SAGE
COMPUTER

SDT
Sage
Debugging
Tool

Copyright (c) 1983, Sage Computer Technology, Reno, NV 89502

All rights reserved. Reproduction or use, without express permission of editorial or pictorial content, in any manner, is prohibited. No patent liability is assumed with respect to the use of the information contained herein. While every precaution has been taken in the preparation of this book, Sage Computer Technology assumes no responsibility for errors or omissions. Neither is any liability assumed for damages resulting from the use of the information contained herein.

SAGE™ is a trademark of SAGE Computer.
p-SYSTEM™ is a trademark of SofTech Microsystems.
CP/M-68K™ is a trademark of Digital Research.

SAGE COMPUTER
4905 Energy Way
Reno, Nv. 89502
(702) 322-6868

December 1983

Table Of Contents

I	**INTRODUCTION**	1
I.01	WHEN TO USE ASSEMBLY LANGUAGE	3
II	**SAGE MEMORY MAP**	8
II.01	RAM	8
II.02	PROM	8
II.03	I/O PORTS (general)	9
II.04	I/O PORTS (specific)	10
II.05	RAM Memory Allocation	14
III	**SAGE PROMS**	15
III.01	PROM VERSIONS	16
III.02	PROM START-UP TESTS	17
III.03	RAM MEMORY TEST	21
	DISABLING THE MEMORY TEST	22
IV	**DISK BOOTSTRAP**	23
IV.01	p-SYSTEM FLOPPY BOOT	24
	FLOPPY FORMAT	26
IV.02	p-SYSTEM WINCHESTER BOOT	26
IV.03	LOADING THE p-SYSTEM BIOS	27
V	**PROM ENTRY POINTS**	30
VI	**SAGE DEBUGGING TOOL**	37
VI.01	SDT PHILOSOPHY	38
VI.02	SDT REGISTER USAGE	38
VI.03	SDT QUICK DESCRIPTION	39
VI.04	SDT DETAILED DESCRIPTION	47
	EXCEPTION ERRORS	81
VII	**LINK INFORMATION FOR THE 68000**	84

TABLE OF CONTENTS

VII.01 WORKING IN ASSEMBLY LANGUAGE 84

VII.02 EXAMPLES 86

VII.03 STACK AND REGISTER USAGE 91

VIII **68000 EXAMPLES** 93

IX **STAND-ALONE LOADER** 99

APPENDIX A: EXCEPTION ERRORS 101

APPENDIX B: FLOPPY DISK BOOT ERRORS 104

APPENDIX C: WINCHESTER DRIVE ERRORS 105

APPENDIX D: PROM ROUTINE ENTRY POINTS 106

APPENDIX E: DEBUGGER COMMANDS 107

INDEX 114

ATTACHMENT: SOFTECH ASSEMBLER MANUAL

ATTACHMENT: SCHEMATICS

I INTRODUCTION :

The purpose of this manual is to provide programmers with the information needed to rapidly develop and debug assembly language programs. The content is aimed at seasoned programmers and is not intended to be a tutorial. If you haven't programmed in assembly language before, we recommend the following reference material.

68000 ASSEMBLY LANGUAGE PROGRAMMING

By Gerry Kane, Doug Hawkings, and Lance Leventhal, published by OSBORNE/McGraw-Hill. This is a good introductory text for programmers who are unfamiliar with assembly language programming.

THE 68000: PRINCIPLES AND PROGRAMMING

By Leo J. Scanlon, a SAMS publication. This is essentially a textbook for learning assembly language programming techniques on the 68000. It includes good discussions on the philosophy of the 68000, excellent programming examples, and valuable reference material.

MC68000 16-BIT MICROPROCESSOR USER'S MANUAL

Available from Motorola Inc., this manual provides reference material intended for use by computer designers, software architects, and design engineers. It contains a complete description of the 68000 command set, essential to anyone who is going to program at the assembly level. Also included are timing information, pin descriptions, and hardware interfacing notes.

INTRODUCTION

There are a number of reasons why advanced programmers use assembly language in the development of their programs, even though assembly language programming ordinarily requires more time and effort than programming in a high-level language such as Pascal, BASIC, or FORTRAN.

One reason is speed. Speed improvements of 10 to 100 times can be achieved by translating time-critical sections of a high-level program into assembly language. The easiest way to do this is to use the "Native Code Generator" utility. (See the Utility section in the PROGRAM DEVELOPMENT MANUAL for more information on the "NCG".)

Another reason programmers use assembly language is to access low-level portions of the computer's hardware. Following this introduction is an example which illustrates some ways to achieve speed and low-level access.

This manual contains information on the memory map of the SAGE, examples on how to link assembly language routines to high-level programs, and details of the operation of the PROM routines. The Index and Table of Contents contain references to all major topics. An appendix provides a list of assembler errors and run-time errors.

I.01 WHEN TO USE ASSEMBLY LANGUAGE :

The decision to use assembly language in a program should not be taken lightly. Although assembly language offers speed, code efficiency, and low-level access, there are also several disadvantages. Assembly code usually takes longer to develop and debug than high-level code, and it is more difficult to modify or expand at a later date. It is also not portable between different machines (to an Apple or IBM system, for instance). The p-code produced by the Pascal, BASIC, and FORTRAN compilers will often run on an Apple or IBM system with little modification. Assembly code produced for the 68000 definitely will not.

The following example illustrates three different approaches to a programming problem. We hope this will help the user select the approach most suited to his or her particular application.

"LED_Test" is a simple program which flashes the SAGE LED status light green and red 50 times. This task requires a bit to be set and cleared in one of the SAGE IV's output ports. Such an access to the low-level hardware normally requires assembly code, but a special UNITWRITE statement (see the TECHNICAL MANUAL) allows us to access the SAGE LED directly from a Pascal program. BASIC and FORTRAN lack this capability, so we would be forced to link to an assembly language routine or a Pascal unit if we were using either of those languages.

The source text of three versions of "LED_Test" follows this text. "LED_Test_1" is the original program written in Pascal. "LED_Test_2" has two additional compiler options which allow the subsequent generation of native code (machine code) from the p-code. This is achieved using the "Native Code Generator" utility (see the PROGRAM DEVELOPMENT MANUAL). "LED_Test_3" is written primarily in assembly code which is linked to a small Pascal program. The assembly code was assembled using SYSTEM.ASSMBLER, the Pascal host was compiled using SYSTEM.COMPILER, and then the two code files were linked using SYSTEM.LINKER (this process is

described in detail on page 87)

Here are the results of a timing test, along with the final code size of each program.

Name		Execution time (seconds)	Code size (words)
LED_Test_1	(p-code)	43.7	108
LED_Test_2	(native)	3.7	163
LED_Test_3	(assembly)	0.6	78

As you can see from this example, translation to native code using the "Native Code Generator" offers an attractive compromise between pure p-code and pure assembly code. Unfortunately, the dramatic increase in speed is accompanied by a considerable increase in code size.

The example also shows that assembly language is very desirable in environments where speed and efficiency are essential. "LED_Test_3" ran 6 times faster than the translated native code version, and it took only half the space. It also ran 72 times faster than the original Pascal program. However, these benefits must be weighed against the costs of increased program development time, more difficult modification of the program in the future, and loss of portability between different microprocessors.

LED_Test_1

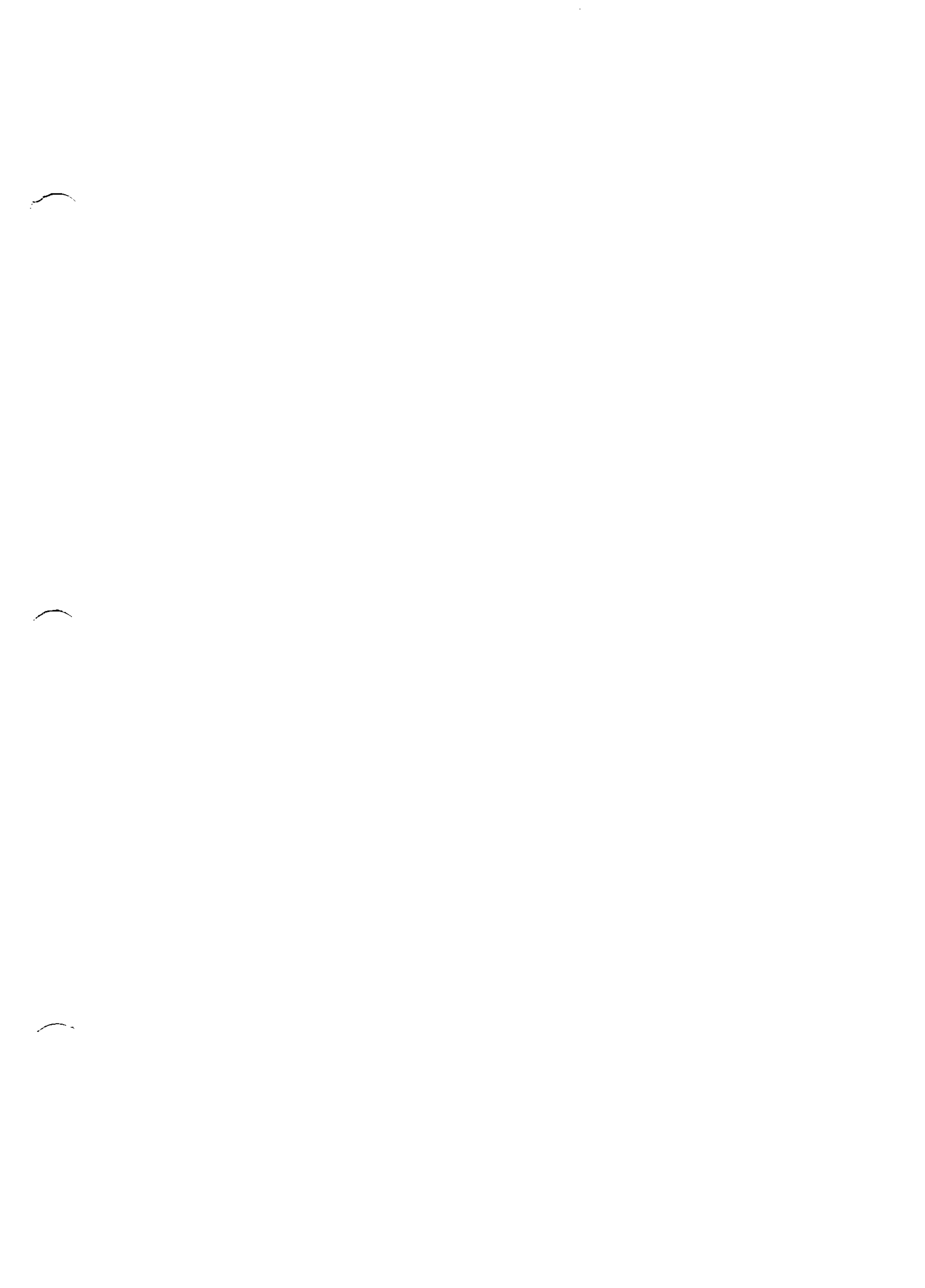
```
PROGRAM LED_Test_1;            { *** P-code version *** }

CONST LEDLO = -16281;          { Address of LED }
      LEDHI = 255;

VAR I,J: INTEGER;
    Red, Green: PACKED ARRAY[0..1] OF 0..255;

BEGIN
WRITELN('Start',CHR(7));       { Beep bell to start time test }
Green[0]:=6; Red[0]:=7;        { Define values to turn LED green and red }
FOR I:=1 TO 50 DO              { Flash LED 50 times }
    BEGIN
    UNITWRITE(130,Green[0],1,LEDLO,LEDHI);      { Turn LED green }
    FOR J:=1 TO 5000 DO;                        { Waste some time }
    UNITWRITE(130,Red[0],1,LEDLO,LEDHI);        { Turn LED red }
    FOR J:=1 TO 5000 DO;                        { Waste some time }
    END;
WRITELN(CHR(7),'End');            { Beep bell to end time test }
END.
```

LED_Test_2

```
PROGRAM LED_Test_2;           { *** Native Code Generated version *** }

CONST LEDLO = -16281;         { Address of LED }
      LEDHI = 255;

VAR I,J: INTEGER;
    Red, Green: PACKED ARRAY[0..1] OF 0..255;

{$N+}                         { *** Start native code translation *** }
BEGIN
WRITELN('Start',CHR(7));      { Beep bell to start time test }
Green[0]:=6; Red[0]:=7;       { Define values to turn LED green and red }
FOR I:=1 TO 50 DO             { Flash LED 50 times }
    BEGIN
    UNITWRITE(130,Green[0],1,LEDLO,LEDHI);      { Turn LED green }
    FOR J:=1 TO 5000 DO;                        { Waste some time }
    UNITWRITE(130,Red[0],1,LEDLO,LEDHI);        { Turn LED red }
    FOR J:=1 TO 5000 DO;                        { Waste some time }
    END;
WRITELN(CHR(7),'End');           { Beep bell to end time test }
END.
{$N-}                            { *** End native code translation *** }
```

LED_Test_3

```
PROGRAM LED_Test_3;              { *** Linked assembly code version *** }

PROCEDURE AssemblyProg; EXTERNAL;  { Link to assembly program which }
                                   { will do all the work }
BEGIN
WRITELN('Start',CHR(7));             { Beep bell to start time test }
AssemblyProg;                        { Assembly program does the rest }
WRITELN(CHR(7),'End');               { Beep bell to end time test }
END.


;  *** This is the assembly language file which is linked to the
;  *** above Pascal program after it is assembled.

        .RELPROC AssemblyProg

LEDLOC  .EQU    0C067H          ;Address of LED control port (FFC067H)

        MOVEQ   #19.,D0         ;Enter supervisor mode so we can access
        TRAP    #14.            ;  I/O area without a bus error
        MOVEQ   #6.,D1          ;Initialize GREEN value
        MOVEQ   #7.,D2          ;Initialize RED value
        MOVEQ   #49.,D0         ;Initialize counter for 50 iterations

$10     MOVE.B  D1,LEDLOC       ;LED = GREEN
        MOVE.W  #5000.,D3       ;Do nothing 5000 times
$20     DBF     D3,$20          ;Wait

        MOVE.B  D2,LEDLOC       ;LED = RED
        MOVE.W  #5000.,D3       ;Do nothing 5000 times
$30     DBF     D3,$30          ;Wait

        DBF     D0,$10          ;Flash 50 times

        ANDI.W  #0DFFFH,SR      ;Return to user mode
        RTS
        .END
```



II SAGE MEMORY MAP :

These tables show the allocation of memory and I/O addresses for the SAGE IV computer.

II.01 RAM :

Addresses (hex)	Contents
000000 - 0000FF	Interrupt & Exception Vectors (see literature on 68000 processor)
000100 - 0001FF	Debugger (SDT) RAM area
000200 - 0002FF	BIOS RAM area
000300 - 0003FF	Debugger (SDT) system stack
000400 - 0103FF	P-System data area (64K, 64K max.)
010400 - 0133FF	P-System Interpreter (12K)
013400 - 0233FF	P-System code pool (64K, 64K max.)
023400 - BIOS	RAM Disk

(Note: Bios is placed at top of equipped RAM memory. See discussion at end of this section.)

II.02 PROM :

FE0000 - FE3FFF	Current PROMS (16K) include startup tests and simple diagnostic tools; the Sage Debugging Tool, disassembler, and low-level I/O routines. See page 15 for further information on PROMS.
FE4000 - FEFFFF	Reserved for future PROM expansion.

II.03 I/O PORTS (general) :

SAGE has defined a major Input/Output partition for each of up to 16 boards. Each partition is divided into 16 device areas of 64 bytes.

Most peripherals are accessed using bytes instead of words, so they are addressed using the low byte of each word (the odd addresses).

Address	Board	Use
FFC000 - FFC3FF	Board #1	(Main CPU I/O)
FFC400 - FFC7FF	Board #2	(Hard Disk and additional serial ports)
FFC800 - FFCBFF	Board #3	(Reserved for SAGE expansion)
FFCC00 - FFCFFF	Board #4	(" ")
FFD000 - FFD3FF	Board #5	(" ")
FFD400 - FFD7FF	Board #6	(" ")
FFD800 - FFDBFF	Board #7	(" ")
FFDC00 - FFDFFF	Board #8	(" ")
FFE000 - FFE3FF	Board #9	(" ")
FFE400 - FFE7FF	Board #10	(" ")
FFE800 - FFEBFF	Board #11	(" ")
FFEC00 - FFEFFF	Board #12	(" ")
FFF000 - FFF3FF	Board #13	(Reserved for User)
FFF400 - FFF7FF	Board #14	(" ")
FFF800 - FFFBFF	Board #15	(" ")
FFFC00 - FFFFFF	Board #16	(" ")

II.04 I/O PORTS (specific) :

Except for the SAGE LED status light, all I/O ports can be configured and accessed using calls to the SAGE BIOS. We do not recommend that you access these ports directly due to possible conflicts with the BIOS.

Note: The 68000 must be in supervisor mode to access these addresses. Access will be denied and a bus error will result if the 68000 is in user mode.

Board #1 (Main CPU I/O)

Address	Function	Device
FFC001	REAL TIME CLOCK	(A)
FFC003	SERIAL PORT 1 BAUD RATE	(B)
FFC005	SERIAL PORT 2 BAUD RATE	(C)
FFC007	MODE WORD FOR A,B AND C ABOVE	(8253-S)
FFC009 - FFC00F	RESERVED	
FFC011 - FFC01F	IEEE-488 INTERFACE	(TMS9914)
FFC021	GROUP-A DIP SWITCH	(A)
FFC023	GROUP-B DIP SWITCH	(B)
FFC025	FLOPPY CONTROL PORT	(C)
FFC027	CONTROL FOR A,B, AND C ABOVE	(8255A-S)
FFC029 - FFC02F	RESERVED	
FFC031	SERIAL PORT 2 (REMOTE) DATA	(8251A)
FFC033	SERIAL PORT 2 CONTROL/STATUS	
FFC035 - FFC03F	RESERVED	
FFC041 - FFC044	INTERRUPT ENCODER CONTROL	(8259)
FFC045 - FFC04F	RESERVED	
FFC051	FLOPPY DISK STATUS	(NEC 765)
FFC053	FLOPPY DISK CONTROL	
FFC055 - FFC05F	RESERVED	
FFC061	PRINTER INTERFACE PORT A	
FFC063	PRINTER INTERFACE PORT B	

FFC065	PRINTER INTERFACE PORT C (LED status light is controlled by bit 3. 1=red, 0=green)	
FFC067	PRINTER INTERFACE CONTROL	(8255A-S)
FFC069 - FFC06F	RESERVED	
FFC071	SERIAL PORT 1 (TERMINAL) DATA	(8251A)
FFC073	SERIAL PORT 1 CONTROL/STATUS	
FFC075 - FFC07F	RESERVED	
FFC081 - FFC087	REAL TIME CLOCK	(8253-S)
FFC089 - FFC08F	RESERVED	

Board #2 (Hard disk and additional serial ports)

Auxiliary Serial Channel Ports:

Address	Description
FFC401	AUX 4 SERIAL CHANNEL DATA (2651)
FFC403	AUX 4 SERIAL CHANNEL STATUS
FFC405	AUX 4 SERIAL CHANNEL MODE REGISTER
FFC407	AUX 4 SERIAL CHANNEL COMMAND REGISTER
FFC409 - FFC43F	RESERVED
FFC441	AUX 3 SERIAL CHANNEL DATA (2651)
FFC443	" " " " STATUS
FFC445	" " " " MODE REGISTER
FFC447	" " " " COMMAND REGISTER
FFC449 - FFC47F	RESERVED
FFC481	AUX 2 SERIAL CHANNEL DATA (2651)
FFC483	" " " " STATUS
FFC485	" " " " MODE REGISTER
FFC487	" " " " COMMAND REGISTER
FFC489 - FFC4BF	RESERVED
FFC4C1	AUX 1 SERIAL CHANNEL DATA (2651)
FFC4C3	" " " " STATUS
FFC4C5	" " " " MODE REGISTER
FFC4C7	" " " " COMMAND REGISTER
FFC4C9 - FFC4FF	RESERVED

Winchester drive ports:

FFC501	MISC. CONTROL, PORT A	(8255)
FFC503	MISC. CONTROL, PORT B	
FFC505	MISC. CONTROL, PORT C	
FFC507	MISC. CONTROL, CONTROL REGISTER	
FFC509 - FFC53F	RESERVED	
FFC541	STATUS REGISTER	(8259)
FFC543	COMMAND REGISTER	
FFC545 - FFC57F	RESERVED	
FFC581	MAIN DRIVE CONTROL, PORT A	(8255)
FFC583	MAIN DRIVE CONTROL, PORT B	
FFC585	MAIN DRIVE CONTROL, PORT C	
FFC587	MAIN DRIVE CONTROL, CONTROL REGISTER	
FFC589 - FFC5BF	RESERVED	
FFC5C1	D/A CONVERTER DATA STROBE	
FFC5C3 - FFC5FF	RESERVED	
FFC601	COUNTER #0	(8253)
FFC603	COUNTER #1	
FFC605	COUNTER #2	
FFC607	MODE REGISTER	
FFC609 - FFC63F	RESERVED	
FFC641 - FFC77F	UNUSED	
FFC781	CHARACTER REGISTER	(2653)
FFC783	STATUS REGISTER	
FFC785	MODE REGISTER	
FFC787	BLOCK CHARACTER CHECK REGISTER	
FFC789 - FFC7BF	RESERVED	
FFC7C1	RAM BUFFER PORT ADDRESS	
FFC7C3 - FFC7FF	RESERVED	

II.05 RAM Memory Allocation :

The present single user RAM allocation provides a full 64K byte p-System data area. The code pool is also a maximum possible 64K. System managers allocating memory for a multi-user system should refer to the SAGE™ IV TECHNICAL MANUAL for more information.

The suggested allocation gives room in the Interpreter and BIOS areas for growth without requiring a configuration change. The two word floating point interpreter currently occupies about 9.5K and the four word interpreter currently occupies about 10.5K. A 12K area has been allocated for the interpreter. The current BIOS and buffers occupy between 18K and 32K depending on your system (ie., whether you have a hard disk or not). The BIOS takes as much memory as it needs from the top of the system's equipped memory and sets the top of RAM Disk to below its base. The BIOS size is expected to grow as more features are added.

Experienced users may want to reconfigure the starting location and size of the p-System code pool using the SETUP.CODE program. This should be done carefully as no cross checks are made for mistakes which cause overlap of areas. Note also that the base of RAM Disk may be changed with SAGE4UTIL.CODE if the code pool is reduced below 23400H.

Users with 128K floppy based systems will need to reduce the size of the code pool, and possibly the data space. Also, no Ramdisk can be configured on systems this size.

The starting address of the interpreter is hard coded in the p-System bootstrap file SAGE.PBOOT.TEXT. Also hard coded in this file are the base and size of the p-System data area. These values and locations will generally never need to be modified because a full 64k data area is desirable.

III SAGE PROMS :

The present SAGE PROMS occupy 16K at addresses FE0000-FE3FFF (hex). The PROMS contain the following:

THE SAGE STARTUP TEST
System-wide tests (which are switch-selectable) are performed on power-up or reset. These tests include memory sizing and testing, and PROM checksum verification.

INITIALIZATION AND BOOTSTRAP ROUTINES
The PROMS contain routines to initialize the system and boot from a floppy disk or a hard drive.

I/O SUBROUTINES
A set of low-level I/O subroutines is provided to access the user's keyboard and terminal, the floppy drives, and the Winchester drives.

THE SAGE DEBUGGING TOOL
The PROMS contain a powerful debugging tool which provides a complete environment for debugging machine-level programs. Its operation is explained later in this chapter.

ERROR HANDLING
PROM routines handle all exception errors such as bus errors, address exceptions, etc.

Note: The SAGE can accomodate larger PROMS using strapping changes discussed in the TECHNICAL MANUAL.

III.01 PROM VERSIONS :

A list of PROM versions follows:

VERSION #	DATE	DESCRIPTION
1.0	13-JUN-82	SAGE II (floppy-based SAGE IV) initial release
1.2	20-DEC-82	General update
2.0	18-MAR-83	Update to 16K PROMS (2764's) for Winchester disk drives
2.1	08-AUG-83	General update

SAGE users with service capabilities may purchase new PROMs at the normal spare parts cost. No strapping changes are required to upgrade from the 8K to 16K PROMS.

In general, changes to SAGE software and hardware are documented in the "SERVICE MANUAL" which can be ordered through your dealer.

III.02 PROM START-UP TESTS :

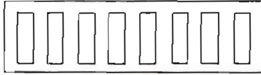
The SAGE performs a number of system-wide tests whenever it is turned on or RESET. This section documents these activities.

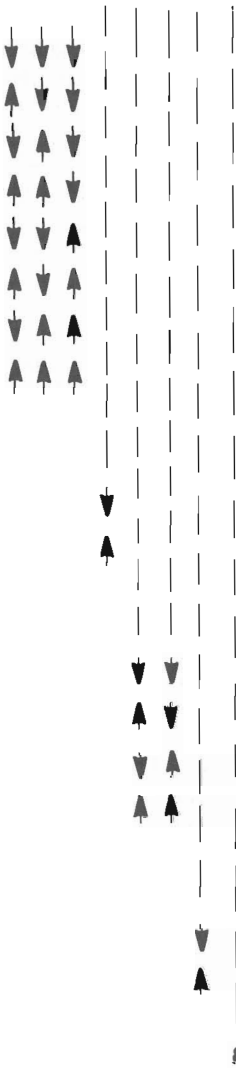
On power-up or when the processor is RESET, the address of the SAGE PROMs changes from FE0000 to 000000. The processor reads the initial stack pointer and initial start vector from PROM locations 0 and 4 respectively. The start vector points to an address where the PROMS normally reside (> FE0000). When this address is executed, the hardware switches the PROMS back to their normal address location. The PROMS remain at their normal location (FE0000 - FE3FFF) until a power-down or RESET.

A processor diagnostic is run on power-up to check the integrity of the CPU. Registers are set and read and a selected instruction set is run. If the test fails, the processor will stop and the CPU light on the front panel will be red.

A PROM test is run next. It calculates a simple checksum on the PROM area to insure that the PROM startup program itself is ok. If an error is detected, the message "PROM 1 Bad" or "PROM 2 Bad" is displayed. Note that if the PROM is bad in a portion of the program needed for the test or printout, the system may fail to respond with any output. PROM 1 refers to the even memory addresses while PROM 2 refers to the odd memory addresses.

Next, the terminal baud rate is determined by reading GROUP-A DIP switches on the rear panel. Communications always uses 8 data bits, 1 stop bit and even parity. (See figure on next page)





TERMINAL BAUD RATE

8 data bits, 1 stop bit and even parity.

19.2 K baud

9600

4800

2400

1200

600

300

reserved, will default to 19.2 K baud

PARITY CONTROL

even parity enabled

disabled

BOOT DEVICE

boot to DEBUGGER

boot to Floppy drive 0.

boot to first partition of Winchester drive 0.

reserved, defaults to DEBUGGER

FLOPPY CONFIGURATION

96 TPI drive

48 TPI drive

8 reserved

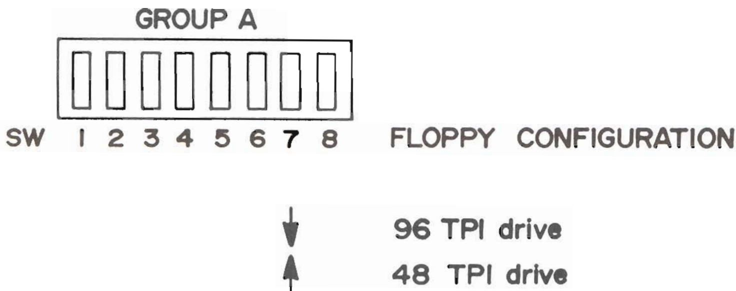
> **Note:** For normal operation, the PROMs require that switch 8 of the Group-A DIP switch be set off (down). Switch 8 is used to cancel the memory test or select various maintenance and debugging options on startup.

On startup, the remote serial channel defaults to 9600 baud with 8 data bits, 1 stop bit, and even parity. Use the PS command under SDT to change the rate for stand-alone applications.

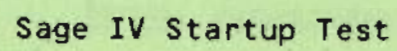
An indication of a USART failure is provided via rapid blinking of the Processor LED. When initially trouble-shooting a "dead" system it is important to know if the processor is communicating with the terminal I/O circuits. If the processor LED blinks rapidly on startup, this indicates that the terminal USART is not responding. The USART transmit flag should go busy when a character is transmitted but should never stick in the busy state.

The processor must be reset to get out of the rapidly blinking LED indication. This check is only present in the PROM resident terminal driver and is not in the BIOS.

The floppy drive option switch is read to determine which drive is installed (always double-density, double-sided format).



After each of these tests have been completed successfully, the display should read:



III.03 RAM MEMORY TEST :

During normal operation, the SAGE Startup Test performs a memory test after the previous tests have executed successfully. Because the SAGE memory test destroys the previous contents of RAM, however, this option may be disabled (see below).

The first 128k of RAM is checked in the following manner:

1) A long word (4 bytes) is set to 00000000 and read back.
2) The long word is set to FFFFFFFF and read back.
3) The long word is set to the value of its own address for later testing, and the test proceeds to the next long word.

When all 128K is done, each long word is read to see if it still contains its address. Then the top word of each 128K bank is read to see if that bank exists. Once the size of the additional memory is determined, it is checked just as the first 128K was.

The memory test takes a few seconds. If no errors are detected, the system displays:

```
RAM SIZE = XXXX
```

If a bad memory location is found, an error message is displayed:

```
BAD memory @ (addr) is xxxxxxxx instead of yyyyyyyy
```

The program stops at the first bad location it finds. Because it re-reads the location to print out the error message, the error value may be the expected value if the RAM is intermittent and reads correctly the second time. The processor will attempt to enter the debugger after a memory error. If the failed memory occurs in the debugger stack area (working down from 400H), the debugger may fail

to operate correctly after the memory error.

DISABLING THE MEMORY TEST

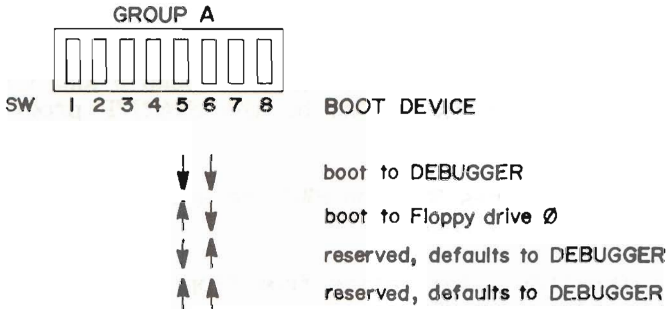
Occasionally while developing assembly code programs it is necessary to look at a post mortem dump of memory after a system lockup. Resetting the SAGE IV normally causes all of memory to be modified during memory testing and causes the default TRAP vectors to be initialized. A method has been provided which will allow entry into the Debugger (SDT) on reset with minimal modification of memory.

To override the normal startup sequence, set Switch 8 of Group A to On, and Switches 5 and 6 of Group A to Off. When you do a RESET now, the terminal will display "SAGE IV Startup Test" followed by "Bypassed Init". The processor immediately enters SDT. The displayed contents of all the registers will be invalid because they are not saved on RESET. SDT uses stack memory from location 400H downward, so a few locations in that area will be modified. A flag at location 104H is cleared so that SDT will not attempt to use the BIOS.

Do not try to use any commands other than Display Memory (DM) before re-initializing the system. Initializing the system may be accomplished with the IS command. Remember to set Switch 8 to Off and Switches 5 and 6 back to your desired bootstrap selection.

IV DISK BOOTSTRAP :

The SAGE STARTUP test ends by reading the bootstrap switches from the GROUP-A DIP switch to determine what device/program to boot to:



The "boot" process requires that the device have on it a small program at most 2 blocks long. The STARTUP program loads and runs this "bootstrap" program which loads and runs an operating system (such as the p-SYSTEM.) The boot program is usually specific to the operating system being loaded. However, it must conform to the following SAGE protocol.

SAGE IV™ bootstrap programs must have the first four bytes of the code (at 400H) set to the ASCII characters 'BOOT'. In Hexadecimal, these bytes are = 42 4F 4F 54 . This data is checked by the STARTUP loader. If it is not present the system displays:

```
Not BOOT Disk
```

If this " header" data is correct the boot program will then be started at location 404H. The bootstrap is entered in Supervisor mode. Details of the bootstrap programs for the p-System follow. For operating systems other than the p-System, refer to the literature you received with your operating system.

IV.01 p-SYSTEM FLOPPY BOOT :

The floppy bootstrap program is located on blocks 0 and 1 of the floppy diskette. It is loaded by the STARTUP program from one of two ways.

1. If the switches SW5 and SW6 are set to boot to the floppy on reset.

2. If the 'IFx', initialize from floppy command, is typed from the SDT.

Either way will cause the first two blocks of the diskette to be read into RAM at location 400H.

If a timeout occurs while trying to access the floppy, the program assumes that there is no diskette there and displays:

```
Put in BOOT disk and press a key (Q -quits)
```

Typing a "Q" will display

```
Boot aborted on drive 0
```

and control will go to the SDT. Typing any other key will cause a re-try to boot from the device.

If a disk error occurs, one of the following error messages will be given:

```
Drive error (code) on drive (0 or 1)
```

where codes are:

- 01 -controller failure
- 02 -invalid command
- 03 -recalibrate or seek failure
- 04 -timeout
- 05 -missing address mark
- 06 -no data found
- 07 -overrun
- 08 -CRC error
- 09 -end-of-cylinder
- 0A -unknown
- 0B -address out-of-range

When the bootstrap is executed, the stack contains a return address which may be used (in case of boot failure) to return to the debugger. Below the return address on the stack (at-4(A7)) is a word containing the drive number: 0 for the left drive or 1 for the right drive.

Then the BIOS program is called. (The last part of this section describes that process.)

The source file of the single-user p-SYSTEM floppy bootstrap is called SAGE.PBOOT.TEXT. The file is assembled normally but not Linked or Compressed. The resulting file SAGE.PBOOT.CODE is installed on a diskette using the Bootstrap Copy facility of the SAGE4UTIL program.

> **Note:** The standard p-System utility BOOTER.CODE should not be used for installing the bootstrap (unless the extra steps of Linking and Compressing are performed). Note that in Version IV code files there is one block plus a 26 byte header ahead of the actual code in an unCOMPRESSED code file.

The routines TERMTEXT, TERMCRLF, and FDREAD in the PROM Debugger are used by the Bootstrap program for terminal and floppy I/O. Note that these routines are accessed via a macro which generates the necessary long absolute addresses.

FLOPPY FORMAT

The default floppy driver is set for 8 sectors/track, 512 bytes/sector. There is no track-to-track skew and no interleaving.

The system can be used to determine the maximum sector on the diskette and allows booting to 8, 9, and 10 sector/track double-sided diskettes (SAGE Format). We do not support booting to a Network Consulting 10 sector diskette which has a different sector numbering scheme and a different sector numbering scheme and a different track layout. Also IBM diskettes cannot be booted because of their different track layout.

SAGE4UTIL may be used to set up a bootable 10 sector format by selecting the SAGE 10 sector format menu. Although this option is available, it is not guaranteed by SAGE to work on all systems and is not recommended as a distribution format.

IV.02 p-SYSTEM WINCHESTER BOOT :

The p-SYSTEM Winchester bootstrap is much like the floppy bootstrap. The bootstrap program is located on blocks 0 and 1 of the Winchester partition. It is loaded by the STARTUP program from one of two ways.

1. If the switches SW5 and SW6 are set to boot to the first Winchester drive partition #1 on reset.

2. If the 'IHx #n', initialize from partition command is typed from the SDT.

Either way will cause the first two blocks of the partition to be read into RAM at location 400H.

If a disk error occurs, one of the following error messages will be given:

```
Drive error (code) on drive (0 or 1)
```

where codes are:

```
  01   -could not initialize VCO
  03   -recalibrate/seek failure
  04   -drive not ready
  08   -CRC error
  0B   -address out of range
  0C   -wrong cylinder
  0E   -bad device number
```

Then the BIOS is called. (The last part of this section describes that process.)

The source file of the p-SYSTEM Winchester bootstrap is called SAGE.WBOOT.TEXT. The file is assembled normally but not Linked or Compressed. The resulting file SAGE.WBOOT.CODE is installed on a Winchester partition using the Bootstrap Copy facility of the SAGE4UTIL program.

> **Note:** The standard p-System utility BOOTER.CODE should not be used for installing the bootstrap (unless the extra steps of Linking and Compressing are performed). Note that in Version IV code files there is one block plus a 26 byte header ahead of the actual code in an unCOMPRESSED code file.

The routines TERMTEXT, TERMCRLF, and WDREAD in the PROM Debugger are used by the Bootstrap program for terminal and Winchester I/O. Note that these routines are accessed via a macro which generates the necessary long absolute addresses. The PROM routine will have already selected the drive, so a call to WSELECT is not required.

IV.03 LOADING THE p-SYSTEM BIOS :

The p-System Bootstrap first reads in the 4 block p-System directory from block 2 of the floppy or Winchester partition. Then the file SYSTEM.BIOS is found and the first block of the file is read. The first four bytes of the BIOS code are checked for the four ASCII characters 'BIOS'. If the proper BIOS data is found the complete SYSTEM.BIOS file

is read in at the top of all RAM memory. Otherwise, the message 'Not BIOS code in SYSTEM.BIOS' is output and the bootstrap returns to SDT.

Here are some items of interest contained in the SYSTEM.BIOS:

Offset 4 from the start of the code in the SYSTEM.BIOS file is the size of the BIOS code.

Offset 6 is the size of the RAM buffer area which is allocated preceding the code. This size is a worst case value for a Winchester based system. Floppy only based systems will determine dynamically at BIOS initialization that they do not need the extra space for the Winchester and extra serial channel buffers.

Offset 8 is the offset of the BIOS Initialization routine address from the beginning of the code.

Also the RAM Disk boot flag and base address are taken from the configuration area in the BIOS file.

The BIOS Initialization routine is executed which sets up all the hardware and drivers and turns on interrupts. The Debugger is set up to use the BIOS terminal driver.

Once the BIOS is initialized, the bootstrap uses the BIOS I/O calls for the remaining disk information. The BIOS channel map is scanned for the Winchester device and subdevice number to find the logical channel used for booting.

The file SYSTEM.INTERP is read into its position in memory above the p-System data area and below the p-System code pool area. If the RAM Disk boot flag is set, files from the disk are copied to the RAM Disk area. The new directory on the RAM Disk is RAMDISK. A file called ENDBOOT on the

source device will terminate the copy process.

Finally the processor is put into User mode and several arguments are put on the User stack for initialization of the p-System Interpreter. The routine then transfers to the beginning of the Interpreter.

V PROM ENTRY POINTS :

The SAGE PROMS contain a set of polled I/O routines to access the terminal, the floppy disk drives, and the Winchester disk drives. Following is a list of fixed entry points which allow bootstraps and other stand-alone (without BIOS) routines access to I/O facilities.

Routines in PROM may be called using MACRO assembly procedures (see the SOFTECH ASSEMBLER MANUAL) to create the long address given. The macro is necessary because the assembler only generates addresses with the short direct addressing mode. The listed offset for each routine should be used as the macro argument. The offset + FE0000H is the actual address of the routine.

The routines must be called in the 68000 SUPERVISOR mode, not user mode.

```
THE LONG JSR MACRO:

        .MACRO  LJSR
        .WORD   4EB9H
        .WORD   00FEH
        .WORD   %1
        .ENDM

THE LONG JMP MACRO:

        .MACRO  LJMP
        .WORD   4EF9H
        .WORD   00FEH
        .WORD   %1
        .ENDM
```

1. KEYBCH - Get a Keyboard Character. Offset=8H

This routine waits for and returns a character from the terminal port. Bit 7 is always cleared and lower case alphabetic characters are converted to upper case. The character is returned as a byte in register D0.

```
On entry:  Use LONG JSR MACRO:  LJSR KEYBCH
On exit:   D0= typed character
           Registers used: D0
```

2. KEYCHK - Check for a Keyboard Character. Offset=0CH

This routine tests the terminal USART to determine if a character is available for input.

```
On entry:  Use LONG JSR MACRO:    LJSR KEYCHK
On exit:   condition code NE if char is available
           "             " EQ "      "   not available
           Registers used: none
```

3. TERMCHAR - Output a Character to Terminal. Offset=14H

This routine outputs the character from the low byte of D0 to the terminal port.

```
On entry:  Use LONG JSR MACRO:    LJSR    TERMCHAR
           D0= character for output
On exit:   Registers used: none
```

4. TERMTEXT - Output a Text String. Offset=18H

This routine outputs a string of characters to the terminal. Register A0 is the address of the base of the string and the string must be terminated with a zero byte.

```
On entry:  Use LONG JSR MACRO:    LJSR    TERMTEXT
           A0 = pointer to first character of string
On exit:   Registers used: A0
           (A0 points to byte beyond zero terminator)
```

5. TERMCRLF - Print a Carriage Return/Line Feed. Offset=1CH

This routine outputs a carriage return and line feed to the terminal. Also five nulls are output after the characters for terminals which need extra time after a vertical positioning change.

```
On entry:  Use LONG JSR MACRO:    LJSR    TERMCRLF
On exit:   Registers used: none
```

6. TERMHEXB - Output a Hexadecimal Byte. Offset=20H

This routine outputs a two-character hexadecimal value contained in the low byte of register D0.

```
On entry:  Use LONG JSR MACRO:    LJSR TERMHEXB
           D0 = byte value to be output
On exit:   Registers used: none
```

7. TERMHEXW - Output a Hexadecimal Word. Offset=24H

This routine outputs a four-character hexadecimal value contained in the low word of register D0.

```
On entry:  Use LONG JSR MACRO:    LJSR TERMHEXW
On exit:   Registers used: none
```

8. FDREAD - Floppy Disk Read. Offset=28H

This routine reads data from a floppy diskette and stores it in memory. The parameters defining the read are passed on the stack. The typical calling sequence is:

```
MOVE.W    BLOCKNUM,-(SP)  ;Logical block no. (2 bytes)
MOVE.L    MEMADDR, -(SP)  ;Memory buffer addr (4 bytes)
MOVE.L    NUMBYTES,-(SP)  ;Number of bytes (4 bytes)
MOVE.W    DRIVENUM,-(SP)  ;Drive no. 0 or 1 (2 bytes)
LJSR      FDREAD
```

```
On entry:
     Use LONG JSR MACRO: LJSR FDREAD
     Stack (from top - last in)

     4 byte return address (stored by LJSR)
     2 byte drive number (0=left drive, 1=right drive)
     4 byte size in bytes
     4 byte memory address
     2 byte logical block # (each block = 512 bytes)

On exit:
     D0 = error type (0 = no error)
     condition code NE if transfer failed.
     Registers used: D0,D1,D2,D3,D4,D5,D7,A1,A4
```

9. FDWRITE - Floppy Disk write. Offset=2CH

This routine writes data from memory to a floppy diskette. The parameters defining the write are passed on the stack (see example in Floppy Disk Read above).

```
On entry:
     Use LONG JSR MACRO:    LJSR FDWRITE
     Stack (from top - last in)

     4 byte return address (stored by LJSR)
     2 byte drive number (0=left drive, 1=right drive)
     4 byte size in bytes
     4 byte memory address
     2 byte logical block # (each block = 512 bytes)

On exit:
     D0 = error type (0 = no error)
     condition code NE if transfer
     Registers used: D0,D1,D2,D3,D4,D5,D7,A1,A4
```

10. BOOTSX - Floppy disk boot. Offset=38H

This routine boots from the floppy disk in the drive specified by the word on the top of the stack. The typical calling sequence is:

```
MOVE.W    DRIVENUM,-(SP)  ;Drive number 0 or 1
LJMP      BOOTSX

On entry:
     Use LONG JMP MACRO:    LJMP  BOOTSX
     Stack (from top)

     2 byte drive number (0=left drive, 1=right drive)

On exit:  Never returns!
```

11. WSELECT - Winchester Select. Offset=40H

This routine selects the Winchester drive and partition that will be accessed. NO registers are preserved.

```
     On entry:  Use LONG JSR MACRO:    LJSR    WSELECT
                Stack (from top - last in)

                4 byte return address
                4 byte partition number (0-15)
                  or pointer to a name (> 16)
                2 byte drive number (0-3)
```

Note that the 4 bytes for the partition can be interpreted two different ways. If the long word value is less than 16 then it is assumed to be the partition number. If greater than 16, the long word is interpreted as an address which points to the partition name. The name must be 8 bytes long with zeros filling any unused bytes.

12. RDCHAN9 - Read Winchester channel 9. Offset=03CH

This routine reads the Winchester partition selected. Note that the partition must have been selected by the WSELECT call.

```
     On entry:
          Use LONG JSR MACRO:    LJSR    RDCHAN9
          Stack (from top - last in)

          4 byte return address
          4 byte length of transfer (in bytes)
          4 byte starting memory address
          4 byte logical block number

     On exit:
          Registers used: D0,D1,D2,A0,A1
```

13. DEBUG - Debugger Entry Point. Offset=30H

This is a non-returning entry point to the PROM Debugger (SDT) for use when terminating a user environment or a failure during a bootstrap.

On entry: Use LONG JUMP MACRO: LJMP to Debug
On exit: Never returns!

VI SAGE DEBUGGING TOOL :

SDT is a powerful tool for analyzing program operation. SDT allows you to display and modify memory and registers, disassemble instructions in memory, trace portions of a program, set breakpoints, and boot to a floppy or hard disk drive.

SDT can be entered several ways:

1) On power-up or reset (Group A switches 5 and 6 off).

2) On all EXCEPTION errors that have not been re-defined by the user.

3) Via a breakpoint (TRAP #15.)

4) Via jump vector in PROM (see page 36.)

5) Via BIOS call TRAP #14., Function=0 (see the TECHNICAL MANUAL).

SDT can use either the PROM polled I/O routines or BIOS routines for input/output. In some circumstances (especially after certain EXCEPTION errors which revert to PROM I/O), conflicts can arise if both types of drivers are in use on the same device at the same time. To resolve such conflicts, use TRAP #14., Function=3 or 4 (see the TECHNICAL MANUAL) to install the proper I/O routines before entering the debugger.

VI.01 SDT PHILOSOPHY :

Most SDT commands consist of two characters followed by optional arguments. SDT prompts the user for a command with a ">". All arguments are assumed to be hexadecimal unless preceded by a "/" to indicate a decimal value.

```
EXAMPLE:

>DM 1000,#A              (Display 10 bytes of memory starting at 1000H)
00001000: 0011 2233 4455 6677 8899  ..

>DM 1000,#/10            (Display 10 bytes of memory starting at 1000H)
00001000: 0011 2233 4455 6677 8899  ..
```

SDT skips over commas and spaces between arguments. No space is required between an SDT command and the first argument ("DM1000" and "DM 1000" are equivalent), but a register specification is interpreted as part of an SDT command and may not be separated by a space:

```
>DD5                     (Display register D5)
>DD 5                    (Illegal syntax)
```

VI.02 SDT REGISTER USAGE :

SDT provides base registers to simplify data entry and address arithmetic. Let's suppose that you have a source listing of a relocatable program. The listing address begins at 0000H, but the beginning of the program in memory might be 5700H. Normally, you would add 5700H to each address in your listing to find the equivalent address in memory. However, if you set an SDT base register to 5700H, SDT will perform the addition for you.

EXAMPLE:
We wish to disassemble the instruction at listing address 1F7EH. Our program starts at 5700H.

```
>S$1 5700                                  (Set base register $1 to 5700H)
>AD $1+1F7E,#1                             (Disassemble one instruction)
0000767E 00001F7E: MULU     (A5)+,D6       (1st address: absolute)
                                           (2nd address: relative)
```

If we wish every address we enter to be added to the same offset, we can define a "standard base register."

```
>S$1 5700                              (Set base register $1 to 5700H)
>$1                                    (Set $1 to standard base register)
$1>AD 1F7E,#1                          (Disassemble one instruction)
0000767E 00001F7E: MULU    (A5)+,D6
```

The SDT prompt now appears "$1>" as a reminder that every address we enter will be added to base register one.

VI.03 SDT QUICK DESCRIPTION :

Detailed descriptions of commands follow in the next section.

Base registers

$0	Absolute base register (always equals zero)
$1	User base register 1
$2	User base register 2

Breakpoint registers

0	User breakpoint register 0
1	User breakpoint register 1

Argument format

Any argument can be specified in decimal if preceded by "/".

addr	An address specified by up to 8 hex digits
[$x+]**addr**	An address added to a base reg. (optional)
'ssss'	Data interpreted as an ASCII string
byte	Data specified by up to 2 hex digits
word	Data specified by up to 4 hex digits
long	Data specified by up to 8 hex digits
#n	A count used to display **n** bytes, disassemble **n** instructions, etc.

Command summary
Arguments enclosed in brackets are optional.

Page

>$x............................ 39
Set standard base reg. x

>$............................. 39
Clear standard base reg.

>AD............................ 64
Disassemble 20 instructions
from current display loc.

>AD [$x+]addr.................. 64
Disassemble 20 instructions
starting at **addr**

>AD [$x+]addr1,[$x+]addr2....... 64
Disassemble instructions
from **addr1** through **addr2**

>AD [$x+]addr,#n................ 64
Disassemble **n** instructions
starting at **addr**

>AR long1,long2................. 52
Arithmetic computation

>DA[x]......................... 53
Display A registers or Ax

>DB[x]......................... 67
Display breakpoint regs.
or breakpoint register x

>DD[x]......................... 53
Display D registers or Dx

>DM.............................. 57
Display 256 bytes of memory
from current display location

>DM [$x+]addr................... 57
Display 256 bytes of memory
starting at **addr**

>DM [$x+]addr1,[$x+]addr2....... 57
Display memory from
addr1 through **addr2**

>DM [$x+]addr,#n................ 57
Display **n** bytes of
memory starting at **addr**

>DP.............................. 54
Display program counter

>DR.............................. 53
Display all registers

>DS.............................. 54
Display status register

>DT[x].......................... 72
Display current trace mode
for all traps or trap x

>DU.............................. 54
Display user stack pointer

>D$[x].......................... 50
Display base regs. or $x

>ER[x].......................... 75
Exercise floppy read

>EW[x].......................... 75
Exercise floppy write

>FB [$x+]addr1,[$x+]addr2, byte. 59
Fill memory **addr1** through **addr2** with **byte**

>FB [$x+]addr, #n, byte......... 59
Fill memory with **n** bytes of **byte** starting at **addr**

>FL [$x+]addr1,[$x+]addr2, long. 60
Fill memory **addr1** through **addr2** with data **long**

>FL [$x+]addr, #n, long......... 60
Fill memory with **n** long words of **long** starting at **addr**

>FW [$x+]addr1,[$x+]addr2, word. 59
Fill memory **addr1** through **addr2** with data **word**

>FW [$x+]addr, #n, word......... 59
Fill memory with **n** words of **word** starting at **addr**

>GC [[$x+]addr]................. 68
Execute program at PC or **addr** if specified

>GO [[$x+]addr]................. 68
Execute program, resetting breakpoint counts

>GS [[$x+]addr]................. 73
Execute subroutine call at PC or **addr**

>IF[x]........................ 47
Boot from floppy drive 0 or x (0=left drive, 1=right)

```
>IFR[x]..........................  48
    Boot from floppy drive 0 or x
    without loading RAMDISK

>IH[x] [#n,name]................  48
    Boot from hard disk 0 or x

>IHR[x] [#n,name]...............  49
    Boot from hard disk without
    loading RAMDISK

>IS.............................  47
    Initialize System

>LA.............................  80
    Load from a remote device
    (Motorola object code format)

>LF[x] block,[$x+]addr,count....  74
    Load count bytes into
    addr from block # block
    of floppy drive 0 or x

>LT.............................  80
    Load from the terminal
    (in Motorola object code format)

>M [$x+]addr1,[$x+]addr2,
                  [$x+]addr3......  60
    Move data from addr1
    through addr2 to addr3

>M [$x+]addr1,#n,[$x+]addr2.....  60
    Move n bytes from addr1
    to addr2

>POB [$x+]addr,byte.............  76
    Output data byte to port
```

>POW [$x+]addr,word............. 76
Output data **word** to port

>PIB [$x+]addr.................. 76
Input data byte from port

>PIW [$x+]addr.................. 77
Input data word from port

>PS x........................... 78
Set remote baud rate

>SA[x] [long]................... 56
Modify A registers or Ax

>SB[x] [[$x+]addr],[passcount]... 67
Set breakpoint regs or Bx

>SD[x] [long]................... 56
Modify D registers or Dx

>SM [$x+]addr................... 58
Modify memory

>SP [long]...................... 56
Modify Program counter

>SR............................. 56
Modify all registers

>SS [long]...................... 56
Modify Status Register

>ST[x] [T,N].................... 72
Set Traps for Tracing

>SU [long]...................... 56
Modify User Stack pointer

>S$[x] [long]................... 50
Modify base regs. or $x

```
>TB [[$x+]addr]................. 70
    Trace without reg. print
    starting at PC or addr

>TE............................. 71
    Terminate trace mode

>TN[x].......................... 71
    Trace next x instructions

>TNI[x]......................... 71
    Trace next x instructions,
    interruptible

>TR [[$x+]addr]................. 70
    Begin Trace Mode with register
    display

>WF[x] block,[$x+]addr,bytecount 74
    Write count bytes from
    addr to block # block
    of floppy drive 0 or x

>XB[$x+]addr1,[$x+]addr2,          61
           byte,[maskbyte]......
    Search memory addr1 through
    addr2 for byte after
    masking with maskbyte

>XW[$x+]addr1,[$x+]addr2,
           word,[maskword]......   61
    Search memory addr1 through
    addr2 for word after
    masking with maskword

>XL[$x+]addr1,[$x+]addr2,
           long,[masklong]......   62
    Search memory addr1 through
    addr2 for long after
    masking with masklong
```

>XM[$x+]paddr1,[$x+]paddr2, 62
[$x+]addr1,[$x+]addr2....
Search memory **addr1** through **addr2** for pattern in memory **paddr1** through **paddr2**

T>.............................. 69
<CR> Trace next instruction (active only when SDT is in Trace Mode)

VI.04 SDT DETAILED DESCRIPTION :

Initialization and boot commands

>IS Initialize system

IS disables interrupts, clears and retests memory, and resets all SDT registers and breakpoints. **IS** performs exactly as a system reset would.

```
>IS  System reinitializing....
SAGE IV Startup Test [2.1]

RAM Size = 1024K

>
```

>IF[x] Boot from floppy disk

IFx boots from floppy drive x (x=0 for left drive, 1 for right drive, no x defaults to left drive).

SDT reads 1K of data from logical blocks 0 and 1 of the diskette into memory starting at location 400H. It then checks that the first four bytes are the ASCII characters 'BOOT' (42H, 4FH, 4FH, and 54H) to verify that the diskette has a bootstrap program installed. The bootstrap routine is then called at location 404H with the booting drive number (0 or 1) previously placed on the stack. If the bootstrap routine returns to the calling program, control reverts to SDT.

```
>IF                               (Boot from floppy drive 0)
Booting from Floppy

UCSD p-System IV.1 Bootstrap      (or whichever operating system you use)
Copying to RAM Disk               (Copy system files to RAM Disk)
                .
                .
                .
```

>IFR[x] Boot from floppy disk, preserving RAMDISK

IFR functions similar to the **IF** command, except RAMDISK is not loaded during the boot process. **IFR** is usually used to recover information in RAMDISK after the user accidentally exits his environment.

The SYSTEM.BIOS and SYSTEM.INTERP are still required to be taken from the floppy diskette (and therefore do not need to be resident in RAMDISK). Once these files are loaded, the system will boot to RAMDISK (if it was originally configured to do so) without first copying the files. This means that SYSTEM.PASCAL and SYSTEM.MISCINFO must still be present in RAMDISK. Note that if the user's exit was caused by a program crash, RAMDISK may no longer contain valid files and the boot may fail.

>IH[x] [#n,NAME] Boot from hard disk

The **IH** command allows the user to boot to a system from a hard drive. All partitions (except the special case #0) are capable of being booted to. A bootable partition must have the **SYSTEM files necessary to run that operating system.** IH has the following formats:

IH Boots to the default partition, drive 0, partition 1. This is the same partition that will boot on power-up or RESET if SW6 and SW5 (port A) are set up and down respectively.

IHx	Where x is drive number 0 through 3. This boots to partition 1 on the drive specified.
IHx #n	Where x is the drive number 0 through 3 and n is the partition number 1-9,A,B,C,D,E,F where A through F represent partitions 10 through 15 respectively.
IH NAME	Defaults to drive 0, where NAME is the partition name.
IHx NAME	Where x is the drive number (0-3) and NAME is the name of the partition.
IHRx #n	The RAMDISK boot command IHR may be used once the system has already been booted to a partition. It is generally used to recover information in RAMDISK when the user accidentally left the partition as IHR does not clear RAMDISK (the IH commands do). The SYSTEM.BIOS and SYSTEM.INTERP are still required to be taken from the partition (and therefore do not need to be resident in RAMDISK). Once these files are loaded, the system will boot to RAMDISK (if it was originally configured to do so) without first copying the files. This means that SYSTEM.PASCAL and SYSTEM.MISCINFO must still be present in RAMDISK. Note that if the partition was left due to a program crash, RAMDISK may no longer contain valid files and the boot may fail.

>**IHR[x] [#n]** Boot from hard disk, preserving RAMDISK

See IH command.

Base register commands

>D$[x] Display base register

D$x displays the contents of base register x (x=0,1,2). If no x is specified, the contents of all three base registers are displayed.

```
>D$                                  (Display all base registers)
$0: 00000000 00005700 00200000
>D$2                                 (Display base register $2)
$2: 00200000
```

>S$[x] [long] Set contents of base register

S$x long sets the contents of base register x (x=1,2) to the value of **long** . If **long** is not specified, SDT displays the current value of the base register and prompts the user for a new value. Type a <cr> to leave the value unchanged at this point. **S$** (no arguments) prompts the user for values for both base registers $1 and $2. **Note:** Base register $0 is a special absolute register. It always contains 0 and may not be modified. See discussion under **$** command.

```
>S$1 FF                              (Set base register $1 to 000000FF)
>D$                                  (Display all base registers)
$0: 00000000 000000FF 00200000
>S$2                                 (Set base register $2)
$2: 00200000: 5555                   (Set to 5555)
>S$                                  (Set both base registers)
$1: 000000FF: 100                    (Set $1 to 100)
$2: 00005555:                        (Type <cr> to leave value unchanged)
>
```

>$[x]   Set standard base register

$x sets base register x (x=0,1,2) as the standard base register. The standard base register is added to any address input which does not have a base register specified. SDT commands display a physical address and an offset from the standard base register if one is active. If no x is specified after the **$** command, the standard base register is disabled. To keep the standard register from being added to an address, type $0+**addr** to specify an absolute address ($0 is permanently defined as zero).

The current standard base register is displayed with the command line prompt as follows:

```
    $1>          ($1 is the current standard base register)
    $2>          ($2 is the current standard base register)
    >            (no standard base register)


>D$                              (Display all base registers)
$0: 00000000 00000100 00005555
>$1                              (Set $1 as standard base register)
$1>AD 352,#1                     (Disassemble 1 instruction at 352+100)
00000452 00000352: ADDI.B  #0E,D3
$1>AD $0+452,#1                  (Disassemble 1 instruction at absolute)
00000452: ADDI.B  #0E,D3         (location 452)
$1>DM $2+1,#2                    (Display 2 bytes at 5555+1)
00005556 00000001: 7468  th
$1>$2                            (Set $2 as standard base register)
$2>DM 1,#2                       (Display 2 bytes at 5555+1)
00005556 00000001: 7468  th
$2>$                             (Disable standard base register)
>
```

>AR long1,long2 Arithmetic computation

AR computes 5 arithmetic results from the two arguments **long1** and **long2:**

(long1+long2) (long1-long2) (long1*long2)
(long1/long2) (remainder long1/long2)

where (long1*long2) is a 16 bit by 16 bit multiply,
(long1/long2) is a 32 bit by 16 bit divide.

```
>AR 4C00,F7FE
+: 000143FE  -: FFFF5402  *: 499F6800  /: 0000  rem: 4C00
```

Note that **AR** can be used to convert a decimal value to hex by typing a zero for one of the arguments:

```
>AR /9652,0                                (Display hex value of 9652)
+: 000025B4  -: 000025B4  *: 00000000     (Division by zero not computed)
```

Displaying 68000 registers

>DR Display all 68000 registers

This command displays the contents of all 8 address registers, all 8 data registers, the Program Counter, the User Stack, and the Status Register (see **DS** command for further information on the SR display).

```
>DR
A0: 00011B56 00000D70 00001238 00010400  00012B6C 000104C8 00000400 0007D782
D0: 00000000 000000E4 00000000 00010000  00010001 00000002 00000000 00000000
PC: 0007E8AE  US: 0000EDDE  SR: 2000 ( S      )
```

>DA[x] Display 68000 address register

DAx displays the contents of address register Ax (x=0-7). If no x is specified, all 8 address registers are displayed.

```
>DA5                              (Display address register A5)
A5: 000104C8
>DA                               (Display all 8 address registers)
A0: 000011B56 00000D70 00001238 00010400  00012B6C 000104C8 00000400 0007D782
```

>DD[x] Display 68000 data register

DDx displays the contents of data register Dx (x=0-7). If no x is specified, all 8 data registers are displayed.

```
>DD3                              (Display data register D3)
D3: 00010000
>DD                               (Display all 8 data registers)
D0: 00000000 000000E4 00000000 00010000  00010001 00000002 00000000 00000000
```

>DP Display 68000 Program Counter

DP displays the current value of the Program Counter along with an offset from the standard base register if one is active.

```
$1>DP                          ($1 = 100)
PC: 0007E8AE ($1: 0007E7AE)
```

>DU Display User Stack pointer

DU displays the current value of the User Stack pointer.

```
>DU
US: 0000EDDE
```

>DS Display 68000 Status Register

DS displays the current value of the 68000 Status Register along with a mnemonic aid to help the user determine which flags are set. Within the parentheses that follow the hexadecimal value of the SR, the presence of the following letters indicate that the corresponding flag is set:

- T - Trace mode
- S - Supervisor mode
- X - Extend bit
- N - Negative flag
- Z - Zero flag
- V - Overflow flag
- C - Carry bit

```
>DS
SR: 2000 ( S     )          (Supervisor mode set; all other flags clear)
```

Modifying 68000 registers

A basic format has been established for all substitute commands. If you wish to modify a single register, you may do so by specifying the register and the new value in a single line:

```
>SA5 6200                    (Set register A5 to 6200)
```

Values may be hexadecimal long values (up to 8 hex digits), decimal values (preceded by "/"), or ASCII strings (up to 4 characters delimited by single quotes). If you do not specify a value in the command line, the current value of the register will be displayed, and you will be prompted for a new value. At this point, you may enter a new value or simply type <cr> to leave the contents unchanged:

```
>SA5
A5: 000104C8: 6200           (Set register A5 to 6200)
```

If you do not specify a register number, you will be prompted for values for each register of the type you indicated:

```
>SA
A0: 00011B56: 250            (Set A0 to 250)
A1: 00000D70:                (<cr> to leave A1 unchanged)
A2: 00001238:                (<cr> to leave A2 unchanged)
A3: 00010400:                (<cr> to leave A3 unchanged)
A4: 00012B6C:                (<cr> to leave A4 unchanged)
A5: 000104C8: 6200           (Set A5 to 6200)
A6: 00000400:                (<cr> to leave A6 unchanged)
A7: 0007D782:                (<cr> to leave A7 unchanged)
```

>SR Modify all 68000 registers

SR prompts the user for a value for each 68000 register. Enter a new value or <cr> to leave the contents unchanged:

```
>SR
A0: 00011B56: 250             (Set A0 to 250)
A1: 00000D70:                 (<cr> to leave A1 unchanged)
A2: 00001238:                 (<cr> to leave A2 unchanged)
A3: 00010400:                 (<cr> to leave A3 unchanged)
A4: 00012B6C:                 (<cr> to leave A4 unchanged)
A5: 000104C8: 6200            (Set A5 to 6200)
A6: 00000400:                 (<cr> to leave A6 unchanged)
A7: 0007D782:                 (<cr> to leave A7 unchanged)
D0: 00000000: 1234            (Set D0 to 00001234)
D1: 000000E4: 'A'             (Set D1 to 00000041)
D2: 00000000: 'BC'            (Set D2 to 00004243)
D3: 00010000: 'DEF'           (Set D3 to 00444546)
D4: 00010001: 'GHIJ'          (Set D4 to 4748494A)
D5: 00000002: /10             (Set D5 to 0000000A)
D6: 00000000:                 (<cr> to leave D6 unchanged)
D7: 00000000:                 (<cr> to leave D7 unchanged)
PC: 0007E8AE:                 (<cr> to leave PC unchanged)
US: 0000EDDE:                 (<cr> to leave US unchanged)
SR: 2000 ( S      ):          (<cr> to leave SR unchanged)
>
```

See previous page for a description of the format of the following commands.

>SA[x] [long] Set 68000 address register

>SD[x] [long] Set 68000 data register

>SP [long] Set 68000 Program Counter

>SU [long] Set User Stack pointer

>SS [long] Set 68000 Status Register

(See **DS** command for information on SR display.)

Memory commands

>DM Display memory

DM has several different forms. If no arguments are given, 256 bytes are displayed following the last displayed location or the last trace or break encountered. Successive **DM** commands can be used to step through memory.

```
Unexpected break point: 0007E8AE: 4A38 TST.B 0258
>DM                        (DM displays memory following break)
0007E8AE: 4A38 0258 67CA 4200 60EA 48E7 4040 5868  J8.XgJB.`jHg@@Xh
0007E8BE: 0008 5828 0004 6708 61AC 12C0 5381 66F8  ..X(..g.a,.@S.fx
    .           .                                    .
    .           .                                    .
    .           .                                    .
0007E99E: 0000 0249 4238 0258 0838 0002 C073 6608  ...IB8.X.8..@sf.
line count error, count= 9
```

The rather odd-looking output at the right of the display is an ASCII interpretation of the data on the left. The ASCII interpretation isn't particularly meaningful in this example, but it is very helpful when locating or decoding strings of ASCII text in memory. Note that non-printing characters are output as periods.

If one address is specified in the **DM** command, 256 bytes are displayed starting at that address:

```
>DM $1+8000                    ($1 = 100)
00008100 00008000: 2074 6869 7320 7472 6170 206F 6363 7572 this trap occur
00008110 00008010: 7320 616E 6420 636F 6E74 726F 6C0D 7265 s and control.r
   |        |           .                                        |
   |        |           .                                        |
   |        `-  address relative to base register              ASCII
   `-  actual memory address
```

If two addresses are specified, all memory between the two addresses is displayed. For example, display memory between 8100 and 810A.

```
>DM 8100,810A
00008100 2074 6869 7320 7472 6170 20  this trap
```

It is also possible to specify a starting address and a byte count:

```
>DM 8100,#/11              (Display 11 (decimal) bytes starting at 8100)
00008100 2074 6869 7320 7472 6170 20  this trap
```

>SM [$x+]addr Set memory at address **addr**

SM allows the user to enter data directly into the computer's memory. The format is similar to the format used to modify registers. The contents of a word (possibly at an odd address!) are displayed in hexadecimal and ASCII. The user is then prompted for a new value. Possible responses are:

<cr>	leave contents unchanged, advance to next memory address
'xx'	load one or two ASCII characters into the word (note that SDT does not allow lower case characters or a single quote to be loaded in this manner)
/n	load n as a decimal value
word	load a word (up to 4 hex digits)

```
>SM 8101
00008101: 7368  th:           (<cr> to leave unchanged)
00008103: 6973  is: 'A'       (Set 8103 to 0041)
00008105: 2074   t: 'BC'      (Set 8105 to 4243)
00008107: 7261  ra: /10       (Set 8107 to 000A)
00008109: 7020  p : 5678      (Set 8109 to 5678)
0000810B: 6F63  oc: .         (Period to terminate SM )
>DM 8101,810C
00008101: 7468 0041 4243 000A 5678 6F63  th.ABC..Vxoc
```

>FB [$x+]addr1, [$x+]addr2, byte

Fill memory with **byte**

FB fills the range of memory from **addr1** through **addr2** with the specified byte value.

```
>FB 1000,1FFF,E5  Filling memory...     (Fill memory 1000-1FFF with E5)
>DM 1000,1FFF                           (Display memory 1000-1FFF)
00001000: E5E5 E5E5 E5E5 E5E5 E5E5 E5E5 E5E5 E5E5  eeeeeeeeeeeeeeee
                    .
                    .
                    .
```

Another form of the **FB** allows you to specify a starting address and the number of bytes to fill:

```
>FB 1000,#/13,1  Filling memory...(Fill 13 bytes with 01 starting at 1000)
>DM 1000,#10                      (Display 16 bytes starting at 1000)
00001000: 0101 0101 0101 0101 0101 0101 01E5 E5E5  .............eee
```

>FW [$x+]addr1, [$x+]addr2, word

Fill memory with **word**

FW fills the range of memory from **addr1** through **addr2** with the specified word value.

```
>FW 1000,1FFF,E5  Filling memory...     (Fill memory 1000-1FFF with 00E5)
>DM 1000,1FFF                           (Display memory 1000-1FFF)
00001000: 00E5 00E5 00E5 00E5 00E5 00E5 00E5 00E5  .e.e.e.e.e.e.e.e
                    .
                    .
                    .
```

The second form of the **FW** allows you to specify a starting address and the number of bytes (not words!) to fill:

```
>FW 1000,#5,FEDC  Filling memory...   (Fill 5 bytes with word FEDC)
>DM 1000,#/16                         (Display 16 bytes starting at 1000)
00001000: FEDC FEDC FEE5 00E5 00E5 00E5 00E5 00E5  ./././.e.e.e.e.e
```

>FL [$x+]addr1, [$x+]addr2, long

Fill memory with **long**

FL fills the range of memory from **addr1** through **addr2** with the specified long value.

```
>FL 1000,1FFF,E5  Filling memory... (Fill memory 1000-1FFF with 000000E5)
>DM 1000,1FFF                       (Display memory 1000-1FFF)
00001000: 0000 00E5 0000 00E5 0000 00E5 0000 00E5  ...e...e...e...e
                    .
                    .
                    .
```

The second form of the **FL** allows you to specify a starting address and the number of bytes (not long words!) to fill:

```
>FL 1000,#9,12345678  Filling memory...(Fill 9 bytes starting at 1000)
>DM 1000,#/16                          (Display 16 bytes starting at 1000)
00001000: 1234 5678 1234 5678 1200 00E5 0000 00E5  .4Vx.4Vx...e...e
```

>M [$x+]addr1,[$x+]addr2,[$x+]addr3

Move a range of memory

The **M** command moves the contents of memory from **addr1** through **addr2** into addresses starting at **addr3** . Overlapping transfers are handled correctly.

```
>M 1000,2672,2000  Moving memory...  (Move memory 1000-2672 to 2000-3672)
```

Another form of the **M** command allows you to specify the number of bytes to be moved:

```
>M 1000,#/325,2000  Moving memory...   (Move 325 bytes from 1000 to 2000)
```

>XB [$x+]addr1,[$x+]addr2,byte,[maskbyte]

Examine (search) memory

The **XB** command searches memory from **addr1** through **addr2** for the first occurrence of **byte** . If **maskbyte** is specified, each word is ANDed with it before the comparison is performed. This option allows non-significant bits to be cleared before the comparison.

When **XB** finds a match, it prints the address and the long word at that address. If you wish to continue the search, you can type a 'C' at this point.

```
>XB 8220,8260,41  Searching...      (Search for an upper-case 'A' = 41H)
Match at 0000822C: 414C2043 - Continue search (C = Yes)? C
Match at 00008231: 414E4E45 - Continue search (C = Yes)? C
No Match found
>XB 8220,8260,'A',5F  Searching...      (Search for an 'A' or 'a')
Match at 0000822C: 414C2043 - Continue search (C = Yes)? C
Match at 00008231: 414E4E45 - Continue search (C = Yes)? C
Match at 00008259: 61756420 - Continue search (C = Yes)? C
Match at 0000825E: 61746520 - Continue search (C = Yes)?    (<cr> aborts)
```

>XW [$x+]addr1,[$x+]addr2,word,[maskword]

Examine (search) memory

The **XW** command searches memory from **addr1** through **addr2** for the first occurrence of **word** . If **maskword** is specified, each word is ANDed with it before the comparison is performed. This option allows non-significant bits to be cleared before the comparison.

When **XW** finds a match, it prints the address and the long word at that address. If you wish to continue the search, you can type a 'C' at this point.

```
>XW 8200,82FF,'SE'  Searching...       (Search for upper-case 'SE')
Match at 0000820E: 53455453 - Continue search (C = Yes)? C
Match at 00008228: 53455249 - Continue search (C = Yes)? C
No Match found
>XW 8200,82FF,'SE',5F5F  Searching...    (Search for 'SE' or 'se')
Match at 0000820E: 53455453 - Continue search (C = Yes)? C
Match at 00008228: 53455249 - Continue search (C = Yes)? C
Match at 0000824C: 73657473 - Continue search (C = Yes)?    (<cr> aborts)
```

>XL [$x+]addr1,[$x+]addr2,long,[masklong]

Examine (search) memory

The **XL** command searches memory from **addr1** through **addr2** for the first occurrence of **long** . If **masklong** is specified, each longword is ANDed with it before the comparison is performed. This option allows non-significant bits to be cleared before the comparison.

When **XL** finds a match, it prints the address and the long word at that address. If you wish to continue the search, you can type a 'C' at this point.

(The following command finds strings in 8500-85FF that begin with ' Th' or ' th')

```
>XL 8500,85FF,20546800,FF5FFF00  Searching...
Match at 00008556: 20746872 - Continue search (C = Yes)? C
Match at 0000856E: 20746865 - Continue search (C = Yes)? C
No Match found
```

>XM [$x+]paddr1,[$x+]paddr2,[$x+]addr1,[$x+]addr2

Examine memory

The **XM** command searches memory from **addr1** through **addr2** for the first occurrence of the pattern contained in memory locations **paddr1** through **paddr2** . No masking is available for this form of the **X** command.

When **XM** finds a match, it prints the address and the long word at the beginning of the match. If you wish to continue the search, you can type a 'C' at this point.

```
>DM 854B,8552                            (Display memory 854B-8552)
0000854B: 6172 6775 6D65 6E74  argument
>XM 854B,8552,8500,85FF  Searching...   (Search for 'argument')
Match at 0000854B: 61726775 - Continue search (C = Yes)? C
Match at 00008560: 61726775 - Continue search (C = Yes)? C
Match at 0000858F: 61726775 - Continue search (C = Yes)? C
No Match found
```

Disassemble memory

>AD Disassemble memory

The form of the **AD** command resembles that of the **DM** command. If no arguments are given, **AD** disassembles 20 instructions starting after the last displayed or disassembled memory location, or from the last trace or break address. Successive **AD** commands may be used to step through memory.

If one address is specified (**AD [$x+]addr**), 20 instructions are disassembled starting at that address.

If 2 addresses are specified (**AD [$x+]addr1,[$x+]addr2**), memory is disassembled from **addr1** through **addr2** .

The **AD** command also accepts a starting address followed by a count which indicates the number of instructions to be disassembled:

```
>AD 7E000,#/10                        (Disassemble 10 instructions)
0007E000: MOVE.B  0224,D0
0007E004: MOVE.W  0222,D1
0007E008: BNE     16[0007E020]
0007E00A: MOVE.B  C021,D1
0007E00E: BTST    #03,D1
0007E012: BEQ     04[0007E018]
0007E014: BSET    #04,D0
0007E018: ANDI.W  #0007,D1
0007E01C: MOVE.B  30(PC,D1.W),D1
0007E020: MOVE.B  D1,C003
```

Note: The **AD** command will not accept an odd address as a starting address because the 68000 does not allow instructions to start on odd addresses.

Breakpoint commands

SDT has two breakpoint registers which allow the user to pause (break) the execution of a program when the 68000 Program Counter reaches a specified address. At this point, memory and registers can be inspected and modified, portions of the program may be disassembled, execution can be resumed, or tracing (single-stepping) can be initiated.

When SDT implements a breakpoint, it replaces the instruction at the specified location with a TRAP #15 instruction. When this trap is executed, control returns to SDT, and it restores the original instruction so that the location can be properly displayed. These substitutions are completely transparent to the user, so you will never see SDT's TRAP instructions in your program unless you enter SDT abnormally (on an EXCEPTION error, for example). Note that breakpoints cannot be set in PROM memory.

Anytime an instruction which has been replaced by a TRAP must be executed, SDT restores the original instruction and then executes it with the trace bit set. The trace bit enables SDT to regain control as soon as the instruction is finished executing. The TRAP is then restored and execution resumes normally. This process is also entirely transparent to the user.

Each breakpoint register has an associated "pass count" which allows SDT to ignore a breakpoint a number of times before breaking. This option is particularly useful when a breakpoint is located inside a loop. Breaking on every iteration of the loop would become very tedious, especially if several thousand iterations were required. If you specify a pass count along with the breakpoint, you can break on every 10th pass, 100th pass, 7295th pass, etc.

You may implement your own breakpoints manually by inserting TRAP #15 instructions in your program. When SDT encounters such a trap, it will realize that it is not one of its own breakpoints and will inform you that an unexpected

breakpoint was encountered:

```
Unexpected break point: 00002532: 4E4F  TRAP   #F  (Easy way to enter SDT)
>GO                   (Continue execution at instr. following TRAP)
```

>DB[x] Display breakpoint register

DBx displays breakpoint register x (x=0,1). If no x is specified, both registers are displayed:

```
>DB                             (Display user breakpoints)
Breakpoint 0: Inactive          (Both breakpoints are inactive)
Breakpoint 1: Inactive
```

>SB[x] [[$x+]addr],[passcount] Set breakpoint register

The format of the SB command resembles that of other S commands (see page 44). A "pass count" (see page 44) may be specified along with the address of each breakpoint. Pass counts may be in the range 0-65535 (0-FFFFH). If no pass count is specified, or the pass count is 0 or 1, SDT will break every time it reaches the breakpoint. A "." is used instead of an address to disable a breakpoint.

```
>SB0 $1+5000                          (Set breakpoint 0 to 5100H)
>DB                                   (Display breakpoint registers)
Breakpoint 0: 00005100 (0000,0000)
Breakpoint 1: Inactive     |    |
                           |    `- passes already made through break
                           `- maximum passes before break
>SB1 2200,/17                         (Set brkpt 1 for 17 passes before break)
>SB                                         (Set both registers)
Breakpoint 0: 00005100 (0000,0000): 1000,FF      (255 passes)
Breakpoint 1: 00002200 (0011,0000):         (<cr> to leave unchanged)
>SB1 .                                      ("." to disable breakpoint 1)
>SB
Breakpoint 0: 00001000 (00FF,0000): .       ("." to disable breakpoint 0)
Breakpoint 1: Inactive:                     (<cr> to leave unchanged)
```

>GO [$x+]addr Execute program

GO starts program execution at **addr** . If **addr** is not specified, execution begins at the address specified by the current Program Counter. (This is handy if you want to resume normal execution of a program after a breakpoint, a break caused by the "break" key, or after tracing.) The **GO** command terminates tracing (if active) and resets breakpoint register temporary pass counts. If you wish to preserve the temporary counts, use the **GC** command.

```
Break: 00008006: 702A  MOVEQ   #2A,D0       (Breakpoint)
>DB                                         (Display breakpoint regs.)
Breakpoint 0: 00008006 (00A0,00A0)          (Break on 160th pass)
Breakpoint 1: Inactive
>GO                                         (Resume execution at 8006)
```

>GC [$x+]addr Execute program

GC is the same as **GO** , except the breakpoint register temporary pass counts are preserved.

```
>GC $2+7500
```

Trace commands
SDT's Trace Mode is invoked using the **TB**, **TR**, **TN**, or **TNI** command. When SDT is in Trace Mode, a "T" appears in front of the standard prompt:

```
T>                                  (Trace Mode)
T$1>                                (Trace Mode with standard base reg.)
```

Any time SDT is in Trace Mode, the next instruction (at the address specified by the Program Counter) can be traced by typing <cr> instead of a command. Trace Mode is terminated with the **TE**, **GO**, **GC**, or **GS** command.

> **Important note:** All trace commands display trace information for the instruction that is about to be executed. A <cr> will execute this instruction and display information for the next instruction.

Tracing TRAP instructions

TRAP instructions are treated in a special manner during tracing. TRAP instructions are used to call special subroutines that the user normally wishes to ignore during tracing. If a TRAP #A is encountered, for example, the user usually wants to continue tracing his program instead of tracing the instructions within the TRAP servicing routine. Furthermore, tracing instructions inside an input/output TRAP can cause very strange things to happen because SDT may use the same TRAP to perform I/O during its operation.

Therefore, SDT normally treats a TRAP instruction as a single indivisible instruction during tracing. It does not enter the TRAP routine and trace the instructions within. Realizing, however, that there are times when this is exactly what a user might want to do, SDT provides a method for tracing or not tracing the interior of TRAPs. The **ST** and **DT** commands allow you to specify which TRAPs you wish to trace and which you want to ignore.

>TB [$x+]addr Begin Trace Mode

TB begins Trace Mode at **addr** or at the address specified by the Program Counter if no **addr** is given.

Important note: The instruction which is disassembled and displayed is about to be executed.

```
>TB                                       (Begin Trace Mode)

Trace: 0007E87E: 46FC  MOVE    #2500,SR
T>Trace: 0007E882: 4241  CLR.W   D1        (<cr> to execute MOVE instr.)
T>Trace: 0007E884: 1238  MOVE.B  024C,D1   (<cr> to execute CLR instr.)
T>
```

>TR [$x+]addr Begin Trace Mode with register display

TR functions identically to **TB** except all 68000 registers are displayed along with the disassembled instruction. Remember that the displayed instruction has not been executed yet, and its effects on the 68000 registers will not be observed until the instruction is executed.

```
>TR                                       (Begin Trace Mode)

Trace: 0007E87E: 46FC  MOVE    #2500,SR
A0: 00011B56 00000D70 00001238 00010400  00012B6C 000104C8 00000400 0007D782
D0: 00000000 00000000 00000000 00010000  00010001 00000002 00003B00 00000000
PC: 0007E87E  US: 0000EDDE  SR: A004 (TS  Z  )
T>Trace: 0007E882: 4241  CLR.W   D1          (<cr> to execute MOVE instr.)
A0: 00011B56 00000D70 00001238 00010400  00012B6C 000104C8 00000400 0007D782
D0: 00000000 00000000 00000000 00010000  00010001 00000002 00003B00 00000000
PC: 0007E882  US: 0000EDDE  SR: 2500 ( S     )
T>Trace: 0007E884: 1238  MOVE.B  024C,D1     (<cr> to execute CLR instr.)
A0: 00011B56 00000D70 00001238 00010400  00012B6C 000104C8 00000400 0007D782
D0: 00000000 00000000 00000000 00010000  00010001 00000002 00003B00 00000000
PC: 0007E884  US: 0000EDDE  SR: A504 (TS  Z  )
T>
```

>TN[x] Trace next x instructions

TNx traces the next x instructions (x in range 0-127 or 0-7FH). If a **TR** command was given beforehand, the 68000 registers will also be displayed. If **TB** initiated Trace Mode, no registers will be displayed. If no x is specified with **TN** , one screenful of instructions is traced.

```
>TB                                  (Initiate trace mode, no reg. display)

Trace: 0007E8B2: 67CA  BEQ     CA[0007E87E]
T>TN4                                (Trace next 4 instructions)

Trace: 0007E87E: 46FC  MOVE    #2500,SR
Trace: 0007E882: 4241  CLR.W   D1
Trace: 0007E884: 1238  MOVE.B  024C,D1
Trace: 0007E888: B238  CMP.B   024D,D1
T>
```

>TNI[x] Trace next x instructions, interruptible

TNIx traces the next x instructions like the **TN** command. The display produced by **TNI** may be paused by typing <CTRL-S> (any character continues), or the trace may be aborted with <CTRL-C>. Although this is a convenient capability, note that **TNI** cannot be used to trace a section of code which is expecting input from the keyboard. This is because **TNI** will grab the input characters before your program can. Use **TN** if this situation arises.

>TE Terminate Trace Mode

TE returns SDT to normal mode.

```
T>TE
>                                    (Prompt indicates normal mode)
```

>DT[x] Display TRAP trace status

DTx displays the Trace/No trace status of TRAP x (x=0-FH). If no x is specified, the status of all TRAPs is displayed. (See **ST** command for further information.)

```
>DT/15                                   (Display status for TRAP #15.)
TRAP #F   No trace
```

>ST[x] [T,N] Set TRAP trace status

STx sets the TRAP trace status of TRAP x (x= 0-FH). If no x is specified, a new status for each TRAP is requested. Three responses are accepted:

<cr>	leave current status unchanged
T	trace the interior code of this trap
N	do not trace the interior code of this trap

A "T" or "N" may also be specified from the command line:

```
>STA T                                   (Trace TRAP #A)
>ST                                      (Set tracing status for each trap)
(T)race or (N)o trace
TRAP #0   No trace : T                   (Trace TRAP #0)
TRAP #1   No trace :                     (<cr> to leave status unchanged)
TRAP #2   No trace :                     (<cr> to leave status unchanged)
TRAP #3   No trace :                     (<cr> to leave status unchanged)
TRAP #4   No trace :                     (<cr> to leave status unchanged)
TRAP #5   No trace :                     (<cr> to leave status unchanged)
TRAP #6   No trace :                     (<cr> to leave status unchanged)
TRAP #7   No trace :                     (<cr> to leave status unchanged)
TRAP #8   No trace :                     (<cr> to leave status unchanged)
TRAP #9   No trace :                     (<cr> to leave status unchanged)
TRAP #A   Trace    : N                   (Don't trace TRAP #10.)
TRAP #B   No trace :                     (<cr> to leave status unchanged)
TRAP #C   No trace :                     (<cr> to leave status unchanged)
TRAP #D   No trace :                     (<cr> to leave status unchanged)
TRAP #E   No trace :                     (<cr> to leave status unchanged)
TRAP #F   No trace :                     (<cr> to leave status unchanged)
```

>GS [$x+]addr Go Subroutine

The **GS** command is a handy way to disable tracing during the execution of a subroutine or a TRAP. Let's suppose that you are tracing a portion of your main program and you come to a BSR instruction. Let's say that the subroutine which is about to be called is a long one, and it has already been thoroughly debugged. Since you know it already works, it would be pointless to spend half an hour tracing it when you want to debug your main program. If you type **GS** , the entire subroutine will be executed, and SDT will break as soon as it returns.

GS works on the following instructions:

BSR BRA TRAP JMP JSR Bcc DBcc

If the instruction at the Program Counter (or **addr** , if specified) is NOT one of the above, **GS** will not attempt to execute. If it is, an internal SDT breakpoint is set at the instruction following the transfer instruction. This means that arguments cannot be passed as data trailing the call.

```
Break: 000041A8  TRAP    #9              (SDT breakpoint)
>TB                                      (Execute TRAP #9)
                                   (TRAP #9 inputs a char., so we type a
                                   character which isn't echoed)
Trace: 000041AA: 6100  BSR     001E[000041CA]
T>GS                                     (Execute this subroutine)

Break: 000041AE: 48E7  MOVEM.L #80C0,-(A7) (SDT breaks when subroutine is
                                            finished executing)
```

Input/Output commands

>LF[x] block,[$x+]addr,bytecount

Load from floppy disk

LF loads data from the floppy disk in drive x (x=0 for left drive, 1 for right drive, no x defaults to left drive) starting at logical block **block** into memory at **addr.** **Bytecount** specifies the number of bytes to load.

```
>LF1 /26,400,/2300           (Load 2300 bytes from drive 1, block
                              26, into memory at 400H)
>DM 400,407
00000400: 1028 5820 202D 2020  .(X  -
```

>WF[x] block,[$x+]addr,bytecount

Write to floppy disk

WF writes data to the floppy disk in drive x (x=0 for left drive, 1 for right drive, no x defaults to left drive) starting at logical block **block** from memory at **addr.** **Bytecount** specifies the number of bytes to write.

```
>WF1 /26,400,/2300             (Write 2300 bytes to drive 1, block
                                26, from memory at 400H)
Line count error, count= 2
```

>ER[x] block Exercise floppy read

ERx block continuously reads 4K (8 blocks) from the disk in floppy drive x (x=0 for left drive, 1 for right drive, no x defaults to left drive) starting at block **block** . A period is displayed for each successful transfer; an X is displayed for each unsuccessful transfer. Typing any character will terminate the **ER** command. **ER** is used mainly for checking drive performance.

```
ER1 /900....................       (Read blocks 900-907, drive 1)
ER /1275XXXXXXXXXXXXXXX            (Read blocks 1275-1282, drive 0)
                                   (Unsuccessful because disk has only
                                    1280 blocks)
```

>EW[x] block Exercise floppy write

EWx block continuously writes 4K (8 blocks) to the disk in floppy drive x (x=0 for left drive, 1 for right drive, no x defaults to left drive) starting at block **block** . A period is displayed for each successful transfer; an X is displayed for each unsuccessful transfer. Typing any character will terminate the **EW** command. **EW** is used mainly for checking drive performance.

Port I/O commands

The following commands provide direct access to I/O ports. Although ports can be accessed just like normal memory locations, the substitute memory commands read from an address (twice) before writing which may be undesirable when accessing an I/O device. The **PO** and **PI** commands allow a port to be read or written in one access. These commands do not verify the existence of a port before accessing it, and will work on memory as well as ports.

>POB [$x+]addr,byte Output byte data to a port

POB writes byte data **byte** to the port at address **addr** .

```
>POB FFC021,'A'                    (Outputs A to port FFC021)
>POB FFC022,10                     (Outputs 10H to port FFC022)
```

>POW [$x+]addr,word Output word data to a port

POW writes the word data **word** to the port at address **addr** (must be on a word boundary).

```
>POW FFC022,'AB'                   (Outputs 'AB' to port FFC022)
>POW FFC022,1007                   (Outputs 1007H to port FFC022)
```

>PIB [$x+]addr Input byte data from a port

PIB reads and displays byte data from the port at address **addr** .

```
>PIB FFC021                        (Inputs a data byte from port FFC021)
00FFC021: 21
```

>PIW [$x+]addr Input word data from a port

PIW reads and displays word data from the port at address **addr** (must be on a word boundary).

```
>PIW FFC022                 (Inputs word data from port FFC022)
00FFC022: 0033
```

>PS x

Set baud rate for remote (modem) serial channel

PS x sets up the baud rate for the remote (modem) serial channel according to the following values for x:

x:	Rate:
0	reserved (currently same as 19200 baud)
1	300 baud
2	600 baud
3	1200 baud
4	2400 baud
5	4800 baud
6	9600 baud
7	19200 baud

On startup, the remote serial channel defaults to 9600 baud with 8 data bits, 1 stop bit, and even parity.

Motorola Object Code Format

Programs and data may be loaded from the Terminal or Modem serial channels using the **LT** or **LA** command. These commands use the standard Motorola object code format. This format consists of ASCII characters formed into records (typically printed on one line). Each record starts with the character 'S' and is followed by a record type number, a byte count, an address, the memory data, and a checksum.

```
record: Stccaaaaddddddd...ddss
 or     Stccaaaaaaddddddd...ddss
```

S — the ASCII character 'S' which always starts a record.

t — type of record (single digit):

- 0 - is the header record which generally contains only a program name in the data field. This record is ignored by the loader routine.
- 1 - indicates an object code record with a two byte address field, 'aaaa'.
- 2 - indicates an object code record with a three byte address field, 'aaaaaa'.
- 9 - is a termination record which indicates that the load is complete.

cc — Hexadecimal byte count of the remaining characters in the record (address, data, and checksum).

aaaaaa or
aaaa — is the hexadecimal memory address where the data which follows is to be loaded. This field is present for all records but ignored for the type 0 (header) and 9 (terminator) records. For type 0, 1, or 9 records the address is contained in 4 hex characters, while for type 2 records the address is contained in 6 hex characters.

dd — represents a two hex character value for each

object code byte. Each record may contain up to 252 bytes of object code although 32 is typical in order to allow a paper listing.

ss is the one's complement of the sum of all the ASCII character bytes from the byte count (including the byte count) to the end of the data.

Note that at the beginning of each record the loader will ignore all characters except 'Q' which will cause the loader to terminate and 'S' which starts the record. This allows a Carriage Return and Line Feed to terminate each line for printout.

Examples:

```
S00600004844521B
S10710801FFE4E728B
S20A010000323C00035641ED
S9030000FC
```

>LA Load memory from auxiliary (modem) port

```
>LA                         (Load data from modem port)
```

>LT Load memory from terminal port

```
>LT                         (Load data from terminal port)
```

EXCEPTION ERRORS

When processing an exception error, interrupts are turned off and the BIOS is disabled, unless the user has re-directed the error to his own error handling routine. Non user-intercepted errors have this format:

```
EXCEPTION: <error type> 'Error at' <8 digit location>
```

Note that the location displayed will sometimes point to the instruction following the instruction that caused the error due to the way the 68000 increments its program counter. Error types are defined below.

Bus Error:
The processor tried to read memory and there was no response. Memory may not exist. A hardware strapping option determines what memory is equipped. Additional information is displayed:

```
Function:<4 digit word>  Access:<8 digit addr> Instr:<4 digit>
```

```
Function: Bits 0-2 ..are the state of the processor
                      function code outputs FC0,FC1 and FC2
          Bit    3 ..is 0 for an instruction, 1 for not
                      an instruction.
          Bit    4 ..is 0 for write, 1 for read.

Access   is the address of the attempt.
Instr    is the instruction being executed
```

Address error: The processor attempted to access a word or long word on an odd address. Additional information is displayed with this error:

```
Function:<4 digit word>  Access:<8 digit addr> Instr:<4 digit>
```

```
Function: Bits 0-2 ..are the state of the processor
                      function code outputs FC0,FC1 and FC2
          Bit    3 ..is 0 for an instruction, 1 for not
                      an instruction.
          Bit    4 ..is 0 for write, 1 for read.

Access   is the address of the attempt.
Instr    is the instruction being executed
```

Illegal Instruction error: There are 2 unused opcodes (Axxx & Fxxx) in the 68000 which are currently undefined and will give this error if an attempt is made to use them. Also, any undefined instruction format or addressing mode will cause this error.

Arithmetic error: An attempt was made to divide by zero or a CHK instruction was executed (user needs to define vector) or a TRAPV instruction was executed (user needs to define vector).

Privilege error: User tried an instruction which requires SUPERVISOR mode.

Reserved TRAP Certain TRAP locations have been reserved by Motorola for future use and should not be used.(This error should never occur.)

Unassigned TRAP error: There are 16 trap locations in the 68000, 0-14 of which are normally unassigned by the Debugger. Trap 15 is used for breakpoints by the Debugger. Traps 8 to 14 are used by the BIOS.

Unassigned Interrupt error: There are 6 maskable auto-interrupt vectors. Normally all of them are unassigned by the debugger.

RAM Parity error: The 7th auto-interrupt vector is non-maskable and is used for RAM parity error reporting. Remember when troubleshooting that the Parity chip itself could be the cause of this error. Note that the location given is where the program was executing and is not necessarily the location with the parity error.

Unknown error: Either the program entered the TRAP handler illegally or the supervisor stack was not set to point to valid RAM.

VII LINK INFORMATION FOR THE 68000 :

VII.01 WORKING IN ASSEMBLY LANGUAGE :

The following files are useful to the assembly language programmer:

SYSTEM.EDITOR The system editor is used to create source text of an assembly language program. The source text is a human-readable (more or less) text file which is then translated into a machine-readable code file using SYSTEM.ASSMBLER.

SYSTEM.ASSMBLER This assembler is used to translate the source text of an assembly program into a code file.

68000.OPCODES This file is necessary to provide SYSTEM.ASSMBLER with information about 68000 assembly code (other OPCODE files enable the Adaptable Assembler to assemble code for other micro-processors).

68000.ERRORS This is an optional file which, if present, will enable SYSTEM.ASSMBLER to print expanded error messages when it encounters errors in the source text.

SYSTEM.LINKER The linker is used to link assembly language routines with other separately assembled routines or high-level programs (examples are shown late in this section).

COMPRESSOR.CODE This utility transforms a code file generated by SYSTEM.ASSMBLER into a ready-to-run memory image (used to develop stand-alone or p-System independent code). COMPRESSOR removes the p-System code file overhead and applies relocation information to the code. See the PROGRAM DEVELOPMENT MANUAL for further information.

Sophisticated users who want to write utilities dealing with assembly code files will find information on code file formats in the p-SYSTEM INTERNAL ARCHITECTURE GUIDE (available from SAGE).

VII.02 EXAMPLES :

Following is a consolidated example of using assembly code files from a Pascal program. It shows how to create, link and run an assembly code program. Later examples explain some of the operations used here.

EXAMPLE 1: LINKING ASSEMBLY ROUTINES TO A PROGRAM

First, the Editor is used to create a Pascal program which references two assembly code procedures (denoted by EXTERNAL). This program is S(aved under the name MAINPROG.TEXT.

```
{ This is a sample Pascal program which uses assembly
language procedures for 32 bit addition and subtraction }

PROGRAM MainExamp;

TYPE INT32 = RECORD                 { Define a 32-bit integer }
              H:INTEGER;
              L:INTEGER;
             END;

VAR Val1,Val2,Val3: INT32;

PROCEDURE ADD32(VAR Result,Arg1,Arg2:INT32); EXTERNAL;

PROCEDURE SUB32(VAR Result,Arg1,Arg2:INT32); EXTERNAL;

BEGIN { MainExamp }
  Val1.H:=0; Val1.L:=-1;    { Set up value of 65535 }
  Val2.H:=0; Val2.L:= 4;    { Set up value of     4 }
  { Val3 := Val1 + Val2 }
  ADD32(Val3,Val1,Val2);
  WRITELN('Addition: High word = ',Val3.H,
          ' Low word = ',Val3.L);

  Val2.H:=0; Val2.L:=-2;    { Set up value of 65534 }
  { Val1 := Val3 - Val2 }
  SUB32(Val1,Val3,Val2);
  WRITELN('Subtraction: High word = ',Val1.H,
          ' Low word = ',Val1.L);
END.
```

Next, the two assembly procedures are created with the Editor and S(aved in the file ASMPROGS.TEXT.

```
;       Example assembly routines:
;       Procedures for 32 bit arithmetic

        .RELPROC        ADD32,3
        MOVEA.L (SP)+,A0        ;Save return address
        MOVE.W  (SP)+,D7        ;Get address of second argument
        MOVE.W  (SP)+,D6        ;Get address of first argument
        MOVE.L  0(A6,D6.L),D0   ;Get first argument
        ADD.L   0(A6,D7.L),D0   ;Add in second argument
        MOVE.W  (SP)+,D7        ;Get address of result
        MOVE.L  D0,0(A6,D7.L)   ;Save result
        JMP     (A0)            ;Return

        .RELPROC        SUB32,3
        MOVEA.L (SP)+,A0        ;Save return address
        MOVE.W  (SP)+,D7        ;Get address of second argument
        MOVE.W  (SP)+,D6        ;Get address of first argument
        MOVE.L  0(A6,D6.L),D0   ;Get first argument
        SUB.L   0(A6,D7.L),D0   ;Subtract second argument
        MOVE.W  (SP)+,D7        ;Get address of result
        MOVE.L  D0,0(A6,D7.L)   ;Save result
        JMP     (A0)            ;Return
        .END
```

Now **compile** the Pascal program.

SCREEN DISPLAYS: **YOU TYPE:**

```
Main prompt line                            C     - for compile
Compiling...
Compile what text?                          MAINPROG <CR>
To what codefile?                           MAINPROG <CR>
Output file for compiled listing?           <CR>  - for none
Pascal compiler
<   0>.............
ADD32
<  13>..
SUB32
<  15>...
MAINEXAM
<  18>............
  30 lines compiled

MAINEXAM .
The code is stored in the file MAINPROG.CODE.
```

Next **assemble** the 68000 assembly code routines.

SCREEN DISPLAYS: **YOU TYPE:**

```
Main prompt line                            A     - for assemble
Assembling...
Assemble what text?                         ASMPROGS <CR>
To what codefile?                           ASMPROGS <CR>
68000 Assembler [IV a.0]
Output file for assembled listing           <CR>
<   0>..
ADD32
<   2>..........
SUB32
<  12>.........
Assembly complete:    21 lines
     0    errors flagged on this assembly
```

LINK INFORMATION FOR THE 68000 EXAMPLES

Now the assembly routines must be **linked** to the Pascal program. Make sure that the specified output file has the extension '.CODE' or the file will not execute.

SCREEN DISPLAYS: **YOU TYPE:**

```
Main prompt line                         L     - for linking
Linking...
Host file?                               MAINPROG <CR>
Opening RAMDISK:MAINPROG.CODE
Lib file?                                ASMPROGS <CR>
Opening RAMDISK:ASMPROGS.CODE
Lib file?                                <CR>
Map file?                                <CR>
Reading MAINEXAM
Reading ADD32
Output file?                             EXAMPLE.CODE
Linking MAINEXAM  #2
      Copying proc ADD32
      Copying proc SUB32
line count error, count= 14
```

Now the final linked result in EXAMPLE.CODE is ready to **eXecute.**

```
Main prompt line                         X     - for execute
Execute what file?                       EXAMPLE
Addition:    High word = 1   Low word = 3
Subtraction: High word = 0   Low word = 5
```

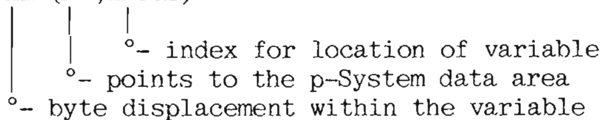
VII.03 STACK AND REGISTER USAGE :

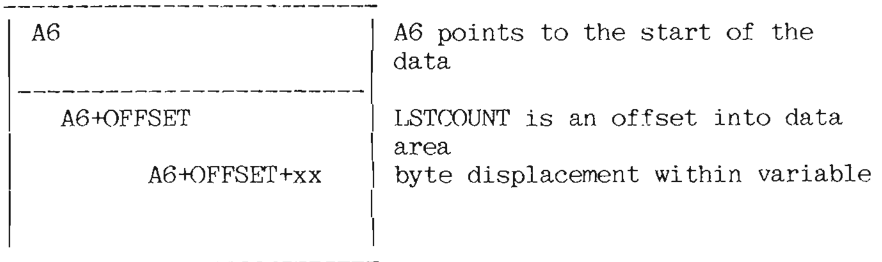
Information from the Pascal calling routine to and from the assembly routine is passed on the User Stack SP=A7. The first 4 bytes are always the return address. Then the arguments, always word values, are passed.

```
PROCEDURE EXAMPLE(arg1,arg2.....argN);EXTERNAL;

SP ----> RET ADDRESS
         argN
         ---
         arg2
         arg1
```

When the argument is specified as VAR, the value passed from Pascal to the assembly routine is a word OFFSET indicating where the argument is in the p-System data area. Register A6 points to this data area. To calculate the true location use the INDIRECT ADDRESSING WITH DISPLACEMENTS AND INDEXES mode:

Affective addr = xx (A6,REG.L)





LINK INFORMATION FOR THE 68000
STACK AND REGISTER USAGE

The OFFSET passed from the Pascal program must not be sign-extended as it is a positive value from 0-64K. This means that the high part of the register REG.L used in the effective address must be zero. D6 and D7 are provided with their high 16 bits already zeroed to make it easy to MOVE.W the OFFSET into the register.

Assembly programs may use the data registers D0-D5 and the low word of D6 and D7 without saving. The high word of D6 and D7 are zero and must stay zero. Only address registers A0-A2 may be used without saving the previous values. All routines are called in USER mode, not Supervisor mode.

VIII 68000 EXAMPLES :

Often when writing an assembly code routine, it becomes necessary to pass the ADDRESS of a Pascal variable through as an argument, so that the assembly routine and Pascal routine can both use the variable.

In the Pascal routine, use the VAR specification for the argument:

```
PROGRAM TRYEXAMPLE;
VAR SHARE:INTEGER;
PROCEDURE EXAMPLE(VAR WESHARE:INTEGER);EXTERNAL;

BEGIN
  EXAMPLE(SHARE);
END.
```

In the assembly code routine calculate the addr:

```
        .PROC    EXAMPLE,1              ;one argument
START   MOVEA.L  (SP)+,A0               ;Save the return address
        MOVEQ    #0,D0                  ;High part of reg should
                                        ;be zeroed
        MOVE.W   (SP)+,D0               ;Get the argument.
        LEA      0(A6,D0.L),A1          ;Calculate addr of SHARE
        .........                       ;body of routine
        JMP      (A0)                   ;return to PASCAL
        .END
```

Variables defined as **.PUBLIC** or **.PRIVATE** are also word offsets to the data area just as VAR arguments are.

Register D6 and D7 already have their high 16 bits zeroed. This provides a convenience in that the user does not have to set up a register himself:

```
        .PROC    EXAMPLE,1            ;one argument
START   MOVEA.L  (SP)+,A0             ;Save the return address
        MOVE.W   (SP)+,D6             ;Get the argument.
        LEA      0(A6,D6.L),A1        ;Calculate addr of SHARE
        .........                     ;body of routine
        JMP      (A0)                 ;return to PASCAL
        .END
```

EXAMPLE 3:PASSING A STRING TO AN ASSEMBLY ROUTINE.

For a string or byte array parameter which is indicated as variable (VAR), the Pascal calling routine passes the offset of the string address to the assembly code routine. In the example the true address of the string is calculated and the string is changed. Control returns to the Pascal routine which prints out the changed string.

> **Note:** If the address is calculated with a zero displacement (giving the start of the string) that address contains the LENGTH of the string not the first character in it.

```
        PROGRAM TRYS;
        VAR S:STRING;
        PROCEDURE EXAMPLE(VAR S:STRING);EXTERNAL;

        BEGIN
          S:='   ';
          EXAMPLE(S);
          WRITELN(' EXAMPLE SAYS:',S);
        END.

        .RELPROC  EXAMPLE,1             ;one argument
START   MOVEA.L (SP)+,A0                ;Save the return address
        MOVEQ   #0,D0                   ;Clear the high part D0
        MOVE.W  (SP)+,D0                ;Get the argument.
        LEA     2(A6,D0.L),A1           ;Calculate S[2] addr.
        MOVE.B  #"H",(A1)+
        MOVE.B  #"I",(A1)+
        JMP     (A0)                    ;return to PASCAL

        .END
```

Strings or byte arrays which are passed as Value parameters (without VAR), must be accessed in a special method using a Segment Pointer. This process is necessary because the string may either be in the data area or in the code area (string constants). The Segment Pointer is passed on the stack as two words. The first word (top of stack) contains either 0 (NIL) or a pointer to a segment environment record. If the first word is 0 then the second word is the offset of the string in the data area and may be accessed as described above. If the first word is not 0 then the access is more complicated because the data is a constant in the code area. The user must also insure that the segment with the constant being accessed is resident in the code pool. The following facts are necessary to track down the constant data.

- First Word (tos) is a pointer to the EREC in the data area.

- Second Word is an offset of the constant into the segment.

- The third word of the EREC points to the SIB (Segment Interface Block) in the data area.

- The first word of the SIB points to the code pool descriptor in the data area.

- The second word of the SIB is the offset of the segment in the code pool.

- The first two words of the code pool descriptor are a long word pointer to the base of the code pool.

```
;       Example of string value parameter access
        .RELPROC YELLOW_BRICK_ROAD,2 ;Actually one parameter
                                     ;  with 2 words
        MOVEA.L (SP)+,A0            ;Save return address
        MOVE.W  (SP)+,D7            ;Get first word parameter
        BNE.S   $10                 ;String is in code area
        MOVE.W  (SP)+,D6            ;Easy access in data area
        LEA     0(A6,D6.L),A1       ;Form address of string
        BRA.S   $20

$10     MOVE.W  4(A6,D7.L),D7       ;Get pointer to SIB
        MOVE.W  0(A6,D7.L),D6       ;Get pointer to pool descriptor
        MOVEA.L 0(A6,D6.L),A1       ;Get base of code pool
        MOVE.W  2(A6,D7.L),D6       ;Get offset of segment in pool
        ADDA.L  D6,A1               ;Form address of segment
        MOVE.W  (SP)+,D6            ;Get offset of string in segment
        ADDA.L  D6,A1               ;Form address of string
$20     ...
```

EXAMPLE 4:A RELOCATABLE ASSEMBLY CODE ROUTINE WITH FIXED PRIVATE AREA.

It is often necessary to set up a data area for an assembly code routine that keeps values between calls to the routine. One way to do this is to make the assembly code a ".PROC" which fixes the entire routine in the heap. This means that there is less room for user data. By making the routine relocatable, and putting only the variables in the data area, the code now resides in the code pool. This is done with ".RELPROC" and ".PRIVATE".

The value of the label specified in the ".PRIVATE" is a word offset to the data area just as VAR arguments are.

```
        PROGRAM TRYIT;                 {simple calling routine}
        PROCEDURE EXAMPLE;EXTERNAL;

        BEGIN
          EXAMPLE;
        END.


        .RELPROC EXAMPLE            ;Relocatable assembly code.
        .PRIVATE LSTCOUNT           ;Setup storage area on heap.


START   MOVE.W  #LSTCOUNT,D6
        ADDQ.L  #4,0(A6,D6.L)       ;Increment count by 4
        RTS                         ;Return addr still on stack
        .END
```

NOTE: Variables defined with ".PUBLIC" are references to data defined in a Pascal program and must also be accessed as an offset within the p-System data area via A6.

IX STAND-ALONE LOADER :

The Stand Alone Assembly Code Loader utility provides a method of loading and starting an assembly language routine which is independent of the p-System (but not the BIOS). This loader prompts the user for the code file name, load address, and starting address. Once this information is given, the routine disables the hooks in the BIOS to the interpreter and sets up the BIOS call to load and start the assembly code. The assembly code may overwrite the p-System data, code, and interpreter but must preserve the BIOS. Once the assembly code is loaded it could disable interrupts and overwrite the BIOS area if necessary.

The LOADASM program is used to load and start a stand alone assembly code program. The assembly code program must be Linked and Compressed to form a file which contains only an image of the assembly code with no header or relocation information.

The LOADASM program will query the user for the file name containing the assembly code. The program will automatically append the '.CODE' suffix if not inhibited with a trailing period. A carriage return here will terminate the program.

The program then asks for the Target memory location for the start of the code file in hexadecimal. A carriage return will default to the typical location 400H. Note that the complete code file is loaded including the remainder of the last 512 byte block past the end of the assembled code.

The program will ask for the Code startup address in hexadecimal. A carriage return here will default the startup address to 400H. Finally the program will query, "Ready to load:" and the user should reply with a Y to initiate the loading and starting of the assembly code program. Any other character will abort the process with a "Program aborted" message.

The load and startup parameters are moved by the BIOS onto the System stack. The original p-System program and interpreter may be overwritten by the new assembly code program. Note that the BIOS may return to the calling program if an error occurs while loading the assembly code file. This may cause a system crash if a portion of the p-System has already been overwritten before the error. If not, a "Load failed" message will be displayed.

Note that a command file might be useful to set up the consistent execution of a stand alone assembly code program. Such a file (LOADCMD.TEXT) might look like:

```
XLOADASM
CODEFILE
1000
1000
Y
```

To start the routine type in X I=LOADCMD.

The LOADASM.CODE file contains the SIO_Unit from the SAGE Toolkit and the FILE_INFO Unit from the p-System distribution.

APPENDIX A: EXCEPTION ERRORS

When processing an exception error, interrupts are turned off and the BIOS is disabled, unless the user has re-directed the error to his own error handling routine. Non user-intercepted errors have this format:

```
EXCEPTION:  <error type>  'Error at' <8 digit location>
```

Note that the location displayed will sometimes point to the instruction following the instruction that caused the error due to the way the 68000 increments its program counter. Error types are defined below.

Bus Error: The processor tried to read memory and there was no response. Memory may not exist. A hardware strapping option determines what memory is equipped. Additional information is displayed:

```
Function:<4 digit word>  Access:<8 digit addr> Instr:<4 digit>
```

Function: Bits 0-2 ..are the state of the processor function code outputs FC0,FC1 and FC2
Bit 3 ..is 0 for an instruction, 1 for not an instruction.
Bit 4 ..is 0 for write, 1 for read.

Access is the address of the attempt.
Instr is the instruction being executed

APPENDIX A:
EXCEPTION ERRORS

Address error: The processor attempted to access a word or long word on an odd address. Additional information is displayed with this error:

```
Function:<4 digit word>  Access:<8 digit addr> Instr:<4 digit>
```

Function: Bits 0-2 ..are the state of the processor function code outputs FC0,FC1 and FC2
Bit 3 ..is 0 for an instruction, 1 for not an instruction.
Bit 4 ..is 0 for write, 1 for read.

Access is the address of the attempt.
Instr is the instruction being executed

Illegal Instruction error: There are 2 unused opcodes (Axxx & Fxxx) in the 68000 which are currently undefined and will give this error if an attempt is made to use them. Also, any undefined instruction format or addressing mode will cause this error.

Arithmetic error: An attempt was made to divide by zero or a CHK instruction was executed (user needs to define vector) or a TRAPV instruction was executed (user needs to define vector).

Privilege error: User tried an instruction which requires SUPERVISOR mode.

Reserved TRAP Certain TRAP locations have been reserved by Motorola for future use and should not be used.(This error should never occur.)

Unassigned TRAP error: There are 16 trap locations in the 68000, 0-14 of which are normally unassigned by the Debugger. Trap 15 is used for breakpoints by the Debugger. Traps 8 to 14 are used by the BIOS.

Unassigned Interrupt error: There are 6 auto-interrupt vectors. Normally all of them are unassigned by the debugger.

RAM Parity error: The 7th auto-interrupt vector is non-maskable and is used for RAM parity error reporting. Remember when troubleshooting that the Parity chip itself could be the cause of this error. Note that the location given is where the program was executing and is not necessarily the location with the error.

Unknown error: Either the program entered the TRAP handler illegally or the supervisor stack was not set to point to valid RAM.

APPENDIX B: FLOPPY DISK BOOT ERRORS

```
Not BOOT disk
Boot aborted on drive 0
Drive error (code) on drive (0 or 1)
```

01	- controller failure
02	- invalid command
03	- recalibrate or seek failure
04	- timeout
05	- missing address mark
06	- no data found
07	- overrun
08	- CRC error
09	- end-of cylinder
0A	- unknown
0B	- address out-of-range

APPENDIX C: WINCHESTER DRIVE ERRORS (while booting)

01	Could not initialize VCO.
03	Recalibrate/seek failure.
04	Drive not ready.
08	CRC error.
0B	Address out of range.
0C	Wrong cylinder.
0E	Bad device number.

APPENDIX D: PROM ROUTINE ENTRY POINTS

```
KEYBCH....   Get a Keyboard Character  ..........loc=FE0008H
KEYCHK....   Check for a Keyboard Character  ....loc=FE000CH
TERMCHAR..   Printout a Character to Terminal  ..loc=FE0014H
TERMTEXT..   Printout a Text String  ............loc=FE0018H
TERMCRLF..   Print a Carriage Return/Line Feed ..loc=FE001CH
TERMHEXB..   Printout a Hexadecimal Byte  .......loc=FE0020H
TERMHEXW..   Printout a Hexadecimal Word  .......loc=FE0024H
FDREAD....   Floppy Disk Read  ..................loc=FE0028H
FDWRITE...   Floppy Disk write  .................loc=FE002CH
BOOTSX....   Floppy disk boot   .................loc=FE0038H
WSELECT...   Winchester Select  .................loc=FE0040H
RDCHAN9...   Read Winchester channel 9  .........loc=FE003CH
DEBUG.....   Debugger Entry Point  ..............loc=FE0030H
```

MACROS USED WITH PROM ROUTINE ENTRY POINTS:

LONG JSR MACRO:

```
.MACRO   LJSR
.WORD    4EB9H
.WORD    00FEH
.WORD    %1
.ENDM
```

LONG JMP MACRO:

```
.MACRO   LJMP
.WORD    4EF9H
.WORD    00FEH
.WORD    %1
.ENDM
```

NOTE: These routines must be called in 68000 SUPERVISOR mode.

APPENDIX E: DEBUGGER COMMANDS

Page

>AD.............................. 64
Disassemble 20 instructions
from current display loc.

>AD [$x+]addr.................... 64
Disassemble 20 instructions
starting at **addr**

>AD [$x+]addr1,[$x+]addr2....... 64
Disassemble instructions
from **addr1** through **addr2**

>AD [$x+]addr,#n................ 64
Disassemble **n** instructions
starting at **addr**

>AR long1,long2................. 52
Arithmetic computation

>DA[x].......................... 53
Display A registers or Ax

>DB[x].......................... 67
Display breakpoint regs.
or breakpoint register x

>DD[x].......................... 53
Display D registers or Dx

>DM............................. 57
Display 256 bytes of memory
from current display loc.

>DM [$x+]addr.................... 57
Display 256 bytes of memory
starting at **addr**

>DM [$x+]addr1,[$x+]addr2....... 57
Display memory from
addr1 through **addr2**

>DM [$x+]addr,#n................ 57
Display **n** of memory
starting at **addr**

>DP............................. 54
Display program counter

>DR............................. 53
Display all registers

>DS............................. 54
Display status register

>DT[x].......................... 72
Display current trace mode
for all traps or trap x

>DU............................. 54
Display user stack pointer

>D$[x].......................... 50
Display base regs. or $x

>ER[x].......................... 75
Exercise floppy read

>EW[x].......................... 75
Exercise floppy write

>FB [$x+]addr1,[$x+]addr2, byte. 59
Fill memory **addr1** through
addr2 with **byte**

>FB [$x+]addr, #n, byte......... 59
Fill memory with **n** bytes
of **byte** starting at **addr**

>FL [$x+]addr1,[$x+]addr2, long. 60
Fill memory **addr1** through
addr2 with data **long**

>FL [$x+]addr, #n, long......... 60
Fill memory with **n** long
words of **long** starting
at **addr**

>FW [$x+]addr1,[$x+]addr2, word. 59
Fill memory **addr1** through
addr2 with data **word**

>FW [$x+]addr, #n, word......... 59
Fill memory with **n** words of
word starting at **addr**

>GC [[$x+]addr]................. 68
Execute program at PC or **addr**
if specified

>GO [[$x+]addr]................. 68
Execute program, resetting
breakpoint counts

>GS [[$x+]addr]................. 73
Execute subroutine call at
PC or **addr**

>IF[x]......................... 47
Boot from floppy drive 0 or x
(0=left drive, 1=right)

>IFR[x]........................ 48
Boot from floppy drive 0 or x
without loading RAMDISK

>IH[x] [#n,name]................ 48
Boot from hard disk 0 or x

>IHR[x] [#n,name]............... 49
Boot from hard disk without
loading RAMDISK

>IS............................. 47
Initialize System

>LA............................. 80
Load from a remote device
(Motorola format)

>LF[x] block,[$x+]addr,count.... 74
Load **count** bytes into
addr from block # **block**
of floppy drive 0 or x

>LT............................. 80
Load from the terminal
(Motorola format)

>M [$x+]addr1,[$x+]addr2,
[$x+]addr3...... 60
Move data from **addr1**
through **addr2** to **addr3**

>M [$x+]addr1,#n,[$x+]addr2..... 60
Move **n** bytes from **addr1**
to **addr2**

>POB [$x+]addr,byte............. 76
Output data **byte** to port

>POW [$x+]addr,word............. 76
Output data **word** to port

>PIB [$x+]addr.................. 76
Input data byte from port

>PIW [$x+]addr.................. 77
Input data word from port

>PS x.......................... 78
Set remote baud rate

>SA[x] [long].................. 56
Modify A registers or Ax

>SB[x] [[$x+]addr],[passcount]... 67
Set breakpoint regs or Bx

>SD[x] [long].................. 56
Modify D registers or Dx

>SM [$x+]addr.................. 58
Modify memory

>SP [long]..................... 56
Modify Program counter

>SR............................ 56
Modify all registers

>SS [long]..................... 56
Modify Status Register

>ST[x] [T,N]................... 72
Set Traps for Tracing

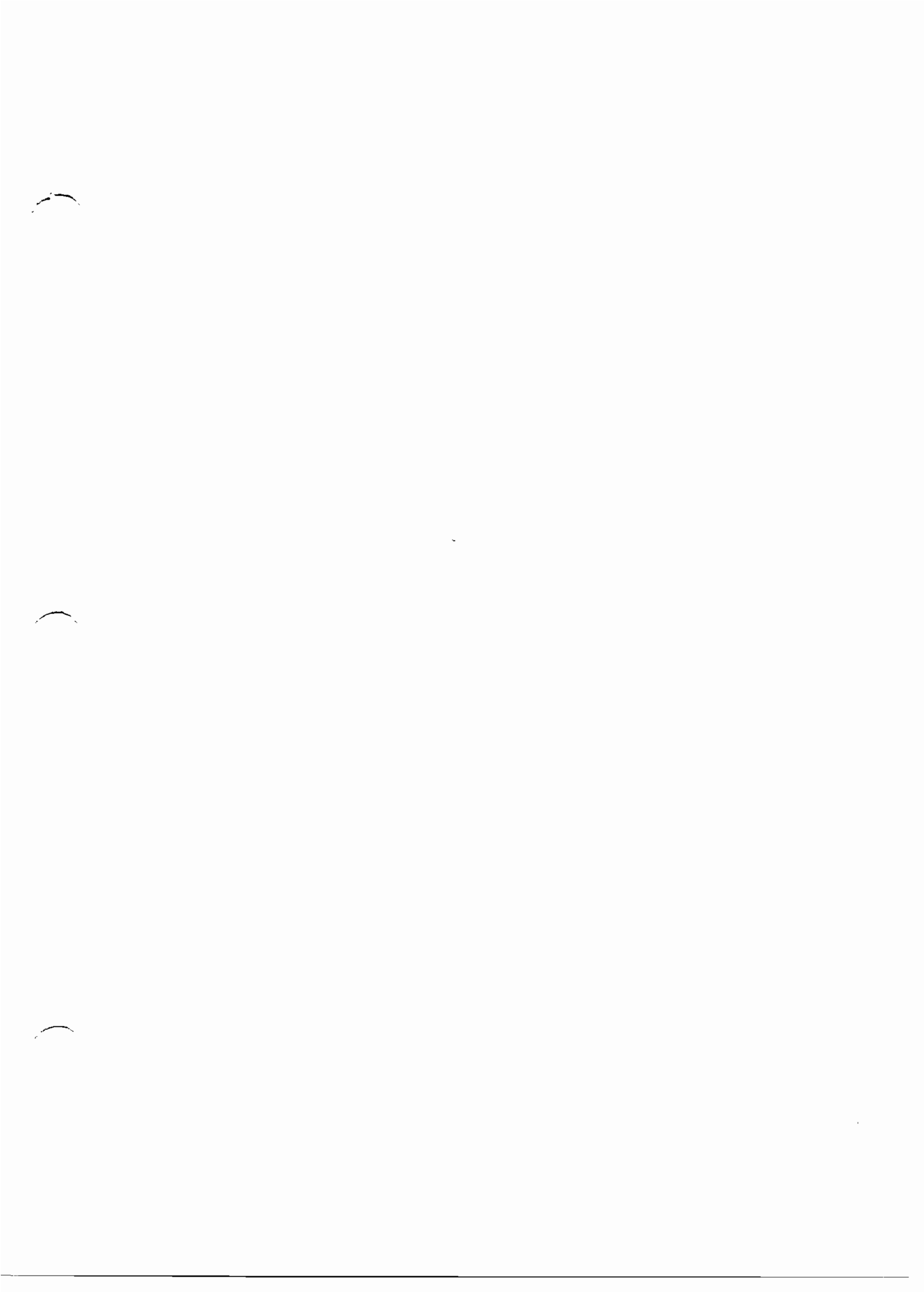
>SU [long]..................... 56
Modify User Stack pointer

>S$[x] [long].................. 50
Modify base regs. or $x

>TB [[$x+]addr]................ 70
Trace without reg. print
starting at PC or **addr**

```
>TE.............................. 71
    Terminate trace mode

>TN[x]........................... 71
    Trace next x instructions

>TNI[x].......................... 71

>TR [[$x+]addr].................. 70


>WF[x] block,[$x+]addr,bytecount  74
    Write count bytes from
    addr to block # block
    of floppy drive 0 or x

>XB[$x+]addr1,[$x+]addr2,          61
           byte,[maskbyte]......
    Search memory addr1 through
    addr2 for byte after
    masking with maskbyte

>XW[$x+]addr1,[$x+]addr2,
           word,[maskword]......   61
    Search memory addr1 through
    addr2 for word after
    masking with maskword
    masking with masklong

>XL[$x+]addr1,[$x+]addr2,
           long,[masklong]......   62
    Search memory addr1 through
    addr2 for long after

>XM[$x+]paddr1,[$x+]paddr2,
      [$x+]addr1,[$x+]addr2....
    Search memory addr1 through
    addr2 for pattern in memory
    paddr1 through paddr2
```

T>.............................. 69
<CR> Trace next instruction (active only when SDT is in Trace Mode)



INDEX

– $ –

$ 51

– A –

AD 64
Address errors 81, 101
Address of a Pascal Variable 93
AR 52
Arithmetic errors 83, 102
Assembler 84
    linkage 93
Auxillary serial channels 12

– B –

BAD memory msg 21
basereg 39
Boot
    PROM 17
    switch settings 21
BOOTER.CODE 25, 27
Bootstrap 23
    Header 24
BOOTSX 34
Break points, Debugger 37
Bus error 81, 101
Bypassed Init 22

– C –

Compressor 84

– D –

D$ 50
DA 53
DB 67
DD 53
DEBUG 36
Debugger
    Calling 37
    Examples 47
    Quick description 39
    Register Usage 38
    SAGE Debugging Tool 47

Dip Switch 10
DISABLING THE MEMORY TEST 22
DM 57
DP 54
DR 53
DS 54
DT 72
DU 54

– E –

ER 75
Errors
    address 81, 101
    arithmetic 83, 102
    bus 81, 101
    exception 81, 101
    illegal instruction 83, 102
    privilege 83, 102
    RAM Parity 83, 102
    Unassigned Interrupt 83, 102
    Unassigned TRAP 83, 102
    unknown 83, 102
EW 75
EXCEPTION ERRORS 81
Exception errors 81, 101
EXTERNAL procedures 84

– F –

FB 59
FDREAD 25, 33
FDWRITE 34
FL 60
Floppy disk
    boot 17, 24
    Control 10
    control port 10
    Status port 10
FLOPPY FORMAT 26
FW 59

-G-

GC 68
GO 68
GROUP-A switches 17
GS 73

-I-

IEEE-488 10
IF 47
IFR 48
IFx boot command 24
IH 48
IHR 49
IHx boot command 26
Illegal instruction errors 83, 102
Indirect addressing 84
Interrupt
    drivers 17
    ENCODER 10
I/O Ports
    (general) 9
I/O Ports (specific) 10
IS 47

-K-

KEYBCH 30, 31
Keyboard, read 31
KEYCHK 31

-L-

LA 80
LF 74
Link
    assembly 93
    register use 92
    68000 specifics 84
link 87
Linker 84
LT 80

- M -

M 60
MACROS 30
Memory test 21
    disable 22
Motorola Object code 79

- O -

Object code, Motorola 79

- P -

Parity, errors 83, 102
PIB 76
PIW 77
POB 76
Polled drivers 17
POW 76
Power-up 17
Printer Port 10, 11
PRIVATE areas 98
Privilege errors 83, 102
PROM 8
    address switching 17
    entry points 30
    Error Handling 15
    Initialization and Bootstrap Routines 15
    I/OSubroutines 15
    SAGE Startup Test 15
    The SAGE Debugging Tool 15
PS 78
p-System, Floppy bootstrap 24
p-System, Winchester boot 26

- R -

RAM 8
    memory test 21
    parity error 83, 103
RAM SIZE =XXXX 21
RDCHAN9 35
Real Time Clock 10
Registers 38
Registers, 68000 92

68000, registers 92
Relocatable code 98
Reserved TRAP 83, 102

-S-

S$ 50
SA 56
SAGE Debugging Tool 47
SAGE IV STARTUP TEST 22
SAGE.PBOOT.TEXT 25
sageproms 15
SAGE.WBOOT.CODE 27
SAGE.WBOOT.TEXT 27
SB 67
scmds 44
SD 56
SDT, calling 37
Serial Port 1 . . . . 10, 11
Serial Port 2 10
SM 58
SP 56
SR 56
SS 56
ST 72
Stand alone, environment 84
SU 56
SUPERVISOR mode 30, 92
Switches
    boot device 21
    GROUP-A 17
    power-up options 17
SYSTEM.BIOS 27
SYSTEM.INTERP 28
system lockup dump 22

-T-

T 69
TB 70
TE 71
TERMCHAR 31
TERMCRLF 25, 27, 32
TERMHEXB 32

Terminal, boot 17
TERMTEXT 25, 27, 32
TN 71
TNI 71
TR 70
trace status 69
TRAPs
    errors 83, 102
    to debugger 37

- U -

Unassigned Interrupt error 83, 102
Unassigned TRAP error 83, 102
Unknown error 83, 102
USER mode 92
User stack 84

- V -

Vectors, interrupt 17

- W -

WDREAD 27
WF 74
Winchester
    boot 26
    ports 13
WSELECT 35

- X -

XB 61
XL 62
XM 62
XW 61

NOTES

NOTES

NOTES

NOTES

NOTES