into the buffer beginning at $1000. Thus when disassembling remember the displacement factor.

The status information is 4 lines of labels and numbers.

TRACK STRT >   All numbers in hex
NUM TRACKS >
BUFFER STR >
BUFFER END >

The first is track start and shows the first track in the buffer. The next is the number of tracks in the buffer. The next is the buffer start (always $10 for $1000 hex). The last is buffer end , for the last page of the buffer. The buffer does include through this page.

Buffer start and buffer end are only given as page numbers. Parts of pages are never written or read.

This information is given to facilitate storage of the buffer to a DOS 3.3 isk. Since the editor runs under DOS the user could read in a buffer....exit to basic.... and store the buffer with a binary save command.

There is a checksum calculation on the read data. If the data does not read reliably the program will stop and inform you. It then returns to the main menu.

2. DISASSEMBLE BUFFER

Once some data has been read in the user can disassemble the data using this command. The status information is giver here also to show the limits of the current buffer.

On entry the user is asked for the address to start dissasembl$. Enter the number and hit return. Then 20 lines of assembly code will be shown. If you hit a space the next 20 lines will be shown, etc. If you hit return you go to the main menu.

Remember the code is not placed back to where it was copied and is displaced.

3. EXIT TO MONITOR

This option allows you to enter the monitor and make some changes to the code in the buffer. Once the changes have been made reenter the program with a command:

*800G (CR)

The buffer and pointers are still intact and you can write the buffer out with the next option.

4. WRITE BUFFER BACK OUT

This option writes out the buffer back to disk.

******PLEASE BE CAREFUL HERE!!!
******
******

When you hit the next key after the prompt the disk in slot six drive one will be written on!!! If it is not the CRACK-SHOT disk it will be overwritten. If it is write protected the program will stop.

The program writes out to disk the same track(s) read by option one. There is a verify after write on this option. If the disk cannot verify the written code it will tell you this and return to main menu. This is unlikely since you just used this disk to read in the code.

5. EXIT TO BASIC

This option allows you to save the buffer or parts of to a DOS 3.3 disk using the BSAVE command. The status information tells you the end of the buffer. The buffer begins at $1000 hex always.

TRACK STORAGE ALL NUMBERS IN HEX

TRK	PAGES
0	02-0B
1	0C-15
02	16-1F
03	20-29
04	2A-33
05	34-3D
06	3E-47
07	48-51
08	52-5B
09	5C-65
0A	66-6F
0B	70-79
0C	7A-83
0D	84-8D
0E	8E-97
0F	98-A1
10	A2-AB
11	AC-B5
12	B6-BF
13	0,1,4 AND RESTART

INTENTIONALLY LEFT BLANK

6.0 PACKER UTILITY

INTRODUCTION

This program will aid in creating a binary DOS 3.3 file from your CRACK-SHOT copy disk. This will compact your programs and the number of disks necessary to store them as well.

BASIC THEORY OF PACKING

Shown below is a map of the apple ram (random access memory). This does not include the 16k upper ram of a ramcard. The CRACK-SHOT disk does not have the upper 16k stored on it.Therefore the upper 16k are never used or discussed. Many programs do not require this upper 16k.

All numbers are in hex

```
                                                      PAGES

$0   $8        $20            $40            $60        $96     $CO
---------------------------------------------------------------------
!     !          !              !              !          !        !
!     !          ! Hi-Res 1     !  Hi-Res 2    !          ! DOS    !
!     !          !              !              !          !        !
---------------------------------------------------------------------
```

In this document and in the PACKER program all numbers preceded with a $ symbol are in hex. For those of you that have never used hex it is quite easy to learn. If you need to learn or want a refresher see appendix A.

A convenient way of referring to parts of memory in the APPLE is to use pages. A page is 256 bytes of memory. On the CRACK-SHOT disk there are $CO pages of memory. $C in base 10 is 12. Therefore there are 12*16 decimal pages of memory stored on a CRACK-SHOT disk. These are the first $CO pages of memory on the APPLE.

The CRACK-SHOT disk contains all $CO pages plus the necessary data to restart the program that was running at time of copy. A reference of what pages are on each track of the CRACK-SHOT disk is given by the PACKER program and at the back of this document in appendix B.

You have made a copy of the original program with the CRACK-SHOT card. The CRACK-SHOT disk now contains $13 tracks of data. When the copy switch was hit CRACK-SHOT copied memory pages $0-$CO to the CRACK-SHOT disk. From tracks 0-$12 there are 10 pages of memory saved on each track. Track $0 stores pages $02-$0B, track 1 contains $0C-$15,.....track $12 contains pages $B6-$BF. Then on track $13 CRACK-SHOT put the necessary data to restart the copied program.

The CRACK-SHOT disk can't be booted as a DOS 3.3 disk. It

can be booted by the CRACK-SHOT card,or other utility programs operating under DOS 3.3. Although the bootup is very fast ,10 seconds, it requires a dedicated disk. The PACKER program is used to pack the CRACK-SHOT disk to a binary file and save it to a DOS 3.3 disk. This binary file can then be executed from normal DOS.

The program is called PACKER for a definite reason. The largest binary file you can read/write to a DOS disk is $80 pages long. There are $C0 pages stored on a CRACK-SHOT disk. Obviously you can't store the CRACK-SHOT disk as one big binary file. What must be done is to decide what parts of the original $0-$C0 memory pages , stored on the CRACK-SHOT disk, to save. You will load in parts of the copied memory into a buffer and look at it with some utilities. When you find a part of memory in $0-C0 range you want to save use option 8 to add it to the binary file you are building. There are utilities to aid you with selection and packing.

This binary file you build will contain a routine (placed automatically by PACKER) to relocate memory modules correctly and restart the program copied by the CRACK-SHOT card. A memory module is a section of memory you chose to include to the file.

EXAMPLE: Say you have a program on CRACK-SHOT disk. Let's say also you know the program has parts at $800-$1200, $6000-$8000, and $B000-$BF00. Then there are three modules: one module for each memory section mentioned above. You could say why not make one module from $800 to $B800. Well the longest binary file you may save/read is $7A00 long ($9100 with language card PACKER). The above file is $B000 bytes long and cannot be loaded by normal DOS. So the PACKER program takes only the required memory and packs them together.

Then at execution time a small relocate program places these packed modules to their correct locations. This relocate program is called RERUN. It not only puts the modules back to their proper position but reloads processor registers and restarts the program copied by the CRACK-SHOT card.

The PACKER program loads a binary file ,called the STORE, with modules and restart data. This is the file ,that once completed, will be saved on a DOS 3.3 disk. This is the file you will execute in replacement of the CRACK-SHOT or original disk.

Below is a map of memory while running the PACKER program.

```
$0    $8       $20              $67       $9A         $C0
-----------------------------------------------------------
!     !        !                !                       !
!     ! PACKER !     STORE      ! BUFFER  !    DOS      !
!     !        !                !                       !
-----------------------------------------------------------
                                !->
```

The PACKER program resides in memory from page $08 to page $20. At page $20 the STORE begins and builds up. The language card version has the PACKER program stored from $D000 up, and the STORE begins at $0800. The term lc will stand for language card. The buffer starts at page $67 but as the STORE increases in size the buffer will draw away and shrink, this is automatic. The buffer cannot overwrite DOS as DOS is needed to write the binary file to disk once the file is built. The buffer is used to read in parts of the CRACK-SHOT disk for analysis. The user decides whether to include the memory in the buffer or not. The section added will be a new memory module.

This completes the Basic theory. We know the CRACK-SHOT disk contains all memory $0-$C0. The PACKER program is used to select parts of the CRACK-SHOT disk to be included in the binary file. These parts are called modules. The binary file is made up of these modules and a program called RERUN, that puts the modules back to their correct place and restarts the program copied by the CRACK-SHOT card. The next sections will expand greatly on performing these actions.

PROGRAM OPERATION

Run the program PACKER and the following menu should appear.

```
PHYSICAL $67  $6C
LOGICAL  $0C  $11

                                TRACK START        >01
                                NUMBER OF TRACKS   >01
                                PHYSI BUFFER START>67
                                LOGIC BUFFER START>0C
                                PHYSIC BUFFER END >7B
                                PHYSICAL FILE END >20
                                LOGICAL FILE END  >20


PACKER MENU

OPTIONS:

1. INITIALIZE
2. READ TRACKS
3. DISSASSEMBL$
4. ASCII DISPLAY
5. ASCII MARK
6. CODE  MARK
7. FULL  MARK
8. ADD TO FILE
9. PAGES STORED
A. SAVE FILE
B. EXIT TO BASIC
C. RUN COM FILE
E. EXECUTE CURRENT STORE

(ESC) CATALOG DOS 3.3 DISK
>
```

First let us look at the upper right part. This is the status area and will give information needed on the current state of the buffer and STORE. There are seven different labels and values with each label.

From the top the first label is TRACK START. This is the first track stored in the buffer. On the CRACK-SHOT disk there are $13 tracks of stored data. During packing you will read in several of these tracks to the buffer. TRACK START tells you what track starts the buffer.

The next value down is NUMBER OF TRACKS. The number of tracks contained in the buffer is displayed with this label.

The following label is LOGIC BUFFER START. I now must introduce two new concepts. They are physical and logical addresses. Data in the buffer has a physical address in memory when you run the PACKER program, the physical address is the data's address at that instant. But it contains data read from the CRACK-SHOT disk. That data from the CRACK-SHOT disk has an

address called the logical address. EXAMPLE: Say the buffer start is $6700. Then the data's physical address is $6700. Now say we read in track 1 from a CRACK-SHOT disk. In appendix B we see that track 1 contains data copied from $0C00-up. The data was at $0C00 during original program execution but the CRACK-SHOT card copied it and put it on track one. The logical buffer start is then $0C00. That is where it was copied from . That is where the data should actually go. We cant put it there because it interferes with the PACKER program and the STORE, so we put it in the buffer.

The next label is PHYSICAL BUFFER END. This is the position in memory where the buffer ends. Beyond that point there is nothing of interest. So in between PHYSICAL BUFFER START and PHYSICAL BUFFER END is the data read in from the CRACK-SHOT disk.

Next on the list is PHYSICAL FILE END. The PACKER program builds a binary file from the CRACK-SHOT disk with your help. The binary file in memory is called the STORE. To give you an idea how large a binary file you have built so far the physical STORE end is given. The STORE always begins at $2000($0800 for lc version).

Last on the list is LOGICAL FILE END. The binary file you are building will execute somewhere in memory. PACKER allows you to build it with a starting address anywhere between $800 and $A000. The file is built from $2000 ($0800 for lc version) up but it is modified to be able to run in the above range. You will need to specify the starting address when you initialize' the STORE. More on this later. With LOGICAL FILE END you can monitor where your binary file ends.

On the lower left is the menu of options. We will cover each option in sequence. First another addition to the status area. On the top of the screen aid you in keeping track of the buffer status and contents.

Remember our discussion on PHYSICAL and LOGICAL? The top two lines will give a translation every 5 pages between the physical current address, and the LOGICAL address the data belongs, every five pages. Going back to our previous example we read in track 1 to the buffer. The buffer starts at $6700 and the data was read from the CRACK-SHOT disk to that address. We see from appendix B that the logical address for the data is at $0C00. Five pages into the buffer at $6C00 the logical address is $1100.

We can see from the status area on the right that the TRACK START is 1, the number of tracks is 1, the buffer starts at $6700 and ends at $7B00. The LOGICAL BUFFER START is $0C00. The more you use these the easier will become reading them. On to the MAIN menu.

1. INITIALIZE STORE

This is the first step to start building a file. You need to have two things worked out before entering here. The first is the starting address of the binary file you are about to build. The STORE (the binary file as it is built) is going to be built for you from $2000 ($0800 for lc version) to ??00. This does not mean you will run it from that address. The STORE will be modified to run within a certain range of memory, you supply the starting address.

```
                              PAGES

$0    $8       $20        $40         $60           $96   $C0
-------------------------------------------------------------
!     !        !          !           !             !     !
!     !        ! Hi-Res 1 !  Hi-Res 2 !             ! DOS !
!     !        !          !           !             !     !
-------------------------------------------------------------
```

When you enter initialize the program will prompt you for your CRACK-SHOT disk. Then it will ask you two questions. The first is to enter the starting address of the binary program you will build. That address can be between $0800 and $A000. This is the address where the binary file you packed will start loading in at. Once packed and saved to a DOS 3.3 disk, to execute the packed program you will type 'BRUN NAME,A$1A00' where $1A00 is the starting address in this example. On receipt of this command DOS will go to the disk and look for file NAME, if found it will load the binary file into memory starting at $1A00,(1A00 for this example). The first question in INITIALIZE is to enter a starting address of your choice.

The reason that you cannot load below $08 is because part of the initializing routine is to read in those 8 pages and put them in your STORE. This is **always** done. The lower 8 pages are used in several different ways by programs. One obvious way is that they are the video mapped memory used by the APPLE. Many programs use this area of memory. Down at pages 0 and 1 are many system and processor locations. The 6502 uses page one as the stack, thus it must always be copied.

PACKER does not give you the option on this count. The first 8 pages of memory are automatically included into your STORE. The first 8 pages of every file you create with PACKER will always be the lower 8 pages of memory. They are also the first 8 pages to be relocated. This is all automatic and is only given for full information. The program RERUN is appraised of this also.

To get back to the two questions asked, the first requests the starting address of your binary file. Now a little logic on selecting this value.

You are going to build a binary file made up of memory modules. These modules will be relocated to their correct

addresses and then RERUN starts the program that we packed. The memory modules are sections of memory you choose to include in the STORE(binary file). The best way to explain this is with an example.

The memory map below shows the APPLE memory with a program marked. The program is in 3 sections and is the parts of the map marked in stars.

```
                                                 PAGES

$0    $8        $20           $40          $60        $96    $C0
-----------------------------------------------------------------
!      !         !             !            !           !      !
!      !         ! Hi-Res 1    !  Hi-Res 2  !           ! DOS  !
!      !         !             !            !           !      !
-----------------------------------------------------------------
            !******!       !*****!               !******!
          $18     $22     $40    $4A            $90     $A1

        !=============!            !++++++++++!
      $08           $37           $50        $7F

                           !+++++++++++++!
                          $40           $6F
```

There are three program areas. Their total length is $28 pages. Don't forget about the lower 8 pages that are included automatically. With their inclusion the length of the binary file is now $30. What is needed is the ability to take their compacted form and fit it into memory somewhere. There are several criteria for this placement. First of course is the limit set by the program of starting between $08 and $A0. Second is to avoid suicide. This happens when the program attempts to write over itself.

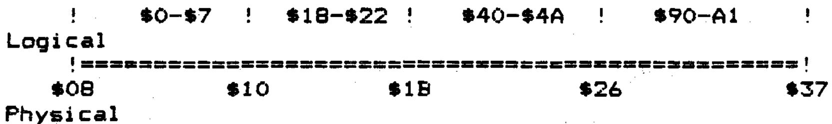
As an example three alternative packing locations are given. The two locations marked with plus signs are OK. You could pack the binary file between $50 and $7F. This is beneficial in that it does not interfere with the relocations. Remember in the memory between $50 and $7F are the modules that need to be relocated to the three starred areas. Since the $50 area is not overlapping any stars there is no chance for a conflict. Another alternative would be to pack it between $40 and $6F. This is really just as good as the previous location. Even though it resides where one of the modules goes if you check the sequence of placement you will see there will be no conflict. By the time the module at $40-$4A is to be placed the binary file is up around $50. The lower 8 pages have been moved out and the range of $18-$22 has been moved.

The last alternative is the bad selection. This attempts to pack the three modules and put the binary file starting at $0800. This leads to trouble. The first 8 pages are relocated without any problems. But now we wish to move the first set of stars out to their proper location at $1800 to $2200. The

module is stored at $1000 inside the binary file (remember it loads at $0800 and first 8 pages are lower mem). It extends up to $1A00. An error will occur$!!. The module's destination is $1800. If RERUN puts the module out to its correct location it will overwrite the end of the module stored in the binary file.

The map below should help visualize the movements for the bad selection.

BINARY FILE



This may be confusing but sit down and read through this a couple of times. One benefit that PACKER provides is error checking. When you try to add a module to the STORE the program PACKER will check for errors. Also at the time of saving to a DOS 3.3 disk the file is checked again. This second check will become obvious later.

So for this example the input to the first question could be $50. The program wants only the page number of the starting location. You cannot start the binary file in the middle of a page. Later in the text we will refer to the value entered here as the DESTination, or DEST for short.

The second question asked in this initialization is placement of the restart program RERUN. All this time we have been talking about moving the memory modules around something must be in control and doing the moving. That something is the RERUN program. It is a one page restart program built into your binary file. It will be modified to fit wherever you want it to go. As with the first question there are limits and placement logic. The limits for RERUN placement is it must be after the DESTination and not more than $7A ($90 for lc version) pages after DESTination. It cannot be in the first 8 pages either as they are reserved for the lower 8 pages of memory that are copied every time.

The logic for selecting RERUN's location is not too hard. RERUN is inside the binary file you are building. It is always in control and therefore can never be overwritten. Pick a single page of memory that is not used by your copied program and put RERUN there. Remember the aforementioned limits. You can usually find a page such as that pretty easily. There are utilities and text to help you decide where to put the 1 page of RERUN program. See option 6 code search in the documentation.

In the above example a good choice could be $70. Enter the page number only. This entry in later text will be referred to as REDEST. This page was not used by the program we copied. Ah.. you say it is in the middle of one of the modules. Not so.. the values of DEST and REDEST are set before

anything else. They define the building points of the binary file, the STORE. Later when you begin to add modules the PACKER program will automatically split modules to fit snugly around the RERUN program. In other words if you want to add a module to your STORE, and it will try and overwrite RERUN the Packer program will split the module and make two modules with the RERUN program between them. This is automatic and will be taken care of for you.

After the two questions are answered the disk drive will spin shortly. This is Packer loading the first 8 pages to the STORE and loading the restart data into RERUN.

Next you pick the screen that is to be shown at execution time. It could be hires 1,2 text 1,2 etc... You will be asked four questions on screen control. This is the same as the options menu on the CRACK-SHOT card restart program. Packer needs to know what screen to build into RERUN. There are 4 questions with two options for each. Option 1 is default, if you hit any key but 2 for the alternate, option 1 will be chosen. They are self explaining.

We spent a long time in this section as initialization is extremely important. Proper selection of the binary file location is important. More examples will be given later in the text.

That finishes INITIALIZE!!!!

2. READ TRACKS TO BUFFER

This is a complete copy from the EDITOR program. You will be asked to enter the range of tracks you wish to load into the buffer for working with. Please note one added feature. Just above the first query of 'START TRACK' is a status line of 'TRACKS /LOAD '. This is the maximum number of tracks you may load in from this section.

As in the EDITOR program there is a listing of the tracks and the pages stored on each track. If you hit return when asked for the starting track the program will goto the MAIN menu.

3. DISASSEMBLE CORE

This option allows you to use the APPLE disassembler on any area of memory. This will be one of your major means of determining what parts of the CRACK-SHOT disk to include.

You could read in a section of the copied program, from the CRACK-SHOT disk. Then use this option to disassemble the buffer area. The limits of the buffer and the logical memory area are all given in the status area.

EXAMPLE:

You have made a copy of a program. Now you wish to pack it. Using option 2 you read in a section of memory. Now you can use option 3 to disassemble the buffer. On entry

to this option the status areas will be displayed and the computer will ask you for the starting address to begin decoding. Enter this as a full address, not as a page number. The computer , through calls to the mini disassembler in rom, will show you 20 lines of code. At that time it will halt and wait for a keypress. If Return is hit the computer will exit to the MAIN menu. If anything else is pressed the next 20 lines of code will be shown. This will continue to loop until a Return is hit.

The benefit is the ability to look at sections of the CRACK-SHOT disk and see if a program is there. The disassembler will list the address, the instruction codes and the assembly mnemonics. For the novice at assembly language appendix C will help you in spotting possible program areas.

4. DISPLAY MEMORY AS ASCII

The program you copied will not only have instruction codes, the machine code that actually 'runs' the program, it will also have some parts of memory set aside for ASCII storage. ASCII is a term used to represent the codes of letters,numbers,symbols...etc. There is a table of reference such that every letter, number..... all are represented by a unique hex number. The program uses these codes when it wants to print something on the screen. This program has several pages of nothing but ASCII codes.

An example would be the MAIN menu. That menu is put on the screen by a driving program. The driving program has a certain place in memory where a copy of the menu is stored in ASCII form. Most programs use areas of storage for ASCII. You will need to look for these also and include them in your STORE. If you did not then how would you run the program if no menu or prompts were put on the screen? Also the programs usually expect the ASCII to be there, if not then the program will sometimes blowup or go off into never never land.

On entry to this section the usual status will be displayed. Then the program will prompt you for the starting address to begin showing memory. Enter a complete address, not a page number.

The program will then begin displaying memory as if it was ASCII stored. It is easy to spot the areas you want. You know what menus, prompts etc are part of the program you copied. Look for them. You might be surprised what else you can see. Some programs I have looked at actually have parts of other programs or source code for development. The display on the screen will give you 32 characters per line with an address on the left. As in the disassembler section if you hit Return the program goes to the MAIN menu. If any other key is pressed the next series of memory is displayed.

5. ASCII SEARCH AND MARK

Here is a search utility to help you. This section will

search the buffer for occurrences of ASCII. The program will start with the first page of the buffer and count the number of stored bytes with values between $A0 to $D0. This is the normal range of ASCII for letters and numbers. At the end of the page it has a count. It compares this count against a stored value. If the count from the page it just checked is greater than the stored value it 'marks' that page. All pages of the buffer are checked. At the end of the buffer all the 'marked' pages are shown to you. Say the buffer starts at $6700 and has one track in it. If you use option 5 all 10 pages of the buffer are checked. At the end the program shows you which pages in the buffer were marked.

At the first of this section we talked of the stored value. This is the value the count must equal or pass to 'mark' the page. This value is set in the program but can be changed by the user. On entry to this option the program displays the current value and requests a new one. If the user likes the current value just hit return and the program continues with analysis. If the user wishes to change the value he enters the new one. The program will then use this as the default value until changed.

After the buffer is checked the marked pages are shown. They are listed in the format shown below.

```
$67...$71 $79...$82 $90...$90
```

There will be from 0 to ?? pairs of numbers listed. The first number of a pair is the first page marked in a continuous series. The second number after three dots is the last page marked in a series. All numbers in between the values are marked also.

6. CODE SEARCH AND MARK

This section is very similar to the previous. It is a utility to search the buffer memory for possible areas of instruction codes.

The buffer is searched page by page as in the previous section. This time the program counts the number of assembly lines it can find in a page. When the APPLE disassembler is called an address is passed to start the disassembly at. What this section does is pass the disassembler a page of memory and counts the number of lines of assembly code it generates. For each page it gets a count. It compares this count against another stored value different from the ASCII value. If the count for the checked page is lower than the stored count the program 'marks' that page. The count for each page is displayed on the screen before checking. When you run this section you will see a block of numbers displayed on the screen. These are the counts for the pages the program is checking. At the end of the buffer all marked pages are displayed just as in the previous section.

On entry to this section the current stored value, called sensitivity, is displayed. The user may change it the same as

in the previous section. Now a few words on the sensitivity. The fewer lines of assembly code the disassembler generates the more likely the page being checked is a valid page made up of instruction codes. Therefore to make the program very sensitive to finding pages ,and finding some borderline or trash, make the sensitivity value high. A good value found currently is around $A0. Assuming an average of two bytes per line of assembly code comes out to a sensitivity of $80. Try different values and see what results you get.

There is another searching and marking algorithm used in this section. This is transparent to the user but is done automatically. Any page that is marked with having possible code is checked again. The next check is for 16 bit indexing. This is a popular means of moving data, doing table lookups and many other objectives. PACKER cannot spot data or table areas easily. The data or table can be made up of any values. One way to try and mark those pages and include them into your STORE is to look for the program using them. That is done in this part. PACKER will take any marked page with a count of less than $80 and do a search for 16 bit indexed used. Any pages indexed into by the program will be marked for you, also any pages jumped to by a JMP or JSR are marked. That is why strange pages may appear in the marked display. Your buffer may only contain between $02-$33 but the marked pages when shown may have other values marked. These were obtained by looking for these indexing opcodes.

Here is a good place to pick a value for REDEST. Before INITIALIZING come to this option and do code search. Choose an unmarked page and disassemble it with option 3, if it looks like trash or unused then use it as REDEST.

See appendix C for more detail on assembly code searching.

7. FULL CHECK OF CRACK-SHOT DISK

This section will do a full search and mark on the entire CRACK-SHOT disk. All memory pages will be loaded and searched. Both the code and ASCII search will be used. On exit the program will display the marked pages the same as in options 5 and 6.

There is one important thing to stress here. Use of this option after you have started building the STORE is not recomended. The original full size buffer will be used for speed. If your STORE has a physical STORE end past $67 then you will lose that data. Remember as the STORE gets larger the buffer shrinks to draw away from the end of the STORE. This section will not use the small buffer size but use the original full buffer. If you have started building and the physical end is not at or beyond $67 then you can use this option safely.

The use of this full check can greatly help spotting the useful areas of memory. You may obtain a hardcopy of the marked pages the same as in the previous sections by entering the printer address when requested. If a carriage return is

hit the output will go to the screen.

Since this section calls the code and ASCII mark options the current sensitivities before entering are used. If you want a more optimistic marking make both sections 5 and 6 more sensitive.

8. ADD TO FILE

This section is used to take portions of memory in the buffer and move then to the STORE. You must have previously INITIALIZED. You have looked at the buffer with options 3-7 and now you are ready to move a portion from the buffer to the binary file, STORE, you are building.

On entry the usual status areas are displayed. The program will prompt you for the beginning physical page number to add. If you hit return the program goes back to the MAIN menu. Two things to emphasize here!! The first is you will enter the **Physical** page number, not the logical. The translation table at the top of the page will help you recognize what logical parts of memory you are adding. The second thing is to stress that you are to enter the page number only, not the full address. All additions to the STORE are in full page increments. No half or partial pages are included.

The program will prompt you for the ending physical page number. Again this is the physical, and you must use page numbers. The number you enter will be included also. So if you enter $67 first and then $6B the program will take pages $67.68.69.6A.6B and move them into the STORE as a memory module.

Before moving the pages to the STORE the program will first do an error check. The first error check done is to ensure that you do not attempt to overwrite REDEST. In the initialization you entered a logical address for the RERUN program to go. You cannot save a module which will attempt to relocate over REDEST. Remember, you must pick a page for REDEST that will not be used by the program you copied. If you do attempt to overwrite REDEST the program will print an error message and ask for the starting address again.

The next error check is for an attempt to overwrite itself. This is the same error that we covered in the initialize section. Any attempt by the module to overwrite itself or another unplaced module is not allowed. An error message is printed and you are prompted for the starting physical page again. At this time you have two options. The first is to start over again and change the location of DEST. By doing that you may be able to move the binary file around enough to stop any overwrite errors. This is not the best solution but you may have to resort to it sometimes.

The other alternative is to delay adding this section. This is going to be tricky but let's try with an example. Lets go back to our old friend the example used in INITIALIZE. Below is the map again.

BINARY FILE

```
    !     $0-$7   !  $18-$22 !   $40-$4A   !   $90-A1     !
Logical
    !=====================================================!
   $08          $10         $1B          $26            $37
Physical
```

This setup causes an error when the module stored at $10-$1A moves out to its correct position at $18-$22. A solution to this would be to delay including this module until later. If you added the module for $40-$4A before the $18 module then there would not be a problem. The new map shows the memory setup.

BINARY FILE

```
    !     $0-$7   !  $40-$4A !   $18-$22   !   $90-A1     !
Logical
    !=====================================================!
   $08          $10         $1B          $26            $37
Physical
```

The module placement above would be correct. The modules are always moved one page at a time starting with the lowest page in a module. Thus the page at $1A physical has a logical address of $1B and can be moved down. The module below has already been moved out so the space can be written on.

This brings up the order of placement. The modules that you add to the STORE are placed back into their correct memory locations. They are done one module at a time starting with lowest module in memory and moving up. Thus any memory below the current module can be written to if desired. The only exception to this is when the RERUN program is below the current module. You can write anywhere but that particular page. In the initialize section we said you must pick a page not used by the copied program.

For final execution it does not matter in what order the modules are placed. All modules are relocated before program restart occurr$. Placement order only matters when trying to solve problems as shown above.

9. SHOW CURRENT PAGES STORED

This option will allow you to determine what pages you have already moved into the STORE. On entry to this section the computer will ask for your printer card address. If you want a hardcopy of the data enter the address. If you only want to see the data on the screen then hit return.

For the printer card address enter a full address. If you have a special printer driver just enter its address.

The usual status areas are displayed then DEST and REDEST with labels. Below this is a printout similar to the marked pages output from code or ASCII search and mark. The computer will print several pairs of numbers. Each pair of numbers is a module. The first number is the logical start of the module. The next number, after three dots, is the last page included in the module. With this data you can keep a record of the areas of memory that were used to build the binary file. It is a good idea to use this option with the printer just before saving the binary file. That way you can keep a record of the pages used. If the program doesn't fully execute try adding more pages, or use more sensitive values in the search and mark routines.

A. PUT FILE TO DOS BINARY

This is the final step of using this program to build a file. In this section two important things occur. The first is a final error check of the stored modules. This is the second error checking mentioned in the INITIALIZE section.

The purpose of this check is to stop any overwriting of the binary file by itself. You might think the error checks in ADD section did all of this. Well they did to the best of their capability at that time. The problem then was that the program had no means of knowing how large the binary file would become. Neither does the user on the first try. The best way to explain this is with another example.

EXAMPLE: The user has initialized the STORE and added two sections. They are shown in the map below.

```
                                BINARY FILE

   ! $0-$8 !  $08-$28       !$50-$52!
Logical
   !================================!
  $08    $10                $30     $32
Physical
```

The user has added the $08-$28 module and the $50-$52 module. Neither of these cause an error at this time. I am assuming RERUN is placed somewhere out of harms way. The error happens later when the user adds the section from $A0-$C2. The

map below now shows the status of the binary file when it tries to execute. The lower 8 page module relocates ok, the next module of $08-$28 also has no problems. The problem of overwrite occurs when the module $50-$52 attempts to relocate. It will overwrite part of the last module which has not been moved yet.

```
                                       BINARY FILE

    ! $0-$8 ! $08-$28          !$50-$52!    $A0-$C2         !
Logical
    !=================================================!
   $08     $10                 $30     $32                $54
Physical
```

This error can only be caught when the entire file has been built. When you enter option 9 the program checks for these errors. If found it will tell you which module and where the overwrite occurred. The solution is to use option 9 to get a copy of the pages you included. Then use option 1 to re-initialize the STORE.

With the knowledge of where the problem of overwrite occurred, you can move the value of DEST to try and avoid this. If you increase dest by $04 this would solve the problem. You can't decrease it anymore because it is already at the minimum. Another solution would be to delay including that module as in the previous example. Use the same DEST but this time add the module $A0-$C2 before the module $50-$52. With so much ability to move the program start, change order of modules..etc there should be a solution to almost all packing problems.

The question about an extra module only occurs in the language card version of PACKER and is expanded in the last section on packing.

Once the error checking has occurred, assuming no errors, the program does the final touchups on the binary file. It will display the command you need to use to execute the binary program. An example is shown below.

BRUN CRACK-SHOT,A$1000

One note is for very large files that have a LOGEND at or near $9A00 there might be a problem. Issue the command 'MAXFILES 1' before the BRUN command and there should be no problems.

Packer will prompt you for a file name to save the binary file under.

The file is stored from the range of $2000 ($0800 for lc version) -->$??00 . This is where the file resides in memory as it is being built. To execute the file you use the brun command, but you must use an address extension as shown in the above example. The PACKER program will give you the full command to use. If you don't want to use the address extension

all the time the file can be changed. Instead of giving a BRUN command you could BLOAD the CRACK-SHOT file. In the example above you would 'BLOAD CRACK-SHOT,A$1000'. Next you would save the file again at its proper location. This way all you need to do to run the file is to type BRUN CRACK-SHOT. To save the file you need the starting address $1000 and the length. PACKER gives you the length of the file at the same time that it saves it. The length shown is the hex value. Let's say for the example above the length is $40. That is $40 pages of length.

You would use PACKER to build the file and then save it to a binary disk. Then exit PACKER and issue the command:

BLOAD CRACK-SHOT,A$1000

Then you would give the command:

BSAVE FILENAME,A$1000,L$4000.

The length of the file goes after the L$ and put two 0's after the length.

The only exception to this is if the file needs to do a master relocate. The modules will be relocated by RERUN. A master relocate is when the entire binary file must be moved. This happens when the file you built wants to load where DOS is. You use DOS to load the file so you cant overwrite that. The solution is to load the file in lower memory and then relocate up to higher memory. This is automatically done for you. If you choose DEST such that the binary file will overwrite DOS when it loads the PACKER program adjusts for this. It builds the file at $2000 ($0800 for 1c version) and modifies it to relocate to higher memory once loaded. You can tell if the binary file will do a master relocate by looking at the LOGEND value at time of saving. If LOGEND is greater than $9A00 then the binary file you save will do a master relocate.

The exact sequence for a master relocate is as follows. When option A is hit PACKER checks LOGEND. If LOGEND is >$9A then a flag is set in RERUN. Some parameters are set in RERUN that give the desired destination of the binary file. The binary file is always loaded in initially at $0800 (for 1c) or $2000 for lower memory PACKER. Once in memory the first 3 bytes of the binary file jump to RERUN, and the flag in RERUN is checked. If set RERUN relocates itself up into memory. It can overwrite DOS at that time because it is not needed any more. Once relocated the operation is the same, all modules are placed and program restart occurr$. All of this action is transparent to the user.

This takes you back to the main menu.

B. EXIT TO BASIC

This will exit the user to APPLESOFT.

C. RUN COMMAND FILE

This option will pack a binary file for you if a command file is available for the program you have. A command file is a short binary program containing all the information necessary to pack a program copied by a CRACK-SHOT card. There are several command files on the disk with the packing program. They all start with C.name. For example the command file for lower memory PACKER is called C.LOWPACK.

If a user has made a copy of the packing program with the CRACK-SHOT card then he could use the command file to pack it into a binary file. This is only for example as the packing programs are given to you anyway unlocked. The user would run the packing program and choose option C. On entry the program will ask you for the command file name. Put the disk with the command file in drive one and enter the name C.LOWPACK. The computer will load the command file and ask you for the CRACK-SHOT disk that contains a copy of low memory packer. Insert that disk in drive one and hit return. The program will then pack that disk into a binary file in memory.

On exit the program will tell you to use option A to save the binary file. Hit A for saving. When asked for extra module hit return. The program will ask for a DOS3.3 disk and request a program name to save the file with. It will also give you the command to use to execute the stored binary program.

That is all there is to command file packing. If you wish to create new command files for other programs see section 7.0 of this manual.

E. EXECUTE CURRENT STORE

This will execute the current store you have built in memory. The packer program will move it to where it would be loaded in by DOS and then jump to the starting location.(DEST)

It is recommended that you use option A to save the store first as using E will kill packer and your STORE. This allows a user to build a store, save it and immediately test it without exiting to reload...etc.

Users with language card may use the lc version of the PACKER program. This version has a few enhancements over the lower memory PACKER. The reason for this is the extra memory of the lc allows greater expansion of the PACKER program.

1) In INITIALIZE you are given the option to clear hires 1 on restart. This is given to clean up the hires screen before starting for programs that use hires.

2) The lc version allows packing of binary files with lengths up to $9100 bytes. PACKER patches DOS to allow the longer files.

3) The main improvement to the system is the addition of an extra module in the language card. The limit to the packing of the binary file is $9100 bytes but with a new addition to the program you can now store $B900 bytes.

The way this is done is with two binary files. One file is the regular large file of up to $9100 bytes. The other file is an additional file containing one memory module of up to $2800 bytes long. This additional module will go into the language card. The main program will load into lower memory as usual but will also access and relocate the language card module. Below is an example.

Say a program uses memory $0-$1F00, $4000-$BFFF. This is a large amount of memory and not packable before. The procedure is as follows.

Run PACKER, use option 1 to initialize a STORE. Set the beginning as $17 and RERUN at $1F. Then use option 2 to load tracks $F-$12. With those tracks in the buffer save the logical pages $A0-$BF to the STORE. Then use option A to save the file. Save it with name file1. In option A hit return when asked for an extra module. The file you just saved is the extra module that will go into the language card.

Now use option 1 to Initialize the STORE. When the program asks you to clear hires one on restart answer yes (Y). Set the DEST as $0800 and RERUN as $2000. Now pack pages $0-$1F and pages $40-$9F. Save the file with option A. This time when asked for the extra memory module enter $D0. The next question will ask for the module end, enter $F0 (actual end + 1), the last question will ask for the module destination, enter $A0. The module will be loaded into the language card from $D0 to $F0 and it belongs at $A0. Save this file as file2.

Now exit packer program and run the program EXEC FILE BUILDER. This will build you an exec file. The exec file is the file you execute to run the copied program. The exec file will load the extra module into the language card and then run the main binary file.

There are a few questions to answer for this exec file to be built for you. The first is to tell it the name of the extra module file. This is file1 from the example above. The program will load file1 and strip off the RERUN program and the automatic lower 8 pages of memory. All that is wanted in this file is the memory module. Then enter the name of the main binary file. This would be file2 from the example above. Enter the starting address (DEST) of $0800.

Last it asks you for a name to give this exec file it will build. To keep matters straight you might have called file1=namehigh called file2=namelow and the exec file name

Once this is entered the program will build the exec file for you. All three programs file1, file2 and the exec file must be on disk to execute. You must also have a language card to execute. To execute the program type EXEC NAME.

The language card is turned off immediately before program execution by RERUN.

This increases the size of the binary file you can pack.

7.0 COMMAND FILE CREATE

There is a program called command file create on your program disk supplied with the CRACK-SHOT card. This program will create a command file to be used by the PACKING programs. This file contains all the necessary data for the PACKER program to access a CRACK-SHOT copy disk and pack it into a binary file. Each copied program is unique and needs a command file.

When you run command file create the program will prompt you for the needed data. It will then store the command file on a DOS3.3 disk.

The first question asked is for the name of the command file. This is the name it will be stored under on a DOS disk. Next it will ask you for DEST. This is the beginning page address of the binary file you will pack. Enter a two digit hex number for the page number where the binary file will begin.

Next the program requests REDEST. This is the address of the RERUN miniprogram in your binary file. Enter a two digit hex number representing the page where RERUN will go.

For more information on DEST and REDEST see option 1 on PACKER program.

Now the program requests which screen setup to build into the program. This is the same set of questions as on the CRACK-SHOT card and in the PACKER program.

The next question asks how many modules there will be. This is the number of memory modules ***excluding*** the lower 8 pages memory module. That module is always present.

The program then goes into a loop depending on the above entered number of modules. It requests the starting and ending page address of each module. Enter the addresses as two digit hex numbers.

You would want to use this program command file create when you have successful packed a binary file and know what pages to pack. A good idea is to use option 9 in PACKER once you have a file packed. This option gives you DEST, REDEST, and all modules you packed, and it gives the modules in the correct order.

If you are building a command file from a parameter list you will need to pay close attention to the sequence of modules. Option 9 will give the correct sequence of modules to load. The parameter lists may only list areas of memory and not a good sequence of modules.

For instance the parameter list of a program that has 0..1A $20..65 $90..9A cannot be entered directly into a command file. You must use packer to find a workable sequence of modules. In this case......

8..1A 20..33 34..65 90..9A

The Packer buffer can only hold so much at one time. Some parameter lists may be in a good form for entry into command file create. Look for comments on the parameter lists.

Once the last module has been entered the program will prompt for a command file disk and save the file. Any file name can be used.

When running PACKER this command file can be used to pack any copy of that particular program.

8.0 EXTRA USES FOR CRACK-SHOT CARD

Besides being a program copy system the CRACK-SHOT card can also be used for program analysis. The ability to halt program execution at any time and have all the restart data saved for you is a nice utility.

If you wonder how a program works or where the microprocessor is currently executing code at any time you can use the CRACK-SHOT card. Simply boot the program you wish to analyze, let it proceed to the point you are interested in and make a copy at that time.

Most programs on the market these days are in assembly language. There are some programs that are in applesoft for sell. To find out where the APPLESOFT program is executing look at the zero page locations storing the current line. See the APPLESOFT manual for those locations and others.

If the program you copied is in machine language the program counter was saved at time of copy. To see the program counter you have two options. One is to use the EDITOR program or the better alternative is to use the PACKER program. The PACKER program is a better choice because it has more utilities for analysis built in.

To see the program counter (PC), Stack pointer (SP), and other register data load track $13 hex. From the documentation on CRACK-SHOT operation look up the storage pattern for the restart data. This tells you what data is saved and where. On track $13 the 4th page is the restart data. The PC is stored at $xxF9 and $xxFA, low byte first. Other register data is documented.

A copy of the stack is stored as the second apge of track $13. With this data you can follow the program execution and find out where the code segments are for different subroutines or other operations.

EXAMPLE:

Some programs use a set track for copy protection. This track may not contain sector data, it could contain only a set sequence of bytes from 1 to ?? that the program looks for. Bit copiers have problems copying tracks like this without help from users. Nibbles Away II has a track/bit editor for working on a track but without knowing what sequence of bytes the program expects the user may have a hard time. The CRACK-SHOT card can be used to solve this problem.

Make a copy of the disk using a bit copier. Then boot the copied disk. If the disk boots part ways but then hangs on a given track the program may be looking for a sequence of bytes and cant find them. This could be a synchronized copy also. Either way the program does not like this particular track for some reason. Well....let's find out why!!!

This time boot the copied disk and right when the program moves to the track it does not like make a copy with the

CRACK-SHOT card. This may necessitate removing the cover of the disk drive to watch the disk booting a couple of times. A good calibrated eyeball helps. Watch the read/write arm and copy the program right when it stops on the track giving problems.

Now you have a copy of the program as it is reading that track. To proceed from here you must run the PACKER program. Once in PACKER use option 2 to read in track $13. Then find the value of the PC at time of copy. If the PC value is above $F800 then the program was using a monitor routine. You need to find the SP value and locate the stack top. The top two bytes on the stack are where the program was executing before it called the monitor routine, unless the monitor routine called anthoer monitor routine. Once you have the location in lower memory where the calling routine is you want to look at that area.

Use option 2 to load the buffer with memory from around the above value. Then use option 3 to disassemble and analyze the code in that area. Find out where the disk is being read. The code will look something like below.

```
0000 LDA $C088,X
0003 BPL $0000
```

This is the code to look at the disk input data latch. After this code is either storage of the loaded data or comparison to a set value (such as $D5). This will look something like:

```
0003 BPL $0000
0005 CMP #$D5
0007 BEQ $0020
0009 (PRINT ERROR MESSAGE AND REBOOT)
       "     "
0020 (ALL OK PROCEED)
```

Usually the code will look for a sequence of bytes so there may be several sets of the above code. If the loaded data does not match then the program will usually print an error message and reboot or crash. You have two options at this point.

One is to use the track and bit editor of NIBBLES AWAY II and make the track look like the program wants. The other is to modify the code of the program. The first is easy if not many bytes are checked, if the track contains a significant amount of data then forget it!!!

The second is more appealing as it does away with the protection scheme. To do that you need to modify the code as it resides on disk. This can get mighty tricky and time consuming. Utilities such as the INSPECTOR and Dr. WATSON come in handy here. They allow access to the disk on sector level. The procedure is sometimes not easy and may require a couple of iterations. Make the copy of the program as described above. Now what you must do is analyze the code and find out where an error is detected. A code to look for is:

JMP $C600

This reboots the disk in drive 1. So any code that executes that is error code. Some programs as mentioned above will look for a sequence of bytes. If any one of the bytes is not correct it will branch to an error code area, look for this. Also look for any code that prints 'error' or other error indicators such as a letter in the upper left corner. Find all entries to such code.

Once you think you have located all the error code and entries to them you must change that code. The procedure to try first is to put nop's ($EA) where the program branches to that code. In other words change the program so that there is no entry to the error code. You can change the program in memory but that won't do you much good. What you must do is to change the code on the copied disk. Change the disk that was copied by the bit copiers, not the disk made by CRACK-SHOT. We only used the CRACK-SHOT disk to find the code areas for analysis.

Here is the fun part, sometimes it can be quick but other times it can take some time. A printer is a definite help. What you must do is to find out where the code you have analyzed is stored on the bit copied disk. It could be on almost any track. Use logic!!! If the disk booted track zero and then checked track one for a sequence of bytes then the code to find must have come off track zero. But....if the program loaded several tracks before checking one track the code you want to find can come off any one or several of the loaded tracks.

Find that code!!! Load sectors and disassemble, compare against the code you want to find. Nibbles Away II has a nice utility for reading a sector and then disassembling that sector in memory. When you have a match enter your changes and save the sector back to disk. Of course if they are using different address markers you may have to modify DOS some. Find a source listing of the RWTS DOS code and there are a few bytes to change that allow different address markers to be used. That goes into another realm of analysis.

So the basic procedure is as follows:

Make a copy of disk with bit copier
Run copy and find track it hangs on
Boot copy and make CRACK-SHOT copy when the trouble track reached.
Analyze the code reading the track
Find all error code and entries to such

Options:
A) Make track fit desired with track editor
B) change code on bit copied disk to ignore all error conditions.

GOOD LUCK!!!

MULT-ACCESS PROGRAMS

One trick to obtain a working backup copy is as follows. Make a bit copy of the disk. If that copy boots and runs then you have a backup.

If not you may still be able to have a backup. On the back of the bit copied disk put a CRACK-SHOT copy of the main or only program. Boot that copy and flip the disk when it is running. Then when it goes back to disk it goes to the bit copied disk that won't boot. The format may have been copied good enough to allow access by the main program.

If this doesn't work try finding the code that accesses the disk repeatedly. Some programs only access the master disk to check for a copy, no data is loaded. Find the code using techniques described above and branch around it. Even if 2-a few bytes are loaded you might want to simply modify the code around that area to reset those bytes and then continue.

APPENDIX A

The number system everyone uses normally is base 10. The computer uses binary numbers, which is base 2. The binary numbers can be read easier in base 16 which is hex. For an example of each see below.

BASE 10
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
BASE 16
0 1 2 3 4 5 6 7 8 9 0A 0B 0C 0D 0E 0F

In hex you use alphabet letters to count once you get past 9.

BASE 10 decimal	BASE2 binary	BASE 16 hex
09	0000 1001	09
14	0000 1110	0E
24	0001 1000	18
32	0010 0000	20

In base 16 you count up to 15 before going to a second digit, 0 1...E F 10. In base 10 you count up to 9 before going to a second digit , 0 1...8 9 10.

In base 10 every digit position is a power of ten. Starting with 10 to the 0=1.

Take the number 123.

3 times 10 to the 0 = 3
2 times 10 to the 1 =20
1 times 10 to the 2 =100

total=123

It is the same with hex. every digit is a power of 16.

Take the number 123 in hex. Translate to decimal.

3 times 16 to the 0 = 3
2 times 16 to the 1 =32
1 times 16 to the 2 =256

total=291 decimal

In the document all numbers preceded by a $ are given in hex.

INTENTIONALLY LEFT BLANK

APPENDIX B

The tracks on a CRACK-SHOT disk contain the memory of the APPLE at time of copy. The reference table below will show you what pages are stored where.

Track storage all numbers in hex.

TRK	PAGES
00	02-0B
01	0C-15
02	16-1F
03	20-29
04	2A-33
05	34-3D
06	3E-47
07	48-51
08	52-5B
09	5C-65
0A	66-6F
0B	70-79
0C	7A-83
0D	84-8D
0E	8E-97
0F	98-21
10	A2-AB
11	AC-B5
12	B6-BF
13	0,1,RESTART

INTENTIONALLY LEFT BLANK

APPENDIX C

The next few pages give examples of assembly language code listed using the APPLE disassembler. They are commented as to what to look for.

INTENTIONALLY LEFT BLANK

```
1000L

1000-   8D 2C 17    STA   $172C
1003-   20 3E 14    JSR   $143E
1006-   B0 E3       BCS   $0FEB
1008-   20 6C 12    JSR   $126C
100B-   AE 22 17    LDX   $1722
100E-   4C 5B 0F    JMP   $0F5B
1011-   20 58 FC    JSR   $FC58
1014-   A9 1F       LDA   #$1F
1016-   A2 22       LDX   #$22
1018-   20 FD 16    JSR   $16FD
101B-   20 E1 16    JSR   $16E1
101E-   4C 3C 0C    JMP   $0C3C
1021-   20 58 FC    JSR   $FC58
1024-   A9 67       LDA   #$67
1026-   8D 19 17    STA   $1719
1029-   A9 05       LDA   #$05
102B-   8D 16 17    STA   $1716
102E-   A9 20       LDA   #$20
1030-   8D 17 17    STA   $1717
1033-   A9 8D       LDA   #$8D
*
```

Here is an example of normal program code. Notice no ?? , every line has an assembly mnemonic on the right. 53 decimal bytes were used for 20 lines giving roughly 100 lines of code per 256 byte page. That is $64 bytes of code for $FF byte page. With the sensitivity set above $64 hex this page would be marked if the rest of the page was similiar.

```
*5000L

5000-   00          BRK
5001-   00          BRK
5002-   00          BRK
5003-   00          BRK
5004-   00          BRK
5005-   00          BRK
5006-   00          BRK
5007-   00          BRK
5008-   00          BRK
5009-   00          BRK
500A-   00          BRK
500B-   00          BRK
500C-   00          BRK
500D-   00          BRK
500E-   00          BRK
500F-   00          BRK
5010-   00          BRK
5011-   00          BRK
5012-   00          BRK
5013-   00          BRK
*
```

Example of memory all zeroes. Every line is BRK which is the assembly mnemonic for 00 hex byte. Not marked.

```
6000L

6000-    FF          ???
6001-    FF          ???
6002-    FF          ???
6003-    FF          ???
6004-    FF          ???
6005-    FF          ???
6006-    FF          ???
6007-    FF          ???
6008-    FF          ???
6009-    FF          ???
600A-    FF          ???
600B-    FF          ???
600C-    FF          ???
600D-    FF          ???
600E-    FF          ???
600F-    FF          ???
6010-    FF          ???
6011-    FF          ???
6012-    FF          ???
6013-    FF          ???
*
```

Example of memory all hex $FF. Notice the large number of ?? meaning the APPLE disassembler cannot decode this section. $FF is not a valid opcode instruction so the disassembler prints ??

```
]CALL -151

*A900L

A900-    55 CE       EOR   $CE,X
A902-    56 45       LSR   $45,X
A904-    52          ???
A905-    49 46       EOR   #$46
A907-    D9 00 21    CMP   $2100,Y
A90A-    70 A0       BVS   $A8AC
A90C-    70 A1       BVS   $A8AF
A90E-    70 A0       BVS   $A8B0
A910-    70 20       BVS   $A932
A912-    70 20       BVS   $A934
A914-    70 20       BVS   $A936
A916-    70 20       BVS   $A938
A918-    70 60       BVS   $A97A
A91A-    00          BRK
A91B-    22          ???
A91C-    06 20       ASL   $20
A91E-    74          ???
A91F-    22          ???
A920-    06 22       ASL   $22
A922-    04          ???
*
```

Not program code. Note large amount of repeating BVS instruction. Several ??? are present. If a lot of ??? are present then the section is probably not valid code. Use the ASCII search or display to see if there are menus or prompts here. Also several BRK are present then code is probably not valid.

A950L

```
A950-   02          ???
A951-   01 C0       ORA   ($C0,X)
A953-   A0 90       LDY   #$90
A955-   00          BRK
A956-   00          BRK
A957-   FE 00 01    INC   $0100,X
A95A-   00          BRK
A95B-   02          ???
A95C-   00          BRK
A95D-   01 00       ORA   ($00,X)
A95F-   07          ???
A960-   00          BRK
A961-   01 00       ORA   ($00,X)
A963-   FF          ???
A964-   7F          ???
A965-   00          BRK
A966-   00          BRK
A967-   FF          ???
A968-   7F          ???
A969-   00          BRK
*
```

Not program code. Note several ??? and many BRK

BD00L

```
BD00-   84 48       STY   $48
BD02-   85 49       STA   $49
BD04-   A0 02       LDY   #$02
BD06-   8C F8 06    STY   $06F8
BD09-   A0 04       LDY   #$04
BD0B-   8C F8 04    STY   $04F8
BD0E-   A0 01       LDY   #$01
BD10-   B1 48       LDA   ($48),Y
BD12-   AA          TAX
BD13-   A0 0F       LDY   #$0F
BD15-   D1 48       CMP   ($48),Y
BD17-   F0 1B       BEQ   $BD34
BD19-   8A          TXA
BD1A-   48          PHA
BD1B-   B1 48       LDA   ($48),Y
BD1D-   AA          TAX
BD1E-   68          PLA
BD1F-   48          PHA
BD20-   91 48       STA   ($48),Y
BD22-   BD 8E C0    LDA   $C08E,X
*
```

Good program area, 35 decimal bytes gives 20 lines of code. This comes out to 146 lines of code per 256 byte page. With sensitivity set at #92 or above this page would be marked if the rest of the page was similiar to this.

```
*1393L

1393-   20 6C 8A    JSR   $8A6C
1396-   4C 44 14    JMP   $1444
1399-   20 6F 8F    JSR   $8F6F
139C-   97          ???
139D-   0F          ???
139E-   20 72 8C    JSR   $8C72
13A1-   20 DF 8A    JSR   $8ADF
13A4-   20 9B 8F    JSR   $8F9B
13A7-   F5 0A       SBC   $0A,X
13A9-   20 79 8A    JSR   $8A79
13AC-   A9 01       LDA   #$01
13AE-   8D A9 0A    STA   $0AA9
13B1-   4C 44 14    JMP   $1444
13B4-   20 6F 8F    JSR   $8F6F
13B7-   E8          INX
13B8-   0F          ???
13B9-   20 72 8C    JSR   $8C72
13BC-   20 DF 8A    JSR   $8ADF
13BF-   20 6F 8F    JSR   $8F6F
13C2-   AB          ???
*
```

Tricky example of good code. There are many ??? in here but the rest of the printout looks good. This is an example of data passing information to a subroutine. The address of the data is on the stack. The data passed is not valid instruction opcodes. The subroutine uses the data and then returns to the calling routine with an adjustment for the data.

There are 48 bytes used giving 108 lines of code per 256 byte page. Any sensitivity above $68 would mark this page.

INTENTIONALLY LEFT BLANK

TIPS ON COPYING AND PACKING BINARY FILES

The CRACK-SHOT card will allow you to copy many different programs for backup, quick restart, and packing to binary files. There are a couple of tips for operation of the card. We will go over some of these here.

Remember that the CRACK-SHOT card will copy total load programs only. This does not limit you from programs as say the word processors. You could copy the editor program which will let you build and edit files. You could then copy the printer program which will access your text file for printing. Most word processors store your files in text files on normal DOS 3.3 disks. Keep examples like this in mind.

Some programs may not boot with the CRACK-SHOT card in a slot and enabled. The transparency switch should be set down for programs like this. Then when the program is in memory and executing you may turn the switch up. Some programs still look for the card while executing. It may be necessary to turn the transparency switch up and then immediately copy the program to stop it from finding the card. This may not work in all cases. Some programs at certain times do nothing but look for a funny card such as CRACK-SHOT and scan the keyboard for a input. The procedure for these programs is to copy them when they are executing a function such as a game or computation. These are time intensive actions and the programs generally are not looking at the slots at that time.

There are several considerations to keep in mind when packing a binary file. First is the CRACK-SHOT copy itself. You want to make the CRACK-SHOT copy at the best time for later restart.

The largest binary file you can load in from a disk is about $9A hex pages long. This is because you can only load into memory not taken up by DOS. To pack programs that are extremely long you may not be able to pack the HIRES page in with the code and data. This means that when the program starts the HIRES page is blank or full of garbage. A nice fix for this is to copy the original program in the act of 'refreshing' or 'redrawing' the HIRES page. Most all programs at some time or another will erase and redraw the HIRES page. Each program is different but find the sequence or command that will force this action. Then right as you issue the command hit the switch for copy. Timing is tricky, it might take a couple of tries.

The benefit here is that when you pack the binary file you don't need to include the HIRES page. It will be redrawn for you by the program when it restarts.

INTENTIONALLY LEFT BLANK

PACKING EXAMPLES

Here are two examples for packing a binary file. The second example uses the language card to store an extra module. The first goes through the steps for packing the program LOCKSMITH. There is a command file for this program but we will go through the entire sequence of making a copy, analyzing code, packing a binary copy, and creating a command file.

The first thing you would do is to make a copy with the CRACK-SHOT card. This gives you a copy in CRACK-SHOT format. You could execute this copy with the CRACK-SHOT card but you would like to put it into a binary file for easier use and storage. Boot the PACKER program and put in the copy disk. Now for a little thinking.

This program is a bit copier. It has large sections of code for analysis and operation. It also has large buffers for reading in data from disk for analysis and storage back out. The buffers do not need to be copied. They are not necessary. Programs that have buffers are bit copiers, communications programs (for modems), word processors, spelling correctors....etc. You do not need to copy the buffers of these programs, only the code. Other areas that need not be copied are the hires pages of APPLE MEMORY for programs that use the hires pages such as plotters, games etc.... As mentioned above if the user copies the program at the correct time the hires pages need not be copied. If you did not copy a program at the proper time then the hires pages need to be copied because the program is executing on restart and the hires page is full of garbage. The hires pages take up a large block of memory and if you copy them your binary file is growing rapidly in length. If the program you copied has a large code and data segment you may not be able to fit in everything. An alternative for language card users is shown later.

For now we want to find all the code areas of the LOCKSMITH program. The first thing to do when starting analysis on a program is to use option 7. This will access all the program stored on disk and do a code and ascii search and mark. For more description of this option see the documentation. When this option completes the computer will show you a list of all page it considers valid program or data. It is recommended to enter a printer address and obtain a hardcopy of this list. With this list you will proceed to analyze the memory and select the pages to store to a binary file. The PACKER program shows code from 0-$20 and $80-$C0. This is not readily evident from the list output but let's look at it. There are several broken blocks above $80 but a large amount of code is shown. The same can be said about the section between $0-$20. The broken sections are simply code

pages with a little different number of assembly lines than other pages. There is code stored there. The means of finding this out is to use options 2 and 3. Option 2 will load any section of memory you desire into the buffer. Option 3 will disassemble any part of memory. Use this to analyze the buffer. Load a section of memory from tracks $C to $10. This corresponds to the memory from $7A to $AB. Now use option 3 to look at memory starting at logical $7F00. You will see that as logical $8000 is displayed valid program code is shown. From there up to $BF00 needs to be copied. The same can be done for lower memory.

The hardest part of a program to find and include is the data areas. There is no valid code in these areas and to disassemble them will do little good. The PACKER program will try to detect these areas by analyzing the valid program code found. The data areas will be accessed by the program code. When PACKER finds a valid program code area it looks for any 16 bit indexed addressing and marks any pages found (this addressing mode is used to access data). There are other addressing modes that access data and the largest used is called zero page indexed. The PACKER program cannot easily spot this type, read an assembly manual and use option 3 for this. PACKER will also mark any pages accessed by a JSR or JMP, subroutine call or goto statement. Almost all code and data areas are found. These pages are marked. This indexed search is where stray pages will show up in the marked display.

The best procedure to take is to obtain a working packed file and then if desired try to reduce it further by taking out the pages not needed. Therefore pack all the pages you can that are marked. If there are a couple of skipped pages between 20 or 30 marked pages include them also.

For LOCKSMITH the hard code areas were packed into a binary file. This file was executed and worked. Options 2 and 3 were used to look at the memory and a decision was made to pack $0-$1F and $80-$BF. The procedure for packing is below.

Use option 1 to initialize the STORE. Set DEST to $08 and REDEST to $20. Page $20 is not going to be copied so put the RERUN program there. For the screen setup use text, page 1, all text. Do not clear the hires screen on startup. The program will return to the menu.

Use option 2 to read in tracks 0-4. Then use option 8 to add to the store. Hit 8 and then $6D for start and $84 for end physical addresses. Hit option 2 and read tracks $0C-$10. Now use option 8 again. Enter $6D for start and $98 for end physical addresses. Now hit option 2 and read in tracks $11 and $12. Use option 8 and enter $84 for start and $97 for ending physical addresses. That completes the packing!

Now use option A to save the packed file to disk. When asked for extra module (if using language card version) hit a return, only use this in cases shown below. PACKER will prompt for a name to store the file under. Also given is the command to execute this stored file. ***NOTE THIS COMMAND. YOU MUST USE THE ADDRESS EXTENSION.*** The command will be :

BRUN 'NAME',A$XXXX

Where the XXXX is the address to start loading in the binary file.

After you have saved your file you could type E to execute the current STORE. This will test your packing. See option E in PACKER documentation.

Test the file out by execution. If it works you are ready to build a command file. Run COMMAND FILE CREATE.

The first question asked is for the name of the command file. To identify these it is recommended to use a C. prefix. It is not necessary though.

Then the program proceeds to ask you the questions needed to build the command file. All that is needed can come off a hardcopy listing if you use option 9 in PACKER once you have packed the file. Option 9 shows the values of DEST, REDEST and all the modules in proper order of inclusion.

The create program asks for these values. The only other questions are for screen setup. Remember text page 1 was used. Once you have entered these parameters insert a DOS 3.3 disk to store the command file to.

One note....when entering the modules exclude the lowest module of $0-$08. This is always entered by the program.

Now you have a command file to pack the program at any time with minimal user action.

EXTRA MODULES WITH LANGUAGE CARD

There are two command file modules given for use with a language card. These will pack a high module and low binary program for any CRACK-SHOT copy. The procedure is as follows.

Run packer, load command file 'C.EXMOD $40-$60. Insert your CRACK-SHOT copy disk and hit return. When packing is finished save this file under namehigh, name is anything. This packs a high language card module containing pages $40-$5F. When asked for extra module hit return, this ***is*** the extra module.

Now load command file 'C.NO HIRES 1/2. This will pack all memory except from $20-$5F. Now with the language card module you can have all memory except $20-$3F. This is a very large relative size file of $A0. Once the command file is loaded insert your CRACK-SHOT copy disk and hit return. When packing is over use option A to save the file. Now when asked for extra module start enter $D0, this is where the module will reside in the language card. When prompted for module end enter $F0, this is the end of the module +1. When prompted for the module destination enter $40.

Save this file as namelow. Exit PACKER and run EXEC FILE BUILDER. This program will create an exec file that will load the extra module into the language card, then load and execute the lower memory binary program namelow.

The first question asked is for the extra module file name, this is namehigh from above. The builder program will load and shorten this file to get rid of the lower 8 pages automatically packed. Then the builder programs asks for the name of the lower memory file, namelow from above. Last the program will ask for the starting address DEST of namelow. The command file used above set DEST to $08. The user can find this out with option 9 in PACKER.

Insert a DOS 3.3 disk and the program will store the exec file when you give it a name to save it under. To execute this file you would then type from basic:

EXEC NAME

You now have a matched set of 3 files that must be present to execute, a language card must be in the system for this option to work.

The following names are copyrighted or trademarks of companies.

APPLE--APPLE COMPUTING COMPANY

D.C. HAYES MICROMODEM--HAYES MICROCOMPUTER INC.i

LOCKSMITH, THE INSPECTOR, WATSON--OMEGA MICROWAVE,INC.

NIBBLES AWAY II--COMPUTER: applications INC.

APPENDIX F

TROUBLE SHOOTING

When reading CRACK-SHOT disk program hangs.

Probably a bad copy. The disk read routine

cannot find valid data. If it cannot sync
in at all it will hang. Recopy.

Command file give file to long error.

When made the command file was incorrect or..
using a command file meant for language card
packer program and currently running lower
memory packer. The max pack size is different.

System crash when try to copy with card.

Cannot have ROM or RAM card in system when
copy made. Also some cards use same expansion
memory as CRACK-SHOT. Cards known are D.C. HAYES MODEM
some video cards also mapped into same memory.

ERRORS DURING PACKER PROGRAM EXECUTION

DISK I/O ERROR
FUNCTION NOT DONE

An error occurred during an APPLE DOS command, such as catalog or bsave (issued in option A). The requested command was not performed. Either not a DOS 3.3 disk or a blown copy of DOS 3.3.

ERROR OCCURRED, ATTEMPT TO OVERWRITE SELF.

This error happens when the user attempts to pack a binary file in such a manner that a module relocates itself over another unused module. This error can appear in either option 8 or A, it is checked for at both places. See documentation on options 1,8 and A for more on module placement.

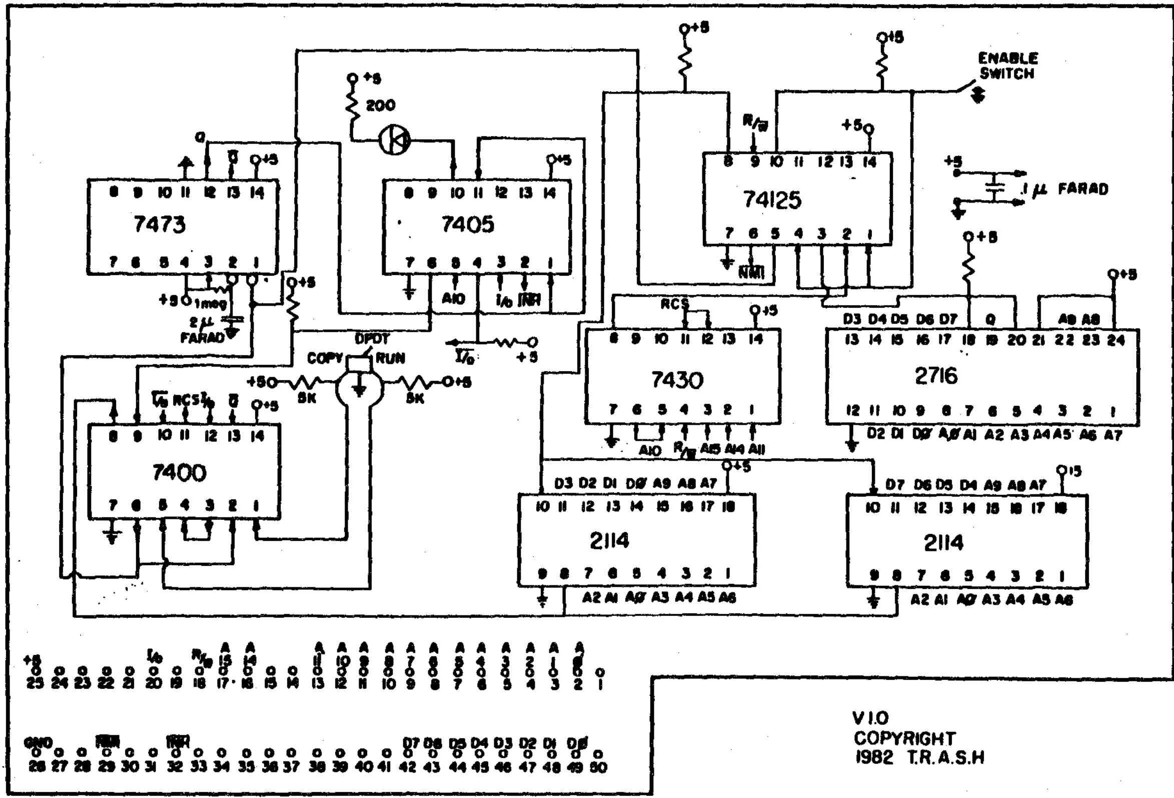
ERROR OCCURRED, ATTEMPT TO OVERWRITE RERUN.

Error happens when the user adds a memory module to the store that at execution time will relocate itself over the RERUN program. This is not allowed as RERUN is the master relocate program and cannot be overwritten. Pick a page in memory for RERUN that does not need to be copied. See options 1,8 for more information.

I/O ERROR WHEN BRUN PACKED FILE

The PACKER program assumes uses of all available memory when the packed binary file is executed. The binary file could extend up to $9A00 hex which is in DOS buffer area. Issue command MAXFILES 1 if you get an I/O error when executing a binary packed file.

G. SCHEMATIC



All resistors unless marked are 5k

INTENTIONALLY LEFT BLANK

APPENDIX H

ADDITIONAL INFORMATION

It is important to **stress** a part of PACKER. When using option A in PACKER to save the packed file the PACKER program will show you the *exact* command to give to execute that file. Right before you enter a name to save the file under the program will print 'execute as BRUN NAME,A$????'

This is the command needed to execute the saved file. What is important to include is the address extension, the ',A$????'. This tells DOS where to load and execute.

The D.C.Hayes Micromodem card has been giving problems on CRACK-SHOT operation. The memory used by the modem card is mapped into the same area as CRACK-SHOT and will have to be removed when using the CRACK-SHOT card.

MODIFIED PACKER PROGRAM

One more version of the PACKER program is supplied on disk. It is not in the menu and must be run seperately.

This packer is a language card version the same as in the domumenation but with a couple of small changes. In some cases of packing it might be advantageous to have full control of packing. In this version the lower 8 pages are **not** packed automatically. The user has full control of placing DEST and REDEST. They both can have the same value now. The user must specifically pack the lower 8 pages. They can be copied from:

$02-$07 on track 1 of CRACK-SHOT copy.
$0,1 on track $13 " "

page $4 on track 1 is not totally valid, it was modified some at time of copy. A complete valid copy of page 4 is on track $13, the third page on that track. It is not normally accessed. You cannot pack it by loading that track and moving the third page to the STORE. For this program there is an option * which will load the track $13 itself and put page 4 as a 1 page module in the current STORE.

This version is for more advanced packers. See the documentation on packing program.

To run this version, have a language card in the system and type:

EXEC EXMODPACK

There are two types of switches used on CRACK-SHOT cards. One switch is two position, up and down. A second type switch is three position...up middle down. Both switches operate exactly the same for the CRACK-SHOT system. Ignore the middle position on the 3 position switches. Not all cards have both types.

APPENDIX I

PARAMETERS FOR CRACK-SHOT PACKER PROGRAM

This is the beginning of a parameter list for CRACK-SHOT disks. Programs will be listed by name. Each program will have the necessary pages to copy given. Use the PACKER program to access a CRACK-SHOT copy disk and pack the given pages.

The term LCMOD means the following module is to go into the language card. It will be a seperate file. See enhanced PACKER documentation for language card users.

The * after a program name means supplied by alternate and not tested yet.

The ! after a program means the modules given are in correct sequence for entering into command file create.

NOTE: The following program titles are copyrighted by software companies.

Current date 10/9/82

NORAD ! $8...$1A $20...$33 $34...$65 $90...$9A
DEST=08 REDEST=1C

TORAX * $40...$60 $0F...$20

INVASION FORCE ! $8...$33 $34..$40
DEST=08 REDEST=42

BUG ATTACK * $09...$15 $40...$84

SPACE RAIDERS $20...$84

TWERPS * $5F...$BF $1F...$3B

APPLE PANIC * $8...$20 $60...$C0

FIREBIRD ! use C.NO $20 $50

HORIZON V * $8...$20 $90...$BF

BEER RUN * $8..$20 $60...$8F $A0...$BF

DUET ! $8...$33 $34...$40 $9D...$BF $4C...$4E

DEST=08 REDEST=42

STARBLAZER $8...$1F $40...$9F LCMOD $A0...$BF

CHOPLIFTER $8...$1E $60...$BF

JAWBREAKER $8..$1F $60..$83 $8E..$92 $96..$BF

ROCKET COMMAND $8...$40 $5E..$62

LOCKSMITH 4.1 ! $8..$1F $80..$BF
& NIBBLES DEST=08 REDEST=29
AWAY II

SENSIBLE SPELLER ! $8...$33 $34...$3F $60...$8D $8E...$96
DEST=1F REDEST=42

APPLE LINK ! $8...$33 $98...$A0 $60...$8D $8E...$96
DEST=22 REDEST=42

LASERSILK ! $8...$33 $34...$3F $89...$9F $60...$6F
$70...$83 $84...$88
DEST=8 REDEST=42

CANNONBALL BLITZ ! $8...$1F $98...$BF $51...$79 $7A...$83
$84...$8D $8E...$96
DEST=8 REDEST=20

SEAFOX * ! $08...$1F $96...$BF $8E...$94 $59...$5F $60...$8D
DEST=8 REDEST=95

SCANNER V1.6 * $08...$28

DISK ORGANIZER V2.6 * $08...$24 $30...$31 $37...$3A $80...$8F
DEST=10 REDEST=25

BACK IT UP II+ V2.4 $9F...$BF $80...$8F
DEST=20 REDEST=28

SNOGGLE ! USE C.NO $20 $50

RASTER BLASTER * $0A..$20 $40...$8B00

APPLE OIDS * $08...$1F $40...$80

VISICALC ! $08...$33 $34...$65 $66...$79 $7A...$7F
DEST=08 REDEST=80

STAR CRUISER * $08...$1F $40...$80

VISIDEX * $08...$1F $97...$C0 $60..$61

MAGIC WINDOW * $08...$48 $96...$C0

THE ELIMINATOR ! $90...$AF $08...$1F $40...$6F $70...$83

$84...$86   DEST=08 REDEST=20

CROSSFIRE ! $08...$33 $34...$65 $66...$79 $7A...$7F
DEST=08 REDEST=88

SNACK ATTACK ! $08...$1F $9A...$9A $29...$51 $52...$83
$84...$8D $8E...$97 DEST=08 REDEST=20