QPAK-68 User's Guide

Chapter 3 is based on the "S-C Macro Assembler" manual for the 6502 Macro Assembler, which is copyrighted by S-C Software Corporation. The manual has been enlarged and updated for 68000 operation for inclusion in this manual.

The 68000 Macro Assembler (3 versions) on the diskette are supplied under license from, and are copyrighted by, S-C Software Corporation.

Apple is a trademark of Apple Computer, Inc.

APPLE COMPUTER, INC. MAKES NO WARRANTIES, EITHER EXPRESS OR IMPLIED, REGARDING THE ENCLOSED COMPUTER SOFTWARE PACKAGE, ITS MERCHANTABILITY OR ITS FITNESS FOR ANY PARTICULAR PURPOSE. THE EXCLUSION OF IMPLIED WARRANTIES IS NOT PERMITTED BY SOME STATES. THE ABOVE EXCLUSION MAY NOT APPLY TO YOU. THIS WARRANTY PROVIDES YOU WITH SPECIFIC LEGAL RIGHTS. THERE MAY BE OTHER RIGHTS THAT YOU MAY HAVE WHICH VARY FROM STATE TO STATE.

DOS 3.3 is a copyrighted program of Apple Computer, Inc. licensed to QWERTY, INC to distribute for use only in combination with QPAK-68. Apple software shall not be copied onto another diskette (except for archive purposes) or into memory unless as part of the execution of QPAK-68. When QPAK-68 has completed execution Apple software shall not be used by any other program.

DISCLAIMER OF ALL WARRANTIES AND LIABILITY

Qwerty, Inc. ("Qwerty") makes no warranties, either express or implied, with respect to this documentation or with respect to the software described in this publication, its quality, performance, merchantability, or fitness for any particular purpose. Qwerty software is sold or licensed "as is". The entire risk as to its quality and performance is with the buyer. Should the programs prove defective following their purchase, the buyer (and not Qwerty, its distributor or its retailer) assumes the entire cost of all necessary servicing, repair, or correction and any incidental or consequential damages. In no event will Qwerty be liable for direct, indirect, incidental or consequential damages resulting from any defect in the software, even if Qwerty has been advised of the possibility of such damages. Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you.

WARRANTY

Qwerty, Inc. ("Qwerty") warrants the Q-68 PC card to be free from defects in material and workmanship for a period of ninety (90) days from the date of original purchase for use. Qwerty agrees to repair or, at its option, replace any defective units without charge during this warranty period upon prepaid shipment and receipt of the unit at Qwerty Inc, San Diego, Calif. Qwerty assumes no responsibility for any direct, indirect, incidental, special or consequential damages resulting from any defect in the hardware. Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty is non-transferrable. This warranty is valid when registered within ten (10) days of purchase date. (QPAK-68 Registration Card enclosed). No other warranty, written or verbal, is authorized by Qwerty.

QPAK-68
System Reference Manual

Table of Contents.

Part I. The QPAK-68 System

Section II -- 68000 Design Topics

Chapter 7 -- Hardware Topics

Chapter 8 -- Software Topics

Chapter 1 -- GETTING STARTED

WHAT YOU BOUGHT

The QPAK-68 is your window into the fascinating world of the 68000 microprocessor. It allows you to turn any Apple II, II+ or IIe computer into a full function 68000 development system.

WHAT YOU NEED

You need a 48K Apple, and a single disk drive, as a minimum.

A 16K (language) card (or IIe) is highly recommended, so that you can write large programs without having to save sections of them on disk. Two versions of the 68000 cross assembler are supplied, one for use with a language card, one for use without.

WHAT'S IN THE BOX

First, some books. These are the best books on the 68000 microprocessor we've found.

1. The 68000: Principles and Programming, by Leo Scanlon. Scanlon is one of the best technical writers going. He uses an easy, clear style, and really knows his stuff. This is an ideal first book on the 68000.

s

3. MC68008 16-Bit Microprocessor With 8-Bit Data Bus. Motorola ADI-939.

This is the data sheet for the 68000, the CPU used in the QPAK-68. You hardware types will love the fold-out timing diagram at the rear of the book.

4. The 68000 User's Guide (Motorola).

The standard reference for programming the 68000. Each instruction is described in detail. A bit dry, but very informative.

5. QPAK-68 User's Guide.

This book (which you're reading now) provides installation and operating instructions for the QPAK-68.

Part 1 deals with the system components--editor, assembler, debugger, and Q-68 board information.

Just as we searched far and wide for the best 68000 books, we also looked for and found the best (we think) cross assembler for the 68000. It is written by S-C Software Corp.

This is a great Macro Assembler. It is simple, powerful and easy to use. And, there is a hidden bonus to learning and using this assembler. S-C also makes cross assemblers for other CPU's. You can get them for the 8048, the Z80, the 6809, even the PDP-11. And they're all available at very reasonable cost.

The beauty of the S-C approach is that you learn the operation of the assembler only once. All of their other cross assemblers use exactly the same syntax, data types, pseudo-ops, etc. This means that you can move from CPU to CPU without seeming to change assemblers. This is a sensible and practical approach.

Tucked into this User's Guide is a diskette containing the Macro Assembler. The diskette is not copy protected; back it up immediately and use only your backup copies.

Part 2 of this User's guide contains tutorial information on the 68000 microprocessor. Hardware and software topics are discussed and illustrated with actual 68000 programs. These programs are included with the QPAK-68 diskette so you can run and watch them yourself.

6. An Apple II hardware board, called the Q-68. This is the magic part of the package. It adds a 68008 processor to the Apple computer, and allows you to actually run the code you developed using the Macro Assembler. To aid you in finding program bugs, the board also includes a firmware monitor/debugger (in EPROM) which allows you to see exactly what's happening inside the 68008.

7. A pocket guide to the 68000 instruction set. If this little book is not dog-eared after about 6 months, you aren't making full use of your QPAK-68 system!

8. A 68000 folding programming card. The official reference card from Motorola. Put this card in your shirt pocket and impress your friends.

INSTALLATION AND STARTUP

1. Look for the box with the Q-68 board in it.

2. Turn off your Apple II and remove the cover. You need an Apple II, II+ or IIe with 48K of RAM, and a disk. Although you can run the system with or without a language (16K RAM) card, you should have one if you plan to write and assemble large programs. If you have a IIe, the "language card" is built-in.

3. Insert the Q-68 board into a vacant slot. The system is initially configured for operation in slot 4, the fourth slot from the RIGHT. If you already have something in slot 4, use another slot and remember which one for step 6.

4. Leave the cover off, so you can see the two lights on the Q-68 board. Locate the diskette inside the cover of this book.

5. MAKE A COUPLE OF BACKUP COPIES OF THIS DISKETTE. We did NOT copy protect this diskette. Always use backup copies for your work, never the original. The "COPYA" program on your DOS 3.3 System Diskette works fine for making a copy.

6. Put the diskette in drive 1, and power up. The QPAK-68 system automatically loads, and you are given five

choices from an onscreen menu.

If you didn't use slot 4 for the Q-68 board, choose menu item 1 and reconfigure the system for your particular slot. When the slot shown at the top of the screen matches the slot your Q-68 card occupies, select item 6.

If you have a language card or a IIe, select item 4 from the menu, "RUN ASSEM. AT $D000". If you don't have a language card, select item 6.

NOTE: Item 6 is not shown on the menu. It loads a special version of the assembler at $4000 for this demonstration only.

This special version allows short programs which use HIRES screen #1 to run without the language card version of the assembler (the language card version avoids the HIRES memory).

If you do not have a language card and wish to write programs that use the HIRES screen #1, you can use this version of the assembler.

7. After about 20 seconds, you should be on the air with the EDITOR/ASSEMBLER. You'll know you are in the EDIT mode by the colon (:) prompt.

   Type LOAD DEMO1, followed by RETURN. (All commands are followed by RETURN).

8. Type LIST, RETURN, to see a real live 68000 program.

9. Back at the (:) prompt, type ASM, RETURN. This assembles the DEMO1 program. Quick, isn't it?

10. Now type QON, RETURN. This starts the 68008. The red light on the Q-68 board should be ON. Do you see wild gyrations on your Apple II screen? If you do, all is well. If you don't, something is wrong. Go directly to Chapter 5, "Self-Test".

    Notice that the four line text window at the bottom of the screen is still active. The 68008 is running the top of the screen, and the Apple's 6502 is running the text window, both at the same time.

11. Type MNTR, RETURN. This takes you to the Apple monitor, with the familiar (*) prompt.

12. Type CØC3, RETURN. You should hear a "ZZZip" sound, and the screen should stop cycling during the sound. If this happens, you have successfully activated the Q-68 interrupt. Don't worry about the numbers shown at the bottom of the screen. The Apple has read location $CØC3, and found meaningless information there. The act of simply accessing this memory location does the job of interrupting the Q-68 board.

13. Hit the RESET key (ctl-RESET on the IIe). This stops the Q-68 board, and takes you back to the EDITOR.

14. Now type DBUG, RETURN. This starts the Q-68 board again; this time running the DEBUG program out of onboard EPROM.

    Use the right and left arrows to see the five DEBUG screens.

You're in business!

If you didn't get the expected results, proceed directly to Chapter 5, "Self-Test".

Chapter 2: The Q-68 BOARD

INTRODUCTION

The Q-68 board adds a second processor, the 68000, to the Apple II computer. It plugs into any slot (#1-#7) and shares the 64K memory on the Apple's motherboard. If the Apple II has a 16K language card, this additional memory is also accessible by the 68000.

The 68000 runs simultaneously with the Apple's 6502 on a cycle sharing basis. Whenever the Q-68 board needs access to an Apple resource (such as the graphics screen or keyboard) it lengthens the 6502 clock and inserts a memory cycle of its own.

The Q-68 board is controlled--turned on, turned off and interrupted--by the Apple II computer.

The Q-68 board makes no distinction between User and Supervisor states. This means that you can use either operating mode, and switch freely between them.

The 68000 is capable of addressing a megabyte of memory. It is convenient in this application to consider the megabyte as sixteen 64 Kilobyte areas.

MEMORY MAP

The complete memory map for the 68000 on the Q-68 board is shown on the following page.

$F0000-$FFFFF	Open
$E0000-$EFFFF	.
$D0000-$DFFFF	.
$C0000-$CFFFF	.
$B0000-$BFFFF	.
$A0000-$AFFFF	.
$90000-$9FFFF	.
$80000-$8FFFF	for
$70000-$7FFFF	.
$60000-$6FFFF	.
$50000-$5FFFF	.
$40000-$4FFFF	.
$30000-$3FFFF	.
$20000-$2FFFF	options
$18000-$1FFFF	Onboard RAM, 2 Kilobytes (expandable to 8 Kilobytes)
$10000-$17FFF	Onboard EPROM, 8 Kilobytes (expandable to 32 Kilobytes)
$00000-$0FFFF	Apple II memory

The lowest 64-Kbytes of 68008 address space is occupied by all the memory in the Apple II. The next 64-Kbyte space is occupied by memory on the Q-68 board. Locations $10000 through $17FFF are occupied by an onboard 8-Kbyte EPROM, and locations $18000 through $1FFFF are occupied by an onboard 2-Kbyte RAM. Both of these memory spaces "wrap around". For example, the 2-Kbyte RAM can be addressed starting at $18000, $18800, $19000, and so on up to $1F800 (sixteen places).

If larger memories are installed on the Q-68 board (for example, an 8 Kilobyte RAM or a 16 Kilobyte EPROM) the starting addresses are the same as for the smaller memories.

The other fourteen 64K areas of 68008 memory are free for expansion. An expansion connector on top of the board is provided for interface to future options such as memory and video boards.

We need to take a careful look at the bottom 64K of memory. This is the Apple memory, which is shared by the 68008. When you run the Macro assembler, it puts 68008 object code here.

Both the 6502 and the 68008 can read and write this low memory. We will thus refer to the shared memory as being accessed from the Apple (6502) side and the Q-68 (68008) side. When you do an assembly of 68008 code, it is put into the memory from the Apple side. When you actually run the code by turning on the Q-68 board, it is accessed from the Q-68 side.

LOW MEMORY IS IMPORTANT TO THE 68008

The low memory for the 68008 has special significance. All of the exception vectors reside on pages 0 through 3 ($000000 through $0003FF). We'll talk a lot more about exceptions, but for now let's just say that important information required for 68008 operation is stored in these bottom three pages.

For example, when the 68008 first turns on, it fetches two 32 bit values from locations 0 and 4 (the 68008 addresses memory in bytes, so a 32 bit word spans four bytes). The 32 bit contents of locations 0-3 are loaded into the SSP (system stack pointer), and the 32 bit contents of locations 4-7 are loaded into the PC (program counter). 68008 operation then begins at the address held in the PC.

Take a look at page 5-4 of the MC68008 data booklet supplied with your QPAK-68. This Vector Table shows what the 68008 expects to find in the bottom 1K of memory. If you are curious about these functions, refer to the application section on "exceptions" in Section 7.1 of this manual.

LOW MEMORY IS ALSO IMPORTANT TO THE 6502

The low memory (bottom 256 bytes) of the Apple II also has special significance. Page 0 is the only section of 6502 memory which can hold memory pointers which are used for indirect addressing. Also, special 6502 instructions can access "page zero" locations faster than elsewhere in memory. For these reasons, all Apple II software, including the monitor, DOS, and Applesoft BASIC make heavy use of these precious 256 bytes on page 0.

Page 1 is used by the 6502 to hold the stack. Page 2 is used by the monitor as a keyboard buffer. Page 3 is used to

store 6502 vectors--addresses of routines for handling the RESET key, and the BRK instruction, for example.

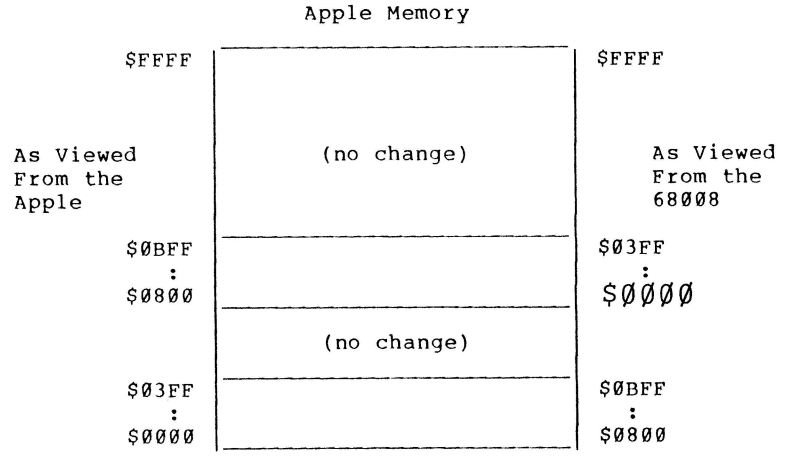
THE DOUBLE PAGE ZERO DILEMMA

We seem to have a problem here. In order to prepare for turning on the 68000, the Apple (actually the Assembler, running in the Apple) must load certain values into low memory. But loading the values necessary for 68000 startup at locations 0-7 (and others on page 0) would interfere with proper operation of the Apple. For this reason, special circuitry on the Q-68 board moves the bottom of 68000 memory out of the bottom of the Apple's memory map.

Specifically, it swaps pages $0-$3 with pages $8-$B. This means, for example, that to put a number at 68000 location $0004, the Apple writes it at location $0804. The fact that these memory spaces are swapped means that the assembler can also access 68000 pages $8 through $B--by writing pages $0 through $3.

All other memory is left alone. Location $2ABC in the Apple corresponds to location $02ABC in the 68000, for example.

The following diagram shows this memory swapping in detail.



Note that the right half of this diagram shows that writing 68008 locations $800 through $BFF corresponds to Apple pages $0 through $3. Using these 68008 locations is definitely not recommended due to the overlap of the critical Apple low memory. If you really need to put data there, you're on your own as far as the Apple is concerned.

Here are some practical examples of how this memory swapping works.

Every 68008 program requires that the RESET vectors be initialized. The diagram above shows that 68008 page 0 corresponds to Apple page 8. So code of the following form is required somewhere in the 68008 source. Keep in mind that this is code "as viewed by the Apple", since it is generated by the cross assembler. The vector addresses shown are for example purposes--they can be anything.

```
.OR     $800            ;Locates this code at 68008 PAGE 0
.DA     $00005000       ;SSP (Supervisor Stack Pointer)
.DA     $00001000       ;PC--PROGRAM START ADDRESS
```

The .DA directive means, "store the numeric constant in memory". This code establishes the SSP at $5000, and begins execution at $1000.

If a routine is to be called as an exception, its address is placed at the proper place in the 68008 page 0 vector table. For example, the routine whose address is placed at Apple location $880 (68008 location $80) will be executed whenever a TRAP #0 instruction is encountered. If you use the QPAK-68 system disk to start up the Q-68 board, you don't need to worry about installing the startup vectors. The system does this for you. Appendix D shows you how this is done.

THE WATCHDOG TIMER

In the course of writing and debugging machine code, it is possible that you will inadvertently attempt to access memory which does not exist in the Q-68 system. If no special measures are taken to allow for this possibility, your program can hang up indefinitely. What actually happens is that the 68008 patiently waits for a DTACK signal (Data Transfer Acknowledge) which never comes. See Section 7.3 for more information about DTACK.

The Q-68 board incorporates a special timer which detects an attempted access to nonexistent memory, and activates the BERR (Bus ERRor) signal on the 68008 if this happens. This triggers an exception sequence, which you (and the supplied DEBUG program) can use to recover from the error.

If a memory does not respond within 180 microseconds after being accessed, the watchdog timer trips a Bus Error exception. When this happens, the 68008 stacks the PC, Status Register, and other processor status information, then jumps to the address held at 68008 location $00008-$0000B. (These addresses correspond to $808-$80B in the Apple II).

If you have an application which calls for a longer response time than 180 microseconds (an example might be a video system where the CPU waits for vertical blanking before updating the video memory), the watchdog timer can be disabled by removing jumper plug #2 on the Q-68 board.

SYSTEM TIMING

The 68008 is clocked by the Apple II's internal 7.16

Megahertz clock. Depending on whether or not the 68000 is accessing its "own" (not Apple's) memory or not, the 6502 processor in the Apple II runs at two different speeds:

> When the 68000 accesses memory in the Apple II memory space (the bottom 64K), the Apple's 6502 runs at half speed.
>
> When the 68000 accesses memory outside of the bottom 64k, the Apple's 6502 runs at full speed.

There is a simple way to gauge how much of the 6502's processing time is being "robbed" by the 68000. Simply hit "CTRL-G" to sound the Apple's beep. If the familiar high pitch beep is heard, the 68000 is not using any Apple II memory. If the pitch is lower, the 68000 is getting into the Apple memory at least some of the time. If the beep is an octave lower than normal, the 68000 is running continuously out of the Apple II memory.

As for the 68000, it too runs approximately twice as fast when running out of its own memory than when running out of the Apple II's memory.

Two simple programs on the supplied demonstration disk show this speed difference. After loading the Macro Assembler, type "LOAD VIDEOTEST" (follow all commands with a RETURN). To see the program, type LIST. This program continuously increments all values in the Apple HIRES screen. Now type ASM (assemble), and when the assembly is finished, type QON. The HIRES screen should cycle continuously.

Now hit CTRL-G and listen to the tone. Hear the difference? The tone is an octave lower than normal, because the Apple's 6502 and the Q-68 are running simultaneously.

Take a look at the screen and get a rough idea of how fast the display is changing.

Now stop the 68000 by hitting RESET on the Apple.

Next, type LOAD MEMOVE. If you LIST this program, you'll see the VIDEOTEST program, with a section of code added to the beginning. This added code moves the entire VIDEOTEST program into the 68000's onboard RAM, and then jumps to it. The program is thus identical to the VIDEOTEST program, but it runs outside of the Apple memory.

Type ASM, then QON. See the difference on the screen? The display really moves now. Now type CTRL-G and listen.

There's the familiar tone, but higher than when you ran VIDEOTEST. If you have a sharp ear, you might notice that it still is not as high in pitch as the normal Apple beep. This is because MOST of the 68000 memory accesses are coming out of the onboard RAM (remember, this is where the program was moved to). But every time the Apple's HIRES video screen is accessed, the Apple shifts to half speed.

Before, in the VIDEOTEST program, all memory used by the 68000 was in the Apple. The program executed out of Apple memory, and the video "writes" were made to Apple memory.

In MEMOVE, the program part is moved out of Apple memory and into the Q-68 board RAM. So the pitch of the Apple's beep is lowered only by the proportion of the program execution time that the 68000 uses Apple memory. This happens for video memory updates only.

If the 68000 uses none of the Apple memory, both the 6502 and the 68000 run at full speed (1 Megahertz for the 6502; 7.16 Mhz for the 68000).

MORE USES FOR Q-68 MEMORY

The RAM on the Q-68 card can be used to move programs out of the Apple memory space for faster execution, as demonstrated above. Two other uses are recommended for the Q-68 RAM.

If you wish to do stack operations with your 68000 program, you need to locate the 68000 stack somewhere in memory. The perfect place for it is in the Q-68 RAM, since it does not get in the way of any Apple II code, and stack operations execute quicker here than out of the Apple memory.

If you are using DEBUG, you can use the top two pages of Q-68 Board RAM for anything you wish. These two pages are at addresses $18600-$187FF. A good place to put a stack would be $18800. The stack works downward in memory as it is pushed; upward when popped. DEBUG places its internal stack at $18600.

Your 68000 program will probably use "scratch" locations to store variables. The Q-68 board RAM is a good place to put them.

If you don't need DEBUG, the Q-68 EPROM can be used to store your "final" routines in permanent form. Again the two

"onboard" advantages are realized--faster execution and noninterference with Apple programs.

Q-68 BOARD USER OPTIONS

There is a three position jumper block (marked "TB1") on the upper right corner of the Q-68 board. The board is shipped with little jumper plugs at positions #2 and #3. This section explains the purpose of these jumpers.

Jumper position #1 disables access to the Apple memory if a jumper plug is installed. This is provided for service reasons. Chapter 5 describes a test technique which involves temporarily installing a shorting plug at this first position. For normal operation, jumper #1 should be left open (no shorting plug).

Jumper #2 connects the watchdog timer to the 68008 BERR (Bus Error) input. Its shorting plug should be left in place for normal operation. If you wish to deactivate the watchdog timer, remove the jumper. But remember that any attempted access to nonexistent memory will hang up the system if jumper #2 is not there.

Jumper #3 maps the Q-68 board for operation in the QPAK-68 system. The jumper plug should be in place for normal operation. Removing this jumper plug swaps the Apple II memory space with the onboard ROM/RAM. Without jumper #3, the onboard ROM occupies locations $00000-$0FFFF, and the Apple memory occupies $10000-$1FFFF. Why do this?

There is a short diagnostic routine at the beginning of the onboard EPROM. If you are trying to isolate a fault in the Q-68 board, and there is a problem in the Apple bus timing, you will never be able to start up the board because the RESET vectors are in the Apple RAM. So this jumper allows you to fire up the system without access to the Apple Bus, and run the diagnostic program out of EPROM.

Another reason, which might be used for a special application, is to allow the board to run without using any Apple memory. You could code a custom routine into the onboard EPROM, and by removing the third jumper start up the 68008 using the EPROM rather than the Apple memory.

X1-X2 JUMPER

The 28-pin socket at location "C2" has a 24-pin IC installed. This is the Q-68 board's 2 kilobyte RAM.

This socket will also accept an 8 Kilobyte RAM (Hitachi HM6264-15 or equivalent). If this part is installed, the jumper at X1 must be cut, and the jumper at X2 must be connected.

X3-X4 JUMPER

The 28-pin socket at location "B2" accepts a 2764 or 27128 EPROM (Intel compatible) with no alterations. The socket can also accept a 27256 (Intel compatible) or ROM equivalent by cutting X3 and bridging X4.

EXPANDING THE Q-68 BOARD

All signals required for 68008 expansion are available on a 50-pin connector at the top of the Q-68 board. These signals include the 20-bit address bus, the 8-bit data bus, and control signals. The following table shows the pin assignments for this connector.

NOTE: Signals with an asterisk are pulled up to +5 volts through a 1000 ohm resistor.

1		A3	2		INTACK/
3		A4	4		A2
5		A5	6		A1
7		A6	8		A0
9		A7	10		FC0
11		A8	12		FC1
13	*	IPL0-2/	14		FC2
15	*	IPL1/	16		A9
17	*	BERR/	18		A10
19	*	VPA/	20		A11
21		A12	22		E
23	*	RESET	24		A13
25	*	HALT/	26		A14
27		GND	28		GND
29		A16	30		A15
31	*	BR	32		A17
33		BG/	34		E0000/
35		A18	36		Watchdog Timer Output
37		R/W	38		A19
39		DS/	40		D7
41	*	AS/	42		D6
43		D0	44		D5
45		D1	46		D4
47		D2	48	*	DTACK
49		D3	50		N.C.

NOTE: The following signals are not direct 68008 signals:

1. E0000/. This is an address decode of $E0000-$EFFFF, conditioned by AS/.

2. Watchdog Timer Output. This is the output of a divide-by-256 counter which is clocked by the 68008 "E-clock" and reset by DTACK/ going high.

3. INTACK/. This is the NAND'ed combination of FC0, FC1, and FC2.

Signals shown with a "/" are active low.

The six 68000 input signals IPL0-2/, IPL1/, VPA/, RESET/, HALT/, and BERR/ are driven by Q-68 board open-collector drivers and 1K pullup resistors. This allows external circuitry to share these 68000 pins.

POWER CONSUMPTION AND FANS

The Q-68 board is specified to use 400 milliamps (max) at 5 volts. The 5 volt supply is the only one used by the board.

The Apple II manual says that 500 milliamps (1/2 Amp) of 5 volt power is available for all of the expansion slots combined. We have found this spec to be extremely conservative. Most Apple II's that we've seen (including ours) are well stuffed with peripheral cards--disk controller, printer cards, serial cards, 80 column cards, etc. A four or five card complement of peripheral cards can easily consume upwards of 2 Amps, and run without problems.

We've found that the real worry is not current consumption, but HEAT.

Beyond about three or more peripheral cards, you should get a fan for your Apple II. Many are available at computer stores, from simple adhesive foam installation types (with on-the-cord lamp type power switches) to side mounted units which contain power line filters and convenience power outlets. These side units are highly recommended.

By the way, if you touch the 68000 on the Q-68 board, you'll feel where most of the board's power is consumed. The 68000 runs very hot. Don't worry, this is normal--the 68000 is specified to 70 degree Centigrade operation.

Editor/
Assembler

Editor/ Assembler

Chapter 3 -- THE MACRO ASSEMBLER

Section 3.1 -- INTRODUCTION

THE MACRO ASSEMBLER

The Editor/Assembler provided with the QPAK-68 is written by S-C Software Corporation. It is a powerful macro assembler, which runs under the Apple II Disk Operating System.

We refer to the S-C package as the "MACRO Assembler" throughout this manual. To be technically correct, it is actually an EDITOR/Macro Assembler. The EDITOR program is built into the package. This is one of the best features of the S-C program. You don't need to shuttle diskettes back and forth as you load an EDITOR program, edit, save source, load ASSEMBLER, assemble, discover errors, reload EDITOR, etc.

The whole edit/assemble package is in memory all at once, so it takes you exactly zero time to move between the edit and assemble modes.

To further speed up the edit/assembly process, the S-C assembler allows your source and object files to be in memory at the same time. This means that for a reasonable size program, there are absolutely no disk accesses required. The source is read from memory, the assembler runs from memory, and the object code is stored in memory. Quickly!

To speed things up even further, you can tell the assembler to not list the program on the Apple screen as it assembles. Believe it or not, the screen writing takes most of the time in the assembly process.

How large a program can be handled without using the disk? You can fit about 8 kilobytes of object code, along with its accompanying source, in memory and still run the "all-RAM" system (language card/IIe version).

If your programs are longer, don't worry. The assembler allows you to load source code in sections from disk, and to store object code back to disk.

THE EDITOR

The EDITOR portion of the assembler allows you to type in and edit source programs in preparation for assembly by the Macro Assembler.

There are numerous functions in the Editor which are tailored for efficient source program entry. Some of these are:

1. Line numbers. These make inserting and deleting of program lines easy. The line numbers are implemented much as in BASIC. You can insert, delete, and renumber program lines.

2. Automatic line numbering. Hitting CONTROL-I generates the next line number for you. The actual "next line" number chosen depends on what you have selected as the increment. You can override this by typing in the line number yourself.

3. An EDIT command. This allows you to edit characters within a line.

4. Search and Replace functions. These automatically search for a character string which you specify. Once found, you can replace the string with another one. You can choose to do the whole listing automatically, or search only a line or group of lines.

5. A COPY command. This lets you rearrange your code. A line or group of lines can be copied to anywhere in your program.

Section 3.2 -- 68000 Assembler Syntax

The opcode and register syntax are taken from the Motorola 68000 User's manual. Before delving into the detailed workings of the assembler, we'll present a general overview of how the Macro Assembler handles 68000 program code.

ADDRESS MODES

The addressing modes and assembler syntax are shown below. An or Dn specifies a register, where n is a digit from 0 to 7; "num" represents a number, label, or expression; and "d" is a displacement.

Dn	Data register direct
An	Address register direct
(An)	Address register indirect
(An)+	Address register indirect with post-increment
-(An)	Address register indirect with pre-decrement
d(An)	Address register indirect with displacement
d(An,An) or d(An,Dn)	Address register indirect with index and displacement
num	Absolute short or Absolute long number (see below)
<num	Absolute short number (16 bit)
>num	Absolute long number (32 bit)
d(PC)	PC relative
d(PC,An) or d(PC,Dn)	PC relative with index and displacement
#num	Immediate

ABSOLUTE ADDRESS MODES

In the absolute address modes, either 16-bit or 32-bit addresses may be forced by preceding the address expression with "<" or ">". If neither is present, the assembler must decide which size to use.

If you do not specify the address size, the assembler tries to select the most reasonable size. There are three factors considered in deciding which size to use: the current assembly address (program counter), whether or not the address expression involves a forward reference, and the actual value of the address expression.

If the address expression is defined before it is encountered and it has a value in the range $000000-$007FFF or $FF8000-$FFFFFF, the assembler uses the 16 bit mode.

If the address expression has not yet been defined (it is defined in a forward reference), and the program counter is in the range for a 16-bit address, the 16 bit mode is used. If the address expression value is not in the range of a 16 bit value after forward references are resolved, a RANGE ERROR is generated. To assemble under these conditions, use the ">" prefix to force 32-bit mode.

If this seems a bit mysterious, take a look at the discussion of SIZES in Chapter 8. Also take a look at "Gotcha #1" in Chapter 9.

PC RELATIVE MODE

The program counter relative modes must have PC as the first register. The Kane book says it may be omitted in the simple indexed form, i.e. d(An) instead of d(PC,An). This will not work. The "PC" is necessary for the assembler to distinguish this mode from the d(An) mode.

The expression, "d" is an actual address, or a label corresponding to an address. The assembler subtracts the current value of the program counter from the d-expression to get a displacement. Do not try to specify "d" directly as a displacement. For example, do not write:

```
ADD       TABLE-*(PC),D3
```

Instead, write:

```
ADD        TABLE(PC),D3
```

68000 EXTENSIONS

Many opcodes allow an extension specifying the length (byte, word, or long) of the data. If no length is specified, a 16 bit word size is assumed. An extension consists of a period immediately following the opcode (No SPACES here) and the letter "B", "W", "L", or "S". The "S" is used for short branches only.

The opcodes which allow an extension are:

ADD	ADDA	ADDQ	ADDI	ADDX
AND	ANDI	ASL reg	ASR reg	CLR
CMP	CMPI	CMPM	EOR	EORI
EXT	LSL reg	LSR reg	MOVE	MOVEA
MOVEM	MOVEP	NEG	NEGX	NOT
OR	ORI	ROL reg	ROR reg	ROLX reg
ROXR reg	SUB	SUBA	SUBQ	SUBI
SUBX	TST	Bcc		

All other opcodes may not have an extension.

Note that ANDI, EORI, MOVE, and ORI, when used with the registers CCR, SP, or USP, do not need an extension.

In the indexed addressing modes, the second register (An or Dn) may have a .W or .L extension. This specifies the data length to use from the register. If you specify the .W extension, the 16 bit data is sign extended before adding.

```
MOVE       $34(A3,A5.W),(A4)+
```

Here the source address is formed by adding the 32 bit contents of A3, the sign extended 16 bit contents of A5, and the 32 bit value $34.

Relative branch size is indicated by the suffix ".S" or ".L". ".S" after the opcode forces a short (8-bit)

displacement. ".L" forces a long (16-bit) displacement. If neither is present and the branch is to a previous location, either the 8- or 16-bit displacement is picked by the assembler, as needed. However, if the branch is to a forward location, a 16-bit displacement is assumed. You can override this with the ".S" extension. This will provide a short branch to a forward reference. Note that only BRA, BSR, and Bcc use an extension. The DBcc instruction always uses a 16-bit displacement.

CONDITIONAL BRANCH INSTRUCTIONS

The conditions for use with Bcc, DBcc, and Scc are as follows:

HI	high
LS	low or same
CC	carry clear
CS	carry set
HS	high or same (same as CC)
LO	low (same as CS)
NE	not equal
EQ	equal
VC	overflow clear
VS	overflow set
PL	plus
MI	minus
GE	greater or equal
LT	less than
GT	greater than
LE	less than or equal

In addition, the following may also be used with DBcc and Scc:

T	true
F	false
RA	same as false, used for DBRA

MISCELLANEOUS NOTES

Although some instructions, like ADD, have different forms (ADD, ADDA, ADDQ and ADDI), you don't always have to specify

the A, Q, or I at the end. The assembler is smart enough to do this for you. For example, if you write the instruction

```
ADD     D1,A5
```

the assembler converts it to

```
ADDA    D1,A5
```

This works for ADD, AND, CMP, EOR, MOVE (converts only to MOVEA, not MOVEQ), OR, and SUB.

If the instruction has a Quick Immediate form (ADDQ, SUBQ), the assembler automatically uses that form if the operand is immediate and has a value from 1 through 8. If it is out of this range, the regular (longer) Immediate form is used. The immediate form can be forced by using ADDI, SUBI, etc. The Quick mode cannot be used if the data is out of range.

When in doubt, use the I, Q, A, or whatever at the end. NOTE: ABCD, ADDX, MOVEP, MOVEM, MOVEQ, SBCD, and SUBX must have the specified last letter at the end.

68000 instructions must start on a word boundary (at an even address). If an instruction does not, the assembler gives an error. This is most likely to occur after a memory area that contains byte-size data. Note that byte data can be placed anywhere, but word and long word data must also start on a word boundary. This is not checked by the assembler.

If you get this error, check the preceding lines of your code. There will probably be a data area there. Just put a .BS 1 or a .DA #0 after the byte data to force the program counter back to an even address.

When using the MOVE, OR, AND or EOR instructions to change the status register, the assembler sizes the data depending on whether you use CCR or SR as the destination. If "SR" is used, the assembler uses word size; if "CCR" is used the assembler uses byte size. The assembler does not accept size extensions (".B" or ".W") in these cases.

Section 3.3 - - TUTORIAL

In this section, we'll go step-by-step through the process of writing a small program with the Macro Assembler.

ENTERING A PROGRAM

First start the Macro Assembler by booting the QPAK-68 system disk, and selecting menu item 3 or 4.

Then type in the following short program. Use the line of periods to help gauge where to put spaces.

If you make a mistake, simply retype the line. As in BASIC, retyping a line replaces the previous line of the same line number.

```
.....................................
1000         .OR      $1000
1010 TIME    .EQ      $100
1020 *
1030 START   MOVE.W   #TIME,D4
1040 BEEP    TST.B    $C030      ;SPKR
1050         MOVE.W   D4,D5       NEW TC
1060 PERIOD  DBF      D5,PERIOD
1070         DBF      D4,BEEP
1080         BRA      START
1090 *
1100         .OR      $800
1110         .DA      $8000
1120         .DA      $1000
```

NOTE: All commands in this manual such as "LIST" are enclosed in quotes. Do not include the quotes when you type them.

Now type "LIST" to see the program as the computer has it. The display should look like this:

```
:LIST

1000          .OR      $1000
1010 TIME     .EQ      $100
1020 *
1030 START    MOVE.W   #TIME,D4
1040 BEEP     TST.B    $C030      ;SPKR
1050          MOVE.W   D4,D5       NEW TC
1060 PERIOD   DBF      D5,PERIOD
1070          DBF      D4,BEEP
1080          BRA      START
1090 *
1100          .OR      $800
1110          .DA      $8000
1120          .DA      $1000
```

This listing is called a source program. It is the text form of an assembly language program. Later we will go through the steps necessary to convert it to a form which can be executed by the Q-68 board, but for now let's observe what the source form looks like.

The first column contains line numbers. These are always 4-digit numbers. Assembler line numbers work just like BASIC line numbers for editing, inserting and deleting lines.

The second column contains labels or an "*" to indicate a comment. There are two kinds of labels, memory markers and variables.

The labels START, BEEP and PERIOD are used to direct program flow. Line number 1080, for example, says to branch to the label START, which has been defined in line 1030. The label START actually corresponds to the memory location which holds the instruction "MOVE.W #TIME,D4".

You might be tempted to write line 1080 as

```
1080              BRA    1030
```

thinking that the target is line number 1030. Not so! Remember that line numbers are used only for editing, inserting and deleting program lines. Never try to reference them in your source program code.

The second kind of label is a variable name. In line 1010, the variable TIME is set to the number $100, using the .EQ (equate) directive. Any time the assembler sees the word

TIME, as in line 1030, it will substitute the number $100.

The third column contains opcodes (OPeration codes). These are either standard 68000 instructions, or special "directives" to the Macro Assembler. In our example, the instructions MOVE.W, TST.B, DBF and BRA are used. The directives, identified by two letters preceded by a period, are .OR, .EQ, and .DA.

The fourth column contains operands. Operands further define the action of the opcode. The operand can contain a number, a label, or an arithmetic expression. A few instructions, such as NOP (no operation), do not require an operand.

The fifth column is used for comments. Comments are used to add clarity to your assembly listing. The assembler requires at least one space before the comment to set it apart from the end of the operand field. Some assemblers require a semicolon before a comment. As line 1040 shows, you can use semicolons if you wish. As line 1050 shows, however, they are not required.

Lines 1020 and 1090 are comment lines with only an asterisk in the left margin. These are used to separate blocks of code and produce a neater listing.

SAVING A SOURCE PROGRAM ON DISK:

To save the program on your disk, type "SAVE NOISY". This is a standard DOS SAVE command, just like you would use in BASIC.

When you look at one of these saved files using the DOS CATALOG command, you'll notice that they are flagged as "I" files, which signifies Integer BASIC programs. They are only marked this way--they are not actually BASIC programs. They are 68000 source programs, and cannot be run by Integer BASIC.

NOTE: You do NOT need Integer BASIC in your Apple to use the Macro Assembler.

To clear memory for a new program type "NEW". Now type "LIST" to verify that nothing is there. To reload the program from disk type "LOAD NOISY". Now type "LIST" and you'll see your program back in Apple memory.

ASSEMBLING A SOURCE PROGRAM:

A program must be assembled into binary form before it can be executed. The command to assemble a program is "ASM". Try it now....

Our program is now assembled into memory starting at address $1000.

If you are using a 40 column screen, the display should look like this:

```
:ASM

                        1000          .OR
$1000
00000100-               1010 TIME     .EQ
$100
                        1020 *
00001000- 383C 0100     1030 START    MOVE.W
#TIME,D4
00001004- 4A39 0000
00001008- C030          1040 BEEP     TST.B
$C030      ;SPKR
0000100A- 3A04          1050          MOVE.W
D4,D5       NEW TC
0000100C- 51CD FFFE     1060 PERIOD   DBF
D5,PERIOD
00001010- 51CC FFF2     1070          DBF
D4,BEEP
00001014- 60EA          1080          BRA
START
                        1090 *
                        1100          .OR
$800
00000800- 0000 8000     1110          .DA
$8000
00000804- 0000 1000     1120          .DA
$1000


SYMBOL TABLE

00001004- BEEP
0000100C- PERIOD
00001000- START
00000100- TIME

0000 ERRORS IN ASSEMBLY
```

If you are using an 80 column screen, the display should look like this:

```
:ASM

                        1000          .OR       $1000
00000100-               1010 TIME     .EQ       $100
                        1020 *
00001000- 383C 0100     1030 START    MOVE.W    #TIME,D4
00001004- 4A39 0000
00001008- C030          1040 BEEP     TST.B     $C030       ;SPKR
0000100A- 3A04          1050          MOVE.W    D4,D5        NEW TC
0000100C- 51CD FFFE     1060 PERIOD   DBF       D5,PERIOD
00001010- 51CC FFF2     1070          DBF       D4,BEEP
00001014- 60EA          1080          BRA       START
                        1090 *
                        1100          .OR       $800
00000800- 0000 8000     1110          .DA       $8000
00000804- 0000 1000     1120          .DA       $1000


SYMBOL TABLE

00001004- BEEP
0000100C- PERIOD
00001000- START
00000100- TIME

0000 ERRORS IN ASSEMBLY
```

Notice that two more columns have been added to the left of those we saw when we typed "LIST". The first new column contains the memory addresses (in hexadecimal) into which the program was loaded as it was assembled. The second column shows the numbers (in hexidecimal) which occupy the memory locations.

The Symbol Table is a list of all the labels and the values assigned to them.

The program is now in memory in two forms. The source program is there, right beneath DOS (starting at $9600, and working downward in memory). The executable form, called the "object" program, is in memory from $1000 through $1015. There is also a small section of code at $800 through $807. This is initialization information required by the 68008.

EXECUTING THE OBJECT PROGRAM:

To run the program, type "QON", (return). Do you hear "zip" sounds coming from your Apple speaker? The sound is produced by continually toggling the Apple speaker by addressing location $C030.

Congratulations! You've just written, assembled, and executed your first 68000 program! As soon as you get tired of the noise, hit the RESET key. This turns off the Q-68 board and re-enters the Assembler. (If you have an old version of the F8 ROM, RESET will return you to the Apple monitor. Type "3D0G" to return to the Macro Assembler.)

Now we have walked through the entry, assembly, and execution of a very small program. You'll use exactly the same steps for a large program. In addition, there are other features built into the Macro Assembler which simplify the process you just went through. To take a look at some of these, let's modify the basic NOISY program.

MODIFYING A SOURCE PROGRAM:

First type "LIST", and look over your program.

Let's modify one line of the program to produce a different sound. Type the following line:

```
1010 TIME   .EQ     $80
```

This replaces the line 1010 that was there before.

Now type "ASM", to create the new version of your program in executable (object) form. And execute it, by typing "QON". Hear the difference?

You changed line 1010 the hard way, by retyping the whole line. A line edit command is available to allow you to change only part of a line and leave the rest alone. Type "control-E" for EDIT. The word "EDIT" appears, with the flashing cursor next to it. Type in the line you wish to edit, 1010, and (return).

Now you see line 1010, with the cursor positioned over the

first letter of the line. The left and right arrow keys move the cursor over the line. When you reach the part you want to change, simply type in the new information. The line will be accepted exactly as it appears on the screen, so be sure you wipe out all unwanted letters by typing a space over them.

EASIER ENTRY OF SOURCE PROGRAMS:

Now let's try an easier way to enter source lines. First save the latest version of your program on a disk by typing SAVE NOISY again. Then type "NEW" to erase the source program from memory. (It is still on the disk).

Now hold down the CTRL key ("CTRL" means "control"), and type the letter I. We call that typing "control-I". Look at the screen. You will see that the Apple printed "1000", and the cursor is blinking after that. Type "control-I" again, and you will see the cursor move over about 7 character positions.

Whenever you type "control-I" inside a line (beyond the left margin), the cursor moves to the next tab stop. Play with this a little, and you will find that the tab stops line up nicely with the source program column format.

After entering a line and typing RETURN, the next CTRL-I automatically generates the next line number for you.

Why don't you try typing in the NOISY program again, with the help of the "control-I" key? You'll find that mastering the "control-I" function will save you a lot of time when you write and edit programs.

Start by typing "NEW", to insure that there are no stray program lines left over from some previous work.

Then type in the program again, this time using the CTRL-I key to generate the line numbers. Type "ASM" again, and "QON". The program should work just like it did the first time. Stop it by pressing "RESET".

With this brief introduction, you should now be ready to dive into the following chapters. As you study each new command or feature, experiment with it until you really understand what is happening.

Section 3.4 -- SOURCE PROGRAMS

Source programs are entered a line at a time, with a line number identifying each line. The line numbers run from 0 through 9999. The automatic line numbering and the RENUMBER command use default line numbers from 1000 through 9999. Source program lines are kept sorted in line-number order. The numbers are used for editing purposes, just as in BASIC.

A blank must always follow the line number. After the blank there are four fields of information: the label, opcode, operand, and comment fields. Fields must be separated by at least one blank. Lines may be up to 248 characters long.

AUTOMATIC LINE NUMBERING

You may type the line number yourself, or use the two means for automatic line number generation.

The first method is really semi-automatic, because you type a control-I to get the next line number. Any time the cursor is at the beginning of a line (right after the prompting colon), typing control-I causes the next line number to be generated.

Immediately after loading the Macro Assembler or typing "NEW", the first line number generated is 1000. The number is displayed as four digits and a blank. The cursor is then in position for the first character of a label, or the asterisk denoting a comment line. If you type the control-I in any other position than the beginning of a line, it causes a tab to the next tab stop.

The second method is invoked by typing the "AUTO" command, with or without a starting line number. In the "AUTO" mode, the next line number is automatically generated at the beginning of every line. If you don't want to use the line number, or want one out of sequence, you can backspace to the colon prompt and type a new line number or command. The "AUTO" mode is terminated by the MANUAL command, by hitting "RESET", or by any error message.

The next line number is the value of the previous line number plus the current increment. The standard increment is 10, but you can change it to any integer value with the

INCREMENT command.

BUILT-IN TAB STOPS

Although the opcode, operand, and comment fields are not required to begin in any particular column, it is neater to align them. Therefore, tab stops are included in the Macro Assembler at columns 14, 22, 27, and 32.

Control-I is the tab command used by the Macro Assembler. Normally control-I generates enough spaces to move the cursor to the next tab stop. If control-I is typed at the beginning of a line, the next line number and one space is generated. If you are already past or at the last tab stop, control-I generates a single space.

Some printer interface cards with firmware drivers use control-I for setting various modes. If you wish to change the tab character to avoid interfering with such a card, you may do so. The ASCII code for the tab character is stored at location $D00F. An alternative is to change the printer interface control character, which is usually stored at $06F8+slot#.

Space is reserved inside the Macro Assembler for a total of five tab stops. They are stored in locations $D010 through $D014, as column numbers. You may change them if you wish. If you want fewer than five tab stops, set the remaining ones to zero.

For the $3000 (no language card) version of the Macro Assembler, the tab character code is at $300F, and the tab stops are at $3010-$3014.

LABEL FIELD:

The label field may be left blank, or may contain a label. There are three types of labels used in the Macro Assembler: normal labels, local labels, and private labels. The first character of the label must be in the second column after the line number.

```
1000 START.HERE     (normal label)
1010 .23            (local label)
```

```
1020 :12            (private label)
```

You may not use a label which corresponds to one of the 68000 register names. These "reserved" labels are D0, D1, D2, D3, D4, D5, D6, D7, A0, A1, A2, A3, A4, A5, A6, A7, CCR, SR, USP and PC.

Normal Labels: Used to name places in your program to which you will branch, as well as constants and variables. Normal labels may be up to 32 characters long. The first character must be a letter. Subsequent characters may be letters, digits, or the period character ("."). The period is useful for making long labels readable. For example, a subroutine to extract the next character from a buffer might be named, "GET.NEXT.CHAR.FROM.BUFFER".

The standard tab stops assume your labels will be six or fewer characters long. However, since the assembler is relatively free-format, you may type any length label followed by a blank and the opcode, operand, and comment fields. Or, if you wish, you may type the long label on a line all by itself. In this form, the label is assigned the current value of the location counter, just as if you had appended ".EQ *" to the line.

```
1000          .OR      $1000
1010 *
1020 SOURCE.POINTER       .EQ  $40
1030 DESTIN.POINTER       .EQ  $44
1040 TRANSFER.COUNT       .EQ $48
1050 *
1060 TRANSFER.BLOCK.OF.LONG.WORDS
1070          MOVE.L  SOURCE.POINTER,A1
1080          MOVE.L  DESTIN.POINTER,A2
1090          MOVE.W  TRANSFER.COUNT,D0
1100 LOOP     MOVE.L  (A1)+,(A2)+
1110          DBF     D0,LOOP
```

Local Labels: Used to name branch points within a module. The main purpose for local labels is to make programs more readable by reducing the number of label names you must invent. As a side effect, local labels save considerable space in the symbol table during assembly; they only require two bytes each. The use of local labels also encourages structured programming habits.

Local labels have a period as the first character, followed by one or two digits. Any label from ".0" through ".99" may be used. (Please note that these are label names, not

decimal fractions. Consequently, the label ".1" is treated as exactly equivalent to the label ".01"; in fact, it will be listed in the symbol table listing as ".01".)

A local label's location is internally defined as a distance from a normal label which precedes it in the source program. (There must be one before it, or you will get an error message.) The local label must be no more than 255 bytes away from a preceding normal label.

Since each set of local labels is associated with a particular normal label, you may re-use the same local labels as often as you wish.

Here are two routines within the same code segment which use the local labels ".1" and ".2" twice. This is perfectly acceptable because of the intervening normal label, "ANOTHER.NORMAL.LABEL".

```
1000          .OR      $1000
1002 *
1005 NORMAL.LABEL
1006 *
1010          CLR      D0
1020 .1       MOVE     #$2000,A0
1030          MOVE     #$1000,D1
1040 .2       MOVE     D0,(A0)+
1050          DBEQ     D1,.2
1060          BRA      .1
1070 *
2000 ANOTHER.NORMAL.LABEL
2002 *
2010          MOVE     #$100,D4
2020 .1       TST.B    $C030
2030          MOVE     D4,D5
2040 .2       DBF      D5,.2
2050          DBF      D4,.1
2060          RTS
```

The assembly listing shows how these local labels are represented in the symbol table:

```
 :ASM

                      1000          .OR      $1000
                      1002 *
                      1005 NORMAL.LABEL
                      1006 *
00001000- 4240        1010          CLR      D0
00001002- 307C 2000   1020 .1       MOVE     #$2000,A0
```

```
00001006- 323C 1000 1030          MOVE     #$1000,D1
0000100A- 30C0       1040 .2      MOVE     D0,(A0)+
0000100C- 57C9 FFFC  1050         DBEQ     D1,.2
00001010- 60F0       1060         BRA      .1
                     1070 *
                     2000 ANOTHER.NORMAL.LABEL
                     2002 *
00001012- 383C 0100  2010         MOVE     #$100,D4
00001016- 4A39 0000
0000101A- C030       2020 .1      TST.B    $C030
0000101C- 3A04       2030         MOVE     D4,D5
0000101E- 51CD FFFE  2040 .2      DBF      D5,.2
00001022- 51CC FFF2  2050         DBF      D4,.1
00001026- 4E75       2060         RTS


SYMBOL TABLE

00001012- ANOTHER.NORMAL.LABEL
.01=00001016,.02=0000101E
00001000- NORMAL.LABEL
.01=00001002,.02=0000100A

0000 ERRORS IN ASSEMBLY
```

Private Labels: Used within macros as branch points. Private Labels are maintained in a separate symbol table, and hence do not interfere with either normal labels or local labels. Each private label is associated with a particular invocation of a macro, so that the assembler treats the recurrence of the same label number as a unique label. Private labels are discussed in more depth in Section 3.8, on Macros.

OPCODE FIELD:

The opcode field contains a machine language operation code, a macro name, or an assembler directive. If you are using the tab stops, the opcode field normally starts in column 14. However, opcodes may begin in any column after at least one blank from a label or at least two blanks from a line number.

The Macro Assembler uses the standard 68000 instruction mnemonics as defined by Motorola. Macros and assembler directives are discussed later in this chapter.

OPERAND FIELD:

The operand field further specifies the action of the opcode field. This field may contain register expressions, arithmetic expressions, variables, and constants in various combinations.

Operand expressions have a range of 32 bits. This allows for the 16 megabyte address range of the 68000, and for 32-bit constants. Although 68000 addresses are represented by 24 bits, and 68008 addresses are represented by 20 bits, the assembler accepts addresses of 32 bits without an error. Future versions of the 68000 will probably be able to use a full 32 bit address.

You may put as many spaces as you want between the opcode and operand fields. However, there may not be any spaces in the operand field. If there are two operands, they must be separated by a comma.

The following program code demonstrates some operand examples:

```
                        1000          .OR       $1000
                        1010 *--------------------------------
00000002-               1020 TWO      .EQ       $2
00000006-               1030 SIX      .EQ       $6
                        1040 *--------------------------------
00001000- 2038 0008     1050          MOVE.L    TWO+SIX,D0
00001004- 223C 0000
00001008- 0008          1060          MOVE.L    #TWO+SIX,D1
0000100A- 18F2 B015     1070          MOVE.B    $15(A2,A3),(A4)+
0000100E- 4E43          1080          TRAP      #3
00001010- 4E75          1090          RTE
```

SYMBOL TABLE

```
00000006- SIX
00000002- TWO
```

In line 1000, the operand is the constant $1000, a value for the .OR directive. Lines 1020 and 1030 define the variables TWO and SIX, and the operand field specifies the numeric values of these variables.

The instruction at line 1050 says to move the data from memory location 8 into data register D0. Location 8 is defined as the sum of two variables, TWO and SIX.

The instruction at line 1060 says to move the number 8 into data register D1.

Line 1070 shows a fancy addressing mode provided by the 68000. Line 1080 contains the opcode TRAP, whose operand is a number (#) from 0 to $F. And finally, line 1090 contains an instruction which does not require an operand, RTE (Return From Exception).

COMMENT FIELD:

Comments are separated from the operand field by at least one blank. Tab stops are set at columns 27 and 32 for the comment field. In the assembly listing, tabbed comments begin in column 51.

COMMENT LINES;

Full lines of comments may be entered by typing an asterisk (*) in the first column of the label field. This kind of comment is useful in separating various routines from each other, and labelling their functions. It is analogous to the REM statement in BASIC.

Lines which are completely blank are also treated as comments.

ESCAPE-L:

A special comment line is built into the Macro Assembler. If the cursor is one space to the right of a line number, typing ESC-L generates a built-in comment line. This consists of an asterisk (*) followed by a line of dashes which just fill one line on the 40 column screen. The comment line is commonly used to set off blocks of comments.

If a comment line of dashes is not your favorite, you may change the repeated character. The ASCII code of the character is kept at $D015 ($3015). It is currently $AD, which is ASCII for "-".

If you type ESC-L at the beginning of a line (before a line number), it has a different meaning. In this case it causes the first six characters of the line on the screen to be changed to "LOAD". Then the rest of the line is read from the screen, and issued as a LOAD command. With this feature you can LOAD a file simply by typing CATALOG, ESC-I,I,...I,L

CURSOR CONTROLS:

The standard Apple II screen editing tools are supported by the Macro Assembler. You can edit lines of assembly language source in the same way that you edit lines in your Integer BASIC or Applesoft program.

Whether or not you have the Autostart ROM, you may use the new Apple standard cursor movement controls: Escape-I, -J, -K, and -M. The older Escape-A through Escape-F and Escape-@ are also supported by the Macro Assembler.

Section 3.5 -- COMMANDS

You will use three types of commands in the Macro Assembler: Assembler commands, DOS Commands, and Monitor Commands. The Assembler Commands allow you to edit, assemble, and execute your assembly language programs. Most Apple Monitor and DOS commands are also recognized. Commands are typed immediately after the prompt symbol, which is a colon (:).

ASSEMBLER COMMANDS

There are 29 commands recognized by the Macro Assembler. Assembler Commands may be abbreviated by typing the first three letters. All of the letters of the command names you do type are checked for spelling.

(Two DOS Commands, LOAD and SAVE, are used so frequently that they might be thought of as Assembler commands. However, they are DOS commands, and as such cannot be abbreviated to the first three letters.)

The 29 Assembler Commands can be conveniently grouped into source commands, editing commands, listing control commands, 68000 execution commands, object commands and miscellaneous commands.

Group	Commands
Source	NEW, LOAD, SAVE, TEXT, HIDE, MERGE, RESTORE
Editing	EDIT, COPY, LIST, FIND, REPLACE, DELETE, RENUMBER
Listing	SLOW, FAST, PRT, (")
Execute 68000	QON, DBUG
Object	ASM, MGO, VAL, SYMBOLS
Miscellaneous	AUTO, MANUAL, INCREMENT, MEMORY, MNTR, RST, USR

SOURCE COMMANDS: ..... NEW, LOAD, SAVE, TEXT,
HIDE, MERGE, RESTORE

NEW Command: ..... :NEW

Deletes the current source text from memory and restarts the Macro Assembler. Clears the screen, writes "S-C MACRO ASSEMBLER 68000 V1.0" on the top line, restarts the automatic line numbering at 1000 and waits for you to type a source line or another command.

LOAD Command: ..... :LOAD
:LOAD filename

Deletes the current source program (unless it is "hidden" with the HIDE command), and then reads in a new one from cassette tape. The LOAD command works exactly the same as the load command in Integer BASIC or Applesoft.

If you type a filename after the LOAD command, it is intercepted by DOS and a source program is loaded from disk instead of tape.

```
:LOAD              (Load from tape)
:LOAD BANANA       (Load disk file named "BANANA")
```

SAVE Command: ..... :SAVE
:SAVE filename

Writes the source program currently in memory to cassette tape. It works exactly the same as the SAVE command in Integer BASIC or Applesoft.

If you type a filename after the SAVE command, it is intercepted by DOS to write the source program on disk rather than tape. A saved file appears in the disk directory as a type "I" file.

```
:SAVE                (save on tape)
:SAVE BANANA         (save on disk file "BANANA")
```

TEXT Command ..... :TEXT filename
:TEXT# filename
:TEXT/ filename

Saves the source program to disk as a DOS text file, so it can be edited with another text editor. There are three forms of this command:

TEXT writes the lines with no line number. This is very handy for building EXEC files for use with DOS, BASIC, or any general use. Text files which are created this way can be read back into the Macro Assembler by turning the AUTO line numbers on (see AUTO command), and using the EXEC command.

TEXT# writes lines with line numbers, exactly as they appear on the screen. These files can be EXECed into Applesoft, or back into the Macro Assembler.

TEXT/ writes the lines with a control-I in place of the line number. You can keep disk files of often-used routines that can be EXECed into a program wherever they are needed. The control-I at the beginning of each line causes a line number to be generated when the line is read by the Macro Assembler.

```
:TEXT ROUTINE   (writes the current source
                 program to a text file
                 named ROUTINE, with no line numbers)

:TEXT# ROUTINE  (writes the current source
                 program to a text file
                 named ROUTINE, with line numbers)

:TEXT/ ROUTINE  (writes the current source
                 program on a text file
                 named ROUTINE, with control-I's
                 in place of line numbers)
```

HIDE and MERGE Commands: ..... :HIDE
:MERGE

These two commands, used together with the LOAD command, allow you to join a program from disk or tape to a program that is already in memory. To remind you that you are HIDE-ing, the prompt symbol changes from ":" to "H:". After HIDE-ing a program, you can LOAD another one from disk or tape with the LOAD command. Then you type MERGE to join the two programs together.

After this sequence of commands is executed, the program which was already in memory follows the program just LOADed. If the line numbers of the merged programs are not correct, you should use RENUMBER to assign new ones.

For example, suppose that we have two source programs on the disk named "SRCONE" and "SRCTWO". We want to join them together so that "SRCONE" precedes "SRCTWO". Here are the two programs:

```
:LOAD SRCONE
:LIST

1000 *   PROGRAM NUMBER ONE
1010 MAIN    BSR SUBROUTINE
1020         RTS

:LOAD SRCTWO
:LIST

1000 *   SUBROUTINE TO DO SOMETHING
1010 SUBROUTINE
1020         MOVE BLAH.BLAH.BLAH,D0
1030         MOVE D0,SOMEWHERE
1040         RTS
```

Now let's HIDE the source of SRCTWO, LOAD in the source from SRCONE, and MERGE them together.

```
:HIDE
H:LIST                  (Note that nothing lists, because the
H:LOAD SRCONE            source has been hidden).
H:LIST

1000 *    PROGRAM NUMBER ONE
1010 MAIN    BSR SUBROUTINE
1020         RTS

H:MERGE

:LIST

1000 *    PROGRAM NUMBER ONE
1010 MAIN    BSR SUBROUTINE
1020         RTS
1000 *    SUBROUTINE TO DO SOMETHING
1010 SUBROUTINE
1020         MOVE BLAH.BLAH.BLAH,D0
1030         MOVE D0,SOMEWHERE
1040         RTS
```

You can see that both programs are now in memory, but the line numbers are not in sequence. RENUMBER fixes the line numbers.

```
:RENUMBER
:LIST

1000 *    PROGRAM NUMBER ONE
1010 MAIN    BSR SUBROUTINE
1020         RTS
1030 *    SUBROUTINE TO DO SOMETHING
1040 SUBROUTINE
1050         MOVE BLAH.BLAH.BLAH,D0
1060         MOVE D0,SOMEWHERE
1070         RTS
```

RESTORE Command ..... :RESTORE

Restores the root source program if an assembly is aborted while inside an "included" module.

The "root source program" is the source program that is in memory at the time you issue the ASM command. If this

source program uses the .IN directive to include additional source files, it is possible that assembly might be aborted while the "root" program is "hidden". An assembly may be aborted either manually by typing the RETURN key while the assembly is in progress, or automatically due to an error in the source program.

If the assembly is aborted during the time that the root program is hidden, the prompt character changes from ":" to "I:". The RESTORE command resets the memory pointers so that the included file is released, and the root program is no longer hidden. The prompt character then changes back to ":".

You do not have to use the RESTORE command after an aborted assembly unless you wish to get back to the root source program for editing purposes. If you type the ASM command, the assembler automatically RESTOREs before starting the assembly.

If an assembly aborts due to an error in a source line, you may correct the source line, SAVE the module on the appropriate file, and type ASM to restart the assembly.

EDITING COMMANDS ..... LIST, FIND, EDIT, REPLACE
DELETE, RENUMBER, COPY

The editor in the Macro Assembler combines the Apple screen editing features with a BASIC-like line editor. Source programs are entered and edited in almost exactly the same way you would enter and edit an Integer BASIC or Applesoft program.

Editing commands allow you to list your source program, delete lines, search for lines, replace portions of selected lines, renumber lines, and copy blocks of lines from one location to another. There is also a powerful EDIT command which allows you to edit characters within a line.

RANGE PARAMETERS:

Most editing commands (LIST, FIND, EDIT, REPLACE, AND DELETE) can use range parameters to operate on just part of the program. A range parameter may be written with one or

two line numbers, or in most cases it may be omitted. If there are two line numbers, separate them with a comma. If there is only one line number, it may stand alone, or with a comma; the comma may precede or follow the line number. Each of these five possible arrangements has a specific meaning:

(no number)	Specifies the entire source program.
,	Specifies the entire source program.
#	Specifies line number #.
,#	Specifies lines from the beginning of the source program through #.
#,	Specifies lines from # through the end of the source program.
#1,#2	Specifies lines from #1 through #2.

Here are some specific examples:

```
:LIST                (lists all lines)
:LIST 2000           (list line 2000 only)
:DEL 2000,3000       (deletes lines from 2000-3000)
:EDIT 2000,          (edits all lines from 2000 through
                     the end of the program)
:FIND ,2000          (finds all lines from the beginning
                     of the program through line 2000)
```

You can also use a period (.) to mean "the last line entered". The period, or "dot", is defined as the number of the last line entered or deleted from the source program.

STRING PARAMETERS:

Some commands (LIST, FIND, EDIT, AND REPLACE) can also use a search string parameter to operate only on lines containing that string. The search string is of the form *string*, where (*) is a delimiter of your choice. The delimiter can be any printing character that does not occur in your search string, except comma (,), period (.), or a digit (0-9). Some examples:

```
:LIST /COUT/          (lists all lines containing COUT)
:EDIT "/DEST"         (edits all lines containing /DEST)
```

You can use a wildcard character in search strings if you want to operate on all lines containing partial matches to your search string. The standard wildcard character is control-W. You have to first type a control-O, and then a control-W. The control-O character is an override to allow the insertion of control characters in commands and source lines. The control-W appears on the screen in inverse video. For example:

```
:FIND ?ASWTA?       (Pretend that the "W" is a control-W)

         MOVE       AS.DATA,D4

1120     MOVE       AS.DATA,D6

1200     MOVE       BASKETA,D2
```

```
LIST COMMAND: ..................... :LIST
                                    :LIST range
                                    :LIST *string*
                                    :LIST *string* range


FIND COMMAND: ..................... :FIND
                                    :FIND range
                                    :FIND *string*
                                    :FIND *string* range
```

Actually, FIND is just an alternate name for the LIST command. Many people find it more natural to use LIST with line number ranges and FIND with a search string, but either command works with either parameter (or both parameters!).

Both FIND and LIST list a single line, a range of lines, or an entire source program. If you specify a search string, only those lines which match the string are listed.

While a program or range of lines is listing, you can momentarily stop the listing by hitting the space bar. Tapping the space bar again restarts the listing. You can abort the listing by hitting the RETURN key. The SLOW and FAST commands allow you to control the listing speed. If

you list a single line, it is displayed on the screen in a position which makes it easy to edit using the Apple screen editing tools.

```
:LIST                (list entire program)
:LIST 1230           (list only line 1230)
:LIST 1230,2890      (list lines 1230 through 2890)
:LIST 1230,          (list all lines from 1230 through end)
:LIST ,1230          (list all lines from beginning through
                     1230)
:FIND /ASCII/        (list all lines containing the string
                     "ASCII")
:FIND "BI",1200      (list all lines through 1200 that
                     contain the string "BI")
```

```
EDIT COMMAND: ..................... :EDIT
                                    :EDIT range
                                    :EDIT *string*
                                    :EDIT *string* range
```

Allows easy editing of program lines. Since this command is typed so frequently, a short form is provided. Instead of typing "EDIT", you just type control-E, and the word, "EDIT" magically appears on the screen. You fill in the line number, and proceed to edit the line.

If you specify no range or string, the whole source program, one line after another, is displayed for editing. If you specify a range, those lines in the range are displayed for editing. If you specify a search string, only those lines matching the string are displayed.

EDIT displays a line for editing by printing the line, clearing from the end of the line to the bottom of the screen, and placing the cursor at the beginning of the label field. You can edit the line with the following commands:

control-B	Move cursor back to beginning of the label field.
control-D	Delete character under cursor.
control Fx	Move cursor to next occurence of "x" in line (if any). You may type any character you wish for "x".
control-H	Move cursor left.

(left arrow)

control-I	Begin insertion mode; characters are inserted until another control character is typed.
control-L	Store current edited line and start editing the next line.
control-M (RETURN)	Store the edited line. The complete line is stored regardless of the cursor position.
control-N	Move cursor to end of line.
control-O	Begin insertion mode, but allow next character typed to be inserted even if it is a control-character
control-Q	Finish edit mode, chopping off all characters from cursor to ehd of line.
control-R	Restore the original line without leaving edit-mode.
control-T	Move cursor to next tab stop.
control-U	(right arrow) Move cursor right.
control-X	Abort the EDIT command.
control-@	Erase from cursor to end of line without leaving edit-mode.

```
REPLACE ........ :REPLACE *s-string*r-string*
COMMAND:          :REPLACE *s-string*r-string* range
                  :REPLACE *s-string*r-string* options
                  :REPLACE *s-string*r-string* range options
```

NOTE: "s-string" stands for search string; "r-string" stands for replacement string.

Searches for and replaces character strings in your source code. REPLACE operates on all fields, from the first character in the label field through the end of each line. The search can include the entire program, or it can be restricted to a range of line numbers by specifying which lines are to be searched (range parameter).

When REPLACE finds your search string it prints the line with the matching string shown in inverse video. The program then asks, "REPLACE?", and waits for you to type "Y", "N", or some other character. If you type "Y" the corrected line is listed, and the search continues. If you type "N" it simply continues searching. If you type some other character, the REPLACE command terminates.

There are two options which may be selected by appending the letters "A" or "U" to the command line. The letter "A" on the end of the command line causes automatic operation, without the prompting at each replacement. The letter "U" means ignore the difference between upper and lower case letters.

It is possible to replace more than one matching string in the same source line.

You may use wildcard characters in the search string. The entire matching string is replaced by the replacement string. Do not put any wildcard characters in the replacement string.

DELETE COMMAND: ..... :DELETE range

Deletes a line or range of lines from your source program, just as in BASIC. Another way to delete a single line is to type its line number followed immediately by a RETURN, or by a space and RETURN.

(Warning: DELETE followed by a file name is a DOS command. This command will delete a file from your disk!)

DELETE must be followed by a range parameter and cannot have a search string parameter.

```
:DEL                (doesn't work)
*** SYNTAX ERROR
:DEL 1230           (delete only line 1230)
:DEL 1230,2890      (delete lines 1230 through 2890)
:DEL 1230,          (delete all lines from 1230 through
                    end)
:DEL ,1230          (delete all lines from beginning
                    through 1230)
:DEL ,              (delete entire program)
```

RENUMBER COMMAND: ..... :RENUMBER
:RENUMBER base
:RENUMBER base,inc
:RENUMBER base,inc,start

Renumbers all or part of the line. in your source program with the specified starting line number and increment. There are three optional parameters for specifying the line number to assign the first renumbered line (base), the increment, and the place in your program to begin renumbering (start). There are four possible forms of the command:

:REN	Renumber the whole source program: BASE=1000, INC=10, START=0
:REN #	Renumber the whole source program: BASE=#, INC=10, START=0
:REN #1,#2	Renumber the whole source program: BASE=#1, INC=#2, START=0
:REN #1,#2,#3	Renumber from line #3 through the end: BASE=#1, INC=#2, START=#3

The last form is useful for opening up a "hole" in the line numbers for entering a new section of code.

```
:LIST
1000 *    LITTLE RENUMBER EXAMPLE
1005 SAMPLE        MOVE D4,D5
1006               MOVE D7,D2
1010               RTS
```

```
:RENUMBER
:LIST
1000 *    LITTLE RENUMBER EXAMPLE
1010 SAMPLE        MOVE D4,D5
1020               MOVE D7,D2
1030               RTS
```

```
:RENUMBER 100
:LIST
0100 *    LITTLE RENUMBER EXAMPLE
0110 SAMPLE        MOVE D4,D5
0120               MOVE D7,D2
0130               RTS


:RENUMBER 2000,4
:LIST
2000 *    LITTLE RENUMBER EXAMPLE
2004 SAMPLE        MOVE D4,D5
2008               MOVE D7,D2
2012               RTS


:RENUMBER 3000,10,2008
:LIST
2000 *    LITTLE RENUMBER EXAMPLE
2004 SAMPLE        MOVE D4,D5
3000               MOVE D7,D2
3010               RTS
```

COPY COMMAND: ..... :COPY range, target

Copies a range of lines from one place in the program to another. A copy of all the lines in the range specified is placed just before the target line.

If the target line does not exist, the lines within range are copied where the target line should have been. If the target is line 9999, and there is no line 9999, the copied lines are placed at the end of the source program.

COPY does not delete the original section or renumber the copy, so this command should be followed immediately by a RENUMBER command.

```
:LIST
1000 *    LITTLE COPY EXAMPLE
1005 SAMPLE        MOVE D3,D4
1006               MOVE D7,D2
1010               RTS
```

```
:COPY 1005,1006,9999
:LIST
1000 *    LITTLE COPY EXAMPLE
1005 SAMPLE         MOVE D3,D4
1006                MOVE D7,D2
1010                RTS
1005 SAMPLE         MOVE D3,D4
1006                MOVE D7,D2


:RENUMBER
:LIST
1000 *    LITTLE COPY EXAMPLE
1010 SAMPLE         MOVE D3,D4
1020                MOVE D7,D2
1030                RTS
1040 SAMPLE         MOVE D3,D4
1050                MOVE D7,D2


:COPY 1020,1040,1010
:LIST

1000 *    LITTLE COPY EXAMPLE
1020                MOVE D7,D2
1030                RTS
1040 SAMPLE         MOVE D3,D4
1010 SAMPLE         MOVE D3,D4
1020                MOVE D7,D2
1030                RTS
1040 SAMPLE         MOVE D3,D4
1050                MOVE D7,D2
```

LISTING CONTROL COMMANDS: ..... FAST, SLOW, PRT, "

The listing control commands are used to control the speed of display on the screen, and to control printing of listings on other devices. One special command allows sending setup control characters to your printer.

FAST and SLOW commands: ..... :FAST
:SLOW

FAST sets the listing speed to the normal speed, which is

too fast for most people to read. When you start the Macro Assembler, it is set to the FAST mode. If you abort a listing by hitting the RETURN key, the system returns to the FAST mode.

SLOW reduces the listing speed so that you can read it as it goes by on your screen.

In both the FAST and SLOW modes, you can momentarily stop the listing by tapping the space bar (or any other key except RETURN). You can abort the listing by typing the RETURN key. When the listing is stopped, pressing two keys at the same time causes one additional line to be listed.

PRT Command: ..... :PRT

Provides a "hook" for a user-supplied printer driver. If you have and Apple parallel or serial printer board, the usual PR# slot will activate your printer. If you have a printer driven through the game port, or an interface board which requires special handling, you can use the PRT command to turn it on. If you don't need it for a printer, PRT can serve as a second USR command.

PRT executes a JSR $D009 ($3009), where you can put a JMP to your printer driver. Remember that this driver is written in 6502 code, not 68000 code.

(") Command: ..... :"string

Sends the specified string to the currently selected output device. If your printer is currently selected, you can send control-codes to it.

Remember that in order to enter a control-character on an input line, you type the control-O (override) followed by the desired character.

For example, if you are using an MX-80 printer and wish to set the italics mode, type:

"control-O(ESC)4

EXECUTE 68000 CODE COMMANDS: ..... QON, DBUG

These commands start up the Q-68 Board.

QON Command: ..... :QON

This command executes a CALL to Apple II location $30B. It is assumed that the Q-68.STARTUP.BIN program has been loaded at address $300. (This is done automatically when the system is booted with the QPAK-68 system disk).

The routine at $30B turns on the Q-68 board, and then returns to the caller, which in this case is the Macro Assembler. The Macro Assembler continues to run after the Q-68 board is turned on.

Note that when you start the Q-68 board using "QON", your 68000 program must take care of installing the 68000 RESET vectors at $800. See Chapter 2.

DBUG Command: ..... :DBUG

Fires up the DEBUG program in the Q-68 board. You will usually use "DBUG" just after a succcessful assembly of your 68000 code.

This command assumes that Q-68.STARTUP.BIN is in memory at $300. The RESET vectors for starting up DEBUG are installed at $800, the Q-68 card is turned on, and a special 6502 routine is entered.

This 6502 routine continuously checks for a CTL-B or CTL-D command. Since the 6502 is fully occupied with this task, it does not interfere with DEBUG's use of the Apple II keyboard or text screen.

The "Q-68.STARTUP" program is explained in Appendix D.

OBJECT COMMANDS: ..... ASM, MGO, VAL, SYMBOLS

Object commands are used to assemble source programs into object programs, execute 6502 object programs, and to print the value of label expressions after assembly.

ASM Command: ..... :ASM

Initiates assembly of your source program. The Macro Assembler is a two-pass assembler. During the first pass it builds a symbol table with the definition of every label used in your program. During the second pass the assembler stores object code into memory (or writes it on a disk file) and produces an assembly listing on the screen and/or the printer. At the end of the second pass all the labels and their values are listed in alphabetical order.

The assembly listing may be momentarily interrupted and restarted by tapping the space bar. You may abort the assembly by typing the RETURN key. The assembly listing may also be controlled with the .LIST directives, to print any part of it or none at all.

If any errors are detected in either pass, they are printed along with a copy of the offending line. Assembly normally continues after an error, so that you can catch as many errors as possible in one pass. If any errors are detected during pass one, pass two is not attempted. At the end of assembly the total number of errors is printed. The assembly error messages with their probable causes are listed in Appendix B.

MGO Command: ..... :MGO expression

Begins execution of a 6502 object program. An expression or label name must follow the MGO command to define the place to begin execution.

```
:MGO BEGIN          (Start execution at label BEGIN)
:MGO $300           (Start execution at $300)
```

The 6502 program can return to the Macro Assembler either by using an "RTS" instruction, by a "JMP $3D0" (if DOS is active), or by a "JMP $D003" ($3003). You may also abort your program by hitting the RESET key. If your Apple has the Autostart ROM, you will come out in the assembler. If you come out in the monitor, type 3D0G to reenter the assembler.

This command is really designed for use with the 6502 Assembler from S-C Corp. Don't try to jump to a 68000 program with MGO--remember that the Apple's 6502 is running the assembler, and when it sees an MGO, it expects to find executable 6502 code at the "MGO" address.

One use for MGO would be to directly access the various entry points of the Q-68.STARTUP.BIN program. See Appendix D.

VAL Command: ..... :VAL expression

Evaluates any legal operand expression, and prints the value in hexadecimal. It may be used to quickly convert decimal numbers to hexadecimal, to determine the ASCII code for a character, or to find the value of a label from the last assembled program.

```
:VAL 12345
00003039
:VAL -21846
FFFFAAAA
:VAL 'X          (ASCII value)
00000058
:VAL LOOPA+3
000084E
```

SYMBOLS Command: ..... :SYMBOLS

Displays a copy of the Symbol Table, just like the one that is normally printed at the end of pass two of an assembly.

MISCELLANEOUS COMMANDS ..... AUTO, MANUAL, INCREMENT,
MEMORY, MNTR, RST, USR

The last seven commands do not fit into any other category.

AUTO Command: ..... :AUTO
:AUTO #

Turns on automatic line numbering mode. In this mode, a new line number is automatically generated every time you end a line. Lines are ended by typing RETURN, by backspacing over the prompt symbol, or by typing control-X.

If AUTO is used without a parameter, the generated line numbers start with the next number after the last line you entered or deleted. The next number is formed by adding the INCREMENT value. The increment can be changed with the INCREMENT command.

AUTO followed by a line number starts the numbering at that value.

AUTO should be used when EXEC-ing in text files from another source. This way, you can even use the Macro Assembler to edit BASIC programs which have been listed into text files (as long as you don't need to renumber the BASIC line numbers).

You can type commands while in the AUTO mode by typing backspaces to the beginning of the line (next to, not over, the prompt) and then typing the MANUAL command.

The AUTO mode is also terminated by hitting RESET, or after any error message.

MANUAL Command: ..... :MANUAL

Terminates the automatic line numbering (AUTO) mode. To use the MANUAL command, first backspace over the line number, just to the right of the colon, and type MANUAL (or simply MAN).

INCREMENT Command: ..... :INCREMENT number

Sets the increment used for automatic line number generation (both control-I generated numbers and AUTO mode numbers). The increment is normally 10, but you may set it to any value between 0 and 9999. (Of course, an increment of 0 makes no sense. Neither does a large value like 9999. But you can use them if you wish!)

```
:INC 5              (set increment to 5)
:INC 10             (set increment to 10)
```

MEMORY Command: ..... :MEMORY

Displays the beginning and ending memory addresses of the source program and the symbol table.

```
:MEM

SOURCE PROGRAM: $94F3-9600
  SYMBOL TABLE: $3000-3274
```

Memory between the top of the symbol table and the bottom of the source program is free to be used without clobbering anything.

The assembler automatically protects memory (during assembly) from $3000 to the top of the symbol table, and from the bottom of the source program through $FFFF. This insures that your object program does not clobber the assembler, the source program, or DOS.

MNTR command: ..... :MNTR

Enters the Apple system monitor. This is the same as CALL -151 from BASIC. You may reenter the Macro Assembler by typing D003G, 3D0G, or hitting RESET.

```
:MNTR
```

RST Command: ..... :RST expression

Changes the Apple II RESET vector to the specified value. If you are using the Autostart Monitor, pressing the RESET key causes a branch to the address in the RESET vector. Normally this is set to $3D0 by DOS to reenter the assembler, but you may change it to enter the monitor, BASIC, or your own 6502 program.

```
:RST -151          (RESET enters the monitor)
:RST $FF69         (also enters the monitor)
:RST $3D0          (RESET enters DOS and assembler)
:RST $300          (RESET enters program at $300)
```

USR Command: ..... :USR whatever

An open-ended command, waiting for you to design and activate with your own 6502 code.

When you type the command "USR", a JSR $D006 ($3006) instruction is executed. If you have not installed a JMP to your own 6502 program at $D006, the command is equivalent to a "No Operation" command. You can write a 6502 program to process your own command, and put a JMP instruction to it at $D006.

The entire command line is stored in the monitor input buffer, starting at $0200. Your USR command processor can scan the input buffer to pick up any parameters you wish.

Remember that USR calls and runs 6502 code, not 68008 code.

DOS COMMANDS

All the Apple DOS commands are valid, even though you are operating from within the Macro Assembler. This feature allows you to maintain your source and object programs on disk using the LOAD, SAVE, BLOAD and BSAVE commands. Source programs appear in the disk catalog with a type code of "I", just as though they were Integer BASIC programs.

Housekeeping Commands: CATALOG, RENAME, DELETE, LOCK,

UNLOCK, VERIFY, MON, NOMON, and MAXFILES can be used as you desire. They function exactly the same within the Macro Assembler as they do within BASIC,

Source Maintenance Commands: LOAD and SAVE when used with a filename are interpreted by DOS. If no filename is included, the Macro Assembler interprets them as cassette tape commands.

Object Maintenance Commands: BSAVE, BLOAD, and BRUN commands may be used to maintain object programs on the disk and to execute them. Be careful when using BLOAD and BRUN that the program you are loading does not load on top of anything you want to keep. And remember that BRUN applies only to 6502 programs, not those for the Q-68 board!

I/O Selection Commands: PR#, IN#, and EXEC commands may be used. PR#(slot) activates Apple intelligent interfaces for printers and other devices. IN#(slot) may be used with other terminals, modems, et cetera. EXEC executes a stream of commands or reads in a series of source lines from a text file.

BASIC Commands: INT and FP may be used to exit the Macro Assembler and enter either Integer Basic or Applesoft.

Commands you should not use: RUN, CHAIN, and INIT will not do what you expect. Avoid typing the "RUN filename" command, because it would be recognized by DOS as an attempt to load and execute an integer BASIC or Applesoft program. However, since the DOS links have been set up for the Macro Assembler, the program would not execute. It would just clobber memory, possibly your source program or the assembler itself!

The CHAIN command is equally dangerous. INIT will properly format a disk, but it writes your source program (which is not executable) as the HELLO program! It is much better to INIT from within Applesoft or Integer BASIC.

MONITOR COMMANDS

All of the Apple II Monitor commands are available from within the Macro Assembler. You use them by typing a dollar sign ($) after the prompt symbol, followed by any monitor command.

Monitor commands are explained on pages 40-66 of the Apple II Reference Manual. With these commands you may examine, change, move or verify memory; read and write cassette tapes; disassemble 6502 machine language programs; execute 6502 programs; and perform hexadecimal arithmetic.

Section 3.6 -- Directives

Twenty assembler directives are available in the Macro Assembler to control the assembly process and to define data in your programs. Directives are indicated by a period followed by two or more letters.

.OR	Origin	.BS	Block Storage
.TA	Target Address	.LIST	Control Assembly Listing
.TF	Target File	.TI	Title
.IN	Include File	.US	User defined directive
.EN	End of program	.PG	Page eject
.EQ	Equate	.DO	Conditional Assembly
.DA	Data	.ELSE	Conditional Assembly
.HS	Hex string	.FIN	Conditional Assembly
.AS	ASCII string	.MA	Macro definition
.AT	ASCII terminated	.EM	End macro

Origin:..... .OR expression

Sets the program origin and the target address to the value of the expression. The origin is the address at which the object program is to be executed. Target address is the memory address at which the object program is stored during the assembly. The .OR directive sets both of these to the same value, which is the normal way of operating.

The origin of a program may be set to any value from $Ø through $FFFFFFFF. However, if you are assembling to memory, you must specify a target address (where the object code is stored) somewhere in the Apple's memory range. If you use a target file (.TF), the DOS BLOAD address will be the same as the low order 16 bits of the origin. If you want to load it elsewhere from disk, specify the address parameter with the BLOAD parameter in the normal way.

If you do not use the .OR directive, the assembler sets both the program origin and the target address to $1ØØØ. If the expression is not defined during pass one prior to its use in the .OR directive, an error message is printed.

If a .TF (Target File) was active before the .OR directive, it is closed out when the .OR directive is encountered.

Target Address: ..... .TA expression

Sets the target address at which the object code is stored during asssembly.

The target address is distinct from the program origin (which is either set by the .OR directive, or is implicitly set to $1000). The .OR directive sets both the origin and the target address; the .TA directive sets only the target address. Object code is produced ready to run at the program origin, but is stored starting at the target address.

We used the .TA directive to assemble code for the Q-68 board MONITOR. This code, which is in the onboard EPROM, starts at location $10000. This address is unknown to the Apple, since Apple memory extends only up to $FFFF. But we had to store the object code somewhere, in preparation for burning it into an EPROM. The EPROM burner we use looks for data starting at Apple location $4000. So we preceeded the assembly code of the MONITOR program with:

```
.OR     $10000
.TA     $4000
```

Although the object code produced by this assembly is designed to run at starting address $10000, it is stored at starting address $4000.

Target File: ..... .TF filename

Causes the object code to be stored on a binary disk file, rather than in memory. Only the code which follows the .TF directive is stored on the file. Code is stored in the file until another .TF directive is encountered, or until a .TA or .OR directive is encountered.

The filename specifier may include volume, drive, and slot numbers if necessary. If you have both .IN and .TF directives in the same assembly, and the files involved are not on the same disk, you need to specify drive number (and maybe slot number) with every .IN and with every .TF directive.

```
1000    .TF     OBJ,S6,D1
```

This example causes object code to be saved in a disk file called "OBJ", on the slot 6, drive 1 disk drive.

Apple object code, which is saved as a "B" type file, allows only one load address. Therefore, if your program consists of several pieces with different origins, each piece must be stored as a separate disk file. This requires a .TF directive (and a different file name) for each section.

```
1000        .OR         $1000
1010        .TF         PROG1
 :
 :
2000        .OR         $10084
2010        .TF         PROG2
```

If you later do a "BLOAD" of PROG1, it will load at $1000; and if you do a "BLOAD" of PROG2, it will load at $0084 (only the bottom 16-bits are used). The load address may be overridden by including the "A$" directive in the BLOAD operation. For example, "BLOAD PROG2, A$4000" would load PROG2 starting at address $4000.

During assembly, the Macro Assembler temporarily patches DOS to allow a binary file to be handled with text file commands. It also creates a text file with your specified name and uses text file techniques to write the object code into the file. When assembly is complete, or when the .TF range is ended by encountering another .TF (or .TA or .OR), the text file is transformed into a binary file by modifying the DOS directory entry.

If you have typed "MON C" (a DOS command) before assembly, the DOS commands issued by the assembler for the .TF directive are printed on the assembly listing. For each .TF directive, during pass two, you will see the following sequence:

```
OPEN file name
DELETE file name
OPEN file name
WRITE file name
```

If you have typed "MON O" (a DOS command), you see lots of crazy characters on the screen during pass two of the assembly. These are the object code bytes which are being written to the Target File. It is better to not set MON O mode.

Include: ..... .IN filename

Causes the contents of the specified source file to be included in the assembly.

The program which is in memory at the time the ASM command is typed is called the "root" program. Only the root program may have .IN directives in it. If you attempt to put .IN directives in an included program, you will get a "NESTED .IN" error.

When the .IN directive is processed, the root program is temporarily "hidden" and the included program is loaded. Assembly then continues through the included program. When the end of the included program is reached, it is deleted from memory and the root program is restored. Assembly then continues with the next line of the root program.

If you type the MON C command (a DOS command) before beginning assembly, the LOAD commands issued by the assembler are printed with the listing. Each included program is loaded in turn during pass one of the assembly, and again during pass two.

The .IN directive is useful in assembling extremely large programs, which cannot fit in memory all at once. It is also useful for connecting a library of subroutines with a main program. Some programmers prefer this method over the use of macros.

The filename portion of the directive is in standard DOS format, and may include volume, slot, and drive number.

End of Program: ..... .EN

Defines the end of the source program, or of an included (.IN) module. You would normally make this the last line, but you may place it earlier in order to assemble only a portion of your source program. If no .EN is present in your program, the assembler assumes that you meant to put one after the last line. Most assemblers for some strange reason go completely crazy if the .EN directive is missing!

Equate: ..... label .EQ expression

Defines the label to have the value of the expression. If the expression is not defined, an error message is printed. If you neglect to use a label with an equate directive, an error message is printed also.

Data: ..... label .DA exprlist

Creates constants or variables in your program. "Exprlist" is a list of one or more expressions separated by commas. Each expression may be treated as one, two, or four bytes, depending on how it is written.

If a # preceeds the expression, it is treated as an 8-bit value.

If a / preceeds the expression, it is treated as a 16-bit value.

If nothing preceeds the expression, it is treated as a 32 bit value.

You may isolate any 8-bit or 16-bit field within a 32 bit value by using the leading "#" or "/", together with division by an appropriate value. For example, if you want to represent the third most significant byte of a 32 bit number in a .DA directive, you could use

```
.DA     #VALUE/256
```

or

```
.DA     #VALUE/$100
```

Similarly, to specify the high order 16 bits of a 32 bit number, use

```
.DA     /VALUE/65536
```

or

```
.DA     /VALUE/$10000
```

(Don't confuse the two "/" symbols. The first means truncate to 16 bits; the second means divide).

The value of the expression, as one, two, or four bytes, is stored at the current location. If a label is present, it is set to the address where the first byte of data is

stored.

The .DA directive may be used to reserve RAM space for a variable. For example, the code:

```
COUNT     .DA       $Ø
```

reserves four bytes of RAM for later use as the variable COUNT.

If you use .DA to define a variable, it is a good habit to use an expression like "*-*", which has a value of zero. This weird expression might make your program more self-explanatory when you look at it again next year. The funny form emphasizes the fact that the data value put at COUNT has no significance--it merely reserves space.

A common use for the .DA directive is to set up the vectors for the Q-68 board. If you are going to start up the Q-68 board with the "QON" command, the following code must appear somewhere in your 68ØØØ source program:

```
.OR       $8ØØ      ;68ØØ8 page Ø
.DA       INITSP    ;initial stack pointer value
.DA       INITPC    ;initial program counter value
```

More .DA statements will follow this if you use more of the 68ØØ8 exception vectors. (If you don't know about exception vectors, take a look at Section 7).

Hex String: ..... label .HS hhh...h

Converts a string of hex digits (hhh...h) to binary, two digits per byte, and stores them starting at the current location. If a label is present, it is defined as the address where the first byte is stored. If you do not have an even number of hexadecimal digits, the assembler prints an error message.

NOTE: Unlike hexadecimal numbers used in operand expressions, you must not use a dollar sign with the .HS directive.

ASCII String ..... label .AS *aaa..a*

Stores the binary form of the ASCII characters "aaa...a" in sequential locations beginning at the current location. If a label is present, it is defined as the address where the first character is stored. The string "aaa...a" may contain any number of the printing ASCII characters. You indicate the beginning and end of the string by any delimiter (*) you choose.

ASCII character codes are seven bit values. The .AS directive normally sets the high-order, or 8th, bit to zero. Some people like to use ASCII codes with the high-order bit set to one, so the Assembler includes an option for this.

```
.AS *aaa...a*       sets the high-order bits = 0
.AS -*aaa...a*      sets the high-order bits = 1
```

The delimiter (*) may be any printing character other than space or minus.

ASCII Terminated: ..... label .AT *aaa...a*

This works just like the .AS directive, except that the high-order bit of the last byte in the string is set opposite from the preceding bytes. This allows a message-printing routine to easily find the end of a message.

Block Storage: ..... label .BS expression

Reserves a block of bytes starting at the current location in the program. The expression (range:1-65535) specifies the number of bytes to reserve. If there is a label, it assigned the value at the beginning of the block.

The address of the beginning of the block is printed in the address column of the assembly listing.

If the object code is being stored directly into memory, no bytes are stored for the .BS directive. However, if the object code is being written on a disk file by using the .TF

directive, the .BS directive writes <expression> bytes (value: 00) to the file.

Title: ..... .TI expression, title

When .TI is in effect the assembler prints a title line and page number at the top of each page. The expression specifies the maximum number of lines you want to print on each page. The title can be up to 70 characters long and is printed starting at the left margin. "PAGE xxxx" is printed immediately after the title.

If you do not specify a title, only the page number is printed at the left margin. Spacing or centering of the title and page number can be adjusted by adding leading or trailing spaces to the title.

The Macro Assembler issues an automatic formfeed when a page fills up. If you want to end a page early, use the .PG directive. You can use more than one .TI directive in a program if you like. The .TI directive issues a formfeed command when encountered in the listing.

You can turn off titles by using .TI with a pagelength of zero.

Listing Control: ..... .LIST optionlist

Controls the listing output of the assembler. "Optionlist" is a list of one or more of the following keywords:

OFF	Listing off.
ON	Listing on.
MOFF	Macro expansion listing off.
MON	Macro expansion listing on.

If .LIST OFF is put at the beginning of the source program, and no .LIST ON is used, no listing at all is produced. The program assembles much faster without a listing, as most of the time is consumed in putting characters on the screen and scrolling the screen up.

If you put .LIST OFF at the beginning of your source program, and .LIST ON at the end, only the alphabetized symbol table is printed.

You may also use this pair of directives to bracket any portion of the listing you wish to see or not see.

With .LIST MON in effect, the complete macro expansion is listed. The call line is printed with its line number, then the expansion lines, each with a line number of "0000>".

Page Control: ..... .PG

Prints an ASCII Form Feed character ($0C). If the assembly listing is being printed on a printer which recognizes this character, a form feed occurs and the next listing line appears at the top of the next page. The .PG line itself is not listed.

Conditional Assembly: ..... .DO expression
.ELSE
.FIN

With these directives, you can include or exclude a particular section of code in the assembly, depending on a condition set earlier. The operand expression is evaluated as a truth value, and must be defined before the ".DO". Zero means skip source code lines; non-zero means assemble them.

The .ELSE directive toggles the current truth value, allowing an "if...then...else" structure. There may be more than one .ELSE directive within the ".DO-.FIN" block; each time .ELSE is encountered the truth value is switched. .FIN terminates the conditional section, .ELSE is optional but .FIN is required.

".DO-.FIN" blocks may be nested up to 8 deep.

These directives are often used to produce different specialized versions of a program from the same source code. For example, the main memory and language card versions of the Macro Assembler were assembled from the same source code, using a .DO flag called LCASM. When a change

is made to the assembler, only one source line needs to be edited to generate the two different versions. The source program is assembled twice, once with LCASM=1, and once with LCASM=Ø.

".DO-.FIN" blocks can also be used to exclude testing routines from the finished program, to relocate a RAM variable area, or to add or delete extra variables.

The following page shows three ".DO-.FIN" examples.

```
1000 * CONDITIONAL ASSEMBLY DEMO
1010 *--------------------------------
1020 FLAG    .EQ      Ø
1030         .DO      FLAG
1040         BSR      PLACE1
1050         .ELSE
1060         BSR      PLACE2
1070         .ELSE
1080         BSR      PLACE3
1090         .FIN
1100         RTS
1110 *--------------------------------
1120 PLACE1 RTS
1130 PLACE2 RTS
1140 PLACE3 RTS

:ASM

                     1000 * CONDITIONAL ASSEMBLY DEMO
                     1010 *--------------------------------
00000000-            1020 FLAG    .EQ      Ø
                     1030         .DO      FLAG
                     1050         .ELSE
00001000- 6100 0006  1060         BSR      PLACE2
                     1070         .ELSE
                     1090         .FIN
00001004- 4E75       1100         RTS
                     1110 *--------------------------------
00001006- 4E75       1120 PLACE1 RTS
00001008- 4E75       1130 PLACE2 RTS
0000100A- 4E75       1140 PLACE3 RTS

:1020 FLAG    .EQ    1
:ASM

                     1000 * CONDITIONAL ASSEMBLY DEMO
                     1010 *--------------------------------
00000001-            1020 FLAG    .EQ      1
                     1030         .DO      FLAG
00001000- 6100 0008  1040         BSR      PLACE1
                     1050         .ELSE
                     1070         .ELSE
00001004- 6100 0008  1080         BSR      PLACE3
                     1090         .FIN
00001008- 4E75       1100         RTS
                     1110 *--------------------------------
0000100A- 4E75       1120 PLACE1 RTS
0000100C- 4E75       1130 PLACE2 RTS
0000100E- 4E75       1140 PLACE3 RTS
```

Here is a conditional assembly example which illustrates how a program may be assembled in two versions: A RAM version at origin $1000, or a ROM version at origin $10084.

```
:ASM

00000001-            1000 TRUE    .EQ      1
00000000-            1010 FALSE   .EQ      0
                     1020 *--------------------------------
00000001-            1030 RAM     .EQ      TRUE
                     1040 *--------------------------------
                     1050         .DO      RAM
                     1060         .OR      $1000    ;RAM ADDR
                     1070         .ELSE
                     1100         .FIN
                     1110 *--------------------------------
00001000- 4E71       1120         NOP

1030 RAM     .EQ     FALSE

:ASM

00000001-            1000 TRUE    .EQ      1
00000000-            1010 FALSE   .EQ      0
                     1020 *--------------------------------
00000000-            1030 RAM     .EQ      FALSE
                     1040 *--------------------------------
                     1050         .DO      RAM
                     1070         .ELSE
                     1080         .OR      $10084   ;ROM ADDR
                     1090         .TA      $1000
                     1100         .FIN
                     1110 *--------------------------------
00010084- 4E71       1120         NOP
```

Macro Definition: ..... .MA macro name
End Macro: .EM

A macro definition must begin with the directive .MA <macro name>, and end with the .EM directive. For detailed information, see Section 3.8 on macros.

User Directive: ..... label .US whatever

The .US directive allows for possible expansion of the assembler by users. When the .US directive is processed (both in pass 1 and pass 2), a branch is made to location $D00C ($300C). This location normally contains a 6502 JMP instruction, which treats the .US as a comment. The source line (without the line number) is in the keyboard buffer starting at $0200.

If you wish to use the .US directive, change $D00C-$D00E ($300C-$300E) to jump to your own 6502 program. Details of the steps necessary to implement your own directives are published in the September 1981 issue of Apple Assembly Line, available from:

S-C Software Corporation
2331 Gus Thomasson, Suite 125
P.O. Box 280300
Dallas, Texas 75228
(214) 324-2050

Section 3.7 -- OPERAND EXPRESSIONS

Operand expressions are written using elements and operators. The valid operators are +, -, *, /, <, =, and >. Terms may be decimal or hexadecimal numbers, labels, a literal ASCII character, or an asterisk (*). The first term in an expression may be preceded by a + or -.

Operand expressions have 32 bit precision. This gives a range of 0-4,294,967,295 decimal; and $0-$FFFFFFFF hex.

ELEMENTS

Decimal numbers: A number with no prefix is assumed to be decimal (base 10).

```
                MOVE.B    #200,D3
                MOVE.L    #-10,D4
                .DA       35691
FLAG            .EQ       -1
                .DA       4096*256*12
```

Hexadecimal Numbers: Hexadecimal numbers are preceded by a dollar sign, and may have from one to eight digits.

```
                .OR       $1000
                MOVE.B    #$2F,D5
                MOVE.W    D6,$18400
                MOVE.L    #$18000,A2
```

Beware of the missing dollar sign! The assembler may be quite satisfied to think of your hexadecimal number as a decimal one if you omit the "$". In some cases even a number with letters in it, such as 23AB, may be acceptable; it may be interpreted as decimal 23 and a comment "AB".

Labels: There are three types of labels in the Macro Assembler. Normal labels are from 1 to 32 characters long. The first character must be a letter. Following characters may be letters, digits, or periods. Local labels are written as a period followed by one or two digits. Private labels are written as a colon followed by one or two digits.

Labels must be defined if they are to be used in an

expression. Labels used in operand expressions after .OR, .TA, .BS, and .EQ directives must be defined prior to use (to prevent an undefined or ambiguous location counter). Labels are defined by being written in the label field of an instruction or in a directive line.

Literal ASCII Characters: Literal characters are written as an apostrophe followed by the character. The value is the ASCII code of the character (a value from $00 through $7F).

```
LTRA        .EQ         'A
            .DA         #'X,'A
            MOVE.B      #'Z,D3
```

If you wish to use literal ASCII values with the sign bit equal to 1 (codes $80-$FF), you can do so by adding $80 in the operand expression

```
LTRA        .EQ         'A+$80
            .DA         #'X+$80,'A+$80
            MOVE.B      #'Z+$80,D3
```

Asterisk (*): Stands for the current value of the location counter. This is useful for calculating the length of a string.

```
MES         .AS         /ANY MESSAGE/
SIZE        .EQ         *-MES

VAR         .EQ         *-*             ;0
FILLER      .BS         $900-*          ;Fill from here thru
                                        ;$8FF
```

OPERATORS

You can use arithmetic and relational operators in operand expressions. Expressions are evaluated strictly from left to right, with no other precedence implied. Parentheses cannot be used to change this order.

Arithmetic Operators (+ - * /): Any of the four arithmetic operators may be used in an operand expression.

All operations are performed on 32-bit values.

Multiplication returns the low-order 32-bits of the 64-bit product.

Overflow and division-by-zero are not considered assembly errors. Overflow merely truncates, returning the low-order 32-bits. Division-by-zero returns the value $FFFFFFFF.

Relational Operators (< = >): The three relational operators compare two 16-bit values. If the relation is true, the result is 1. If the relation is false, the result is Ø. The result can be used in further calculations, and as the truth value for conditional assembly (.DO directive).

Three elementary operators are available: Less than (<), equal(=), and greater than (>). They cannot be combined as they are in BASIC to form <=, <>, >=. However they may be used with the AND and OR operators described in the next paragraph to achieve these combined operators.

The result of a relational expression is a true or false value. A value of zero is considered to be false, and a non-zero value is considered to be true. You may operate on logical values with * and + operators: * has the effect of the logical AND, and + has the effect of the logical OR operation.

If you are in doubt how an expression will evaluate, you can use the VAL command to find out. Or you can go ahead and assemble your program and see how it turns out.

```
:VAL 56>33
00000001
:VAL 33>56
00000000
:VAL 56>33*33>56
00000000
:VAL 56>33+33>56
00000001
```

Section 3.8 -- MACROS

A macro is a single instruction in your source code, which, when assembled, is replaced by a predefined series of instructions. You can use macros as a shorthand for commonly used code sequences.

A SIMPLE MACRO

Here is a small section of code which adds two 64 bit values. The number starting at memory location $1000 is added to the number at $1008, and the result is stored starting at $1000.

```
MOVEA     #$1000+8,A0
MOVEA     #$1008+8,A1
MOVE      #0,CCR
ADDX.L    -(A1),-(A0)
ADDX.L    -(A1),-(A0)
```

We can define a macro called DBLADD to do this operation. Just add the following two directives to the program code:

```
.MA       DBLADD              ;macro name
MOVEA     #$1000+8,A0
MOVEA     #$1008+8,A1
MOVE      #0,CCR
ADDX.L    -(A1),-(A0)
ADDX.L    -(A1),-(A0)
.EM                           ;end of definition
```

Now to do the same operation in your program, simply put:

```
>DBLADD
```

Macros are indicated by the leading ">" symbol. Whenever the assembler encounters the the macro call ">DBLADD", it replaces it with the actual program lines contained between the .MA and .EM directives.

The object code will be the same with or without macros. If an operation is used only once or twice in a program, it probably isn't worth the effort to define a macro for it. But if you have to do the same operation on several different variables, a macro can save a lot of work. Macros can also help prevent common mistakes, such as incorrectly

specifying data sizes.

CALL PARAMETERS

Suppose you want to generalize the above macro to use any memory locations; not just $1000-$100F. You can "pass" parameters to macros. Parameters are values which are defined when you call the macro.

```
.MA       DBLADD              ;macro name
MOVEA     #]1+8,A0
MOVEA     #]2+8,A1
MOVE      #0,CCR
ADDX.L    -(A1),-(A0)
ADDX.L    -(A1),-(A0)
.EM                           ;end of definition
```

(You enter the "]" character on an Apple II+ by typing shift-M).

The terms]1 and]2 are called dummy variables. "Dummy" means that they are put in as place markers, to be defined later. There can be up to nine of these, from "]1" through "]9". There is also a parameter which is automatically defined. This is "]#", and it takes the value of the number of parameters passed to the macro.

Now when the macro is called, it requires two values to plug into the dummy variables:

```
>DBLADD $1000,$1008
```

Parameters are written in the operand field of the macro call line, separated by commas. If you want a parameter to include a comma or space, enclose the parameter in quotation marks. If you want it to also include a quotation mark, use two quotation marks in a row wherever you want one. For example, in the macro:

```
>SAM JONES,$1234,"ABC DEF","ABC,DEF, "" GHI"

]1 is JONES
]2 is $1234
]3 is ABC DEF
]4 is ABC,DEF, " GHI
]# is 4
```

What is actually passed to a macro parameter is a text string, which literally replaces the corresponding dummy variable.

This means that you can use expressions other than numeric constants. For example, register names may be passed to macros:

```
 :ASM

                    1000 *--------------------------------
                    1010         .MA      A
                    1020         MOVE     D]1,D]2
                    1030         .EM
                    1040 *--------------------------------
00000800-           1050         >A 1,2
00000800- 3401      0000>         MOVE     D1,D2
                    1060 *--------------------------------
                    1070         .MA      B
                    1080         MOVE     ]1,]2
                    1090         .EM
                    1100 *--------------------------------
00000802-           1110         >B D1,D2
00000802- 3401      0000>         MOVE     D1,D2
                    1120 *--------------------------------
                    1130         .MA      C
                    1140         MOVE     ]1,]1+$100
                    1150         .EM
                    1160 *--------------------------------
00000804-           1170         >C $1000
00000804- 31F8 1000
00000808- 1100      0000>         MOVE     $1000,$1000+$100
                    1180 *--------------------------------
```

In macro A, the macro supplies the "D" portion of the data register name, and the parameters 1,2 supply the numbers. In macro B, the entire register names D1 and D2 are passed.

Macro C uses a hex address, and the arithmetic expression "]1+$100" to add $100 to the passed address. Note in the assembly listing that $1000+$100 equates to $1100, as it

should.

PRIVATE LABELS

Private labels are used inside a macro definition to name branch points in the same way that labels are used in the main program. They are written as a colon (:) followed by one or two digits.

The Macro Assembler considers each private label unique to the macro in which it is used. This allows you to re-use the same private labels in different macro definitions. Private labels do not interfere in any way with local labels. Here is an example which uses both private labels and local labels:

```
 1000          .OR      $102A
 1100 *--------------------------------
 1500          .MA      SPIN
 1510          MOVE     ]1,D2
 1520 :1       DBF      D2,:1
 1530          .EM
 1540 *--------------------------------
 1542 *
 1544 LOCAL.AND.PRIVATE.LABELS
 1550          MOVE     #$22,D3
 1560 .1       >SPIN    #$50
 1570          DBF      D3,.1
```

```
:ASM
                     1000          .OR     $102A
                     1100 *-------------------------------
                     1500          .MA     SPIN
                     1510          MOVE    ]1,D2
                     1520 :1       DBF     D2,:1
                     1530          .EM
                     1540 *-------------------------------
                     1542 *
                     1544 LOCAL.AND.PRIVATE.LABELS
0000102A- 363C 0022  1550          MOVE    #$22,D3
0000102E-            1560 .1       >SPIN   #$50
0000102E- 343C 0050  0000>          MOVE     #$50,D2
00001032- 51CA FFFE  0000> :1       DBF      D2,:1
00001036- 51CB FFF6  1570          DBF     D3,.1

SYMBOL TABLE

0000102A- LOCAL.AND.PRIVATE.LABELS
.01=0000082E
```

Each private label requires five bytes of storage during assembly. This storage starts at $0FFF and works downward. Consult Appendix A on memory usage for details.

LISTING THE MACRO EXPANSIONS

There are two directives which control the appearance of macros in the asembly listing. With .LIST MON in effect, the complete macro expansion is printed. The call line is printed first, and then the assembled code on subsequent lines.

The expansion lines have line numbers of "0000>" to indicate a macro, and are indented one space. When .LIST MOFF is in effect, only the macro call line is printed. This saves space and makes the logic of the program easier to follow. You do, however, lose the listing of the object code, which shows exactly what is stored at each address.

USING CONDITIONAL ASSEMBLY IN MACRO DEFINITIONS

You can use the .DO, .ELSE, and .FIN directives inside macro definitions. They are executed during macro expansion, so

that the same macro can be expanded in different ways depending on parameters.

An example of conditional directives inside macro definitions is given in the Nested Macro Definitions section, on the next page.

NESTED MACRO DEFINITIONS

You can call macros within macro definitions. This is not recommended, since it produces convoluted and hard to follow code. Many programmers, however, delight in the intracacies of nested and recursive macros.

Suppose you want to write a macro which can be used to call one or more subroutines on a single source line. For example, CALL SAM should expand to BSR SAM. CALL SAM,TOM should expand to JSR SAM and JSR TOM, and so on. You could do it at least two ways: using conditional directives, or using nested macro definitions.

Using conditional directives is fairly straightforward. The following program shows how. The "]#" parameter is tested to determine how many parameters are passed to the macro, and thus how many BSR's to produce.

```
                        1000          .MA      CALL
                        1010          BSR      ]1
                        1020          .DO      ]#>1
                        1030          BSR      ]2
                        1040          .FIN
                        1050          .DO      ]#>2
                        1060          BSR      ]3
                        1070          .FIN
                        1080          .EM

00001000-               1090          >CALL SAM,TOM,JOE

00001000- 6100 0012     0000>          BSR      SAM
                        0000>          .DO      3>1
00001004- 6100 0012     0000>          BSR      TOM
                        0000>          .FIN
                        0000>          .DO      3>2
00001008- 6100 000C     0000>          BSR      JOE
                        0000>          .FIN

0000100C-               1100          >CALL SAM,TOM

0000100C- 6100 0006     0000>          BSR      SAM
                        0000>          .DO      2>1
00001010- 6100 0006     0000>          BSR      TOM
                        0000>          .FIN
                        0000>          .DO      2>2
                        0000>          .FIN

00001014- 4E75          2000 SAM      RTS
```

```
00001016- 4E75       2010 JOE    RTS
00001018- 4E75       2020 TOM    RTS
```

The other approach uses a nested macro definition--one which includes a call to another Macro. Three macros are set up for each possible number of parameters: CALL1 for one parameter, CALL2 for 2, and CALL3 for three. Then the macro "CALL" is used to call the appropriate one of those. The "nested" macro is in line 1160.

```
                     1000         .MA     CALL1
                     1010         BSR     ]1
                     1020         .EM
                     1030 *-------------------------------
                     1040         .MA     CALL2
                     1050         BSR     ]1
                     1060         BSR     ]2
                     1070         .EM
                     1080 *-------------------------------
                     1090         .MA     CALL3
                     1100         BSR     ]1
                     1110         BSR     ]2
                     1120         BSR     ]3
                     1130         .EM
                     1140 *-------------------------------
                     1150         .MA     CALL
                     1160         >CALL]# ]1,]2,]3
                     1170         .EM
                     1180 *-------------------------------
00001000-            1190         >CALL   SAM
00001000-            0000>         >CALL1 SAM,,
00001000- 6100 0016  0000>>         BSR     SAM
00001004-            1200         >CALL   SAM,JOE
00001004-            0000>         >CALL2 SAM,JOE,
00001004- 6100 0012  0000>>         BSR     SAM
00001008- 6100 0010  0000>>         BSR     JOE
0000100C-            1210         >CALL   SAM,JOE,TOM
0000100C-            0000>         >CALL3 SAM,JOE,TOM
0000100C- 6100 000A  0000>>         BSR     SAM
00001010- 6100 0008  0000>>         BSR     JOE
00001014- 6100 0006  0000>>         BSR     TOM
                     1215 *-------------------------------
00001018- 4E75       1220 SAM    RTS
0000101A- 4E75       1230 JOE    RTS
0000101C- 4E75       1240 TOM    RTS
```

POSSIBLE ERRORS

What happens if you supply more parameters in a macro call line than the macro definition expects? The extra parameters are simply ignored. You can use the]# parameter with conditional assembly directives to test for the correct number, if you wish.

If you do not supply enough parameters on the call line, the missing ones are assumed to be null strings.

The Macro Assembler tests for three error conditions. If you attempt to call a macro which has not been defined earlier in the program, the ***UNDEFINED MACRO ERROR is printed. If you use a .MA directive without a name in the operand field, the ***NO MACRO NAME ERROR is printed. If you use the "]" character without the digit 1-9 or the "#" character,the ***BAD MACRO PARAMETER ERROR is printed.

MACROS AND SUBROUTINES

There is a significant difference between macros and subroutine calls. A subroutine is placed in memory only once, and called from different parts of the program. A macro is inserted into your code every time you issue a macro call. A macro thus executes faster than a subroutine because no BSR-RTS is involved, but it uses more memory because it is repeated in the program whenever it is used.

Debug

The QWERTY Debugger

The Q-68 board contains an 8 Kilobyte ROM which holds a comprehensive debug package ("DEBUG") for testing your 68000 programs. This chapter tells you how to use the many features of the debugger.

WHY USE A DEBUGGER?

Here are some of the things DEBUG does for you:

1. It provides a window into the 68008 processor. You can look at everything inside--the address registers, the data registers, the program counter and the status register.

2. It lets you run your code one instruction at a time. You can pause between instructions and view results.

3. It allows you to stop program execution at any predetermined address and inspect what is going on. This is done by setting breakpoints. You can then resume execution exactly where your program left off.

4. It allows you to inspect and alter memory. You can look at memory in HEX format, in ASCII to spot character strings, or in 68000 instruction form.

5. It lets you put labels on memory locations to help you recognize important information.

6. It provides error notification and recovery for your running program.

7. It can run in a "Remote" mode, in which "keystroke" commands are passed to it from a BASIC (or machine language) program, rather than from the Apple II keyboard.

Considerable emphasis has been given to making the Debugger easy to learn and use. Once you become familiar with it's use, you won't want to test your programs any other way.

Before describing the Debugger, let's consider the kinds of errors you are likely to make in writing 68000 code. We'll see that the most difficult types to track down and fix are handled easily by the Debugger.

TWO KINDS OF ERRORS

There are two general types of errors. Those that are made when you write and assemble the program; and those made as the program actually runs.

ASSEMBLY ERRORS

The Assembler checks your typed instructions for correctness. Syntax errors account for most errors the assembler finds. These are mistyped or incorrectly specified instructions. For example if you type

```
MOV       A2,A1
```

you will get a "BAD OPCODE" error, since you typed MOV instead of the required MOVE.

Likewise, the code line:

```
MOVE      A2,#45
```

will generate a "BAD ADDRESS" error, since you can't move data into an immediate value. Both of these are syntax errors.

The assembler also reports errors related to the correct operation of the assembler. If you define the same label twice, you will get an EXTRA DEFINITION error. If you try to assemble code into memory which contains the assembler program, you will get a MEM PROTECT error.

(Appendix B lists all the Macro Assembler error messages).

RUN ERRORS

You might think that once you have cleaned up all the errors which the assembler has reported, your code is bug-free. Not so! There are errors which no assembler can catch. For example, consider the following statement:

```
MOVE.L    $45(A2,A3),(A4)+
```

The assembler is happy, since you have typed a legal instruction with a legal address mode. But what if, as your code executes, the contents of A2 and A3 become even? Adding $45 would make the resulting address odd, which is forbidden for word and long word addresses. (If you are perplexed about words and long words, see section 8 of this manual on data sizes).

The assembler can't possibly anticipate all values that A2 and A3 will assume as you run your program. This is a classic case of a run-time error which can't be detected by an assembler.

Here's where the debugger helps out. Using the extensive Exception facilities built into the 68000, the DEBUG program catches these errors and notifies you when they occur.

NOTE: If you are not familiar with Exceptions in the 68000, please take a moment to read the EXCEPTIONS section in part 7 of this manual. There you will learn that run-time errors cause program execution to be diverted through preassigned memory locations, where you (or the DEBUG program) can put addresses of routines to handle the error condition.

STARTING DEBUG

Let's walk through a typical assembly/debugging session. Boot the QPAK-68 system disk, and select item 4 if you have a language card or a IIe; item 3 if you don't.

After the assembler is loaded and running, indicated by the ":" prompt, type "LOAD DEBUG.TEST". Back at the ":" prompt again, type "LIST".

This is the test program we are going to run with DEBUG. It simply increments the registers. For a little variety,

Register D5 is incremented five times. The program loops continuously, and never stops.

Now type "ASM". The Assembly listing will scroll by. The 68000 object code is placed at location $1000.

At this point you want to actually run your code and test it. To do this, type "DBUG". This starts up the DEBUG program. You should see this on your Apple II screen:

```
REGISTERS
***REG**CONTENTS*******REG**CONTENTS****
*                                       *
*                                       *
*   D0 >00000000         A0  00000000   *
*   D1  00000000         A1  00000000   *
*   D2  00000000         A2  00000000   *
*   D3  00000000         A3  00000000   *
*                                       *
*   D4  00000000         A4  00000000   *
*   D5  00000000         A5  00000000   *
*   D6  00000000         A6  00000000   *
*   D7  00000000                        *
*                    SSP:A7  00018000   *
*                    USP:A7  00000000   *
*                                       *
**********************T-S--III---XNZVC**
*   PC  00001000   SR 0010011100000000 *
*---------------------------------------*
*                                       *
* 001000  ADDQ.W  #1,D0                 *
*---------------------------------------*
*      SET CONTENTS:                    *
*****************************************
```

Once DEBUG is running, you can start up your program code. Before doing so, you can examine your code, set up breakpoints, and initialize registers.

DEBUG gives you a unique way to select which of its options you wish to use. Each debug mode uses a distinctive display screen. Rather than memorizing several commands to select the various modes, you use the arrow keys to cycle from one screen to another.

The screens cycle between these five modes:

Registers
Memory
Disassembler
Breakpoints
Help

Each screen has a title block at the top of the display, so you won't have any problem knowing which of the five you are viewing.

Now experiment a bit with the left and right arrow keys to look at the five DEBUG screens.

All screens contain a command window that shows you the available options for the screen. Hit the ESC key to cycle the options in any window. We'll describe the five screens in detail, but for now remember this simple two step process:

1. Select the display you want by cycling the screens with the right and left arrow keys.

2. Select the operation you want inside a screen by cycling the COMMAND window with the ESC key.

FOR THOSE WHO CAN'T WAIT

Now we're going to quickly do some things with DEBUG. If you're the methodical type, and wish to completely learn the DEBUG system before actually trying it out, please skip ahead to the next section, "THE FIVE DEBUG SCREENS".

Using the left or right arrow key, select the DISASSEMBLY screen. See the DEBUG.TEST program? The default display address is $1000, which just happens to be the default origin for the Macro Assembler.

Hit RETURN a few times to scroll the disassembly listing. As you scroll past location $101E, the end of the test program, you will see some interesting garbage. This is DEBUG's attempt to translate what happens to be in Apple memory into 68000 instructions. Some memory values will translate into legal instructions; some will not. Those which don't are indicated by "????".

Now hit the comma key a few times (this is the same as shift-comma, or left caret). The listing scrolls

backwards! This is no mean trick--we'll tell how we do it later.

Let's disassemble a different part of memory. See the "SET ADDRESS" message at the bottom of the screen? Hit ESC a few times and this line changes to SET COMMENT and back to SET ADDRESS. This is the command window showing you the available DISASSEMBLY screen options. Cycle it to SET ADDRESS.

Now type 1Ø84, return. This is the starting address for DEBUG. DEBUG is looking at itself!

Enough of this. We don't want to tell you how easy it is to disassemble the DEBUG program.

Use the right or left arrow key to select the REGISTERS screen.

See the "PC" value of ØØØØ1ØØØ? This is where program execution will start. The small window at the bottom of the screen shows you the disassembled instruction residing at $1ØØØ--ADDQ.W #1,DØ, the first instruction of our test program.

Now hit CTL-T once. Three things happen:

1. Register DØ changes from Ø to 1.
2. "PC" changes from 1ØØØ to 1ØØ2.
3. The disassembled instruction scrolls up, and a new one at address 1ØØ2 is shown.

You actually saw the "ADDQ #1,DØ" instruction execute! The disassembly window shows the next instruction up for execution. Hit CTL-T again and this one executes. Keep hitting CTL-T and watch the instructions and the registers. You see everything: The last instruction, the next instruction, and the effect on the registers!

Those of you who are lucky enough to own an Apple IIe, hold down the CTL and T keys. This produces a fast automatic single-step. You II and II-plus owners have to hold down three keys, REPT, CTL and T. CTL-T means TRACE, which lets you execute instructions one step at a time.

Now type CTL-G. This runs your program at full speed. Unfortunately, this program continuously loops, and there is no way to stop it!

Except by hitting CTL-B (BREAK), that is. Try it now. Hear

the little "zip" sound? You'll hear this anytime a break occurs (when you use CTL-B, or when a breakpoint is encountered). Now look at the registers. They have counted up pretty far, haven't they? Why is D5 different than the others? Because 5 was added every time 1 was added to the other registers.

Try a few more CTL-T's. Now you're single stepping from where the break occured. You can use CTL-T, CTL-G and CTL-B all you want, and DEBUG keeps the program executing on track.

You can watch any of the DEBUG screens as your program executes. For example, you probably would want to watch the MEMORY screen if your program alters memory values. You can switch back and forth between screens as you single step or run your code with CTL-G.

Now for a surprise. Hit CTL-V. There's the source listing that was on the screen before you issued the "DBUG" command. CTL-V lets you watch Apple II video screens as your code runs. You're now looking at the Apple's TEXT screen. Hit CTL-V again, and then again. You should see graphic garbage. These are the (uninitialized) Apple HIRES screens #1 and #2. One more CTL-V takes you back to the DEBUG screen.

Now let's set a breakpoint. Make sure that you are viewing the DEBUG screen (use CTL-V if you're looking at an Apple screen). Now select the BREAKPOINTS screen with the left or right arrow.

Notice that the command window shows "SET ADDRESS". Type 101E, RET. You've just set your first breakpoint. Go back to the REGISTERS screen. Let's start the program at the beginning, address $1000. Hit CTL-P. The command window now shows, "SET PC:". Type 1000, RET. This is how you specify the run address.

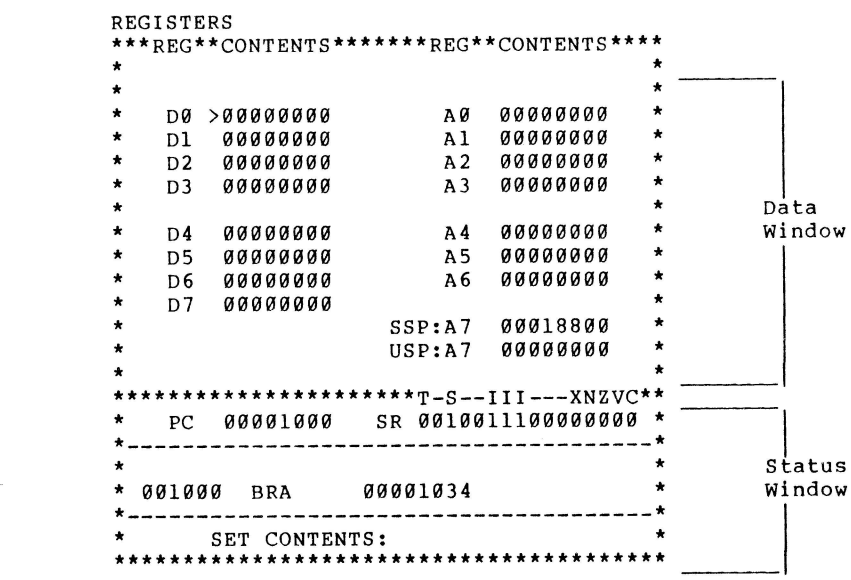
Now hit CTL-G for GO. The program starts at your "PC:" address, $1000. Hear the Breakpoint sound? The message at the bottom of the screen shows you which breakpoint you hit. Now hit CTL-G a few more times. Each time Breakpoint #0 is encountered, all the registers in the program have been altered.

If you wish to clear the BREAKPOINT #0 message, simply select a different screen, and then reselect the REGISTERS screen.

With this brief tour of DEBUG's capabilities, we're now ready to investigate the intricacies of the DEBUG program.

THE FIVE DEBUG SCREENS

Before looking at the five modes in detail, let's take a look at the general screen layout.



The screen is divided into two windows, the Data Window, and the Status Window. As you cycle through the five screens, the Data Window changes and the Status Window remains the same.

THE DATA WINDOW

The top line shows the title of the active screen.

The second line contains the column headings for the data displayed in the window.

The next fourteen lines show the various DEBUG data screens. Here you will see registers, memory locations, disassembled 68000 instructions, and breakpoints.

THE STATUS WINDOW

The Status Window appears in the bottom eight lines of the screen.

The first line, which forms the top of the Status Window, contains the Status Register bit names. These are:

T	Trace mode
S	Supervisor State
III	Three Interrupt Level bits
X	Extend bit
N	Sign (negative) bit
Z	Zero bit
V	Overflow bit
C	Carry bit

Unused bits in the 16 bit Status Register are indicated by a dash (-).

The second line shows the values of the Program Counter (PC) and the Status Register (SR).

When you start execution of your code with a CTL-G command, your program starts (or resumes) at the address shown by "PC". If you are single stepping through your program, the "PC" value indicates the address of the instruction which will be executed next. If you hit a breakpoint, the "PC" value indicates the next instruction to be executed.

You may use the CTL-P command from any Data Window to change the value of "PC". For example, if you want to start executing a piece of test code at address $2200, use CTL-P to set "PC" to $2200; then type CTL-G (GO).

The remainder of this line shows the individual bits in the Status Register. You may use CTL-S from any screen to change these.

The next two lines show disassembled 68000 instructions. The lower line shows the instruction to be executed next. It is the one which resides at the indicated "PC" address. The upper line shows the instruction which was last executed. In the Trace mode, where you use CTL-T to single-step through your program, the two line instruction window "scrolls" up every time you trace one instruction.

The next line in the Status Window shows the available commands for the currently displayed screen. Whatever you type at the keyboard is echoed on this command line. The commands in this window cycle when you hit the ESC key. In this way you can immediately see the available options for every screen mode without referring back to this manual.

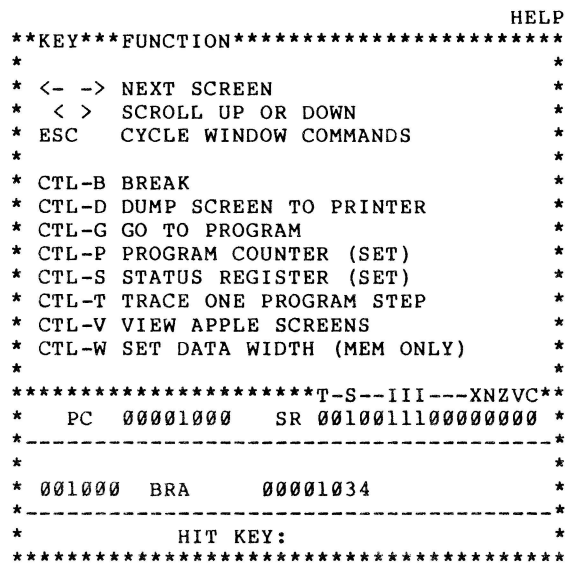
The bottom line of the Status Window looks like the bottom of a frame most of the time. But when an exception or breakpoint is encountered, the appropriate message appears here.

The Program Counter and Status register are included in the Status Window portion of the screen to allow you to jump back to your program from any DEBUG display screen. To do this, you type CTL-G or CTL-T to resume operation at the PC value shown in the window, or type a new PC value then CTL-G to resume somewhere else.

Now that we've discussed how the screens work, let's take a look at each of them in detail.

We'll start with the HELP screen, and use it to describe most of the command options for DEBUG.

THE HELP SCREEN



Left and Right Arrow Keys.....CYCLE SCREENS

The right arrow key moves you "forward" one screen, and the left arrow key moves you "back" one screen. Notice that the titles at the top of the screen are arranged like index tabs, which move in the direction of the arrow keys.

Left and Right Caret Keys.....SCROLL SCREEN UP OR DOWN

These keys move data in the Data Window. In the MEMORY and DISASSEMBLY screens, they scroll data up and down.

In the REGISTERS screen, they move the register pointer.

In the BREAKPOINTS screen, they move the breakpoint pointer.

When using "CTL-S" to set the Status Register bits, they move the cursor which indicates the update bit.

These keys are actually "Shift-comma" and "Shift-period" on the Apple II keyboard. You don't need to use the shift key. DEBUG interprets (,) as left caret and (.) as right caret.

Escape Key.....CYCLE WINDOW COMMANDS

"ESC" cycles the command options in the Status Window to show you the available choices.

CTL-B.....BREAK

CTL-B initiates a 68000 program break. It stops execution of a program started by CTL-G from DEBUG, and returns control to the DEBUG program. When a CTL-B break occurs, the PC value in the Status Window indicates the address of the next instruction to be executed.

CTL-B functions like a breakpoint you have inserted using the BREAKPOINT screen, except it is manually activated from the Apple II keyboard. Unlike a breakpoint, you can't determine in advance where the break will occur in your program. CTL-B is usually employed after long periods of embarrassing silence, when your program seems to be running but apparently is not doing the right thing.

After using CTL-B, you can single step your program (with CTL-T) to find out what is going on.

CTL-D.....DUMP SCREEN TO PRINTER

When you hit CTL-D, whatever is on the Apple II screen is sent to the currently selected output device. Nonprintable characters, such as the inverse blank used for the frame,

are printed as asterisks (*).

If you have not previously initialized a printer card, the screen dump appears on the Apple II screen in a rather bizarre form. Because the character codes on the Apple II screen do not represent true ASCII values, they must be modified before sending them to a printer.

These ASCII values appear as inverse and flashing characters on the Apple II screen.

If you see this type of display after using CTL-D, either your printer card is not selected, or it is set to echo printed characters to the screen. To clear the garbage from the screen, simply use the arrow keys to go to another DEBUG screen, then back to the desired one.

CTL-D works only if DEBUG has been started with the assembler "DBUG" command.

CTL-D prints the entire 24 line DEBUG screen, followed by nine blank lines. This fits two DEBUG screens onto a standard 66 line printout.

Appendix D contains a discussion of how the CTL-D function is implemented.

CTL-G.....GO

CTL-G starts or resumes execution of the 68Ø08 program from the address shown as "PC" in the Status window. If this is not the address you want, change it with the CTL-P operation first.

CTL-P.....Set Program Counter

CTL-P allows you to change the value of the Program Counter displayed in the Status window. The CTL-G (GO) and CTL-T (Trace) commands start execution of your 68Ø08 program at this address.

CTL-S.....Set Status Register

Just as you can specify the starting PC value, you can also preset the Status Register bits before executing your code. To save the trouble of entering sixteen ones and zeros every time, a screen cursor is used to select individual bits. Use the left and right caret keys to cycle the cursor over the sixteen bits. To update a bit, type the value 1 or Ø. When you are done, press RETURN.

DEBUG remembers the last bit updated, and positions the cursor over that bit the next time you type CTL-S. This allows you to quickly change the same bit every time you use the CTL-S function.

NOTE: You can change the unassigned Status Register bits (shown with a dash) if you wish, but they have no effect when the 68ØØ8 starts running.

CTL-T.....TRACE ONE PROGRAM STEP

CTL-T lets you trace your program one step at a time. In Trace, a single instruction is executed, and then control is returned to DEBUG. Whichever screen you are viewing remains displayed during TRACE.

For example, if you wish to watch the registers change instruction by instruction, select the REGISTER DISPLAY screen and activate Trace. Every time you hit CTL-T, you will see the register activity.

Note that you can single-step quickly through your code by holding CTL-T down on the IIe, or using the REPT key (as you hold down CTL-T) on the II+.

To start the trace operation at any address, first set "PC" using the CTL-P function, then use CTL-T.

CTL-V.....VIEW APPLE SCREENS

The Apple II contains preassigned memory buffers for three

displays. These are shown below:

$0400-$07FF	Text Screen
$2000-$3FFF	Hi-Res graphics Page 1
$4000-$5FFF	Hi-Res graphics Page 2

The View command lets you look at the Apple Text Screen and the two high resolution graphics screens while debugging your 68000 code. View is a cyclic command. Repeatedly pressing CTL-V causes four displays to cycle:

DEBUG
TEXT
HIRES Graphics Page 1
HIRES Graphics Page 2

Pressing CTL-F steps the display screens in the reverse order. This command is not shown in the HELP screen.

DEBUG uses the Text Screen for display. What if you are debugging a 68008 program which also writes to the text screen? Whatever your program puts on the text screen would be wiped out every time DEBUG uses the screen.

To prevent this, if you are viewing the Apple text display during GO or TRACE, the DEBUG program copies the text screen data into it's onboard RAM prior to using the screen. Then, just before your code is executed, the contents of the RAM are copied back into the primary text screen memory space.

IMPORTANT NOTE: This screen save occurs ONLY if you are viewing the Apple II's primary text screen during debugging.

While you trace your program, DEBUG remembers which display you last viewed. For example, if you are debugging a plotting routine which uses Hires screen #2, you can use CTL-V to display this graphics screen, then repeatedly use CTL-T to watch the plotting taking place one instruction at a time.

If you want to check the register contents during such a trace, you can use CTL-V to take you back to the DEBUG screen, then press the right arrow key, if necessary, to view the REGISTERS screen. Using CTL-V again will then take you back to the HIRES screen display.

CTL-W.....Select displayed data width

When viewing the MEMORY screen, you can display three different data sizes--byte, word, and long word. Hitting the W key in MEMORY mode cycles between the three.

THE REGISTERS SCREEN

```
REGISTERS
***REG**CONTENTS*******REG**CONTENTS****
*                                       *
*                                       *
*   D0 >00000000         A0  00000000   *
*   D1  00000000         A1  00000000   *
*   D2  00000000         A2  00000000   *
*   D3  00000000         A3  00000000   *
*                                       *
*   D4  00000000         A4  00000000   *
*   D5  00000000         A5  00000000   *
*   D6  00000000         A6  00000000   *
*   D7  00000000                        *
*                    SSP:A7  00018800   *
*                    USP:A7  00000000   *
*                                       *
**********************T-S--III---XNZVC**
*   PC  00001000   SR 0010011100000000 *
*---------------------------------------*
*                                       *
* 001000  BRA     00001034              *
*---------------------------------------*
*       SET CONTENTS:                   *
*****************************************
```

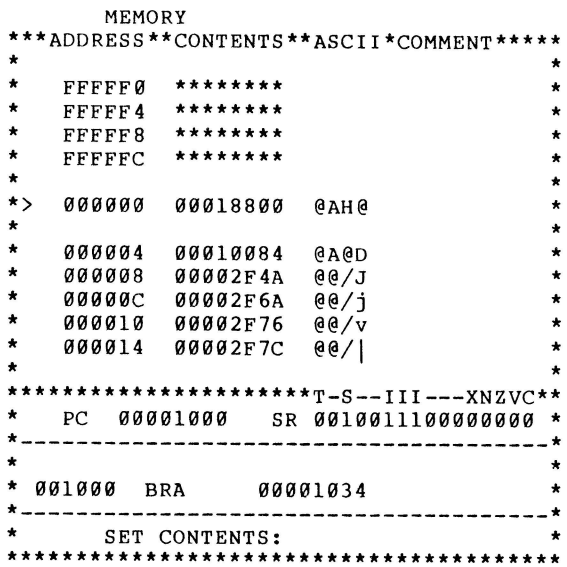
This screen displays the eight Data Registers and the nine Address Registers inside the 68008. Register "A7" is shown as two registers: "SSP:A7" and "USP:A7". This helps to clarify the Motorola terminology in which two system stack pointers have the same name, "A7". (See Chapter 7, "Two Stack Pointers Named A7").

A single command is used in the REGISTERS screen: "SET CONTENTS". The left and right caret keys move a pointer to each of the registers. To change a register value, simply type in the new value, followed by RETURN.

If you type more than 8 digits, the excess leading digits are ignored. If you type fewer than 8 digits, the remaining leading digits are set to zero. For example, typing the value "1" and RETURN enters the value "00000001". All data is in hexadecimal. You do not have to supply a leading "$".

When you use GO or TRACE, the values shown on the REGISTERS screen are copied into the 68008 registers prior to beginning program execution.

THE MEMORY SCREEN



This screen allows you to scan the 68000 memory space. This includes all of the Apple memory, as well as the Q-68 board ROM and RAM. You may view and alter the contents of memory with this screen.

The ">" and "<" keys are used to scroll data up and down. The RETURN key also scrolls the screen up.

The memory data is shown in 32 bit (long) form. To change the data width, use CTL-W. This cycles the displayed data width in the following sequence:

Long--Word--Byte--Word--Long--Word--Byte--Word, (etc.)

DEBUG remembers the data size for the MEMORY screen. If you have selected Word length (for example) and leave the MEMORY screen, the data size will still be "word" next time you select the MEMORY screen.

Memory data is displayed and entered in hexadecimal. No leading "$" is required.

The contents of non-existent memory are shown as "****". The ASCII column is blank for non-existent memory.

Pressing the ESC key selects three commands:

1. SET CONTENTS Insert data into a memory location.
2. SET ADDRESS Select a new memory location.
3. SET COMMENT Type up to 10 characters which "mark" a particular memory location.

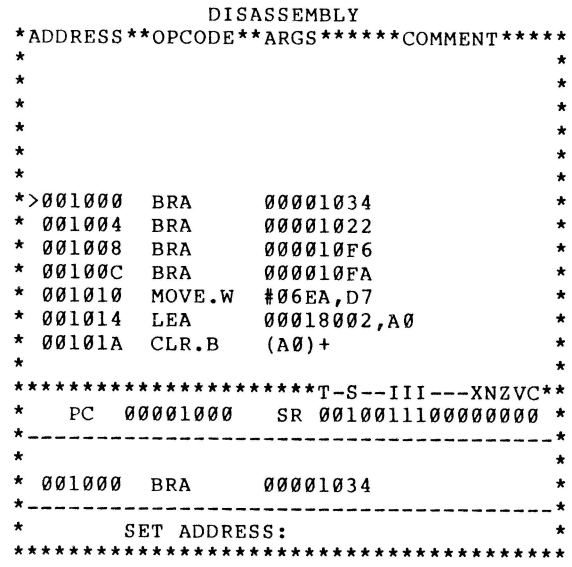
SET CONTENTS lets you type in new memory values (in hex).

SET ADDRESS allows you to view/change a different portion of memory. This is the command which appears whenever you select the MEMORY screen.

SET COMMENT allows you to "tag" certain memory locations with words of your choice. These comments are stored in a table and shown anytime the corresponding memory locations are displayed. Up to sixteen ten-character comments may be stored.

One use for the comments is to assign labels to breakpoint addresses. Then when the breakpoint is hit, the word (which might say something like "delay") will appear on the screen.

THE DISASSEMBLY SCREEN.



Two commands apply to the DISASSEMBLY screen, "SET ADDRESS" and "SET COMMENT". These function exactly as in the MEMORY screen.

The ">" key (or RETURN) steps the display up to advance one disassembled instruction. The "<" key moves the display one instruction back in memory. As with all disassemblers, you must start the display on a legal instruction address for the listing to make sense. Memory values which do not correspond to legal 68000 instructions are shown as "????".

The scroll keys allow you to move up and down in the disassembled listing. When you scroll down (moving backwards through memory) DEBUG goes back to your starting address and works forward one less instruction. You'll notice that the further away you are from the starting address, the longer the backwards scroll takes. To speed it up, simply specify a DISASSEMBLE start address closer to the code you are examining.

A POINTED ISSUE

When you use the SET ADDRESS command in the MEMORY or the DISASSEMBLER screens, should the display address for the other screen change to reflect the new display address? This was a hotly debated issue among the Qwerty staff.

One school of thought has it that you want the display addresses to "track". When you scroll one of the screens, the other should follow it in memory, even though it is not currently displayed.

The other view is that you might want to watch a particular section of memory as you trace program execution. A good use of this feature would be to watch a stack in operation. For example, suppose the stack is at $18400, and your program is at $1000. If you are watching your program on the DISASSEMBLER screen and then switch to the MEMORY screen to look at the stack, you DON'T want the memory screen to show you your program at $1000. You want it to stay at $18400, where the stack is.

Both points are valid, so we have provided a mechanism to operate both ways. There are two rules:

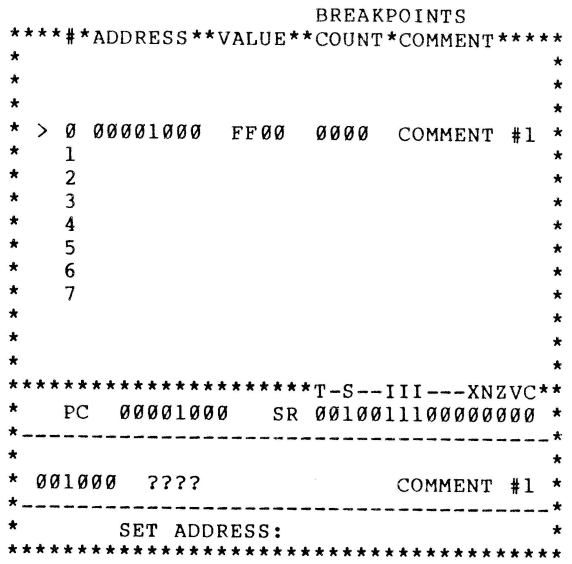
1. Whenever you set the display address to the same value for the MEMORY and DISASSEMBLER screens, the MEMORY display address tracks the DISASSEMBLY display address. In other words, the MEMORY screen "scrolls" when you scroll the DISASSEMBLY screen, even though you are not viewing the MEMORY screen.

   This is the way the screens are initially set in DEBUG, with the initial display address for both screens set to $1000.

2. Whenever you set different addresses for the two screens, they independently show their own display addresses.

An inverse "L" appears in the lower right corner of the Status window when the display addresses are locked together.

THE BREAKPOINTS SCREEN



There are three command choices in the BREAKPOINTS screen.

SET ADDRESS.

Set the address at which a break is to occur.

SET COUNTER

The count value (in hex) tells DEBUG how many times to skip over the break address before actually doing a break operation.

The Counter value is convenient for testing repetitive subroutines. You might want to see the result of a subroutine after the hundredth time through, for example. In this case, set the counter to 63 (hex for 99).

As the breakpoint address is encountered, the onscreen COUNTER value is decremented. When it hits zero, the program break is initiated.

SET COMMENT

This is the same "SET COMMENT" function as used in the MEMORY and DISASSEMBLY screens. The comment appears in the Status Window when the breakpoint is hit.

TRAP #15

When you set a breakpoint, DEBUG goes to the address you have specified and retrieves the 16-bit word it finds there. The "Value" column in the BREAKPOINTS Data Window shows this 16-bit word.

When you use CTL-T or CTL-G to begin execution of your program, the word at each breakpoint address is replaced with a "TRAP #15" instruction. When you hit a breakpoint, the original value at the breakpoint address is restored.

Therefore, you will never see the TRAP #15 instruction used for breakpoints when you disassemble your program, even if breakpoints are set.

The "TRAP #15" instruction used by DEBUG is stored in the first word of onboard Q-68 RAM ($18000). You can change this to another trap value if you wish, by replacing these two bytes with a different TRAP instruction. This replacement must be done by your test code, after DEBUG has started.

EXCEPTION VECTORS

DEBUG installs nine exception vectors when it is started. The vector addresses point to routines within DEBUG which display the exception type when any of the nine is encountered. The following table shows the vector types and the 68000 addresses at which DEBUG installs them:

$008	Bus Error
$00C	Address Error
$010	Illegal Instruction
$014	Divide by Zero
$018	CHK Instruction
$01C	TRAPV Instruction
$020	Privilege Violation
$028	1010 Instruction
$02C	1111 Instruction

(Remember that to translate these to Apple II addresses, add $800).

When any of these exceptions happens, DEBUG shows the corresponding message on the bottom line of the DEBUG screen. The message appears no matter which DEBUG screen you are viewing at the time. You must be viewing a DEBUG screen, however, to see the message--it will not appear on the Apple text or HIRES screens.

If you wish to override these vectors with your own program, simply add code to write new values (addresses) to the vector locations. Remember, though, every time you start DEBUG (with the "DBUG" command), the above DEBUG message vectors are (re-)installed.

REMOTE MODE

DEBUG has a remote mode, in which you can write APPLESOFT BASIC programs that mimic the Apple II keyboard. In the Remote mode, DEBUG looks at location $00FD for keyboard input, rather than the normal $C000.

The Remote mode is started by executing a call or jump to Apple location $303 (771 decimal). The "Q-68.STARTUP.BIN" program must be in memory for this to work. Appendix D gives details on how to use the Remote mode.

DEBUG MEMORY USAGE

DEBUG is contained in the 8 Kbyte EPROM on the Q-68 board. The first 132 bytes of the EPROM contain the self-test program described in Chapter 5. DEBUG starts at $10084 and ends at $11FFF, the top of EPROM memory.

DEBUG uses the 2 Kbyte RAM on the Q-68 board for all working storage. The SSP is located at $18600, and the 68000 stack works downward from there. The top two pages of onboard RAM, from $18600 through $187FF, are unused by DEBUG. You may use these 512 bytes for your own programs.

When you start DEBUG, the value of the SSP shown on the screen is $18800. If you do not initialize the SSP yourself (from the Registers screen or in your program code), your supervisor stack will be at $18000 (a good place for it, by the way).

If your code is going to use the USP, be sure to initialize the USP register to point to the place in memory which you want to locate the user stack. (Section 8.4 discusses stack pointers).

The only Apple memory used by DEBUG is the TEXT screen, from $400 through $7FF. DEBUG updates this screen by writing the memory directly.

USEFUL DEBUG ROUTINES

DEBUG contains some 68000 subroutines which you can call from your own programs. These routines let you disassemble one line of 68000 code, display hexadecimal numbers on the Apple II screen, and make the breakpoint "zip" sound.

DISASSEMBLE ONE LINE OF 68000 CODE.

Initial Conditions:

Register A0 is loaded with the address of the the first byte of the instruction to be disassembled. This must be an even address (not checked by this subroutine).

Calling Address: $10E4A

Result:

The disassembled code is written to a 64 byte buffer starting at address $181CE. This buffer will contain a replica (in Apple display character codes) of the result you see on the Apple screen when 68000 code is disassembled.

For example, if you put the address of a non-68000 instruction in A0 and call this routine, you will find the display codes for the address, two spaces, and four question marks in the buffer.

Note that the 64 byte buffer can accomodate the maximum length 68000 instruction. DEBUG copies the first 40 columns of this opcode onto the 40-column Apple screen.

Exit Conditions:

For a legal opcode, A0 is advanced to the byte following the disassembled instruction. This will normally be the starting byte of the next instruction. Repeated calls to this routine will thus disassemble sequential instructions without the necessity of reloading A0 every time.

For an illegal opcode, A0 will be advanced two bytes.

Register Usage: all except A7

PUT MESSAGE ON APPLE 40 COLUMN SCREEN

Initial Conditions:

A1: Screen starting address ($400-$7FF)
A3: Start of message (Apple display codes)

Calling Address: $11AB8 for normal video
$11AC8 for inverse video
$11ACE for flashing video

Result:

The message is written to the Apple text screen.

The message string must have all character codes except the last one stored with the MSB set to 1. (This is done in the Macro Assembler by preceeding the string with a "-"). The last character code of the message has its MSB set to 0.

This is the "terminator" which signals the end of the message.

Exit Conditions:

A1: One byte past the end of message on the screen.
A3: One byte past the terminator.

NOTE: DEBUG makes no check for messages that spill off the display. You need to make sure this does not happen. This would happen, for example, if you try to put a 20 character message starting at column 30.

Running off the screen can have disasterous results in Apple II operation. For example, you could overwrite peripheral card scratch locations.

Register Usage:

Normal display:	D0
Inverse:	D0, D6
Flash:	D0, D6

DISPLAY HEX AND BINARY DIGITS

Initial Conditions:

D0: data (right justified hex digits or 16-bit binary)
A1: starting screen address ($400-$7FF).

Calling Address:

One hex digit:	$11BEE
Two hex digits:	$11BF4
Four hex digits:	$11BFA
Six hex digits:	$11C00
Eight hex digits:	$11C06
16-bit binary:	$11C24

Result:

Value displayed on screen.

Exit conditions:

A1: next display address

Register Usage:

Hex: D0, D2, D3
Binary: D0, D2

ZIP SOUND

Initial Conditions: none

Calling Address: $11EEC

Result:

Sound heard when you hit a breakpoint.

Exit Conditions: none

Register Usage:

D4, D5

Self-Test

Chapter 5 -- SELF TEST

This section tells you how to use the self test feature of the Q-68 board. We'll describe a test procedure which uses the onboard diagnostic program. For every symptom, we'll tell you which Integrated Circuits are suspect. All IC's on the board are in sockets, so they are easy to replace.

CAUTION: When removing or inserting IC's be sure that the power to the system is OFF. Changing components with the power on can cause multiple component failures, and severely aggravate any problems you may be experiencing.

PRELIMINARY TESTS

Before testing the Q-68 board, you should answer the following questions:

1. Does your Apple II run with the Q-68 board removed? If not, you have a system problem which should be fixed before troubleshooting the Q-68 board.

2. Is the Q-68 board in the correct slot? The software supplied on disk assumes slot #4 operation. You can use a different slot if you reconfigure using item 1 of the menu you see when you boot the QPAK-68 disk.

3. Is the board pushed all the way down in the slot connector? If not, try removing and reseating the board.

4. Are there two little jumper plugs in the upper right corner of the board? Compare your board with the board photo at the end of this section. Your board should have jumper plugs in positions 2 and 3.

Still with us? Too bad. It appears that something might be wrong with the Q-68 board.

Since the board was thoroughly checked out and burned in at the factory, the most likely cause of a fault is an IC that went south. The following procedure takes you through a procedure which is designed to isolate a bad chip.

This test procedure assumes a few things. First, that you don't wish to take the time for a warranty repair, but prefer to try to fix it yourself. Second, that you can remove and replace socketed IC's without mangling the pins. And third, that you have access to TTL parts, and can replace certain ones which the procedure reveals to be suspect.

THE OBLIGATORY WARRANTY STATEMENT

Don't be afraid that using the following test procedure will void your warranty. As long as you put the board back into the shape that you received it before sending it back, we'll honor the warranty.

The warranty will be INSTANTLY VOIDED, however, if you touch the board with a soldering iron, or manhandle it. It doesn't take much scuffing to break the delicate PC board traces.

The main thing is to get you back on the air, so let's get started.

NOTE: When the procedure calls for removing an IC, use the following method:

Use a small blade screwdriver to pry up one side of the IC just a little. Then pry up the other side of the IC just a little. Alternate your prying on both sides to "rock" the IC out of the socket. This technique prevents bent pins, so the IC can later be reinserted. Make sure that you don't mar the surface of the circuit board with your screwdriver.

TEST PROCEDURE

1. Turn off the Apple.

2. Remove the Q-68 board.

3. Look at the upper right corner of the board. You'll find a six pin header, with two little jumper plugs installed. The plugs should be in positions 2 and 3.

   Remove the jumper plug from position 3. There should

now be a single jumper plug in the middle (#2) position.

4. If there is a board in slot #4, remove it. Then install the Q-68 board into slot #4. Make sure the board is pushed down all the way.

5. Turn on the Apple. You can boot DOS if you wish, although this is not necessary. You should get to the APPLESOFT prompt (]).

6. Type CALL -151 (ret). This takes you to the Apple II monitor.

Now you are going to issue some control commands directly to the Q-68 board. The next step attempts to run the 68ØØ8 system using onboard Q-68 board resources only. When you removed the third position jumper, you enabled the onboard test programs.

If your Apple is dead, and you can't get this far, replace all the chips listed in step 7.

7. Type CØC1 (ret). This turns on the 68ØØ8.

   Look at the red and green lights on the Q-68 board. The RED one should be on, and the GREEN one should be blinking. If all is well, proceed to step 8.

   If the RED light is not on, replace the following IC's and try again:

A2	74LS74
C1	74LSØ5

   If the GREEN light is not flashing, we have a major problem. An OFF Green light means the processor is not running.

First we'll remove some IC's and retest. Turn off the Apple, and remove the Q-68 board.

Remove the following IC's (make sure A2 and C1 are installed, if you used the previous step):

A3	74LS244
B3	74LS373
A4	74LS244
A5	74LS244
C6	74LS05
---	---
D5	74LS393
B6	74LS74

NOTE: The IC's above the dotted line connect to the Apple II bus, so if the Q-68 board hung up the Apple, these are likely culprits.

How's that for a minimum 68008 system? You should have only ten IC's remaining on the Q-68 board. Now plug the Q-68 board back into the Apple and try steps 4, 5 and 6 again. If the green light now flashes, one or more of the chips you removed might be bad--try replacing them.

If chip replacement produces the flashing green light, proceed to step eight. If not, carry on here.

We're down to two possibilities. Either the PC board itself is messed up (check carefully for scratches and bent-under IC pins), or one of the remaining IC's is bad. Replace those if you can. Six of them are garden variety TTL, which you probably can find if you have access to TTL chips. The other four are available from Qwerty:

C3	MC68008 CPU
B2	2764 EPROM with DEBUG 1.0
C2	2 Kilobyte RAM
B4	PAL (QT-3)

If you're here, rather than step 8, plug the chips back in and send the board back to us for repair.

NOTE: The chips all insert so the lettering on them is rightside up.

8. Welcome to step 8! If you got this far, your system works in a standalone mode. Now we'll extend the test to include the Apple II memory.

   The Q-68 board should be running now, with the red light on and the green light flashing.

   There should be a single jumper plug in the middle (#2) position.

   From the Apple keyboard, at the monitor prompt (*), type CØC3 (ret). This interrupts the 68ØØ8, and sends it to a program which rapidly cycles the Apple HIRES screen. The red light should be on, and the green one should now be off (stopped flashing).

   NOTE: The 4 digit Hexadecimal number in the lower right corner of the Apple II screen is the "checksum" of the Q-68 EPROM. It should match the value written on the EPROM label.

Is the screen flashing like vertical window shutters in a storm? If so proceed to step 9.

If not, there are several possible conditions:

If the screen is cycling, but there are not clear vertical divisions (for example if random dots litter the screen) replace A3, a 74LS244.

If the screen just sits there, replace the following IC's:

A3	74LS244
A4	74LS244
A5	74LS244
B3	74LS373
C6	74LSØ5
B5	74LS74
B6	74LS74

If replacing these IC's doesn't fix it, send the board back for repair.

9. If you're here, the screen is cycling. There should be a single jumper plug in position two. Take the jumper plug which you removed from the third position at the beginning of this procedure, and install it at position 1, while the board is running. This forces a watchdog timer BERR. (Read section 7.3 if you don't know what BERR is).

As soon as you install this jumper plug, the screen should freeze, and the green light should begin blinking. If so, you have successfully tested/fixed the Q-68 board. If not, try replacing the following IC's:

D5	74LS393
C6	74LS05

If this doesn't fix it, send the board back for repair.

If you have fixed it, reinsert the jumpers into positions 2 and 3, and reinstall the board.

Appendix

Appendix A

Macro Assembler Operation and Memory Usage

CONFIGURATION REQUIREMENTS

The Macro Assembler runs in any Apple II, or Apple II plus, or Apple IIe with 48K of RAM. A language card is recommended. You will need at least one standard Apple disk.

CONTENTS OF THE DISK

The disk you received with your assembler is a standard 16-sector DOS 3.3 disk. It can be copied with Apple's disk copy programs, and the individual files are copyable with FID.

There are three versions of the S-C Macro Assembler on the disk. "ASM.LC" is the standard (language card) version, which loads at $D000. A second file, "ASM.LC2" is a small segment which loads into the alternate 4K bank of the language card.

The type "T" file named "QLOAD LCASM" is a control (EXEC) file used to load the language card files.

A second version, called "ASM.MB" (for "motherboard") runs without a language card. It loads at address $3000.

A third version, called "ASM.MB.4000" is included to allow programs which use the HIRES screen #1 to run without a language card. It loads at $4000, and thus is clear of the HIRES screen #1 at $2000-$3FFF.

This HIRES screen utilization is provided at the expense of 4 Kilobytes of memory which would have been used for source code and symbol table storage (as with ASM.MB).

The best place for the assembler is in a language card, so that maximum "motherboard" memory can be utilized, including the two HIRES screens.

A number of "I" (assembler source) files are provided on the disk to illustrate 68000 properties described in the second section of this manual.

An Applesoft program called "REMOTE.DEMO" illustrates the remote mode of the Debugger. To run this one, boot the system, choose item 5, and then type, "RUN REMOTE.DEMO".

The 68000 source program called, "OPCODE.TEST.68000" can be used to exercise the assembler. You might want to assemble and print this program to get aquainted with 68000 code syntax.

MEMORY USAGE

The language card version of the Macro Assembler program occupies $D000 through $F7FF in memory. The symbol table begins at $3000 and extends upward; your source program begins at the bottom of DOS ($9600 in a 48K machine) and extends downward.

This leaves an 8 kilobyte space from $1000-$2FFF to store object code.

The EXEC file which loads the assembler into the language card configures it so that DOS thinks of it as the alternate to the language in ROM on the mother board.

During source program entry or editing, memory usage is monitored so that the source program doesn't grow so large as to overlap the symbol table. Overlapping will cause the "MEM FULL ERROR" message to print. During assembly, memory required by the symbol table is monitored, to prevent the symbol table from overlapping the source program. Overlapping will generate the "MEM FULL ERROR" message and abort the assembly.

In addition, memory usage by the object program is monitored, so that it will not destroy the source program, DOS, the S-C Macro Assembler, the symbol table, or switch any I/O addresses. Therefore, if the object program bytes are directed at any memory address between $3000 and the top of the symbol table, or any address above the beginning of the source program, the "MEM PROTECT ERROR" is printed and assembly is aborted.

If you are using macros with private labels, the private

label table extends from $0FFF downward toward $0800. The private label table is also protected during assembly. Each private label uses five bytes in this table.

CAUTION: Location $800-$BFF are used to store 68008 exception vectors. The top half of this area, from $0A00-$0BFF correspond to interrupt vectors, and probably won't be used by your programs. So in effect the usable space for private labels is from $0A00 through $0FFF.

The assembler uses many locations in page zero during editing and assembly. Your programs should not tamper with page zero locations. Remember that Apple II page zero locations correspond to 68008 address $0800 locations.

Page one ($100-$1FF) is used both as a stack and as storage for various items. The high addresses in page one are used for the stack. The low end is used for a symbol buffer and for the pointers to the 27 hash chains used in storing the symbol table. The block from $170 through $1BF is used for holding search and replace strings by the editor, and for .TI titles during assembly.

Page two ($200-2FF) is used as the keyboard input buffer.

The high end of page three ($3D0-3FF) is used by DOS and by the assembler. You must not change any bytes between $3D0 and $3EF. $300-3CF is used by the "Q-68.STARTUP.BIN" program, which is described in Appendix D.

Locations $400-7FF are used by the Apple II as the text screen buffer. 32 of these bytes are unused by the screen. They are used instead as "scratch" locations by peripheral boards such as the disk controller and printer interface boards.

Locations $800-$803 are used to store the 68008 exception vectors.

Private labels, used in Macros, are stored from $FFF downward. If you use private labels, you need to insure that the private label table does not extend down into your 68008 startup vectors. Each private label uses 5 bytes.

ROM USAGE:

The Assembler takes full advantage of subroutines inside the

Apple Monitor ROM. Here is a list of all the subroutines used:

F941	Print 4-digit hex value from A,X
F94A	Print (X) blanks
FB2F	Set text mode, full screen window parameters
FBF4	Advance cursor
FC10	Backspace cursor
FC1A	Move cursor up one line
FC22	VTAB to current CV value
FC2B	An RTS instruction
FC42	Clear to end of page
FC58	Clear screen, Home cursor
FC66	Move cursor down one line
FC9C	Clear to end of line
FCA8	Delay
FD0C	Read next input character from keyboard
FD18	Read next input character through $38,39
FD84	Add char to input line
FD99	Print (Y,X) in hex with dash
FDDA	Print (A) in hex
FDED,FDF0	Print (A) as ASCII character
FE00	Display memory in hex
FE2C	Move block of memory
FE89	Set input to keyboard
FE93	Set output to screen
FECD	Write block of memory on tape
FEFD	Read tape into memory
FF2D	Print "ERR", ring bell
FF3A	Ring bell
FF69	Enter Monitor for MNTR command
FFA7	Get hex number
FFBE	Process monitor command
FFC7	Clear monitor mode byte
FFCC	Table of monitor commands

Appendix B

Assembler ERROR MESSAGES

If you make a mistake, the MACRO Assembler will probably catch you. Here are the error messages you may see.

*** SYNTAX ERROR	There is a misspelled command or bad line number.
*** MEM FULL ERROR	Either you do not have enough memory for the source program, or for the source plus the symbol table or a tape read error has occurred.
*** MEM PROTECT ERROR	Your program tried to assemble into an area of memory occupied by the assembler, the symbol table, or your source code. Use the .TA or .TF directives.
*** RANGE ERROR	The relative offset for a branch instruction was not in range
*** NO LABEL ERROR	There was no label with an equate (.EQ) directive.
*** BAD OPCODE ERROR	The opcode field does not contain a valid opcode or directive.
*** EXTRA DEFINITION ERROR	The same label was defined more than once.
*** UNDEFINED LABEL ERROR	A symbol in the operand field is not defined.
*** BAD SYMBOL ERROR	A character in the label field is not a legal character for a label.
*** BAD ADDRESS ERROR	This one is a catch-all for syntactical errors in the operand expression, as well as for use of a particular

	address mode with an opcode that does not support that mode.
*** VALUE > 255 ERROR	A local label is more than 255 bytes from its normal label.
*** NO NORMAL LABEL ERROR	A local label is used with no normal label present
*** NESTED .IN ERROR	There is a .IN directive within an included file.
*** MISSING .DO ERROR	There is a .FIN or .ELSE without a corresponding .DO.
*** .DO NEST TOO DEEP ERROR	.DO - .FIN blocks are nested more than eight levels deep.
*** KEY TOO LONG	The search string in a command is longer than 38 characters.
*** REPLACE TOO LONG ERROR	The REPLACE command tried to create a line longer than 248 characters.
*** NO MACRO NAME ERROR	The .MA directive has no name in the operand field.
*** UNDEFINED MACRO ERROR	The macro name has not been defined.
*** BAD MACRO PARAMETER ERROR	The character following a square bracket (]) must be a number (1-9) or a (#).

When an error is discovered during assembly, the error message is printed along with the offending line. The assembler then continues its pass, looking for more errors. At the end of the pass it prints "XXXX ERRORS IN ASSEMBLY", where XXXX is the number of errors it found in that pass.

If there are any errors discovered during pass one, assembly does not continue into pass two. Some errors are catastrophic, and abort assembly without continuing to the end of the pass.

Appendix C

Quick Check of the Q-68 Board.

This appendix gives you a quick check procedure for the Q-68 board. The steps outlined below are exactly those used at the factory to perform a quick go/no go check of Q-68 boards prior to burn-in.

If you don't get the expected results, refer to Chapter 5, "Self Test" for comprehensive test information.

1. Remove the jumper plug at position 3 of TB1 (upper right corner of the board). There should be a single jumper plug at position 2.

2. Turn on the Apple II. If you're really in a hurry, hit RESET before the disk boots. It doesn't matter whether the disk boots or not. All you need to do is get to the BASIC prompt "]".

3. From the "]" prompt, type CALL -151 (RETURN). This takes you to the Apple Monitor, signified by the "*" prompt.

4. Now you need to know into which slot the Q-68 board is plugged. We'll assume slot #4, and show the values to type in parentheses. If you have it in a different slot, use the following table.

Slot	Base Address (BA)
1	$C090
2	$C0A0
3	$C0B0
4	$C0C0
5	$C0D0
6	$C0E0
7	$C0F0

5. Type BA+1 (C0C1). The red light on the Q-68 board should turn on; the green one should start blinking.

6. Type BA+3 (C0C3). The red light should stay on; the green one should turn off; and the Apple screen should cycle, accompanied by a ticking sound from the Apple speaker.

In the lower right corner of the Apple screen, you should see a four digit hexadecimal number. This is the checksum of the onboard EPROM. The checksum is calculated by doing a 16-bit addition of all 8-bit bytes in the EPROM, and ignoring overflow. This checksum number should match the number printed on the EPROM label. If it doesn't, you've probably got a bad EPROM.

7. While the screen is cycling, plug the shorting plug you removed from TB1 position 3 into position 1. This places plugs at positions 1 and 2. The screen cycling should stop, and the green light should resume blinking.

Step 7 forces a Bus Error, and thus checks operation of the watchdog timer on the Q-68 board.

8. Reinstall the jumper plug at position 3 of TB1. There should now be two plugs, at positions 2 and 3. Position 1 should not have a jumper plug. You can change these plugs with the power still on, if you possess the manual dexterity. Otherwise, turn off the Apple and temporarily remove the Q-68 board to reinstall the jumpers.

Appendix D

Starting Up QPAK-68

When you boot the QPAK-68 system disk, a number of things happen.

First, the file "HELLO" is automatically loaded and run. This program is a one-line "EXEC" program:

```
10 PRINT CHR$(4); "EXEC X"
```

An "EXEC" file is run exactly as if you had typed the commands contained in the file at the Apple II keyboard.

Now the BASIC program "X" is run ("EXEC'ed", in Apple jargon). This program does two things:

(1) POKE 104,96: POKE 103,1: POKE 24576,0

This moves the start of BASIC program text storage from $800 to $6000. This is done so that you can use the Q-68 board and BASIC together. If the BASIC program storage were left at $800, it would wipe out the Q-68 board startup vectors.

(2) RUN QPAK.STARTUP

The Applesoft program, QPAK.STARTUP is now loaded and run. This program puts up the five item menu screen which allows you to reconfigure the system for a different Q-68 board slot, start DEBUG, start the Macro Assembler, or exit to BASIC.

The first thing QPAK.STARTUP does is to load the binary file, "Q-68.STARTUP.BIN" at address $300-$3CF. This program must be in memory for the Q-68 board to function correctly with the Macro Assembler or with BASIC.

Full listings of QPAK.STARTUP and Q-68.STARTUP.BIN are included at the end of this Appendix.

Q-68.STARTUP.BIN does the following chores:

1. It establishes four fixed Apple II entry points for Q-68 board control:

$300----Starts the Q-68 board (running the DEBUG program) and then enters a 6502 program loop which checks for CTL-D (dump the Apple II text screen), and CTL-B (Break 68008 program execution). This routine is jumped to when you type "DBUG" from the Macro Assembler.

$303----Remote Operation. Call this address from a BASIC program (CALL 771) to fire up DEBUG in the Remote mode. After setting up for remote operation, the routine returns to the caller (usually a BASIC program). No checking is thus done for CTL-B or CTL-D.

$30B----Start the Q-68 board and exit. The Apple II program that did the JSR $30B continues to run, simultaneously with the Q-68 board. This routine is called when you type "QON" from the Macro Assembler.

$3C2----Turn off the Q-68 board. It is sometimes useful to reset the Q-68 board from a BASIC program. For example, the Q-68 board must be off to do a disk access.

2. It contains the 6502 code to dump the text screen to a printer.

3. It contains the 6502 code to check the Apple II keyboard for a CTL-B, and upon finding it, initiating a 68008 program "Break".

Three instructions in the Q-68.STARTUP.BIN program control the Q-68 board. These instructions will be coded differently depending on which slot the Q-68 board occupies.

```
$30B  BIT $C0C1  ;Turn card on
$354  BIT $C0C3  ;Interrupt the card
$3C2  BIT $C0C0  ;Turn card off
```

These instructions are initially set for slot #4 operation. If you put the Q-68 board into another slot, the three instructions must be modified to account for the different slot. This is done for you when you choose item 1 ("SET Q-68 BOARD SLOT") of the boot menu. The three locations are changed to reflect the chosen slot, and then Q-68.STARTUP.BIN is saved back to disk so that you won't have to reconfigure again.

HOW CTL-D WORKS

When you type CTL-D from the DEBUG program, the 68Ø08 does one thing. It places an asterisk (*) in the lower right corner of the text screen. This is a signal to the 65Ø2 Q-68.STARTUP.BIN program to print the screen.

The 65Ø2 now takes over. The screen printing is done in two steps. First, the text screen is copied to $ØCØØ-$ØFFF. Second, it is sent one character at a time to the DOS printer vector location ($9EBD).

This DOS vector is set up by the Apple II when you initialize your printer card. You do this from BASIC or from the Macro Assembler by typing "PR#N" (N is the slot number of your printer card).

Why copy the screen to another RAM area first?

Because your printer card might be set up to "echo" the printout to the text screen. This has the unfortunate effect of scrolling the screen whenever a carriage return is encountered. Trying to print a screen as it scrolls out of existence produces garbage on the printer.

So whether or not your printer card echoes to the screen, the data sent to the printer is accurate, since the memory area $ØCØØ-$ØFFF never scrolls.

You'll know if your card is set for "video echo". As the printer prints, you'll see a screen of inverse and flashing garbage slowly devour the DEBUG screen you are printing. When the printing is finished, simply cycle the DEBUG window back to the one you want, and the garbage will disappear.

HOW CTL-B WORKS

When CTL-B is read from the Apple II keyboard port (location $CØØØ), the 68ØØØ AUTOVECTOR 7 address ($87C-$87F) is loaded with $1ØØ8C, the "break" entry point for DEBUG. The Q-68 board is then interrupted by doing a "BIT CØx3" instruction. "x" is the slot number plus 8.

NOTE: If your 68Ø08 program uses the Apple II keyboard, and you have started DEBUG by typing "DBUG" from the assembler,

remember that it will be "competing" with the 6502 in reading the keyboard strobe to look for the "CTL-B" code.

REMOTE MODE

Anything you can do with DEBUG from the Apple II keyboard, you can also do from a BASIC program.

When you activate the Remote mode (with a CALL 771 from BASIC, for example), DEBUG looks at location $00FD (decimal 771) for "keyboard" input, rather than the Apple II keyboard. DEBUG interprets bit 7 as a strobe bit. When bit 7 is high, DEBUG reads the value at $FD and processes it exactly as if it had been read from the keyboard. When DEBUG is ready to accept another "key", it clears bit 7 of location $FD.

When the Remote mode is first entered, a "dummy" value of $FF should be written to $FD. Then the 6502 program should wait for bit seven to be cleared (in other words, wait for the $FF to change to $7F) before sending "keystroke" data to DEBUG. This startup protocol insures that DEBUG has finished all startup housekeeping before accepting remote input.

To get a feel for the power of this mode, boot the system, and choose item 5, EXIT TO BASIC. Then, type RUN REMDEMO.

Sit back and watch.

What a great way to teach 68000 operation! Here are some hints you can use in your own BASIC programs:

1. Start up the board with a CALL 771 statement.

2. You send a character to DEBUG by doing a POKE (253,val) where "val" is the keyboard code PLUS 128. The high bit must be set for DEBUG to recognize that a new character is being sent.

3. Use CHR$(150) (CTL-V) to switch from the DEBUG screen to the BASIC text screen. Do this before you put up messages using PRINT statements.

4. Use CHR$(134) (CTL-F) to go back to the DEBUG screen, or send three more CTL-V commands to cycle to HIRES screen #1, HIRES screen #2, then back to DEBUG screen.

5. DEBUG indicates that it is ready for another character by clearing bit 7 of address 253 (decimal).

The following BASIC subroutine sends one character to DEBUG, and then waits for it to be accepted before proceeding:

```
10 POKE 253,V
20 IF PEEK(253)>127 THEN 20
30 RETURN
```

Here's another handy one. The character string A$ (which might have come in directly from the keyboard) is sent to DEBUG:

```
100 FOR L=1 TO LEN(A$)
110 V=128+ASC(MID$(A$,L,1))
120 GOSUB 10
130 NEXT
140 RETURN
```

Line 110 sets bit 7 of A$ high, as required by DEBUG.

This 6502 program is loaded into Apple memory (at $300) when you boot the system disk.

```
Q-68.STARTUP PAGE 0001


                1010 * (V07)
                1020 *--------------------------------
9EBD-           1030 HOOK    .EQ $9EBD DOS PRINTER HOOK
0020-           1040 INVBLK  .EQ $20   INVERSE BLANK
07F7-           1050 LOWRT   .EQ $7F7  LO RIGHT CORNER
C000-           1060 KBDATA  .EQ $C000
C010-           1070 KBSTB   .EQ $C010
                1080 * MEMORY MOVE POINTERS
003C-           1090 SRCL    .EQ $3C   (MON:A1L)
003D-           1100 SRCH    .EQ $3D   (MON:A1H)
0042-           1110 DSTL    .EQ $42   (MON:A4L)
0043-           1120 DSTH    .EQ $43   (MON:A4H)
                1130 * DEBUG START ADDRESSES
0084-           1140 MGOAD   .EQ $0084     ;DBUG ENTRY
0088-           1150 RGOAD   .EQ MGOAD+4   ;REMOTE
008C-           1160 CTBAD   .EQ MGOAD+8   ;CTL-B ENTRY
                1170 *--------------------------------
                1180 * FOUR ENTRY POINTS:
                1190 *
                1200 * $300---DBUG COMMAND
                1210 * $303---START DEBUG-REMOTE MODE
                1220 * $30B---QON (START BD AND LEAVE)
                1230 * $3C2---TURN BOARD OFF
                1240 *--------------------------------
                1250 * THE LOCATIONS TURNON+1, INTR+1
                1260 * AND STOP+1 ARE SET FOR
                1270 * SLOT #4 OPERATION.  TO SET A
                1280 * A DIFFERENT SLOT, THE USER
                1290 * SELECTS ITEM (1) FROM THE
                1330 * SYSTEM STARTUP MENU.  THIS
                1340 * CHANGES THE BYTES IN THE
                1350 * "BIT" INSTRUCTIONS TO REFLECT
                1360 * THE PROPER SLOT.  THEN THIS
                1370 * PROGRAM (Q-68.STARTUP.BIN) IS
                1380 * SAVED BACK ON DISK
                1390 *--------------------------------
                1400         .OR $300
                1410 *--------------------------------
0300- 4C 0F 03  1420 BEG     JMP MGO
                1430 *--------------------------------
0303- 20 2C 03  1440 RGO     JSR INSTAL ;PUT IN MGO VEC
0306- A9 88     1450         LDA #RGOAD ;ALTER ONE BYTE
```

```
0308- 8D 07 08 1460          STA $807   ;TURN ON &EXIT
                1470 *
030B- 2C C1 C0 1480 TURNON   BIT $C0C1  ;QON  ENTRY PT
030E- 60       1490          RTS
                1500 *--------------------------------
030F- 20 2C 03 1510 MGO      JSR INSTAL ;RESET VEC'S
0312- 20 0B 03 1520          JSR TURNON
                1530 *
                1540 *--------------------------------
                1550 *  MAIN CONTROL LOOP
                1560 * LOOK FOR ASTERISK IN BOTTOM
                1570 * RIGHT CORNER OF SCREEN
                1580 * AND FOR CTL-B FROM KEYBOARD
                1590 *--------------------------------
0315- AD F7 07 1600 LOOP     LDA $7F7
0318- C9 AA    1610          CMP #$AA
031A- D0 03    1620          BNE .1      ;NO "*" YET
031C- 20 58 03 1630          JSR PRTSCR
031F- AD 00 C0 1640 .1       LDA KBDATA
0322- C9 82    1650          CMP #$82    ;CTL-B?
0324- D0 EF    1660          BNE LOOP    ;NOPE
0326- 20 40 03 1670          JSR INTER
0329- 4C 15 03 1680          JMP LOOP
                1690 *--------------------------------
                1700 * INSTALL STARTUP VECTORS
                1710 * SSP AT $18800
                1720 * PC  AT $100XX (XX FROM #MGOAD)
                1730 *--------------------------------
032C- A0 07    1740 INSTAL LDY #7
032E- B9 38 03 1750 .1       LDA RVEC,Y
0331- 99 00 08 1760          STA $800,Y
0334- 88       1770          DEY
0335- 10 F7    1780          BPL .1
0337- 60       1790          RTS
                1800 *
0338- 00       1810 RVEC     .DA #$00 SSP:00 01 88 00
0339- 01       1820          .DA #$01
033A- 86       1830          .DA #$88
033B- 00       1840          .DA #$00
                1850 *
033C- 00       1860          .DA #$00 PC: 00 01 MGOAD
033D- 01       1870          .DA #$01
033E- 00       1880          .DA /MGOAD
033F- 84       1890          .DA #MGOAD
                1900 *
                1910 * NOTE--INTER AND PRTSCR ARE
                1920 * WRITTEN AS SUBROUTINES SO THAT
                1930 * THEY MAY BE CALLED FROM BASIC
                1940 * IN THE REMOTE MODE
                1950 *
```

```
                    1960 *--------------------------------
                    1970 * CTL-B WAS HIT.  INSTALL AUTOVEC
                    1980 * #7 AND INTERRUPT THE Q-68 BD
                    1990 *
                    2000 * 87C 87D 87E 87F
                    2010 *  00  01  00  NN (FROM CTBAD)
                    2020 *--------------------------------
0340- 2C 10 C0      2030 INTER  BIT KBSTB   ;CLEAR KB STB
0343- A2 8C         2040        LDX #CTBAD
0345- 8E 7F 08      2050        STX $87F
0348- A2 00         2060        LDX #0
034A- 8E 7C 08      2070        STX $87C
034D- 8E 7E 08      2080        STX $87E
0350- E8            2090        INX         ;X=1 NOW
0351- 8E 7D 08      2100        STX $87D
                    2110 *--------------------------------
                    2120 * INTERRUPT THE Q-68 BOARD
                    2130 *--------------------------------
0354- 2C C3 C0      2140        BIT $C0C3 <ALTERABLE INSTR
0357- 60            2150        RTS
                    2160 *--------------------------------
                    2170 * PRINT THE SCREEN.  FIRST,
                    2180 * MOVE SCREEN FROM $0400-$07FF
                    2190 *                  TO $0C00-$0FFF
                    2200 *--------------------------------
0358- A0 00         2210 PRTSCR LDY #0
035A- 84 3C         2220        STY SRCL
035C- 84 42         2230        STY DSTL
035E- A2 04         2240        LDX #$04
0360- 86 3D         2250        STX SRCH
0362- A9 0C         2260        LDA #$0C
0364- 85 43         2270        STA DSTH
                    2280 *--------------------------------
                    2290 * BY A FORTUITOUS COINCIDENCE,
                    2300 * X=4 AND Y=0, WHICH IS NEEDED
                    2310 * FOR THE NEXT CODE SECTION
                    2320 *--------------------------------
0366- B1 3C         2330 MOVE   LDA (SRCL),Y
0368- 91 42         2340        STA (DSTL),Y
036A- C8            2350        INY
036B- D0 F9         2360        BNE MOVE
036D- E6 3D         2370        INC SRCH
036F- E6 43         2380        INC DSTH
0371- CA            2390        DEX
0372- D0 F2         2400        BNE MOVE
                    2410 *--------------------------------
                    2420 * PRINT SCREEN
                    2430 *--------------------------------
                    2440 * CALC. SRCL-H FOR NEXT LINE
                    2450 *--------------------------------
```

```
0374- 8A       2460 NULINE TXA
0375- A0 00    2470        LDY #0  ;CHAR. COUNTER
0377- 4A       2480        LSR
0378- 29 03    2490        AND #$03
037A- 09 0C    2500        ORA #$0C
037C- 85 3D    2510        STA SRCH
037E- 8A       2520        TXA
037F- 29 18    2530        AND #$18
0381- 90 02    2540        BCC .1
0383- 69 7F    2550        ADC #$7F
0385- 85 3C    2560 .1     STA SRCL
0387- 0A       2570        ASL
0388- 0A       2580        ASL
0389- 05 3C    2590        ORA SRCL
038B- 85 3C    2600        STA SRCL
               2610 *--------------------------------
               2620 * GET NEXT CHARACTER IN LINE
               2630 *--------------------------------
038D- B1 3C    2640 INLINE LDA (SRCL),Y
               2650 *--------------------------------
               2660 * TRANSLATE SCREEN CODE TO ASCII
               2670 *
               2680 * 0-1F----40-5F
               2690 * 20------2A (INV BLK TO *)
               2700 * 21-7F---21-7F
               2710 * 80-FF---MASKED TO 0-7F AFTER
               2720 *          INVERSE BLANK CHECK
               2730 *--------------------------------
038F- C9 20    2740        CMP #INVBLK
0391- F0 0B    2750        BEQ AST
0393- 29 7F    2760        AND #$7F
0395- C9 20    2770        CMP #$20  A-20
0397- 10 07    2780        BPL PRINT A>=20
0399- 18       2790        CLC       A<20
039A- 69 40    2800        ADC #$40
039C- 90 02    2810        BCC PRINT
039E- A9 2A    2820 AST    LDA #'*
03A0- 20 BD 9E 2830 PRINT  JSR HOOK
03A3- C8       2840        INY
03A4- C0 28    2850        CPY #40
03A6- D0 E5    2860        BNE INLINE
               2870 *--------------------------------
               2880 * END OF THE LINE
               2890 *--------------------------------
03A8- A9 0D    2900        LDA #13  ;CR
03AA- 20 BD 9E 2910        JSR HOOK
03AD- E8       2920 NEXT   INX   ;BUMP LINE COUNTER
03AE- E0 18    2930        CPX #24
03B0- D0 C2    2940        BNE NULINE
               2950 *--------------------------------
```

```
                2960 * FINISHED.  NOW REMOVE *
                2970 *-------------------------------
03B2- A9 20     2980         LDA #INVBLK
03B4- 8D F7 07  2990         STA LOWRT
                3000 *-------------------------------
                3010 * AND PRINT THE CARRIAGE RETURNS
                3020 *-------------------------------
03B7- A2 09     3030         LDX #9
03B9- A9 0D     3040 .4      LDA #13     ;CR
03BB- 20 BD 9E  3050         JSR HOOK
03BE- CA        3060         DEX
03BF- D0 F8     3070         BNE .4
                3080 *-------------------------------
                3090 * AND RESUME WAITING
                3100 *-------------------------------
03C1- 60        3110         RTS
                3120 *
                3130 *-------------------------------
                3140 * TURN OFF THE Q-68 BOARD
                3150 *-------------------------------
03C2- 2C C0 C0  3160 STOP    BIT $C0C0   ;SLOT #4
03C5- 60        3170         RTS
                3180 *
00C6-           3190 ZLEN    .EQ *-BEG
```

This BASIC program is loaded and run when you start up the QPAK-68 system.

```
50  REM --QPACK.STARTUP
100 D$ =  CHR$ (4)
110  PRINT D$;"BLOAD Q-68.STARTUP .BIN"
120 V = ( PEEK (780) - 129) / 16
140  TEXT : HOME : GOSUB 460
160  HTAB 5: PRINT "    QPAK-68 STARTUP PROGRAM"
170  GOSUB 460: HTAB 5: PRINT "Q-68 BOARD SHOULD BE IN SLOT # ";V:
GOSUB 460
180  PRINT
220  PRINT : PRINT "1....SET Q-68 BOARD SLOT"
230  PRINT : PRINT "2....START Q-68 DEBUG"
280  PRINT : PRINT "3....RUN ASSEM. AT $3000 (NO LANG.CARD)
310  PRINT : PRINT "4....RUN ASSEM. AT $D000 (L.C. OR IIE)
340  PRINT : PRINT "5....EXIT TO BASIC"
350  VTAB 20: HTAB 1: PRINT "YOUR CHOICE?....";
380  GET A$:A =  VAL (A$): IF A < 1 OR A > 6 THEN 350
400  PRINT A: IF A = 1 THEN 620
410  IF A = 2 THEN  CALL 768
420  IF A = 3 THEN  PRINT D$;"BRUN ASM.MB"
430  IF A = 4 THEN  VTAB 24: POKE 34,23: PRINT  CHR$ (4);"EXEC
QLOAD.LCASM": END
432  IF A = 6 THEN  PRINT D$;"BRUN ASM.MB.4000"
440  PRINT "(FOR ANOTHER CHANCE, TYPE "; CHR$ (34);"RUN"; CHR$
(34);")"
450  END
460  FOR J = 0 TO 39: PRINT "-";:NEXT
470  RETURN
620  ONERR  GOTO 810
630  TEXT : HOME : PRINT "QPAK-68 CONFIGURE PROGRAM": PRINT "
------------------------"
640  PRINT : PRINT
650  PRINT "Q-68 CARD IN WHICH SLOT?...";
654  HTAB 28: VTAB 5
660  GET A$:A =  VAL (A$): IF A < 1 OR A > 7 THEN 654
670  PRINT A
680  PRINT : PRINT : PRINT "UPDATING FILE
--Q68.STARTUP.BIN-- "
700 V = A * 16 + 129
710  POKE 780,V: POKE 853,V + 2
712  POKE 963,V - 1
730  PRINT  CHR$ (4);"BSAVE Q-68.STARTUP.BIN,A$300,L$CF"
760  GOTO 120
810  INVERSE
820  PRINT : PRINT : PRINT "THIS DISKETTE DOES NOT CONTAIN
THE FILE --Q-68.STARTUP.BIN--"
```

```
830  NORMAL
840  PRINT  CHR$ (7): PRINT  CHR$(7)
850  PRINT "PLEASE REPLACE THE DISKETTE WITH"
860  PRINT "ONE THAT DOES, AND TRY AGAIN"
870  PRINT : PRINT "(HIT ANY KEY TO RESUME...)": GET Z$:
GOTO 620
```

Appendix E -- Things That Could Go Wrong

In a system as complex as the QPAK-68/Apple II combination, there are a few things that could go wrong. This appendix collects as many as we know about. As a registered QPAK-68 owner, you will receive mailings that update this section (not too extensively, we hope).

BASIC BEHAVES ERRATICALLY

You probably started BASIC the normal way, by booting an Apple system disk. One of the purposes of the QPAK-68 disk boot is to move the BASIC text storage out of the 68000 exception vector memory space starting at $0800.

If you wish to use BASIC with the Q-68 board (for example a REMOTE mode program), be sure to boot the QPAK-68 system disk, and select item 5 (Exit to BASIC).

THE DISK DOESN'T WORK (I/O ERROR)

Your Q-68 board is running. Turn it off by pressing RESET.

Apple saved 75 cents by not including a track 0 switch in their disk drive. Normal disk drives send head motion pulses to the drive until the track 0 switch closes, and then the computer knows that the head is at the "home" position.

Since the Apple disk drive does not have this switch, the only way to insure that the head is homed is to give it the maximum number of step pulses that would home the head from the furthest-out position. This is the clatter you hear when the disk boots or tries to recover from an error. It is the head assembly rattling against the "home" position.

If the clatter is slower than usual, and then a disk error message appears, your Q-68 board is probably running. The 6502 in the Apple must be running at full speed to perform successful disk accesses. This means that the Q-68 board

must be off.

The easiest way to make this mistake is to be in the middle of a DEBUG session, and try to load a disk file.

Recovery is easy: RESET the Apple, taking you back to the assembler (":" prompt), and then use the disk.

APPLE GOES COMPLETELY DEAD

Did you put something into Apple page 0 memory? You can do this by writing locations $00800-$008FF in your 68000 program. This will mess up the Apple for sure.

So will writing Apple locations $3D0-$3FF (68008 $B00-BFF). The vital pointers which control such things as where the Apple goes when you press RESET are in this area of memory.

NOTHING HAPPENS WHEN I TYPE "QON" FROM THE ASSEMBLER

Your 68000 program does not have the required vectors for the 68008 startup operation. You need to put two addresses at Apple locations $800-$807 which correspond to the initial system stack pointer, and 68000 start address, respectively. For example:

```
.OR       $800      ;68008 location 00000
.DA       $18800     ;SSP
.DA       $1000      ;68000 program start
```

TYPING "DBUG" FROM THE ASSEMBLER DOES NOT WORK

The "Q-68.STARTUP.BIN" program is not in Apple memory at $300.

Note: If you type "DBUG", your 68000 program does not need the RESET vectors described above. The "Q-68.STARTUP.BIN" program installs the RESET vectors for you. These vectors are set to start the Q-68 board running DEBUG.

(If your program does install the vectors, they will override those installed by the startup program. Your new ones will be installed when your program is assembled.)

The easiest way to make sure the startup program is in memory is to reboot the system (CTL-OpenApple-RESET on the IIe). This automatically installs the startup program.

If you wish to load the startup program without losing results in progress, from the assembler (":" prompt) type MNTR. Then, from the Apple monitor ("*" prompt) type "BLOAD Q-68.STARTUP.BIN". Make sure the Q-68 board is off for this operation. You can insure this by hitting RESET before typing "MNTR".

I CAN'T PASS CHARACTERS TO DEBUG IN THE REMOTE MODE

Remember to set the MSB of all data sent to DEBUG. Also, if you are sending a bunch of characters in a quick succession, you need to check the MSB of the sending location (Apple $FE) for MSB low before sending each character. This insures that DEBUG won't miss any characters.

THE DEBUG "CTL-D" COMMAND DOESN'T WORK

Is your printer card enabled? This is done by typing "PR#n" where "n" is the slot number which your printer card occupies.

You can type this either from BASIC (if you have selected item 5 from the startup menu), or directly from the editor/assembler ":" prompt.

Is the Q-68.STARTUP.BIN program in memory? This program scans the DEBUG screen looking for the signal to print the screen (as asterisk in the lower right corner), and then does the actual screen printing.

Again, you can install this program from the editor prompt (":") by typing MNTR, then "BLOAD Q-68.STARTUP.BIN".

The 6502 listing for this program is included in Appendix D. If you have some special printer interface card problem and have an Apple (6502) editor/assembler (we recommend the S-C

one), you can use this listing to serve as a guide to write your own printer driver.

CAUTIONS

Here are some general cautions which "beta-site" (first test) users brought to our attention:

1. Be careful not to let your 68000 code clobber the memory at $800-$BFF, which is Apple memory $0-$7FF.

2. Don't do Apple DOS commands while the Q-68 is using Apple memory. The disk I/O is carefully timed by the Apple's 6502 processor. If the Q-68 board is slowing down the 6502 during attempted disk reads, you will get I/O errors. During disk write operations, you could clobber data on the disk.

3. Never work on the board (as in Chapter 5) without first turning off power to the Apple II.

4. Stay clear of Apple memory from $300-$3CF. This memory contains the "Q-68.STARTUP.BIN" program, which must be intact to allow correct operation of the QON and DBUG assembler commands, and the CTRL-D and CTRL-B DEBUG commands.

Chapter 7 - HARDWARE TOPICS

Section 7.1 -- EXCEPTIONS

Question: How many programmers does it take to change a light bulb?

Answer: None. It's a hardware problem.

Have you ever been in the middle of a microprocessor project where something didn't work, and the software and hardware people just pointed at each other, each insisting that the problem was caused by the other group?

This is very common. A microcomputer based system is a very complex animal, and it is sometimes very difficult to exclusively place the blame on either hardware or software.

Motorola has equipped the 68000 with an elegant mechanism, called "exceptions", which stops program execution when something abnormal happens, and lets you poke around to find out what went wrong.

You hardware types can present irrefutable evidence to the software team that somebody tried to access a long word at an odd address. And you software types can prove once and for all that register A4 never exceeded $E0000, and thus the glitches in video memory were caused by hardware.

Catching programming errors is only one of the many things the 68000 exception system can do.

There are five types of exceptions:

1. Exceptions caused by outside stimulation (RESET, Interrupts)

2. Exceptions caused by bugs in the program (Address Error, Illegal Instruction, Divide by Zero, Privilege Violation).

3. Exceptions caused by program instructions (TRAP, CHECKV, CHK).

4. Exceptions caused by a system malfunction (Bus Error, Spurious Interrupt).

5. Exceptions which allow system testing (TRACE).

All exceptions, no matter what the cause, are handled the same way by the 68000. "Normal" processing, where the program executes exactly as you wrote it, is suspended, and a special three-step exception sequence is started.

First, the 68000 state is saved, so that you can find out exactly where in memory the exception happened. Or, in the case of an interrupt, so that you can resume where you left off after finishing the interrupt processing.

Second, the 68000 reads a 32-bit address out of a special area of memory. This address is called an "exception vector" (a vector is something which points; in this case the address points to a subroutine).

Third, the 68000 jumps to this address.

By processing the exception, the 68000 diverts program execution away from the running program, to a special routine which you have written to handle the exception.

How does the 68000 know which exception occured, and therefore which particular vector to use?

At the bottom of the 68000 memory map (at $000000) is a one kilobyte table of exception vectors. Take a look at page 61 of the Motorola 68000 User's Manual. Here you'll see the complete vector table. Rather than duplicating the descriptions found in the Motorola manual, we'll concentrate on the vectors actually used by the QPAK-68 system.

THE RESET VECTOR

The first two vectors are for handling 68000 startup. When a 68000 system is powered up, the RESET line is typically held asserted for a brief time until the other circuits in the system can power up and stabilize. Then RESET is released.

The 68000 then automatically fetches the first eight bytes in memory. The first four constitute an initial System Stack Pointer value. This 32-bit value is loaded into the

SSP. The next four bytes constitute a program start address. This 32-bit value is loaded into the Program Counter (PC). The 68000 then jumps to this initial PC address.

The RESET vector is the only 8-byte vector in the table. All the others are 4-bytes, representing 32-bit addresses. It is up to the system designer to make sure that valid information exists at the first 8 bytes of 68000 memory. When you run the QPAK-68 system, the RESET vectors can be set up two ways:

1. When you start the system with the QPAK-68 system disk, the reset vectors are automatically set up to start up the DEBUG program in the Q-68 board EPROM. The first eight bytes (which start at $800 in Apple II memory) are:

   00 01 88 00 00 01 00 84

   This places the System Stack Pointer for your program at the top of Q-68 board RAM ($18800), and begins 68008 operation at the beginning of DEBUG ($10084). NOTE: The SSP for DEBUG is set to $18600. It won't interfere with your program's SSP at $18000.

2. The assembler can place the eight bytes at $800 (using two ".DA" statements), and thus start itself with SSP and PC values that you choose. Once you write these numbers to $800-$807, you turn the Q-68 board on with the assembler's "QON" command. Take a look at the "VIDEOTEST" program on the system disk for an example of a program which starts this way.

BUS ERROR

This vector is used by the Q-68 board to start exception processing when the system watchdog timer has "timed-out". Technically speaking, if 180 microseconds elapses between the 68008 sending out an address and receiving back a DTACK signal, the BERR pin of the 68008 is asserted and a Bus Error exception is initiated.

DEBUG uses this feature to figure out whether the memory you have selected for display is there or not. DEBUG installs a vector at the Bus Error vector location which points to an internal routine that prints asterisks for "non-memory" data values.

If a Bus Error or Address Error occurs in your program, the 68008 puts the return address and status register on the Supervisor Stack. These values are displayed as PC and SR of the DEBUG Status Window after the error occurs.

Four additional words of status information are also pushed onto the Supervisor Stack for a Bus or Address Error exception. DEBUG copies these four words to RAM starting at $18002. Simply display memory at this location to see these values after a bus or address error.

LEVEL 7 INTERRUPT AUTOVECTOR

An Autovector is one which does not need to supply a vector number to the 68008 using external hardware. The interrupting priority is the autovector number. The Q-68 board uses autovector 7 to interrupt the 68008 from the Apple II. This is the highest priority level, and is equivalent to the "Non-Maskable Interrupt" found in most 8-bit systems.

TRAP 15

The Trap 15 vector is used by DEBUG to implement breakpoints.

Section 7.2 -- 68000 INTERRUPTS

There are two types of interrupts in 68000-type processors. Both are initiated in the same manner--external hardware asserts control signals which request the processor to suspend normal processing and jump to a special interrupt program.

Most 8 bit processors contain two interrupt pins, which implement two priority levels of interrupt. The Non Maskable Interrupt (NMI) interrupts the CPU regardless of what is happening. The maskable interrupt (usually called IRQ for interrupt request) may or may not interrupt the CPU, depending on whether or not the running program has enabled the interrupt.

The user may set interrupt priorities between the two interrupt types. The NMI will typically be connected to something which absolutely MUST interrupt, such as circuitry which detects a power failure. The other interrupt may be honored or not, depending on the requirements of the program. For example, IRQ might be disabled for a time-sensitive operation, such as a disk access.

The 68000 expands this two level scheme into seven interrupt priority levels. Rather than providing seven separate interrupt pins, the interrupt level requested is encoded into three lines and input to the 68000 on three pins called IPL0/, IPL1/, and IPL2/. IPL stands for Interrupt Priority Level, and the slash means that the signals are active low.

As the 68000 is operating, it maintains an internal priority level in three bits of its 16 bit status register. By setting its interrupt priority level to different values, the 68000 can allow some interrupts and ignore the others. This is a great improvement over the two level 8-bit CPU's.

Here's how it works. If the level of the incoming interrupt is greater than the interrupt level set by the CPU, or if it is a level 7 interrupt, the interrupt is recognized and the interrupt processing begins. If not, the interrupt is ignored. For example, if the CPU is running at interrupt level 2 (010) in the status register, and a level two interrupt arrives (IPL2:IPL1:IPL0=010), it is ignored, and normal processing continues.

When an interrupt is accepted, the 68008 must figure out how

to get to the special program for interrupt processing. Let's say you have written this interrupt routine at address AAAA. When the program is interrupted, it is at address PPPP. How do we get from PPPP to AAAA, and back to PPPP when the interrupt routine is finished?

There are two ways. The most general way is for the 68000 to execute a special bus cycle called Interrupt Acknowledge. In this cycle, external interrupt hardware puts an eight-bit vector number on the bus, and this number, from 0 to 255 is brought into the 68000.

VECTORS

A vector is something which points. In this case, the number brought in points to one of 256 locations in the bottom of the 68000 memory space. At these locations are four byte (32-bit) pointers which the user (that's you) has put there ahead of time. Since four bytes are required to define each of the 256 vector locations, there are a total of 1024 bytes in the vector area. Page 61 of the Motorola 68000 User's Manual shows the entire vector table.

Some of these vectors are reached by internal means. You already know about the RESET vectors at locations 0-7. Although the whole 1K area can be vectored to by the external interrupt hardware, the last half of it is completely clear for externally generated interrupt vectors. Locations in the first half either already have preassigned functions such as reset, traps, and errors; or they have the mysterious "reserved by Motorola" designation.

What if you don't want to go to the complexity and expense of adding the hardware required to jam in the interrupt vector pointer? If you can do with 7 or fewer vector pointers, you can use the autovector mechanism provided by the 68000.

AUTOVECTORS

An autovector is one which is selected by internal, rather than external hardware. When the autovector mode is activated, the vector number is simply the interrupt priority level number--those three bits you activated on the

IPL pins to cause the interrupt in the first place. For example interrupt priority 5 selects autovector #5, which happens to be located at memory locations $74 through $77.

How do you tell the 68000 that you want the autovector mode? The system designer does this by forcing the VPA line low during the interrupt. VPA stands for Valid Peripheral Address. Forcing the autovector mode is actually a secondary role for the VPA pin. The primary role is to cause the 68000 to alter its timing so that it is compatible with existing 6800 peripheral chips (the 68000 designers at Motorola are no dummies!). A result of this dual mode is that the autovector mechanism is wrenched into the 6800 bus timing, and is therefore slower than the "normal" vectored interrupt mechanism.

How much slower? Not much. It turns out that the 68008 will wait anywhere between six and 14 extra clock periods to synchronize itself to the 6800-type "E-clock", which makes it 6800 bus compatible. This adds from 0.84 to 1.96 microseconds to the interrupt cycle (7.16 Megahertz clock).

FITTING 64 PINS INTO 48

The 68008 implements a reduced version of the 68000 interrupt system. It functions exactly as in the 68000, with one important limitation. Due to the necessity to reduce the package size from 64 pins (68000) to 48 pins (68008), something had to go.

One of the things done to conserve pins was to tie two of the interrupt request pins together. The IPL0 and IPL2 pins are tied together and designated IPL0-IPL2 on the 68008. This means that interrupt levels 0, 2, 5 and 7 are possible. "Level 0" really means "no interrupts requested" so there are three priority levels in the 68008--2, 5 and 7.

The Q-68 board uses autovector number 7 to allow the Apple II to interrupt normal 68008 processing. This interrupt is implemented in a manner that allows external hardware to have free use of the interrupt system. This means that future peripheral boards for the Q-68 board can support any kind of 68008 interrupts.

EXERCISING THE 68000 INTERRUPT SYSTEM

Here is a program (INTER.DEMO.BEEP) which demonstrates the 68000 interrupt mechanism. We'll take the standard VIDEOTEST program and add code to process an interrupt generated by the Apple II. Then, as the VIDEOTEST program is running, we'll interrupt it from the Apple and see what happens.

Lines 1040 through 1150 are the VIDEOTEST program described in Chapter 2.

```
1000          .TI      60,INTER.DEMO.BEEP
1001 *--------------------------------
1020 TIME     .EQ      $0100
1021 *--------------------------------
1030 *
1040          .OR      $1000
1050 *
1060          CLR      D0
1070 START    MOVE     #$2000,A0
1080          MOVE     #$1000,D1
1090 LOOP     MOVE     D0,(A0)+
1100          DBEQ     D1,LOOP
1110          ADDQ     #1,D0
1120          TST.B    $C050
1130          TST.B    $C053
1140          TST.B    $C057
1150          BRA      START
```

Now we'll put the interrupt handling program at address $1100 (Next page). This "handler" does just enough to let you know it is there. It makes a "zip" sound in the Apple's speaker.

This is done by setting up two delay loops. The value TIME is loaded into D4 to time the duration of the sound event. The inner loop PERIOD decrements D5 and waits for it to hit zero (line 1230). Then D4 is decremented, and if it hasn't hit zero yet, it is decremented and the loop runs again. The ever decreasing value of D4 in the outer loop is loaded into D5 at line 1220. This is what gives the sweep up in frequency which produces the "zip" sound effect.

The actual control of the Apple speaker is done with line 1210. Simply accessing location $C030 in the Apple produces a single "tick" sound from the speaker. The TST.B instruction reads this speaker location and sets 68000

status flags according to what it finds there (we don't care). When this little routine finishes, line 1250 does a Return From Interrupt instruction to resume the interrupted program.

```
1180 *--------------------------------
1190 * INTERRUPT ROUTINE
1192 *--------------------------------
1194         .OR      $1100
1196 *
1200         MOVE     #TIME,D4
1210 BEEP    TST.B    $C030
1220         MOVE     D4,D5
1230 PERIOD  DBF      D5,PERIOD
1240         DBF      D4,BEEP
1250         RTE
1260 *--------------------------------
```

The startup vectors are set as usual. The program starts at $1000, and we'll put the stack at $8000:

```
1260 *--------------------------------
1270 * SET UP START VECTORS (SSP&PC)
1280 *
1290         .OR      $800
1300         .DA      $00008000
1310         .DA      $00001000
1320 *--------------------------------
```

Now to install the interrupt vector. Autovector #7 goes to 68008 location $7C-$7F to pick up the interrupt routine address. Since our interrupt program is at $1100, this is the address we put there.

```
1320 *--------------------------------
1330 * POINT AUTOVECTOR #7
1340 *
1350         .OR      $87C
1360         .DA      $1100
```

We really haven't changed the VIDEOTEST program function. If you run the INTER program, you won't see any difference--the screen cycles continuously.

But now try this. While the above program is running, you can use the S-C assembler (which is also running) to trigger a Q-68 interrupt. Simply type $C0C3, RETURN. This does a read of Apple memory location C0C3 which is the Q-68 location which triggers the autovector 7 interrupt.

Do you realize what you just did? You used one computer, the Apple, to interrupt another, the Q-68 board!

Section 7.3 -- DTACK, BERR and Watchdogs

WHAT IS DTACK, AND HOW DO I USE IT?

The 68000, like all computers, cannot operate all by itself. It requires memory to hold the programs and data which it is to execute. The mechanism by which the 68000 accesses memory is referred to as "executing bus cycles".

A bus is a collection of wires which have a common purpose. Microprocessor systems are generally organized into three busses:

1. Address bus. 16 to 24 lines which carry information to select one particular memory location out of many.

2. Data bus. The "width" (number of wires) of this bus determines the "-bit" size of the microprocessor. 8 data bus lines connect to an "8-bit" microprocessor; 16 data bus lines connect to a "16-bit" microprocessor. Some microprocessor versions such as the 68008 and the 8088 blur this definition by bringing out a 16-bit data bus 8 bits at a time.

3. Control Bus. This is a hodgepodge of control signals such as memory read and write, interrupt, wait, etc.

DTACK

DTACK is a member of the control bus signals. It stands for Data Transfer ACKnowledge. We'll look at a typical bus cycle and see the vital role DTACK plays in the data transfer.

Any 68000 bus cycle consists of eight clock states. This clock is the one attached to the CLK pin of the 68000. With an 8 MHz clock, the time for one cycle of the clock is 125 nanoseconds. Each time state is half of this, 62.5 nanoseconds.

The 68000 is a synchronous machine, meaning that everything that happens inside it is caused by some edge or other of

the clock. If you look at a 68000 spec sheet, you'll see all timing specifications given as such a time after one of the clock edges (an edge is where the clock goes low to high or high to low).

Let's look at the 68000 states one by one

0. STATE 0. This state separates one bus cycle from another. Nothing is asserted during this state.

1. STATE 1. The address bus goes active with the address of the memory location to be read or written. In STATE 0, the address bus was "floating" which means that was neither high nor low. It was simply disconnected from the system.

2. STATE 2. By now the address bus is ready with the desired collection of highs and lows, and in S2 a signal called Address Strobe (AS) goes low to indicate that the signals on the address bus are stable and can be used by the system.

If the cycle is a READ operation (data transferred into the 68000) the data strobe signal also goes low in S2. If it is a WRITE cycle (data coming out of the 68000) the R/W line goes low along with AS in this cycle.

3. STATE 3. Nothing happens here for a READ cycle. The system circuitry is simply given some time to get the data ready for the 68000.

   For a WRITE cycle, the data out of the 68000 is placed on the data bus lines. The data bus was until now floating.

Before we look at STATE 4, let's summarize where the 68000 is after the first four states.

For a READ operation, the address has been sent out, the data bus is set to the input mode to receive the outside data, and the data strobe signal has been activated to tell the outside device(s) to send data into the 68000.

For a WRITE operation the address has been sent out, the R/W line has gone low to tell the system that a write operation is coming, and the data out of the 68000 is sitting on the data bus.

Now what?

The 68000 has done all it can to perform a successful data transfer. It is now up to the outside system to complete the transaction.

In a READ cycle, the outside system (usually a memory) must present the data to be read by the 68000 on the data bus. In a WRITE cycle, the outside system must grab the data off the data bus and store it.

How does the 68000 know that the outside system has successfully completed its half of the data transfer? You guessed it, by looking at the Data Transfer Acknowledge (DTACK) signal.

It is the responsibility of the outside circuitry to pull DTACK low after it is done with the data bus.

For a READ, DTACK is asserted when the outside circuit has placed its data on the data bus. For a WRITE, the device asserts DTACK after it has stored the data from the data bus.

4. STATE 4. At the end of S4 the DTACK signal is checked by the 68000. If it is low by the end of S4, the 68000 proceeds to state 5 to finish the bus cycle. If it is not low yet, the 68000 enters a waiting phase to give the outside device more time to respond.

This is done by substituting WAIT states for states 5 and 6. How many wait states? As many as are needed by the outside device. Each wait state takes one 68000 clock period (125 nanoseconds at 8 MHz). As far as the system is concerned, a wait state behaves exactly like a STATE 4 cycle in that if DTACK goes low before the state ends, the 68000 proceeds to state 5.

For example if three wait states are inserted, the sequence is S0, S1, S2, S3, S4, Sw, Sw, Sw, S5, S6, S7 for a total of 14 cycles instead of the usual eight. Remember that each wait interval takes two state times.

This obviously slows down the 68000, since a memory cycle takes LONGER with wait states inserted.

Once DTACK is asserted, the bus cycle is allowed to finish.

5. STATE 5. Another "do nothing" cycle to give the outside devices time to respond. None of the signals change during this state.

6. STATE 6. We're almost done. Data to be read into the 68000 must be available just before the end of S6, and held until about the end of S7. For a WRITE cycle, nothing happens in S6.

7. STATE 7. Clean-up operations. Address Strobe is deactivated by going high, and likewise Data Strobe returns high. If the R/W line was low for a write operation, it is returned high during S0 of the next cycle.

MORE ABOUT DTACK, ALL ABOUT WATCHDOGS

What's the difference between DTACK and the WAIT line provided by most 8 bit microprocessors? Although they both accomplish the same result of extending CPU bus cycles for slowly responding outside components, the WAIT function works just the opposite as DTACK. The WAIT line is asserted to extend the CPU time; the DTACK signal is asserted to NOT extend the time.

This has some very subtle system implications.

If a WAIT type system tries to access nonexistent memory, the CPU will read or write this "memory" without really knowing that it is not there. For a read, it will bring in garbage (probably the state of the floating data bus lines); and for a write it will write into thin air.

If a DTACK type system tries to access nonexistent memory, the CPU will wait forever for DTACK to be asserted, which of course never happens since there is no memory or DTACK circuitry there. It is the responsibility of the 68000 system designer to make sure that an orderly recovery can be made from this situation.

Who, you might ask, would be dumb enough to try to access nonexistent memory? Believe it or not, I would and you would. Not on purpose, of course, and probably not at first, but as you really get proficient with the 68000, you'll find yourself writing instructions like this:

```
MOVE.L   $45(A2,A3),(A5)+
```

This elegant little program says to move data from one area of memory to another. Where to find the data (the source) is the first expression before the comma; where to put it (the destination) is the second expression.

The $45(A2,A3) says take the 32-bit address in A2, add it to the 16-bit address in A3, and to that add the number $45. That's the source address.

The (A5)+ says use the 32-bit address in A5 as the destination, and then add 4 to it to access the next sequential long (32-bit) word.

Of course, you have loaded A2, A3, and A5 previously, and of course you have made absolutely sure that every possible combination that these values might take on as your program

runs do not send you into non-existent memory land.

You haven't?

The point is that with the fantastic (and complex) addressing modes provided by the 68000, it is fairly easy, especially as you are learning, to try to access memory where it isn't.

What prevents the 68000 from hanging up if this happens?

There is a circuit on the Q-68 board called a watchdog timer which monitors the time between sending out an address and the arrival of DTACK. If no DTACK comes back after 180 microseconds, the timer triggers a 68000 mechanism called BUS ERROR. This is accomplished by pulling the 68000 pin called BERR low.

BERR: ANOTHER WAY TO END A BUS CYCLE

When BERR is asserted, the 68000 enters a special processing mode called an exception. There are various kinds of exceptions which we'll look at later. BERR is an externally generated exception, in that external hardware (in this case the watchdog timer) triggers the exception processing.

What BERR does is to terminate the current bus cycle (the one which had to wait too long for DTACK), and jumps to a routine which you wrote to handle the error condition. The step by step mechanism for this operation is as follows:

1. The Program Counter and Status Registers are pushed on the stack.

2. Four more words which deal with the instruction which was being executed when the BERR happened are pushed on the stack.

3. The 68000 goes to locations $00008-$0000B and picks up a 32 bit address it finds there. This address is loaded into the program counter.

The program now runs at the address you have stored at location $00008-$0000B.

Tricky machine, no?

How do you exit from the Bus Error processing? The normal way to exit from an exception is to execute an RTE (Return From Exception) instruction. This does two things. It pops the stack once to load the status register, and twice again to load the program counter.

This works fine for most exceptions (interrupts, traps and other exceptions), but don't expect it to work with a Bus Error. Why? Because of those other four words which were pushed on the stack for the Bus Error exception.

When your program has completed bus error exception processing it will probably want to jump to an absolute address. Before it does this you need to clean up the stack. You do this by popping 7 words off of the stack. The best way to do this is to add 14 to the stack pointer register A7.

You might be wondering why 14, and not 7 is added to the stack pointer.

The total information saved for the Bus Error exception is the program counter (32 bit--4 bytes), the status register (16 bit--2 bytes), the contents of the 68000 instruction register (16 bits--2 bytes), the contents of the address bus (32 bits--4 bytes), and a special status word which indicates the type of bus cycle which was executing when the error occured (16 bits--2 bytes). You might want to consider this as 7 words, but the 68000 considers memory in byte increments. Therefore to "bump" the stack pointer (A7) by 7 words, add 14 to A7.

TRYING OUT THE BUS ERROR MECHANISM WITH THE Q-68 BOARD

Here is a little program (BERR.EX) which actually lets you see the bus error mechanism in action.

You'll recognize the VIDEOTEST program from Chapter 2 at the start.

```
1050          .OR       $1000
1060          CLR       D0
1070 START    MOVE      #$2000,A0
1080          MOVE      #$00001000,D1
1090 LOOP     MOVE      D0,(A0)+
1100          DBEQ      D1,LOOP
1110          ADDQ      #1,D0
1120 *
1130 * FORCE A WATCHDOG TIMEOUT
1140 *
1150          MOVE      $00030000,D3
1160 *
1170 BACK     TST.B     $C050
1180          TST.B     $C053
1190          TST.B     $C057
1200          BRA       START
1210 *
```

VIDEOTEST uses the Apple II HIRES screen as a giant pilot light to tell you that the 68008 is cranking. To VIDEOTEST we'll add some code to deliberately cause a bus error, and a little more code to handle it.

First, the error itself. Line 1150 tries to read the 16 bit contents of location $30000, and put it into D3. But there is no memory at $30000 in the Q-68 system. The label "BACK" has been added to the instruction following the bogus instruction as a place to return to after the bus error processing is done.

The VIDEOTEST program starts at location $1000. We'll put the bus error handling program at $1100. This small program occupies listing lines 1220 through 1300. All this program does is to delay a while, then jump back to the main program loop ("JMP BACK"). Lines 1270 and 1280 do the delay.

```
1220          .OR       $1100
1230  *
1240  * ADD 7 WORDS TO STACK POINTER TO CLEAN
1250  * UP BERR CONDITION
1260  *
1270          MOVE      #$FFFF,D4
1280  SPIN    DBF       D4,SPIN
1290          ADDA      #$0E,A7   ;ADD 14
1300          JMP       BACK
```

Line 1290 cleans up the stack by adding 14 to the stack pointer. This puts the stack pointer back to where it was before the bus error happened. Line 1300 takes us back into the VIDEOTEST program loop.

```
1310  *
1320          .OR       $800
1330          .DA       $00005000
1340          .DA       $00001000
1350  * BERR VECTOR
1360          .DA       $00001100
```

We're not quite done. The last step is to install the vector at location $00008 to point to the program we just put at $1100. Line 1360 does this. The origin does not need to be set to $8 since it is there anyway due to the previous two .DA statements. The result of line 1360 is to put the address $00001100 at locations $808-$80B. (Remember that this is actually 68008 addresses $00008-$0000B).

Now every time through the VIDEOTEST loop, the program causes a bus error and detours through the routine at $1100, which puts in a long delay. Run the program and satisfy yourself that this is what is happening.

THERE ARE ACTUALLY THREE WAYS...

There are actually three ways to end a bus cycle. DTACK is the normal way. BERR is there for when DTACK isn't (because of a system problem). The third way is with a signal called VPA, Valid Peripheral Address.

The VPA signal is used for two purposes. The primary purpose is to allow peripheral chips which were designed for the MC6800 (8-bit) microprocessor to function with the 16-bit 68000. The second purpose is to activate a special interrupt mechanism called autovectoring. The section on interrupts deals with this second topic.

Take a look at page 6-3 of your 68008 data book (the white one). Figure 6-2 shows a signal called E near the bottom of the timing diagram. This E signal is a 6800 clock signal. All transfers to and from 6800 peripheral devices are synchronized to this E clock.

All 6800 peripheral chips have an E clock input. To run compatibly with the 68000, the 68000 provides an E clock output which simulates the one coming out of a 6800 system. This clock runs all the time, and is derived internal to the 68000 by dividing the 68000 clock by ten. It is high for six clocks, low for four, ad infinitum.

Address and data transfers to and from the 68000 occur at precisely defined states, synchronized to the input CLOCK. The E clock, on the other hand, runs merrily along, without caring what the 68000 is up to. For example, if DTACK comes late in a 68000 cycle, the 68000 pauses for a bit, but the E clock continues to run. What this means is that there is no way to predict the phasing of the E clock with reference to the 68000 system timing.

For this reason, whenever a 6800 type peripheral is to be accessed, the 68000 must slip it's sync until it "lines up" with the E-clock. In other words, it's address and data transfers must be coordinated with the timing (with respect to the E-clock) expected by the peripheral.

How does the 68000 know when to do this?

This is the primary role of the VPA input pin. When this line is asserted (pulled low) at the beginning of a 68000 bus cycle, the 68000 enters a special sequence which inserts enough wait states to synchronize with the E-clock. Figure 6-2 of the 68008 data book shows this.

By the way, figure 6-2 and the second to last paragraph on page 6-1 contain a major inaccuracy. The last sentence of this paragraph reads, "The processor now inserts wait states until it recognizes the assertion of VPA".

Actually, this is backwards. It is the assertion of VPA which tells the processor to insert wait states to synchronize the E clock. VPA is actually sampled at the end of S4 (figure 6-2 shows it being asserted about an inch too far to the right).

This synchronization effect brings up an interesting point. It seems that depending on exactly where the E clock is with respect to the 68000 timing, different amounts of time will be required to synchronize with the 68000 bus cycle. This happens to be exactly correct, as pages 8-6 and 8-7 of the 68008 data manual illustrate. Here the "best" and "worst" cases are shown, the best being the minimum number of wait states required (12), and the worst being the maximum (28).

Chapter 8

Section 8.1 -- DATA SIZES

The 68000 deals with data in three sizes--byte, word and long word. A byte is 8 bits of data. A word is two bytes laid end to end. A long word is four bytes laid end to end.

There are two important facts about 68000 memory addressing.

Fact #1: ALL 68000 MEMORY IS ADDRESSED IN BYTES

If you want to read or write a byte, you can do so at any address. No restrictions, no problems.

Words are considered as two bytes. These two bytes are stored in "high-low" order: The most significant half is at the low (even) address, and the least significant half is stored at the next (odd) address.

Long words are considered as four bytes.

Fact #2: WORDS AND LONG WORDS MUST BE ACCESSED AT EVEN ADDRESSES

Let's suppose that the first four memory locations contain the following data:

```
0    aaaaaaaa
1    bbbbbbbb
2    cccccccc
3    dddddddd
```

Each location contains one byte of data. The 68000 instruction

```
MOVE.B          $0,D1
```

puts aaaaaaaa into the low 8 bits of D1, and the instruction

```
MOVE.B          $1,D1
```

puts bbbbbbbb into the low 8 bits of D1.

Now suppose you want to load more than 8 bits at a time. (This is, after all, one of the main reasons you upgraded from an 8-bit processor). Here are two attempts to do this. Look at them, and and try to figure out which is illegal:

```
#1          MOVE.W          $2,D1
#2          MOVE.W          $3,D1
```

If you're dealing with word-size data, the first four memory locations should be visualized like this:

```
aaaaaaaa:bbbbbbbb----Word Address Ø

cccccccc:dddddddd----Word Address 1
```

Thus WORD Ø is stored as aaaaaaaabbbbbbbb and word 2 is stored as ccccccccdddddddd. If you try to read "word 3", the 68ØØØ won't know what you are talking about. Remember--words exist at even addresses only.

So #1 is ok, and #2 is illegal.

What happens if you actually write the program for try #2? Your assembler should flag this as a "bad address" error. But that's not the whole story on address errors. There are far more subtle ways to generate bad addresses, some of which are not detectible by any assembler! We'll look at how the 68ØØØ deals with these situations in the section on exceptions. For now, let's just say that the 68ØØØ has a way to alert you to address errors as your code is running.

Now for the third case, long words.

Long words are similar to words, in that they are stacked, like so:

```
aaaaaaaabbbbbbbbccccccccdddddddd----Long Word Address Ø


eeeeeeeeffffffffgggggggghhhhhhhh----Long Word Address 4
```

To load the 32 bit word a-b-c-d into D3, you write

```
MOVE.L          $Ø,D3
```

If you try to do this at any odd address, such as

```
MOVE.L          $1,D3
```

your assembler will flag a bad address error.

What if you try this:

```
MOVE.L          $2,D3
```

That's legal, since location 2 is even. But it is not a long word boundary. What will load into D3 is:

```
ccccccccddddddddeeeeeeeeffffffff
```

which is the low half of one long word, and the high half of the other! This is perfectly legal as far as the 68000 is concerned, so be careful to keep your long word boundaries consistent.

The Macro Assembler distinguishes data size by the extensions .B for byte, .W for word, and .L for long word. If you specify nothing, such as

```
MOVE    (A2),(A1)
```

the assembler assumes you meant MOVE.W and uses the 16 bit form.

NUMEROUS PRACTICAL EXAMPLES

Here are programming examples that demonstrate some of the data size distinctions made by the Assembler and by the 68000:

```
                    1030 *--------------------------------
0000F234-           1040 NUM4    .EQ      $F234
12345678-           1050 NUM8    .EQ      $12345678
                    1060 *--------------------------------
```

This code segment illustrates that data values are normally represented as 32-bit quantities. In line 1040, even though NUM4 is defined as the 16-bit value, "F234", the assembler represents it as the 32-bit value, "0000F234" (look at the assembled code in the left column).

LABELS AS DATA VALUES

In 68000 instructions, a label that represents a constant must be preceded by a "#" sign. This tells the assembler that the value is data, and not a memory address. If you are specifying a hexadecimal value, the number is preceeded by "#$".

The size of the data actually used by the 68000 is determined by the instruction extension, ".B", ".W" or ".L".

```
                        1170 *--------------------------------
00001000- 307C F234     1180         MOVE     #NUM4,A0
00001004- 327C 5678     1190         MOVE     #NUM8,A1
                        1200 *--------------------------------
00001008- 247C 0000
0000100C- F234          1210         MOVE.L   #NUM4,A2
0000100E- 267C 1234
00001012- 5678          1220         MOVE.L   #NUM8,A3
                        1230 *--------------------------------
00001018- 123C 0078     1250         MOVE.B   #NUM8,D1
                        1260 *--------------------------------
```

Line 1180: Even though "NUM4" is the 32-bit quantity "0000F234", only the bottom 16-bits are used in the MOVE instruction, since MOVE is the same as "MOVE.W", which specifies word size.

Line 1190: The 32-bit constant "NUM8" is truncated to the low 16-bits for the same reason as in line 1180.

Line 1210, Line 1220: Here the full 32-bit values are loaded due to the ".L" extension on the MOVE instruction.

Line 1250: Here the 32-bit "12345678" is truncated to "0078" because of the ".B" extension. Note that although the constant is coded as "0078", only the "78" loads into D1.

WITH THE "LEA" INSTRUCTION

For the "LEA" instruction, leave off the "#" sign for data values. The Assembler will give a BAD ADDRESS ERROR if you include it. Also note that the LEA instruction is always "long", as if a ".L" were included. (Don't include it though; the LEA instruction does not allow an extension).

```
                        1290 *--------------------------------
0000101C- 49F9 0000
00001020- F234          1300         LEA      NUM4,A4
00001022- 4BF9 1234
00001026- 5678          1310         LEA      NUM8,A5
                        1320 *--------------------------------
```

LABELS AS ADDRESSES

In all of the above examples, the labels "NUM4" and "NUM8" represent data values.

Now let's take a look at labels that specify addresses.

To specify a memory address, leave off the "#" sign.

```
                    1360 *--------------------------------
00001028- 2E39 0000
0000102C- F234      1370        MOVE.L  NUM4,D7
0000102E- 3C39 1234
00001032- 5678      1380        MOVE    NUM8,D6
                    1390 *--------------------------------
```

Now "NUM4" and "NUM8" represent addresses at which the 68000 finds data values. Lines 1370 and 1380 are good examples of the kind of "size" nuances in the 68000 that can trip up the unwary.

There are really two sizes specified by these lines.

The ADDRESS size is 32-bits, as always. You can confirm this by noticing that NUM4 and NUM8 are represented by 32-bit quantities in the opcode (left) column.

The DATA size is determined by the instruction extension, ".B", ".W" or ".L". In line 1370, the 32-bit quantity at address 0000F234 is loaded into D7. In line 1380, the 16-bit quantity at address 12345678 is loaded into D6.

Lines 1370 and 1380 illustrate the "absolute address" mode of addressing. If you wish to force an address expression to 16-bits, you can use the "<" prefix. This saves two bytes of opcode by truncating the 32-bit address to the low 16-bits.

```
                    1440 *--------------------------------
00001034- 2A38 5678 1450        MOVE.L  <NUM8,D5
00001038- 3839 1234
0000103C- 5678      1460        MOVE    NUM8,D4
                    1480 *--------------------------------
```

In line 1450, the "<" prefix has truncated "NUM8" to "5678". Be careful if you do this! The truncated (16-bit) address will be sign-extended before it is sent out as an address. In fact, ANY 16-bit address is sign extended

before it is used. (Section 8.2 deals with sign extension).

Don't be lulled into thinking that the address will be 32-bit because of the ".L" extension in line 1450. As in the previous example, the ".L" specifies the size of the data moved into D4, not the address where the data is found.

By the way, the Assembler also recognizes a ">" prefix which forces the 32-bit form. This prefix is used to force 32-bit branch displacements.

LABELS IN ".DA" DIRECTIVES

We've seen how the "#" sign means a data value, as opposed to a memory address, in 68000 instruction code. In ".DA" directives, the "#" prefix means something else. It means "use the low 8-bits".

Also, "/" means "use the low 16-bits". No prefix means "use the entire 32-bits".

```
                        1550 *--------------------------------
0000103E- 78            1560         .DA      #NUM8
0000103F- 34            1570         .DA      #NUM4
00001040- 5678          1580         .DA      /NUM8
00001042- 1234 5678     1590         .DA      NUM8
                        1600 *--------------------------------
```

A PARTING QUESTION

In program line 1180, what address do you think is loaded into A0? The assembled code says to load "F234".

If you said "0000F234", you'll want to read the next section on sign extension. If you said "FFFFF234", you'll want to read the next section to verify how smart you are.

Section 8.2 -- SIGN EXTENSION

WHEN DOES 25 PLUS 65,530 EQUAL 19?

The 68000's 32-bit address registers introduce some complications not present in 8-bit micros. These complications arise from the fact that the address registers can be loaded with data of two sizes: word or long word.

Motorola could have made life "simple" by insisting that address registers must always be loaded with 32-bit data. For example, the instruction:

```
LEA     #1,A3
```

would contain the operand "00000001", which occupies 4 bytes of data.

It would be nicer to represent the number "1" as "0001", and save the two bytes of leading zeros. A two byte address would accomodate address references from $0000-$FFFF.

The 68000 designers realized this, and wisely provided a mechanism which allows you to specify addresses as either 16-bit or 32-bit data.

Now for a little quiz. What is the value of the following number? (Hint: think of data in two's complement form, as the 68000 does in its address calculations):

```
$0000FFFF
```

You're not supposed to answer a question with a question, but in this case you have to. Before answering the quiz, you must ask the following question:

```
Is this a 16-bit number or a 32-bit number?
```

If you think it is a 16-bit number, your answer should be "-1".

If you think it is a 32-bit number, your answer should be "65535".

Before proceeding, let's quickly review two's complement arithmetic.

SOME TWO'S COMPLEMENT STUFF

The value "-1" is represented as:

$FF	in 8-bit form
$FFFF	in 16-bit form
$FFFFFFFF	in 32-bit form

In two's complement arithmetic, the most significant bit is the sign bit. If the MSB is "1", the number is negative; if it is "Ø" the number is positive.

To interpret a negative number in two's complement form, first complement (flip) all the bits, and then add one. Let's convert the 8-bit $FF. First we note that the MSB is "1", so the number is negative. Next we flip the bits to get ØØ. Then we add one, to get Ø1. The answer: -1.

Now let's do the same for the 32-bit form. MSB is "1": number is negative. Flipping the bits gives ØØØØØØØØ. Adding 1 gives ØØØØØØØ1. Answer, -1, exactly as before.

68ØØØ ADDRESS ARITHMETIC

Let's play 68ØØØ for a minute. We're the 68ØØØ, executing code that some programmer wrote.

We encounter the following instruction

```
LEA     -5(AØ,A1),A4
```

This address mode says we are to add three values: -5, the contents of AØ, and the contents of A1. The "-5" term is an 8-bit quantity; the (AØ) term is a 32-bit quantity; and the (A1) term is a 16-bit quantity. (It might come as a surprise that (A1) is 16-bit--if the programmer wanted it to represent a 32-bit number he or she would have added a ".L" extension, i.e. "A1.L"). The result of this addition is to be placed into A4.

Let's suppose we look inside AØ and A1, and find the following numbers:

```
A0:  00000000
A1:  0000F000
```

Here's the addition:

```
A0:      00000000
A1:    + 0000F000
-5     +       FB
         --------
```

Those of you who charged ahead and got "0000FOFB", stay after class and spend 2 hours writing 8086 code!

The first thing to recognize is that the number "-5" is NEGATIVE, and it thus must be represented as a minus number in 32-bit form. This is done by an operation called sign extension.

Sign extension takes the MSB and copies it into all of the higher bits. The 8-bit quantity is stretched to 32-bit form by sign extension, which transforms "FB" into "FFFFFFFB". So now we can represent the addition as:

```
A0:      00000000
A1:    + 0000F000
-5     + FFFFFFFB
         ---------
```

NOW can we do the addition?

Almost. But first we need to check the A1 value for SIZE. If the programmer meant "0000F000" to be a 32-bit quantity, we can go ahead and add.

But if the programmer intended "0000F000" to be a 16-bit quantity, then "F000" really represents a negative number, and it must be sign extended to 32 bits.

Since the ".L" extension was not supplied in the instruction, we know that the A2 value is supposed to represent a 16-bit number. So we must sign extend it to preserve its minus sign:

```
A0:      00000000
A1:    + FFFFF000
-5     + FFFFFFFB
         ---------

         FFFFEFFB
```

Now we can answer the question posed at the beginning of this section:

When does 25 plus 65530 equal 19?

It's when "FFFA" represents a 16-bit number, and is therefore sign extended to represent "-6".

Before sign extension:

```
25:          00000019
65530     + 0000FFFA
            --------
```

After sign extension:

```
25:          00000019
-6        + FFFFFFFA
            --------
19           00000013
```

What does sign extension do to a positive number? Nothing. Look at these examples of sign extension, and remember that a positive number has an MSB of "0":

8-bit	16-bit	32-bit
$23	$0023	$00000023
$01	$0001	$00000001

DATA REGISTERS AND ADDRESS REGISTERS

Sign extension can produce some unexpected results if you're not used to it. Before looking at its effect in Address Registers, let's first look at how Data Registers behave.

Data Registers act "normally". Data Registers can be loaded with three different sizes, byte, word, and long word. Only the portion of the register you load changes; the remaining bits retain their previous value.

Suppose register D3 contains $ABCDEF00. The instruction

```
MOVE.B          #$23,D3
```

changes the contents of D3 to $ABCDEF23, and the instruction

```
MOVE.W          #$2345,D3
```

changes the contents of D3 to $ABCD2345. What about this instruction:

```
MOVE.L          #$2345,D3
```

This loads 00002345 into D3. The Assembler supplies the leading zeros.

This is all pretty straightforward.

But now suppose Address Register A3 contains $ABCDEF00 and you execute this instruction:

```
MOVEA.W         #$2345,A3
```

You might expect A3 to contain $ABCD2345, right? It doesn't. The reason it doesn't is that the 68000 automatically sign extends "2345" to 32 bits, wiping out the 16 high bits in the process. So the value in A3 is actually $00002345

Consider the following instructions:

```
MOVEA.W         #$2345,A3

MOVEA.W         #$9345,A4
```

The first one loads $00002345 into A3; the second one loads $FFFF9345 into A4.

Here's an important rule to remember:

> 16-bit data which is loaded into an Address Register is always automatically sign extended

If you wish to sign extend a Data register, you can use the EXT (extend) instruction. There are two forms of the EXT instruction. EXT.W extends an eight bit value to 16 bits, and EXT.L extends a 16 bit value to 32 bits. "EXT" works only on Data registers, since sign extension is automatically handled in Address registers.

There is one address mode in which a Data Register is automatically sign extended. Consider this instruction:

```
LEA       $45(A2,D3),AØ
```

We've seen one like this before, where the second register was an address register, not a data register. Because this is an address calculation, it is performed in 32 bits. Thus, since D3 is specified as a 16-bit quantity (no ".L"), it must first be sign extended.

D3 is sign extended for purposes of the address calculation only--its contents are not "permanently" sign extended, and the upper 16 bits are preserved.

As with the address register, you can force the entire 32-bit contents of D3 to be added in the address calculation by putting a ".L" extension onto "D3":

```
LEA       $45(A2,D3.L),AØ
```

68ØØØ SIGN EXTENSION AND THE APPLE MEMORY.

The Q-68 board's low 64 Kbytes are occupied by the 64K memory in the Apple II. As long as a 68ØØØ program is referencing addresses between $ØØØØ and $7FFF, the "short" form of address loading may be used. For example, to load the start of HIRES screen #1 into A3, you may write:

```
MOVEA     #$2ØØØ,A3
```

The 16-bit MOVE is ok since the 15th bit of this address is Ø, and the actual (sign extended) value loaded into A3 will be $ØØØØ2ØØØ.

But what happens when you try this:

```
MOVEA     #$96ØØ,A3
```

You have specified a sixteen bit load (remember that MOVE is equivalent to MOVE.W) into A3, with an address whose 15th bit is a 1. The actual value loaded into A3 is $FFFF96ØØ, which is nowhere near the Apple II memory space!

The moral: whenever you load an address register with an Apple address of $8ØØØ or higher, be sure to specify a full 32-bit load (using the ".L" extension).

Section 8.3 -- The TAS Instruction

The TAS (test and set) instruction is one not usually found in a microprocessor. It's primary use is to allow multiple processors which share the same bus to coordinate activities without running into each other.

TAS is the only "read-modify-write" (RMW) instruction in the 68000. In a RMW cycle,

1. A memory location is read into the CPU (read).

2. The CPU does something with the data (modify).

3. The CPU writes new data out to the same memory location (write).

A TAS instruction implements these specific steps:

1. A byte of data is read into the CPU. The TAS instruction allows byte data only.

2. The Z (zero) and N (negative) condition codes are set according to the byte read in step 1. If all eight bits are 0, the Z flag is set; if the MSB (bit 7) is 1, the N flag is set.

3. The byte is rewritten to the same memory location, with one change: Bit 7 is set to a 1.

The TAS instruction is referred to as "indivisible", meaning that once it starts execution, it can't be interrupted or suspended in any synchronized manner. Of course a system RESET will stop it, but this assumes a catastrophic restart.

What is the use of this strange instruction?

Let's suppose that two 68000's and a single printer are connected to the system bus. A bit is assigned which indicates "printer busy--stay away". Both CPU's can read and set this bit. This bit is called a "semaphore" because it is used as a signal flag between the two CPU's.

It is agreed that either CPU will test this bit before using the printer, to make sure that it doesn't start sending characters to the printer while the other CPU is using it. The agreed-upon format (protocol, in computerese) is that

the bit must be LOW to indicate not busy before either CPU can use the printer, and a CPU which is going to use the printer sets the bit high to keep the other one away.

Let's say CPU-A decides to use the printer. It reads the bit, finds it low, and sets it to a 1 to indicate that it is using it (printer busy). Now let's say CPU-B decides to print something. It checks the bit, finds it high, and waits for CPU-A to finish.

Everything is fine, except for one case. What if CPU-A and CPU-B try to use the printer at exactly the same time? CPU-A reads a zero, indicating ready. It takes CPU-A at least one bus cycle to go back to memory to set the semaphore to a 1. But before it does this, CPU-B reads the bit, and also finds it low!

CPU-B also finds the printer ready, and proceeds to set the semaphore to a one (just after CPU-A does). Both CPU's now happily print away, and the result is garbage.

To prevent this case, the instruction which tests the semaphore bit must also set the bit, so that there is no chance of another CPU sliding in and testing the bit before it is set. This mechanism is what is referred to as an "indivisible cycle"--the testing and setting cannot possibly be disrupted.

The way the 68000 makes the TAS instruction indivisible is to assert Address Strobe for the initial instruction fetch, and then keep it low for the remainder of the cycle which includes the writing back part. Other CPU's sitting on the bus know not to grab the bus while another CPU has pulled AS low, so the non-interfering timing is accomplished.

Section 8.4 -- Two Stack Pointers Named "A7"

There are two system stack pointers in the 68000. Motorola confuses this feature a bit by calling them both "A7".

Before looking at the strange case of the two 32-bit registers which have the same name, let's first clarify the difference between a "system" stack and a "normal" stack.

The 68000 has two addressing modes which automatically "bump" the contents of a memory pointer every time the pointer is used. These are called the predecrement and postincrement address modes. These modes apply only in indirect address modes. Indirect addressing means that the contents of an address register point to a location in memory which holds a data value, rather than indicating the data value itself.

For example, suppose address register A3 contains $1020, and you execute the following instruction:

```
MOVE.W          D3,-(A3)
```

This moves the contents of D3 into the memory locations $101F and $101E. After the instruction executes, A3 contains $101E. This instruction executes in four steps:

1. A3 is decremented by one. This puts its value at $101F.
2. The low byte of the low word in D3 is stored at (A3), which is $101F.
3. A3 is decremented by one again. It is now $101E.
4. The high byte of the low word in D3 is stored at (A3), which is $101E.

What if you try:

```
MOVE.L          D3,-(A3)
```

The 68000 is clever enough to realize that four bytes of data are to be moved (from the ".L" extension), and decrements A3 four times to store the data at $101F, $101E, $101D and $101C.

The position of the minus sign is a reminder that the A3 value is decremented before storing each byte of the value in D3.

Now let's do the reverse, and load the bytes back into D3:

```
MOVE.W          (A3)+,D3
MOVE.L          (A3)+,D3
```

In the first case, A3 is incremented twice to load the two bytes into D3; in the second case A3 is incremented four times due to the long word (.L) specification. The plus sign following the (A3) reminds us that the incrementing of A3 takes place after the data is moved.

In these examples A3 functions as a true stack pointer. We could name these operations PUSH D3 and POP D3 (in a Macro, perhaps?).

A programmer friend measures the value of a program by how many PUSH'es and POP's it contains. He loves stacks. You should have seen the fire in his eyes when he realized that with the 68000 he could have 9 stacks going at once! This is possible because the predecrement and postincrement modes can be used on any of the 9 address registers.

You may have noticed that the 68000 has no "PUSH" or "POP" instructions. You use the predecrement and postincrement address modes to effect these instructions.

THE REAL STACK POINTER(S)

Aside from the complement of address registers which may be used as stack pointers, there is an "official" stack pointer in the 68000. This is register A7. When any exception happens, the "pushed" data goes onto the stack area pointed-to by this register.

A7 is the register with two names (three, actually, if you include "A7"). It is known as the Supervisor Stack Pointer (SSP), and the User Stack Pointer (USP). Which it is called depends on whether the 68000 is running in User or Supervisor state.

SUPERVISOR AND USER STATES

The idea behind these two states is that the 68000, being a

"big" machine, will likely be running an operating system which serves one or more "users". The operating system is written by the system architect, who has the status of "High Priest". Only the High Priest has the power to make the really big mistakes, like running the stack into a data table, or halting the processor.

The User has no such privileges. He is regarded as an unknown quantity. He may be a mischievous hacker trying to erase all the disk files. Or, he might simply be a beginner who is likely to make all kinds of programming mistakes.

In any event, the operating system must be insulated from "users".

Hence the Supervisor state. When anything unusual happens, the supervisor state is entered, and the operating system takes over. This is when A7 becomes the Supervisor Stack Pointer.

It's easy to switch the 68000 to User Mode. Bit 13 of the Status Register is the "S" bit, which stands for "Supervisor". Simply set this bit to 0, and you're in the User mode. Now any reference to A7 will use the User Stack Pointer.

It's not so easy to switch from User to Supervisor mode. If you try to set the Status Register back to a "1", you'll trigger a Privilege Violation exception. If the 68000 is indeed running an operating system, the exception will hand control to the High Priest, who will take appropriate action.

The 68000 has output pins that indicate when it is in Supervisor State or User State. The hardware designer (who works with the High Priest) can enable operating system memory only in the Supervisor State, and thus insure that it is not accessible to probing Users.

Note that certain information is "stacked" during an exception. The A7 register called SSP serves as the stack pointer during exceptions.

THE Q-68 SUPERVISOR/USER DEMOCRACY

The Q-68 board makes absolutely no distinction between Supervisor and User states. All of its memory is accessible

in both states. The REGISTERS screen in DEBUG shows A7 as SSP and USP. Remember that A7 represents either SSP or USP, depending on the state of the "S" bit in the Status Register.

When you start up the 68Ø08 on the Q-68 board, the initial value of the SSP is fetched from 68Ø08 address Ø (Apple II address $8ØØ). This is set to $188ØØ, the top of onboard Q-68 RAM. If you are using DEBUG, don't worry about interfering with DEBUG's stack. It's at $186ØØ.

When you enter the User state, A7 no longer points to the system stack. The USP is now in effect, not the SSP. If you wish to use the USP, you must first initialize A7 to point to the top of your user stack. This will NOT change the value of SSP, since the SSP and USP are two distinct registers, even though they are both called "A7".

Section 8.5 -- 68000 Gotcha #1:

The Case of the Creeping Program Size

A very strange thing happened to us while developing the Qwerty DEBUG program.

The program was developed with the Macro Assembler, with the program origin at $1000. Once the program was debugged (we sure could have used DEBUG to debug DEBUG), it came time to burn an EPROM for use on the Q-68 board.

We made two changes to the DEBUG source program. First, the program origin was changed from $1000 to $10084, the beginning of DEBUG in the EPROM. And second, a "target address" directive was added to put the object code at $1000.

```
.OR     $10084
.TA     $1000
```

When the assembly was complete, we were astonished to find that the program size had increased by 74 bytes!

Same program, only a different origin. What's going on?

Unfortunately, nobody makes a 8266 byte EPROM, so we had to find the reason and put those 74 bytes back into the bit bucket, where they presumably came from.

After considerable head scratching, the solution was found by examining instructions which dealt with program addresses. In the origin $1000 version, all address references were between 0 and $FFFF (in the Apple II memory space). This meant that all addresses could use the 16 bit form.

But when the program was reassembled with an origin of $10084, the address references could no longer fit in 16 bits, so the assembler correctly generated the 32 bit form.

Consider the following pair of instructions. The assembler op-codes are shown to the left of the instructions:

```
47F9 0001 0084      LEA     $10084,A3
47F8 1000           LEA     $1000,A3
```

See the difference? The first one requires six bytes--two for the opcode (47F9) and four for the address (0001 0084). The second one requires only four bytes--two for the opcode (47F8) and two more for the sixteen bit address 1000.

Now that the problem was identified, how to fix it? How do you reduce 32-bit address references to 16 bits?

The answer lies in the 68000 addressing modes called "PC-Relative", which stands for "Program Counter Relative".

This mode is usually touted as the one which promotes position independent code. But it also got us out of the long address pinch. Here's how.

In PC-Relative addressing, the effective address is formed by adding the current value of the program counter to a 16-bit displacement. This 16-bit value is treated as a two's complement number, which means that it can be either positive or negative.

This means that an address is not represented in absolute form (such as $10084), but rather as a distance from the current instruction. How far away can the address be? A 16-bit two's complement number can represent the range of about plus or minus 32,000 bytes.

In our case, the DEBUG program size is 8 kilobytes, so we can easily "reach" anywhere in the program, forward or backward, with a 16 bit offset.

So we went back to the DEBUG source and changed all absolute memory references to PC relative addressing, and the program shrank back to its former size. It now makes no difference where it is assembled--the program is always the same size. This is comforting.

To see how we made these changes, look at the following program:

```
1020         LEA     TABLE,A0
1040         LEA     TABLE(PC),A1
1060 TABLE   .DA     $00000000
```

Line 1020 represents the "before" form. A table is about to be accessed, so its address is loaded into A0.

Line 1040 is the "after" (corrected) form. Same table, but now the PC relative mode is used.

Now let's assemble this program at origin $1000. (We've taken out the source code line numbers for clarity):

```
00001000- 41F8 1008             LEA     TABLE,A0
00001004- 43FA 0002             LEA     TABLE(PC),A1
00001008- 0000 0000     TABLE   .DA     $00000000
```

Both instructions generated four-byte op-codes. In the first line, the absolute address of TABLE (1008) appears in the opcode. In the second line, the displacement (distance) from the address TABLE appears in the opcode (0002). You might notice that the displacement is measured from the end of the first word of the instruction opcode. Don't worry about this. The assembler figures out the correct displacement for you.

So far, no difference between the two instructions. They both use the same amount of memory, and they both do exactly the same thing. But now let's reassemble the same program at $10084:

```
00010084- 41F9 0001
00010088- 008E                  LEA     TABLE,A0
0001008A- 43FA 0002             LEA     TABLE(PC),A1
0001008E- 0000 0000     TABLE   .DA     $00000000
```

Now the first line has grown by two bytes, to hold the full address 0001 008E, which is the new address for TABLE.

But TABLE is still the same number of bytes away from the second instruction as before, so exactly the same instruction (43FA 0002) is generated! The PC relative mode has saved two bytes when the program was assembled at an address above $FFFF.

Just to show you how much work the assembler does for you, let's modify the program to change the distance between the LEA instruction and the address TABLE.

```
00001000- 41F8 100A             LEA     TABLE,A0
00001004- 43FA 0004             LEA     TABLE(PC),A1
00001008- 4E71                  NOP
0000100A- 0000 0000     TABLE   .DA     $00000000
```

We stuck in a harmless "NOP" instruction just before "TABLE". Now take a look at the second word of the second instruction. This is the new displacement to TABLE (0004), which is two bytes larger than it was before we inserted the two byte NOP. This is because "TABLE" is two bytes further away from the "LEA TABLE(PC),A1" instruction than before.

One last item: don't forget that the displacement can be forward or backward. If TABLE is placed before the "LEA" instruction, the offset value calculated by the assembler is negative.

```
00001000- 0000 0000      TABLE  .DA     $00000000
00001004- 41F8 1000             LEA     TABLE,A0
00001008- 43FA FFF6             LEA     TABLE(PC),A1
```

(For those of you without a hex calculator, $FFF6 is -10)

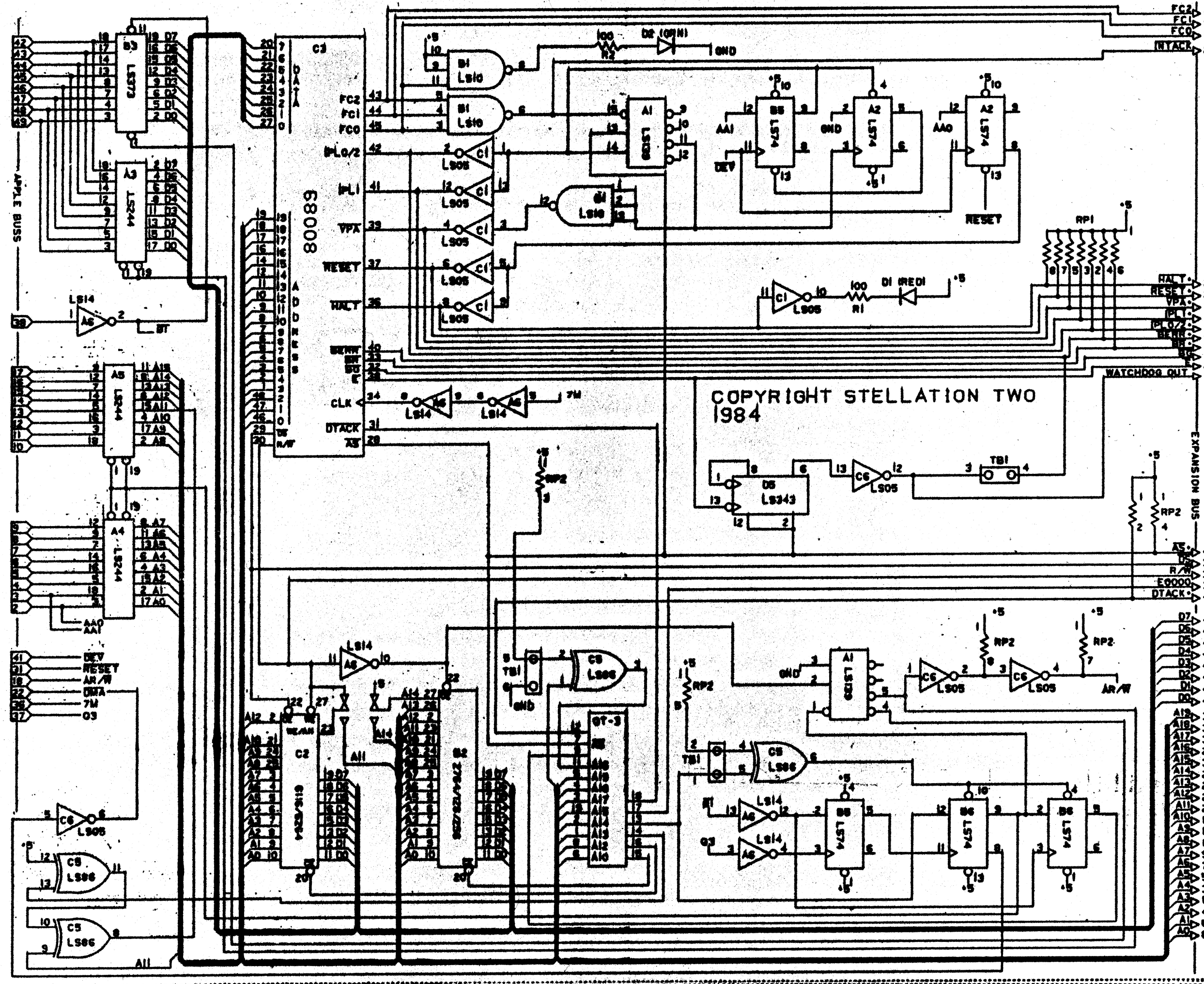
Remember to start counting back from the end of the first LEA opcode word (at the "F8").

Applications
Notes

Inserts-1

Index

INDEX



COPYRIGHT STELLATION TWO
1984
APPLE BUSS
EXPANSION BUS
68008
C3
DATA
ADDRESS
LS373
LS244
LS14
LS10
LS05
LS139
LS74
LS343
LS86
6116/6264
2764/128/256
GT-3
RP1
RP2
TB1
D2 (GRN)
D1 (RED)
GND
WATCHDOG OUT
FC2
FC1
FC0
INTACK
HALT
RESET
VPA
IPL1
IPL0/2
BERR
BR
BG
DTACK
CLK
AS
R/W
E0000
DEV
AR/W
DMA
7M
Q3
AA0
AA1