HEWLETT-PACKARD

Technical Reference Manual

HP-94
Handheld Industrial Computer

HP-94 Handheld Industrial Computer

Technical Reference Manual

HEWLETT PACKARD

Edition 1 February 1987

Reorder Number
82521-90001

Notice

Hewlett-Packard makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Hewlett-Packard shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance, or use of this material.

Hewlett-Packard assumes no responsibility for the use or reliability of its software on equipment that is not furnished by Hewlett-Packard.

Copyright © Hewlett-Packard Company, 1985, 1986.

This document contains proprietary information, which is protected by copyright. All rights are reserved. No part of this document may be photocopied, reproduced, or translated to another language without the prior written consent of Hewlett-Packard Company. The information contained in this document is subject to change without notice.

Epson RTC-58321 Data Sheet © Epson America, Inc., 1986.
All rights reserved. Reprinted by permission.

Hitachi HD61102A Data Sheet © Hitachi America, Ltd. 1986
All rights reserved, reprinted by permission

MS®-DOS is a U.S. registered trademark of Microsoft Corp.

NEC μPD70108 (V20) Data Sheet © NEC Electronics, Inc., 1985.
All rights reserved. Reprinted by permission.

OKI MSM82C51A Data Sheet © OKI Semiconductor, Inc., 1984.
All rights reserved. Reprinted by permission.

Smartmodem™ is a trademark of Hayes Microcomputer Products, Inc.

UNIX® is a registered trademark of AT&T in the U.S.A. and other countries.

Portable Computer Division
1000 N.E. Circle Blvd.
Corvallis, OR 97330, U.S.A.

Printing History **Edition 1** February 1987 Mfg. No. 82521-90002

Contents

1 **Introduction to the Technical Reference Manual**

Part 1 Operating System

1 **Introduction to the Operating System**

Chapter 1 Memory Management

1-1 Hardware Overview
1-1 Software Overview
1-2 Memory Organization
1-7 Reserved Scratch Space
1-10 Directory Table
1-13 File System
1-15 Data Files
1-18 Free Space
1-20 Scratch Areas
1-25 Logical ROMs
1-32 System ROM
1-33 Memory Integrity Verification

Chapter 2 Program Execution

2-1 Running Programs
2-2 Cold Start and Warm Start
2-5 Ending Programs
2-8 Program Structure
2-10 Program Restrictions

Chapter 3 User-Defined Handlers

3-1 Handler Structure
3-3 Channel Input and Output
3-4 Types of Handlers
3-5 Handler Information Table
3-7 Passing Parameters to Handlers
3-10 Handler Linkage Routines
3-12 Handler Routine Descriptions
3-14 CLOSE
3-16 IOCTL

3-20 OPEN
3-22 POWERON
3-25 READ
3-27 RSVD2
3-28 RSVD3
3-29 TERM
3-31 WARM
3-33 WRITE

Chapter 4 Operating System Functions

4-1 Operating System Function Usage
4-1 Operating System Function Descriptions
4-2 BEEP
4-3 BUFFER_STATUS
4-4 CLOSE
4-6 CREATE
4-8 CURSOR
4-9 DELETE
4-11 DISPLAY_ERROR
4-12 END_PROGRAM
4-14 FIND_FILE
4-16 FIND_NEXT
4-19 GET_CHAR
4-21 GET_LINE
4-23 GET_MEM
4-25 MEM_CONFIG
4-27 OPEN
4-29 PUT_CHAR
4-30 PUT_LINE
4-32 READ
4-35 REL_MEM
4-36 ROOM
4-37 SEEK
4-39 SET_INTR
4-41 TIMEOUT
4-43 TIME_DATE
4-45 WRITE

Chapter 5 Hardware Control and Status Registers

5-2 Main Control and Status Registers
5-3 Interrupt Control and Status Registers
5-5 Copies of Write-Only Control Registers

Chapter 6 CPU

Chapter 7 Interrupt Controller

7-1 Procedure for Using a Hardware Interrupt
7-3 Interrupt Control and Status Registers
7-5 When the Operating System Disables Interrupts
7-6 Operating System Functions

Chapter 8 Keyboard

8-1 Keyboard Shift Status
8-2 Display Backlight Control
8-2 Key Buffer
8-2 Waiting for a Key
8-3 Keyboard Scanning
8-5 Keyboard Control and Status Registers
8-6 Operating System Functions

Chapter 9 Display

9-1 Display Backlight Control
9-2 LCD Controllers
9-2 Writing Dots to the Display
9-2 Display Control and Status Registers
9-3 Writing Characters to the Display
9-4 Operating System Functions
9-5 User-Defined Characters

Chapter 10 Serial Port

10-1 Signal Levels
10-1 Enabling or Disabling the Serial Port
10-2 Initializing the Serial Port
10-2 Processing the Serial Port Data Received Interrupt
10-2 Serial Port Control and Status Registers
10-5 Built-in Serial Port Handler
10-9 Operating System Functions

Chapter 11 **Bar Code Port**

11-1 Bar Code Port Power and Transition Detection
11-1 Bar Code Timer
11-1 Initializing the Bar Code Port
11-2 Processing the Bar Code Port Transition Interrupt
11-2 Bar Code Port Timing Constraints
11-3 Bar Code Port Control and Status Registers

Chapter 12 **Timers**

12-1 System Timer
12-3 Bar Code Timer
12-4 Timer Control and Status Registers
12-7 Operating System Functions

Chapter 13 **Power Switch**

13-1 Power Control and Status Registers
13-2 Operating System Functions

Chapter 14 **Batteries**

14-1 Main Nickel-Cadmium Battery Pack
14-2 Backup Lithium Batteries
14-2 Battery Control and Status Registers
14-4 Operating System Functions

Chapter 15 **Real-Time Clock**

15-1 Real-Time Clock Control and Status Registers
15-1 Operating System Functions

Chapter 16 **Beeper**

16-1 Beeper Control and Status Registers
16-2 Operating System Functions

Chapter 17 **Reset Switch**

Chapter 18 **Other Hardware**

18-1 Read/Write Memory (RAM)
18-1 System ROM
18-1 Custom Gate Array
18-2 Earphone Jack
18-2 External Bus Connector

Part 2 BASIC Interpreter

1 **Introduction to the BASIC Interpreter**

Chapter 1 **BASIC Program and Data Structure**

1-1 BASIC Program Organization
1-2 BASIC Program Outline
1-4 Intermediate Code
1-4 Operand Codes
1-9 Variable Area
1-13 Data Structure
1-16 Control Information Save Area

Chapter 2 **Operation Stacks**

2-1 Operation Stack Area
2-2 Control Stack
2-5 Numeric Operation Stack
2-7 Character Operation Stack
2-8 Parameter Table (only for %CALL)

Chapter 3 **Assembly Language Subprograms (Keywords)**

3-1 Program Structure
3-2 BASIC Call and Return
3-6 Access to BASIC Interpreter Utility Routines

Chapter 4 **BASIC Interpreter Utility Routines**

4-1 BASIC Interpreter Utility Routine Descriptions
4-2 ERROR
4-3 GETARG
4-5 IOERR
4-7 SADD
4-8 SDIV
4-9 SETARG
4-10 SMUL
4-11 SNEG
4-12 SPOW
4-13 SSUB
4-14 TOBIN
4-15 TOREAL

Chapter 5 **I/O Statements and Handlers**

5-1 Input Keywords (GET #, INPUT #, INPUT$)
5-4 Output Keywords (PRINT #, PRINT # . . . USING, PUT #)

Part 3 Hardware Specifications

1 **Introduction to the Hardware Specifications**

Chapter 1 **Electrical Specifications**

Chapter 2 **Mechanical Specifications**

2-1 Physical Specifications
2-1 Serial Port Connector Specifications
2-2 Bar Code Port Connector Specifications
2-3 Memory Port Connector Specifications
2-5 External Bus Connector Specifications
2-7 Earphone Connector Specifications
2-7 Battery Pack Connector Specifications

Chapter 3 **Environmental Specifications**

Chapter 4 **Accessory Specifications**

4-1 40K RAM Card Specifications
4-2 ROM/EPROM Card Specifications
4-3 Battery Pack Specifications
4-4 Recharger Specifications
4-5 Level Converter Specifications
4-7 Cables
4-10 Bar Code Readers

Chapter 5 **Data Sheets**

NEC μPD70108 (V20) Microprocessor Data Sheet
OKI MSM82C51A UART Data Sheet
Hitachi HD61102A LCD Column Driver Data Sheet
Epson RTC-58321 Real-Time Clock Data Sheet

Appendixes **Resident Debugger**

A-3 Command Syntax
A-4 D
A-6 G
A-7 I
A-8 L
A-9 M
A-10 O
A-11 R
A-12 S
A-13 X

B-1 **Errors**

C-1 **Keyboard Layout**

D-1 **Roman-8 Character Set**

E-1 **Display Control Characters**

F-1 Memory Map

G-1 Control and Status Register Addresses

H-1 Hardware Interrupts

I-1 Operating System Functions

J-1 BASIC Interpreter Utility Routines

K-1 Program Resource Allocation

Hewlett-Packard Bar Code Handlers

L-1 HNBC Low-Level Handler for Bar Code Port
L-7 HNSP Low-Level Handler for Serial Port
L-14 HNWN High-Level Handler for Bar Code Handlers

Disc-Based Utility Routines

M-2 Utility Routine Descriptions
M-3 BLINK.ASM
M-5 EQUATES.ASM
M-8 FINDOS.ASM
M-10 INTERNAL.ASM
M-14 IOABORT.ASM
M-18 IOWAIT.ASM
M-20 ISOPEN.ASM
M-22 LLHLINKG.ASM
M-34 NOIOWAIT.ASM
M-36 READCTRL.ASM
M-38 READINTR.ASM
M-40 SCANKYBD.ASM
M-42 SETCTRL.ASM
M-44 SETINTR.ASM
M-46 VERSION.ASM
M-49 XIOCTL.ASM
M-51 XTIMEOUT.ASM

Illustrations

Part 1 Operating System

2 Figure 1. HP-94 Hardware Block Diagram

1-3 Figure 1-1. Memory Map of the HP-94
1-4 Figure 1-2. Memory Map of Main Memory
1-6 Figure 1-3. Memory Map of the HP 82411A 40K RAM Card
1-8 Figure 1-4. Memory Map of Reserved Scratch Space
1-10 Figure 1-5. Directory Table Header Contents
1-12 Figure 1-6. Directory Table Entry Contents
1-16 Figure 1-7. File Movement During Data File Expansion
1-18 Figure 1-8. Example of Data File Expansion
1-19 Figure 1-9. Use of Free Space in Main Memory
1-21 Figure 1-10. Defining Scratch Area Data Structure
1-22 Figure 1-11. Blocking a Released Scratch Area
1-23 Figure 1-12. Coalescing Adjacent Released Scratch Areas
1-25 Figure 1-13. Memory Map of the HP 82412A ROM/EPROM Card
1-27 Figure 1-14. Possible Logical ROM Configurations
1-28 Figure 1-15. Memory Map of a 32K Logical ROM in Directory 2
1-30 Figure 1-16. HP 82412A ROM/EPROM Card Circuit Board
1-33 Figure 1-17. Memory Map of the System ROM

2-8 Figure 2-1. Program Headers
2-9 Figure 2-2. BASIC Keyword Structure
2-11 Figure 2-3. Defining Scratch Area Data Structure

3-2 Figure 3-1. Handler Header and Jump Table
3-5 Figure 3-2. Relationship Between High- and Low-Level Handlers
3-7 Figure 3-3. Example of Reading Handler Information Table Entries
3-13 Figure 3-4. Register Save Area

5-2 Figure 5-1. Main Control Register
5-3 Figure 5-2. Main Status Register
5-4 Figure 5-3. Interrupt Control Register
5-5 Figure 5-4. Interrupt Status Register

7-3 Figure 7-1. Interrupt Control Register
7-4 Figure 7-2. Interrupt Status Register
7-5 Figure 7-3. Interrupt Clear Register
7-5 Figure 7-4. End of Interrupt Register

8-1 Figure 8-1. HP-94 Keyboard
8-3 Figure 8-2. HP-94 Keycodes

8-5 Figure 8-3. Keyboard Control Register
8-6 Figure 8-4. Keyboard Status Register

9-1 Figure 9-1. 6 x 8 Character Cell
9-3 Figure 9-2. Keyboard Control Register
9-3 Figure 9-3. Right LCD Driver Data Register
9-3 Figure 9-4. Left LCD Driver Data Register

10-3 Figure 10-1. Interrupt Control Register
10-3 Figure 10-2. Interrupt Status Register
10-3 Figure 10-3. Interrupt Clear Register
10-3 Figure 10-4. Baud Rate Clock Value Register
10-4 Figure 10-5. Main Control Register
10-4 Figure 10-6. Main Status Register
10-4 Figure 10-7. Serial Port Data Register
10-7 Figure 10-8. Baud Rate - Parameter Byte 1
10-8 Figure 10-9. Data Format - Parameter Byte 2
10-8 Figure 10-10. Terminate Character - Parameter Byte 3

11-3 Figure 11-1. Interrupt Control Register
11-4 Figure 11-2. Interrupt Status Register
11-4 Figure 11-3. Interrupt Clear Register
11-4 Figure 11-4. Bar Code Timer Data Register
11-4 Figure 11-5. Bar Code Timer Data Register
11-5 Figure 11-6. Bar Code Timer Control Register
11-5 Figure 11-7. Bar Code Timer Value Capture Register
11-5 Figure 11-8. Bar Code Timer Clear Register
11-5 Figure 11-9. Main Control Register
11-6 Figure 11-10. Main Status Register

12-4 Figure 12-1. Interrupt Control Register
12-4 Figure 12-2. Interrupt Status Register
12-5 Figure 12-3. Interrupt Clear Register
12-5 Figure 12-4. System Timer Data Register
12-5 Figure 12-5. System Timer Control Register
12-6 Figure 12-6. Bar Code Timer Data Register
12-6 Figure 12-7. Bar Code Timer Data Register
12-6 Figure 12-8. Bar Code Timer Control Register
12-6 Figure 12-9. Bar Code Timer Value Capture Register
12-6 Figure 12-10. Bar Code Timer Clear Register

13-2 Figure 13-1. Interrupt Control Register
13-2 Figure 13-2. Interrupt Status Register
13-2 Figure 13-3. Interrupt Clear Register
13-2 Figure 13-4. Power Control Register

14-3 Figure 14-1. Interrupt Control Register
14-3 Figure 14-2. Interrupt Status Register

14-3 Figure 14-3. Interrupt Clear Register
14-4 Figure 14-4. Main Status Register

16-1 Figure 16-1. Main Control Register

Part 2 BASIC Interpreter

1-1 Figure 1-1. BASIC Program Organization
1-2 Figure 1-2. Program Header
1-2 Figure 1-3. Program Code
1-3 Figure 1-4. Variable Descriptor Table
1-3 Figure 1-5. Variable Descriptor Type Byte
1-6 Figure 1-6. Variable Reference
1-7 Figure 1-7. Parameters in the Variable Descriptor Table
1-8 Figure 1-8. Line Reference
1-8 Figure 1-9. DATA Statement Linking
1-10 Figure 1-10. Variable Area Allocation
1-10 Figure 1-11. Allocating and Releasing Variable Areas
1-11 Figure 1-12. Program Code and Variables
1-12 Figure 1-13. BASIC Program and Variable Relationships
1-13 Figure 1-14. Real Numeric Data in the Variable Area
1-13 Figure 1-15. Integer Numeric Data in the Variable Area
1-14 Figure 1-16. Character Data in the Variable Area
1-14 Figure 1-17. Array Data in the Variable Area
1-15 Figure 1-18. Array Data Example: DIM A(2,3)
1-15 Figure 1-19. Array Data Example: OPTION BASE 0 : DIM B$6(4)
1-16 Figure 1-20. Format of the Control Information Save Area

2-1 Figure 2-1. Operation Stack Area
2-2 Figure 2-2. Control Stack Operation
2-2 Figure 2-3. Control Stack During Subprogram Execution
2-3 Figure 2-4. GOSUB Control Element
2-4 Figure 2-5. FOR...NEXT Control Element
2-5 Figure 2-6. Numeric Operation Stack
2-5 Figure 2-7. Real Numeric Data on the Numeric Operation Stack
2-6 Figure 2-8. Integer Numeric Data on the Numeric Operation Stack
2-6 Figure 2-9. Numeric Operation Stack Example: A + B * C → D
2-7 Figure 2-10. Character Operation Stack
2-7 Figure 2-11. Character Operation Stack Example: "ABC" + "DE"
2-8 Figure 2-12. Parameter Table Format
2-8 Figure 2-13. Parameter Table Type Byte

3-1 Figure 3-1. Assembly Language Subprogram Structure
3-3 Figure 3-2. Parameter Table Format
3-3 Figure 3-3. Parameter Table Type Byte
3-5 Figure 3-4. %CALL Example: Calling an Assembly Language Subprogram

4-3 Figure 4-1. GETARG Parameter Processing
4-4 Figure 4-2. GETARG Result Flags (Register CL)
4-9 Figure 4-3. SETARG Parameter Processing

Part 3 Hardware Specifications

2 Figure 1. HP-94 Hardware Block Diagram

Appendixes

F-2 Figure F-1. Memory Map of the HP-94

L-5 Figure L-1. HNBC Valid Data Flag – Parameter Byte 1
L-5 Figure L-2. HNBC Baud Rate – Parameter Byte 2
L-6 Figure L-3. HNBC Parity – Parameter Byte 3
L-6 Figure L-4. HNBC Key Abort – Parameter Byte 4
L-6 Figure L-5. HNBC Good Read Beep – Parameter Byte 5
L-6 Figure L-6. HNBC Terminate Character – Parameter Byte 6
L-11 Figure L-7. HNSP Valid Data Flag – Parameter Byte 1
L-11 Figure L-8. HNSP Baud Rate – Parameter Byte 2
L-11 Figure L-9. HNSP Parity – Parameter Byte 3
L-12 Figure L-10. HNSP Key Abort – Parameter Byte 4
L-12 Figure L-11. HNSP Good Read Beep – Parameter Byte 5
L-12 Figure L-12. HNSP Terminate Character – Parameter Byte 6
L-17 Figure L-13. HNWN Valid Data Flag – Parameter Byte 1
L-17 Figure L-14. HNWN Escape Sequences – Parameter Byte 2
L-19 Figure L-15. Serial Port Configuration Escape Sequence

Tables

Part 1 Operating System

1-1 Table 1-1. HP-94 Memory Configurations
1-2 Table 1-2. Summary of Memory Information
1-13 Table 1-3. Directory Table Sizes
1-28 Table 1-4. Addresses for All Logical ROM Sizes in Directories 1-4
1-29 Table 1-5. Different Organizations of a 96K Application
1-31 Table 1-6. Placing a 96K Application Into Three 32K ICs
1-32 Table 1-7. Placing a 96K Application Into Two 64K ICs
1-34 Table 1-8. Memory Integrity Errors
1-35 Table 1-9. Configuration Map for Valid Memory Configurations

2-3 Table 2-1. HP-94 Status at Cold and Warm Start
2-5 Table 2-2. Cold Start Status of BASIC Programs
2-6 Table 2-3. Ending a Program With END_PROGRAM or FAR RET
2-7 Table 2-4. HP-94 Status in Command Mode

3-3 Table 3-1. Channel Number Assignments
3-6 Table 3-2. Handler Information Table Entries
3-9 Table 3-3. Interpreting the Valid Data Flag
3-11 Table 3-4. Register Usage By Handler Linkage Routines
3-19 Table 3-5. Reserved IOCTL Function Codes
3-24 Table 3-6. Functions Allowed in POWERON Routine

5-1 Table 5-1. I/O Addresses for Control and Status Registers
5-6 Table 5-2. Copies of Primary Control Registers

6-2 Table 6-1. Intel 8088 and NEC V20 Instruction Mnemonics

7-1 Table 7-1. HP-94 Hardware Interrupts
7-2 Table 7-2. Using Hardware Interrupts
7-3 Table 7-3. Interrupt Control and Status Registers
7-6 Table 7-4. Interrupt-Related Operating System Functions

8-4 Table 8-1. ASCII Characters and Keycodes for Each Key
8-5 Table 8-2. Keyboard Control and Status Registers
8-6 Table 8-3. Keyboard-Related Operating System Functions

9-3 Table 9-1. Display Control and Status Registers
9-4 Table 9-2. Display Control Characters
9-5 Table 9-3. Display-Related Operating System Functions

10-2 Table 10-1. Serial Port Control and Status Registers

10-4 Table 10-2. Baud Rate Clock Values
10-6 Table 10-3. Behavior of Built-in Serial Port Handler
10-7 Table 10-4. Errors Reported by Built-In Serial Port Handler
10-7 Table 10-5. Built-in Serial Port Handler Baud Rate Values
10-9 Table 10-6. Control Line Behavior
10-9 Table 10-7. Serial Port-Related Operating System Functions

11-3 Table 11-1. Bar Code Port Control and Status Registers

12-1 Table 12-1. HP-94 Timers
12-1 Table 12-2. Events Checked By System Timer Interrupt Routine
12-4 Table 12-3. Timer Control and Status Registers
12-7 Table 12-4. Timer-Related Operating System Functions

13-1 Table 13-1. Power Control and Status Registers
13-2 Table 13-2. Power Switch-Related Operating System Functions

14-1 Table 14-1. Activities Halted During Default Low Battery Behavior
14-3 Table 14-2. Battery Control and Status Registers
14-4 Table 14-3. Battery-Related Operating System Functions

15-1 Table 15-1. Real-Time Clock Control and Status Registers
15-1 Table 15-2. Real-Time Clock-Related Operating System Functions

16-1 Table 16-1. Beeper Control and Status Registers
16-2 Table 16-2. Beeper-Related Operating System Functions

Part 2 BASIC Interpreter

1-3 Table 1-1. Variable Descriptor Length Byte
1-4 Table 1-2. Intermediate Code
1-4 Table 1-3. Intermediate Code Groups
1-5 Table 1-4. Operand Codes

4-2 Table 4-1. Codes for ERROR Utility Routine
4-3 Table 4-2. GETARG Result Flag (Register CL)

5-2 Table 5-1. Response of Input Keywords to Handler-Generated Errors
5-4 Table 5-2. Response of Output Keywords to Handler-Generated Errors

Part 3 Hardware Specifications

1-1 Table 1-1. Principal Integrated Circuits
1-2 Table 1-2. Electrical Specifications

2-1 Table 2-1. Physical Specifications
2-2 Table 2-2. Serial Port Connector Pin Assignments
2-2 Table 2-3. Serial Port Mating Connectors
2-2 Table 2-4. Bar Code Port Connector Pin Assignments
2-3 Table 2-5. Bar Code Port Mating Connectors
2-4 Table 2-6. Memory Port Connector Pin Assignments
2-6 Table 2-7. External Bus Connector Pin Assignments

3-1 Table 3-1. Environmental Specifications

4-1 Table 4-1. HP-94 Hardware Accessories
4-2 Table 4-2. ROM and EPROM Specifications
4-2 Table 4-3. ROM and EPROM Manufacturers
4-3 Table 4-4. HP 82430A Rechargeable Battery Pack Specifications
4-5 Table 4-5. HP 82431 Recharger Specifications
4-6 Table 4-6. HP 82470A RS-232-C Level Converter Pin Assignments
4-6 Table 4-7. Line Receivers That Do Not Require Level Converter
4-7 Table 4-8. HP-94 to Modem Cable
4-8 Table 4-9. HP-94 to Printer Cable
4-8 Table 4-10. HP-94 to Level Converter Cable
4-9 Table 4-11. HP-94 to Vectra Cable
4-9 Table 4-12. Vectra or IBM PC/AT to Level Converter Cable
4-10 Table 4-13. IBM PC or PC/XT to Level Converter Cable
4-11 Table 4-14. HP-94 Serial Port to Smart Wand Cable

Appendixes

A-1 Table A-1. Resident Debugger Commands
A-2 Table A-2. Resident Debugger Keyboard Map

B-2 Table B-1. Operating System Errors
B-3 Table B-2. BASIC Interpreter Errors

C-1 Table C-1. ASCII Characters and Keycodes for Each Key

E-1 Table E-1. Display Control Characters

G-1 Table G-1. I/O Addresses for Control and Status Registers

H-1 Table H-1. HP-94 Hardware Interrupts

I-1 Table I-1. Operating System Function List

J-1 Table J-1. BASIC Interpreter Utility Routine List

K-1 Table K-1. Error Number Usage
K-2 Table K-2. Hewlett-Packard Handler Resource Usage

L-2 Table L-1. HNBC Statistics
L-3 Table L-2. Behavior of HNBC
L-4 Table L-3. Errors Reported by HNBC
L-5 Table L-4. HNBC Baud Rate Values
L-6 Table L-5. HNBC Parity Values
L-7 Table L-6. HNSP Statistics
L-9 Table L-7. Behavior of HNSP
L-10 Table L-8. Errors Reported by HNSP
L-11 Table L-9. HNSP Baud Rate Values
L-11 Table L-10. HNSP Parity Values
L-14 Table L-11. HNWN Statistics
L-15 Table L-12. Behavior of HNWN
L-16 Table L-13. Errors Reported by HNWN
L-18 Table L-14. Beeps From HNWN for Smart Wand Escape Sequences
L-19 Table L-15. Smart Wand Baud Rate
L-19 Table L-16. Smart Wand Parity Values
L-21 Table L-17. Status Request Escape Sequence Parameter

M-1 Table M-1. Utility Routines on Technical Reference Manual Disc
M-14 Table M-2. Low Battery Interrupt Routine Behavior During I/O
M-15 Table M-3. Power Switch Interrupt Routine Behavior During I/O
M-15 Table M-4. Timeout Interrupt Routine Behavior During I/O
M-22 Table M-5. Handler Linkage Routine List

Introduction to the Technical Reference Manual

The *HP-94 Technical Reference Manual* provides software and hardware reference information about the HP-94 Handheld Industrial Computer. This information should allow software developers to write assembly language programs for controlling the HP-94 hardware resources, and hardware developers to design accessories that connect to the machine. This manual assumes a certain level of familiarity with the HP-94 and 8088 assembly language programming, and that the user will be using Microsoft assembly language development tools (MASM and LINK) or their equivalents. It is a supplement to the HP 82520A *HP-94 Software Development System* (SDS), which includes other information necessary to fully understand the product, as well as software utilities needed to convert and transfer assembly language programs to the machine. The manual is divided into four major parts:

- Operating System
- BASIC Interpreter
- Hardware Specifications
- Appendixes

The first section describes the built-in operating system, which manages and provides programmatic access to the HP-94 hardware: memory, interrupt system, keyboard, display and backlight, serial port, bar code port, internal timers, power switch and power control, low battery detection, real-time clock, and beeper. This section includes topics such as memory management, program execution, writing user-defined handlers (device drivers) for controlling the serial and bar code ports, and using operating system functions to simplify hardware control from assembly language programs.

The second section describes the internal operation of the built-in BASIC interpreter, which provides the ability to execute BASIC programs that were developed on a development system computer using the HP-94 SDS. This section does not discuss the syntax of the BASIC language, or the operation of each BASIC keyword; that information is contained in the *BASIC Language Reference Manual*. Instead, the section discusses the structure and operation of BASIC programs, data structure of BASIC variables, writing new BASIC keywords, and using BASIC interpreter utility routines to simplify the interaction of BASIC and assembly language programs.

The third section contains hardware specifications for the HP-94 in four categories: electrical (voltage and current levels, HP-94 operating conditions), mechanical (dimensions and connector pinouts), environmental (conditions under which the HP-94 will perform properly), and accessory (electrical and mechanical characteristics of plug-in cards, level converter, cables, etc.).

The final section is appendixes containing summaries of reference information for developers. This includes documentation for the utility subroutines on the disc with this manual, and for the built-in assembly language debugger.

Part 1

Operating System

Introduction to the Operating System

This section of the *HP-94 Technical Reference Manual* describes the built-in operating system, which manages and provides programmatic access to the HP-94 hardware. This section includes topics such as memory management, program execution, writing user-defined handlers (device drivers) for controlling the serial and bar code ports, and using operating system functions to simplify hardware control from assembly language programs.

This section also describes the HP-94 hardware: what major hardware elements are present in the machine, what they do, and how to operate them under software control. The major hardware elements are as follows:

- System ROM
- Read/Write Memory (RAM)
- Control and Status Registers
- CPU
- Interrupt Controller
- Keyboard
- Display with Electroluminescent Backlight
- Serial Port
- Bar Code Port
- Timers
- Power Switch
- Nickel-Cadmium (NiCd) Battery Pack
- Lithium Backup Batteries
- Real-Time Clock
- Beeper
- Reset Switch

All these items will be discussed in subsequent chapters. The following is a block diagram showing the major hardware elements and their relationships.

1

Memory Management

Contents

Chapter 1 Memory Management

1-1 Hardware Overview
1-1 Software Overview
1-2 Memory Organization
1-4 Main Memory
1-5 40K RAM Card
1-7 ROM/EPROM Card
1-7 Reserved Scratch Space
1-10 Directory Table
1-13 File System
1-13 File Names
1-13 File Types
1-14 Erasing and Loading Files
1-14 Reserved File Names
1-14 Maximum Number of Files
1-15 Data Files
1-15 File Size
1-15 Size Increment
1-17 End-of-Data Address
1-17 File Access Pointer
1-17 Deleting Data Files
1-17 Interrupts During File Operations
1-18 File Expansion Example
1-18 Free Space
1-19 Usage in Command Mode
1-20 Usage at Run Time
1-20 Scratch Areas
1-20 Allocating Scratch Areas
1-21 Releasing Scratch Areas
1-24 Number of Scratch Areas
1-24 Optimum Memory Use With Scratch Areas
1-25 Logical ROMs
1-25 Logical Structure of the ROM/EPROM Card
1-26 Combining Logical ROMs of Different Sizes
1-29 Selecting a Logical ROM Size
1-29 Physical Layout of the ROM/EPROM Card
1-30 Selecting an IC Size
1-31 Placing Logical ROMs Into Physical ICs
1-32 System ROM
1-33 Memory Integrity Verification
1-34 Checksums Computed at Power Off
1-34 Memory Integrity Tests at Power On

1

Memory Management

This chapter describes memory in the HP-94: its possible configurations, how it is organized, and the memory management software.

Hardware Overview

The HP-94 is available in three memory configurations: HP-94D with 64K RAM, HP-94E with 128K RAM, and HP-94F with 256K RAM. Inside the 94 is a single slot for optional memory accessories. The 94D and 94E allow either the HP 82411A 40K RAM Card or HP 82412A ROM/EPROM Card (holding 32 to 128K of ROM or EPROM) to be plugged in. In addition, the 94E can be expanded to 256K (equivalent to a 94F) with the HP 82410A 128K Memory Board (service upgrade only), which also occupies the accessory slot. The 94F cannot be expanded. The following table summarizes HP-94 memory configurations.

Table 1-1. HP-94 Memory Configurations

Machine	Built-In RAM	40K RAM Card Allowed	ROM/EPROM Card Allowed	128K Memory Board Allowed
HP-94D	64K	Yes	Yes	No
HP-94E	128K	Yes	Yes	Yes
HP-94F	256K	No	No	No

The maximum total user memory in the HP-94, RAM and ROM/EPROM combined, is 256K. This limit is imposed by both hardware and software.

Software Overview

The memory management software in the HP-94 provides a directory structure for major contiguous blocks of memory, such as built-in memory and plug-in memory (RAM and ROM/EPROM cards). Within each directory is a file system that supports four different file types and files in RAM or ROM. BASIC programs (type B), assembly language programs (type A), and user-defined I/O port handlers (type H) execute in place, whether in RAM or ROM. Data files (type D) can be created and deleted dynamically while programs are running, and expand when written to in fixed- or variable-length increments. The operating system also provides for allocation and release of scratch areas, and verifies memory integrity using checksums at power off and power on.

Memory Organization

HP-94 memory is organized into contiguous blocks called directories. The directories fall into three major categories: main memory (built-in memory plus the 128K memory board), plug-in memory (40K RAM and ROM/EPROM cards), and system ROM (built-in operating system and BASIC interpreter). Each block of memory has a fixed-length table at the beginning that describes each file in that block of memory. Since the directory table is fixed-length, the maximum number of files that the directory can contain is also fixed. The directory table also identifies what type of memory it is (main, plug-in RAM, plug-in ROM) and how much memory is encompassed by the directory. Below is a table summarizing important information about HP-94 memory. followed by a memory map that shows the organization of all memory in the HP-94. Note that in the map, the main memory RAM quantities include the RAM for the smaller memory configurations, and the ". . ." indicates unused address space.

Table 1-2. Summary of Memory Information

Name of Memory Area	Memory Size	Directory Number(s)	Max. No. of Files	Min. System Overhead
Main Memory	64K	0	63	3.5K *
	128K	0	63	3.5K *
	256K	0	127	4.5K *
40K RAM Card	40K	1	31	0.5K
ROM/EPROM Card	32K	1-4	31	0.5K
	64K	1-3	31	0.5K
	96K	1-2	63	1K
	128K	1	63	1K

* If a BASIC program is running, there will be an additional 2K used by the BASIC interpreter, plus space for the data in the BASIC program variables.

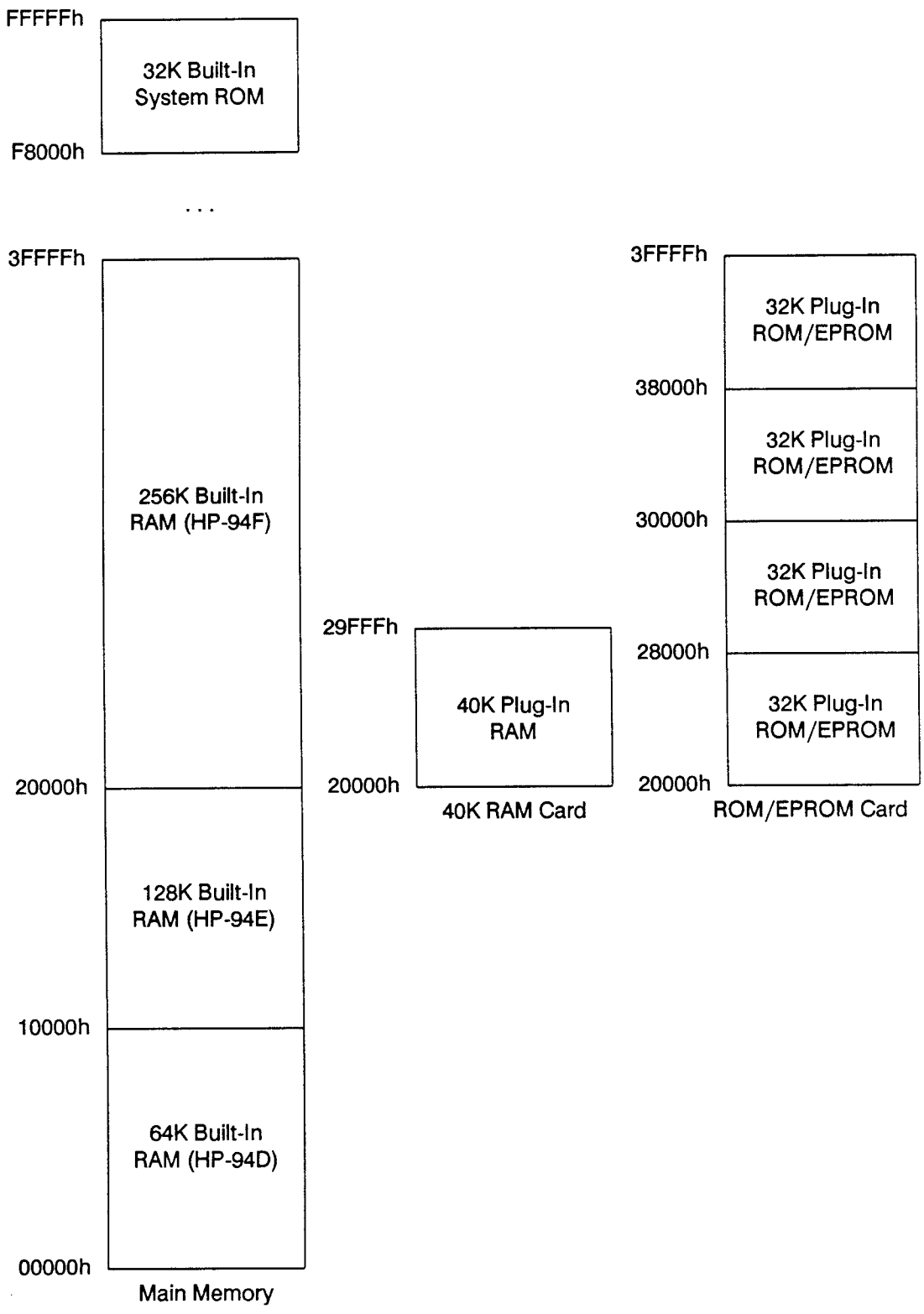


Figure 1-1. Memory Map of the HP-94

Main Memory

Main memory is the first major block of memory, and is called directory 0. It can be 64, 128, or 256K, depending on the memory configuration (94D, 94E, or 94F). Even though the 128K memory board that is used in the 94F or added to the 94E occupies the accessory slot, it is still treated as main memory because it cannot be installed or removed by the user the way the plug-in cards can. The number of files main memory can contain are 63, 63, and 127 respectively for the three memory configurations.

Below is a map of main memory. The pointers on the right side of the memory map correspond to segment addresses maintained in the directory table header (the first entry in the directory table), and will be discussed under "Directory Table".

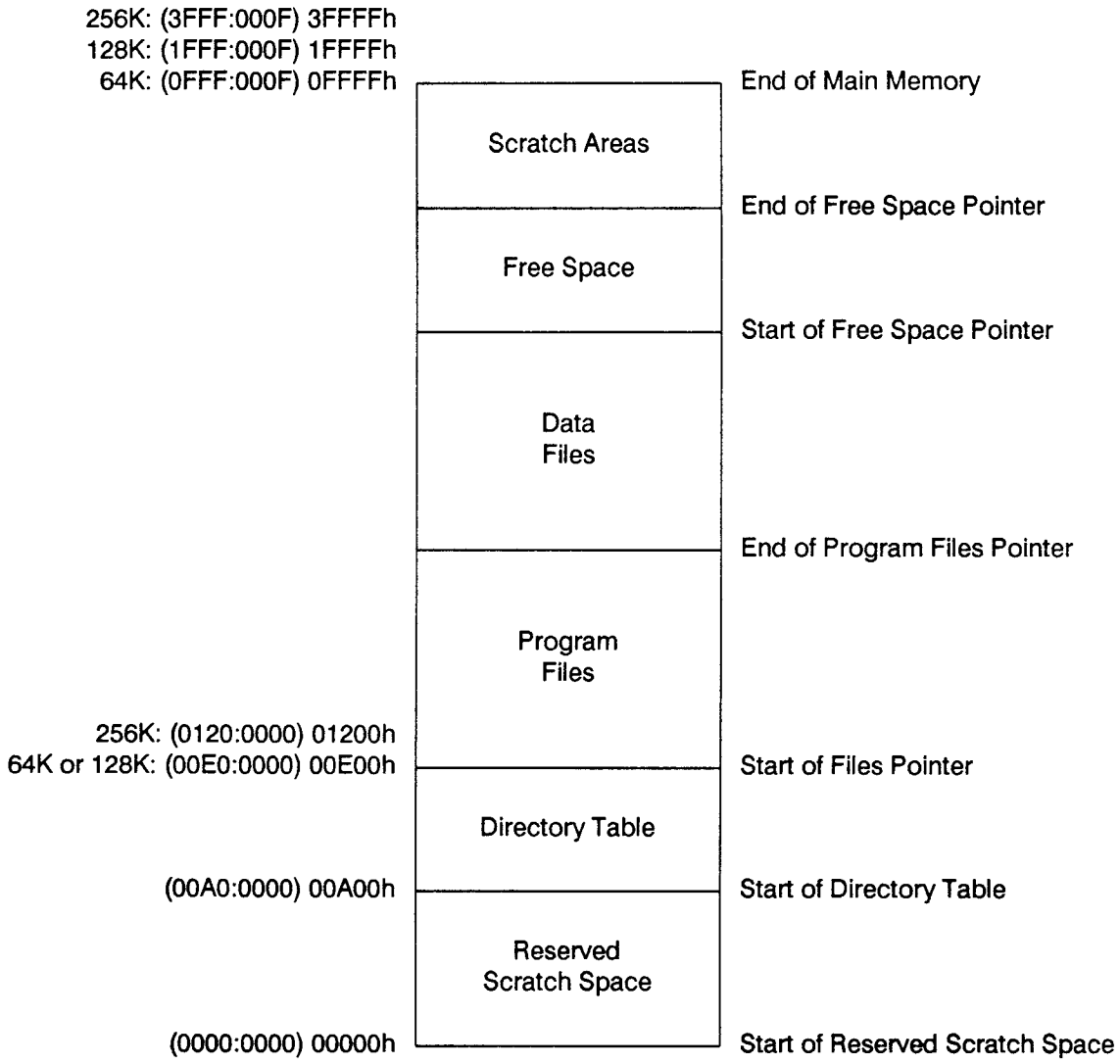


Figure 1-2. Memory Map of Main Memory

The major blocks of memory shown in the memory map are described briefly below. They will each be the subject of a separate section of this chapter.

- Reserved Scratch Space
  This area contains the interrupt vectors for the hardware and software interrupts for the CPU. This area is also used by the operating system to maintain information about the current state of the 94, and for pointers into that information. This area comprises 2.5K of the system overhead.
- Directory Table
  This block describes main memory and all the files contained in it. Files begin immediately after the end of the directory table. This area comprises 1K or 2K of the system overhead, depending on the memory configuration.
- Program Files
  This block is where all non-data files are stored; that is, file types A, B, and H. All program files appear first in the file system. The size of this block changes while programs are loaded, but does not expand or contract at run time.
- Data Files
  This block is where data files are stored. Data files expand by allocating memory from free space, expanding toward higher addresses. When data files are deleted, all their space is returned to the free space area.
- Free Space
  This block is the pool of available memory from which data files are created and expanded and scratch areas are allocated.
- Scratch Areas
  Scratch areas are requested by the built-in BASIC interpreter and by user-written assembly language programs and handlers, and are created by allocating memory from free space, building toward lower addresses. When scratch areas are released, they are returned to free space. Scratch areas are only created in main memory, regardless of which directory contains the program requesting the scratch area. They comprise any additional system overhead requirements.

40K RAM Card

The HP 82411A 40K RAM card is one of the two types of plug-in memory, and is called directory 1. It is 40K long, and can contain a maximum of 31 files. The organization of the RAM card is a subset of the main memory organization — it contains only a directory table, files, and free space. No scratch areas are available, since scratch areas are only allocated in main memory.

Here is a memory map of the 40K RAM card. The pointers on the right side of the map have the same meaning as for main memory.

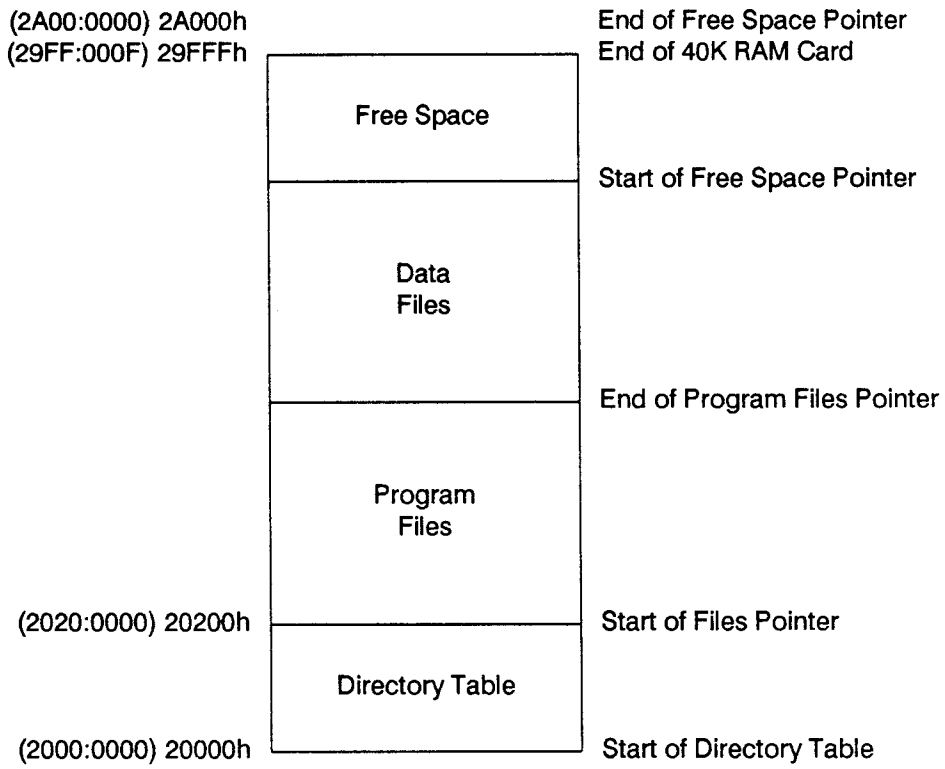


Figure 1-3. Memory Map of the HP 82411A 40K RAM Card

The major blocks of memory shown in the memory map are described below.

- Directory Table
  This block describes the RAM card and all the files contained in it. Files begin immediately after the end of the directory table. This area comprises the 0.5K RAM card overhead.
- Program Files
  This block is where all non-data files are stored; that is, file types A, B, and H. All program files appear first in the file system. The size of this block changes while programs are loaded, but does not expand or contract at run time.
- Data Files
  This block is where data files are stored. Data files expand by allocating memory from free space, expanding toward higher addresses. When data files are deleted, all their space is returned to the free space area.
- Free Space
  This block is the pool of available memory from which data files are created and expanded.

ROM/EPROM Card

The HP 82412A ROM/EPROM card is the other type of plug-in memory, and can contain directories 1 through 4. Files can be put in ROM or EPROM in blocks of four different sizes: 32, 64, 96, and 128K. The number of files each block can contain is 31, 31, 63, or 63 respectively, depending on the ROM or EPROM size. The memory map of the ROM/EPROM card will be discussed in detail under "Logical ROMs" (a *logical ROM* is a ROM in one of the different possible sizes, not necessarily related to the physical IC size actually placed on the ROM/EPROM card).

The organization of each of the four directories within the ROM/EPROM card is similar to the RAM card. They each contain only a directory table, files, and free space. No scratch areas are available, since scratch areas are only allocated in main memory (and could not be allocated in ROM or EPROM anyway).

The memory map of an individual ROM within the ROM/EPROM card is essentially the same as for the 40K RAM card. Unlike the RAM card, data files can only be read — they cannot be created, deleted, or written to. Also, the free space in a ROM or EPROM cannot be used.

The pointers that are shown on the RAM card memory map have the same meaning for an individual ROM or EPROM, but their values vary depending on the size and directory number of the ROM. This will also be discussed in "Logical ROMs".

Reserved Scratch Space

The reserved scratch space is the first 2.5K of main memory. The first 0.5K contains interrupt vectors for CPU, hardware, and software interrupts. It also contains pointers to the next 2K, which is the operating system scratch space. Here is a memory map of the reserved scratch space. The ". . ." indicates unused interrupt vector locations.

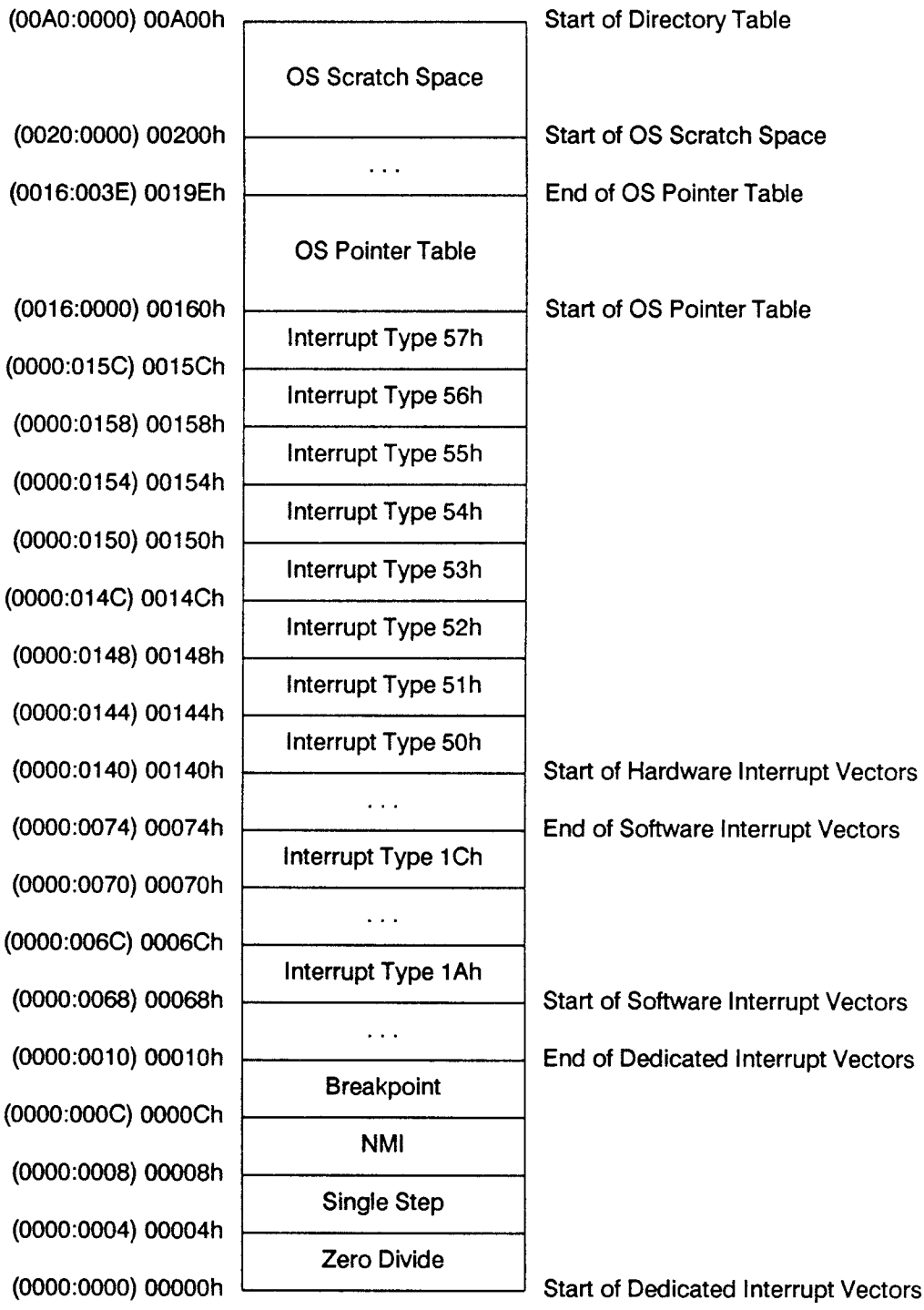


Figure 1-4. Memory Map of Reserved Scratch Space

The major items in the reserved scratch space are described below. The information at the end of each description are the chapters or appendixes where further information can be found about that interrupt. General information about the hardware interrupts (types 50h-57h) is in the "Interrupt Controller" chapter.

- Zero Divide
  Dedicated interrupt vector for divide-by-zero condition. Points to the same location as the breakpoint interrupt vector (appendix A).
- Single Step
  Dedicated single step interrupt vector used for single-stepping the resident debugger (appendix A).
- NMI
  Dedicated non-maskable interrupt vector used to invoke the resident debugger. Points to the same location as the breakpoint interrupt vector (appendix A).
- Breakpoint
  Dedicated breakpoint interrupt vector used for breakpoints in the resident debugger (appendix A).
- Interrupt Type 1Ah
  Software interrupt vector used to invoke the operating system functions (chapter 4).
- Interrupt Type 1Ch
  Software interrupt vector used for the one-second background timer (chapter 12).
- Interrupt Type 50h
  Hardware interrupt vector for system timer (chapter 12).
- Interrupt Type 51h
  Hardware interrupt vector for bar code port timer (chapters 11 and 12).
- Interrupt Type 52h
  Hardware interrupt vector for bar code port transitions (chapter 11).
- Interrupt Type 53h
  Hardware interrupt vector for serial port (82C51 data received) (chapter 10).
- Interrupt Type 54h
  Hardware interrupt vector for low main battery voltage (chapter 14).
- Interrupt Type 55h
  Hardware interrupt vector for power switch pressed (chapter 13).
- Interrupt Type 56h
  Reserved hardware interrupt vector 1 (chapter 7).
- Interrupt Type 57h
  Reserved hardware interrupt vector 2 (chapter 7).
- OS Pointer Table
  These are pointers to various parts of the operating system scratch space. The main pointer of interest to assembly language programmers is the one that points to the handler information table. Refer to the "User-Defined Handlers" chapter for details.
- OS Scratch Space
  This is the space in which the operating system keeps important information about the current state of the HP-94. This area is 2K long. The operating system stack is in this area. It varies in length as it is used, up to a maximum of approximately 600 bytes.

CAUTION The operating system does not initialize or use the overflow interrupt (dedicated interrupt vector 04h, at address 04h * 4 = 00010h). A program that uses the INTO instruction (*interrupt on overflow*) must initialize this interrupt vector to a location in its own program space.

Directory Table

The directory table is organized as a series of 16-byte entries, one per file. The first entry is the directory table header. It identifies the directory, the type of memory (main memory, 40K RAM card, or ROM/EPROM card), and the total amount of memory encompassed by the directory. The header also contains the *pointers* shown on the memory maps. Since all memory areas start and end on paragraph boundaries (a *paragraph* is a block of 16 bytes), pointers are stored in the directory table as segment addresses only.

The contents of the directory table header are shown below. The numbers on the left are hex offsets relative to the start of the header.

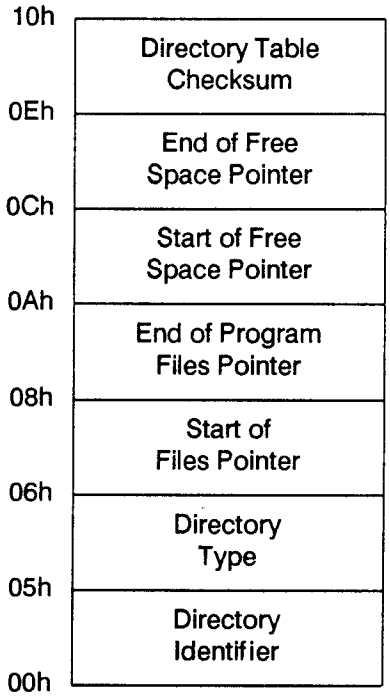


Figure 1-5. Directory Table Header Contents

Refer to the memory maps to see the areas of memory that the pointers refer to.

- Directory Identifier
  The directory identifier always contains the characters `*DIR*`. The operating system uses this to help verify memory integrity.

- Directory Type
  The directory type is the character `M` for main memory, `A` for a 40K RAM card, or `O` for a ROM/EPROM card.

- Start of Files Pointer
  This segment address points past the end of the directory table, and is the beginning of all files. Program files always appear first in the file system.

- End of Program Files Pointer
  This segment address points past the end of the program files, which is the beginning of the data files. Nothing below this address within the directory will move at runtime.

- Start of Free Space Pointer
  This segment address points past the end of the data files, which is the beginning of the free space. Free space is used for data files and scratch areas in main memory, for data files only in a RAM card, and is not available for use in a ROM or EPROM.

- End of Free Space Pointer
  This segment address points past the end of free space. For main memory, it also marks the beginning of scratch areas available for assembly language programmers. If no scratch areas have been allocated, this pointer points past the last byte in main memory — to 1000:0000 (64K), 2000:0000 (128K), or 4000:0000 (256K).

  For the 40K RAM card, this pointer points past the end of the card, since there are no scratch areas. For the same reason, in a ROM, this pointer points past the end of the logical ROM.

- Directory Table Checksum
  This is where the checksum of the directory table is saved when the machine is turned off.

The other entries in the directory table identify the different files. The contents of the directory table entries for files is shown below. Again, the numbers are hex offsets from the start of the entry.

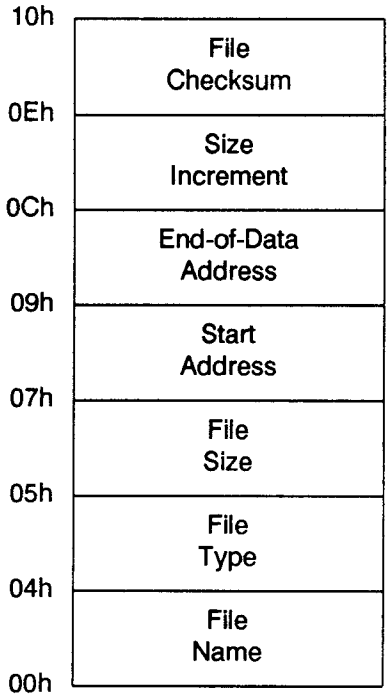


Figure 1-6. Directory Table Entry Contents

- File Name
  This is the name of the file. File names are 1-4 characters long, padded with blanks. If the file had a checksum error at power on, the high bit is set in the first character of the file name (except in ROM files). If a directory table entry is unused, the first byte of this field is set to null (00h).
- File Type
  This is either an A, B, D, or H.
- File Size
  This is the current length of the file in paragraphs. All files are padded with nulls (00h) to the nearest paragraph boundary.
- Start Address
  This segment address is the location where the file starts.
- End-of-Data (EOD) Address
  For data files, this is the offset of the end-of-data within the file, relative to the start of the file. For program files, this is a pointer to the end of the program, which may not be the end of the file because of the null padding. The EOD address is a 24-bit value stored as a two-byte offset and a one-byte segment (low word followed by high byte).
- Size Increment
  For data files, this is the expansion increment, in paragraphs, used when data is written past the end-of-file. It is 0 for program files in RAM and for all files in ROM.

- File Checksum
  This is where the checksum of the file is saved when the machine is turned off.

The space reserved for the directory table is fixed-length, and varies with the total amount of memory. Because the first entry is always reserved for the directory table header, there will be space for one less user file than the size of the directory table would otherwise indicate. The directory sizes and number of files available are shown below.

Table 1-3. Directory Table Sizes

Name of Memory Area	Memory Size	Directory Table Size	Number of Files
Main Memory	64K	1K	63
	128K	1K	63
	256K	2K	127
40K RAM Card	40K	0.5K	31
ROM/EPROM Card	32K	0.5K	31
	64K	0.5K	31
	96K	1K	63
	128K	1K	63

File System

The HP-94 file system allows for multiple files of different types to coexist simultaneously. User files can reside in any of the five user directories (0-4), whether RAM or ROM.

File Names

Each file is identified by a 1-4 character name. File names are composed of uppercase alphabetic characters and numbers only, and must start with a letter. A file name can only exist once in any directory. It is not possible to have the same name but a different type in the same directory. However, the same file name can exist in different directories, with either the same or different type.

File Types

There are four possible file types:

- Assembly Language Program — Type A
  Assembly language programs are either new BASIC keywords, invoked with the %CALL statement, or are entire assembly language applications.
- BASIC Program — Type B
  BASIC programs are a collection of "tokens" that are can be executed by the BASIC interpreter. They are produced by HXC from a BAS file during the file conversion process.

- Data File — Type D
  Data files are simply contiguous blocks of memory.
- User-Defined Handler — Type H
  A handler is a special assembly language program that controls the I/O ports, such as the serial and bar code ports. It has a structure similar in concept to a UNIX or MS-DOS device driver.

Erasing and Loading Files

When files are erased from command mode with the E (*erase*) operating system command, their memory is returned to free space, and files higher in memory move down to fill in the hole. When files are loaded with the C (*copy*) operating system command, existing files with the same name are erased first, and the memory they occupied is reclaimed for other uses. Then memory for the new file is allocated from free space (assuming there is enough room). This ensures that neither file space nor free space are fragmented while erasing or loading files. When data files are deleted with the DELETE function (14h), the memory they occupied is also reclaimed.

Reserved File Names

There are four files with reserved names that must not be used for anything except their current use:

- SYBI — built-in BASIC interpreter
- SYBD — BASIC debugger
- SYFT — user-defined font
- SYOS — built-in operating system

When the BASIC interpreter searches for user-defined keywords with %CALL, the 12 built-in keywords starting with SY will be *not* be overridden by new keyword files of the same name (SYAL, SYBP, SYEL, SYER, SYIN, SYLB, SYPO, SYPT, SYRS, SYRT, SYSW, and SYTO).

In general, Hewlett-Packard uses SY as the first two characters of all its assembly language utilities, and HN as the first two characters of all its user-defined handlers. If you use file names starting with SY or handler names starting with HN, you may have a name conflict. Consequently, you should not use names starting with those characters.

Maximum Number of Files

The maximum number of files that can be placed in any directory was indicated in "Memory Organization" and "Directory Table". The maximum total number of files would occur in a 94D or 94E with a ROM/EPROM card containing four 32K ROMs — 63 files for main memory plus 4 * 31 files for the ROM/EPROM card, for a total of 187 files.

Data Files

Data files are contiguous blocks of memory with a 1-4 character file name name, and file type D. They have no explicit record structure associated with them — it is the responsibility of the application program to impose any record structure needed, and read and write data from the appropriate position within the file. They always appear after all program files in whichever directory the data file resides — between the end of program files pointer and the start of free space pointer.

Data files are created using the CREATE function (11h). When a data file is created, the space requested is taken from free space at the end of the current data files, the directory table header pointers are adjusted, and one entry in the directory table is used to identify the file. Once a file is created, it must be opened with the OPEN function (0Fh) before data can be read or written. Data files are automatically closed at cold start. Data files that were open when the machine was turned off remain open at warm start.

Data files have two characteristics that are defined by the program that creates them (*file size* and *size increment*) and two that are defined automatically (*end-of-data address* (EOD) and *file access pointer*).

File Size

This is the initial size of the file, which is the amount of memory that will be reserved for the file when it is created. It is specified in paragraphs and ranges from 0000h to FFFFh (although the maximum file size is limited by available memory). The space used for the file is automatically initialized to all nulls (00h). A file size of 0 means that the file initially occupies no space, even though the directory table entry still exists to identify the file.

Data files cannot be created in a ROM or EPROM, or in any read-only directory (main memory or the 40K RAM card may be set to be read-only if a checksum error occurred in their directory tables at power on).

Data files can also be created on the development system. Like all development system files, they are converted to Intel MDS format by HXC for transmission to the 94. When no file size is specified, HXC automatically sets it to the actual file size on the development system, rounded up to the nearest paragraph boundary. The 0 to 15 bytes needed to pad the file are automatically set to nulls (00h).

For RAM data files, HXC allows specifying a file size that is larger than the actual size. That way a file could be defined to have a certain amount of data in it, and a fixed amount of unused space in the file. This option is not available for ROM data files, since a program cannot write to unused space in a ROM or EPROM.

Size Increment

This is the expansion increment used to increase the file size when the WRITE function (13h) attempts to write past the end of the file (that is, when the current file size is exceeded). It is specified in paragraphs, and ranges from 0000h to FFFFh (although the maximum expansion is limited by available memory). When a program writes to a data file, and there is no room for the data being written, the operating system will attempt to expand the file by the number of size increments needed, and then the data will be written to the file. For example, a file with a size increment of three (3) paragraphs will expand by as many three-paragraph blocks of memory (48 bytes) as needed to accommodate the

data being written.

Note that the 94 may run out of memory during any of the expansions, leaving a file that has been expanded, but not enough to hold the data to be written. In this situation, no data will be written to the file — data is only written to a file if there is enough room for all it.

When a data file expands, all data files higher in memory move up to accommodate the increased file size. This is illustrated below.

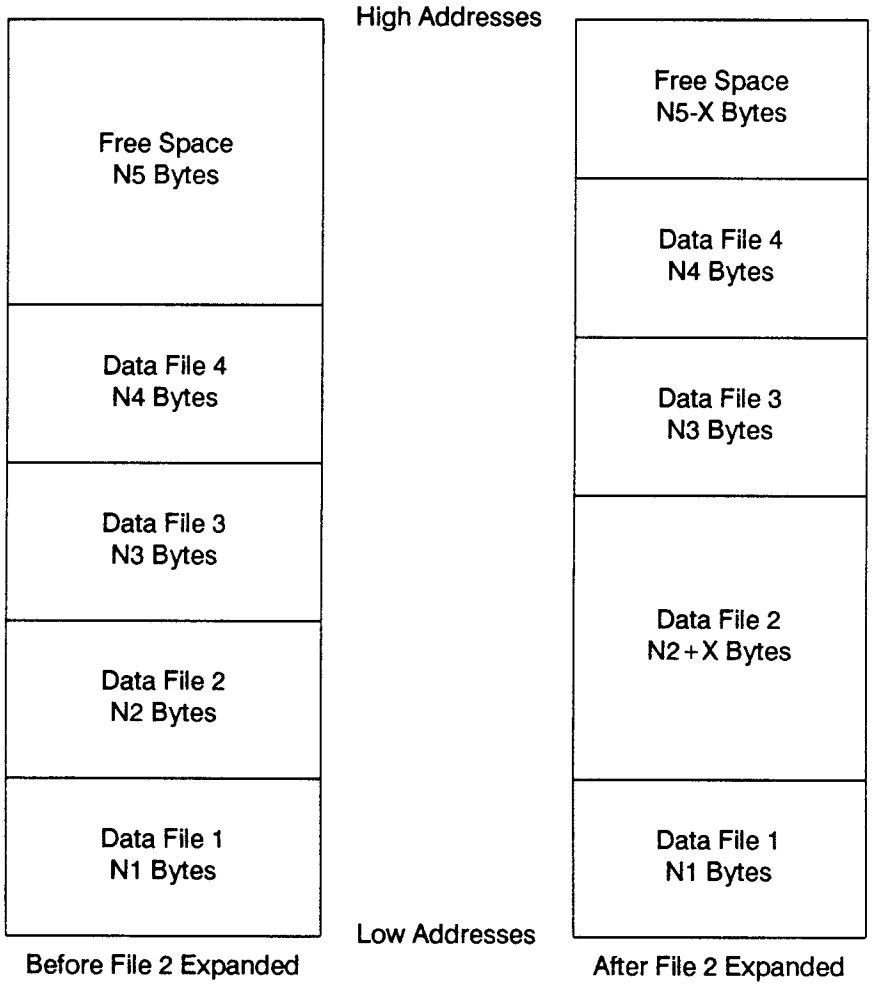


Figure 1-7. File Movement During Data File Expansion

Expansion space added to the file is automatically initialized to all nulls (00h). A size increment of 0 means no expansion will take place — the file will never grow past its allocated size. A size increment of 0 can be specified for any RAM data file; HXC automatically sets it to 0 for ROM data files, since they cannot expand.

File writes are not buffered — they immediately modify the file, provided space is available.

End-of-Data Address

The EOD address is a pointer in the directory table to the location in the data file just past the last byte of data. It is usually not equal to the end of the file (EOF) because files always end on a paragraph boundary. For data files from the development system, HXC sets the EOD address past the last byte of data, even if there is padding to the paragraph boundary or unused space specified beyond the actual file size.

Every time a file write operation writes data past the current EOD or EOF, the EOD is automatically adjusted to reflect the new end-of-data location.

File Access Pointer

This is the single pointer to the current read/write position in the file. The pointer is set to 0 (the start of the file) when the file is opened, and is updated after every file read or write operation. Every time a read or write occurs, the pointer is changed to point past the last byte read or written. Subsequent file read or write operations will begin reading or writing from that updated position. The pointer can be explicitly moved to an arbitrary position between the start of the file and the EOD, or set to the EOD by using the SEEK function (15h). Moves beyond the EOD give an error. It is also possible to force the EOD to be equal to the current file access pointer by performing a zero-length write using the WRITE function (13h). This renders any data after that point inaccessible, but does not collapse the file.

Deleting Data Files

Data files are deleted with the DELETE function (14h), and must be open before they can be deleted. When data files are deleted, all the space occupied by the file is returned to free space. All data files higher in memory move down to fill in the hole. The file space is then available for new data file creation, data file expansion, or scratch area allocation.

Interrupts During File Operations

The power switch and low battery interrupts are disabled during file create, read, write, and delete operations, so they are guaranteed to complete and not be corrupted (unless the reset switch is pressed or the machine turns off automatically because of very low battery). The interrupts are reenabled after the file operation is completed. This disabling and enabling does not change the interrupt status defined by the SET_INTR function (0Ah). What it does is defer the processing (or ignoring) of those interrupts until after the file operation has been completed.

The system timeout only occurs during read operations for channels 0-4 and read/write operations for channels 1-4, so it will not occur during file operations, which use channels 5-15.

File Expansion Example

Assume a data file exists with a current size of 2 paragraphs (32 bytes) and a size increment of 3 paragraphs (48 bytes). The file already contains 25 bytes of data, leaving the EOD at offset 25 relative to the start of the file (the first byte of the file is at offset 0, and the EOD points past the last byte of data). For this example, assume the file access pointer is also at EOD.

When a program tries to write 66 bytes at the file access pointer, there is no room — there are only 7 bytes available. The amount of space required is 66 - 7 = 59 bytes, or 4 paragraphs. Since the size increment is 3, two expansions of 3 paragraphs each will be performed, with a resulting file size of 2 + 2 * 3 = 8 paragraphs (128 bytes). Once the expansion has been completed, the data will be written. The EOD (and the file access pointer) will be moved to offset 25 + 66 = 91, leaving 37 bytes of unused space available at the end. This change to the data file is illustrated below (both decimal and hex offsets are shown).

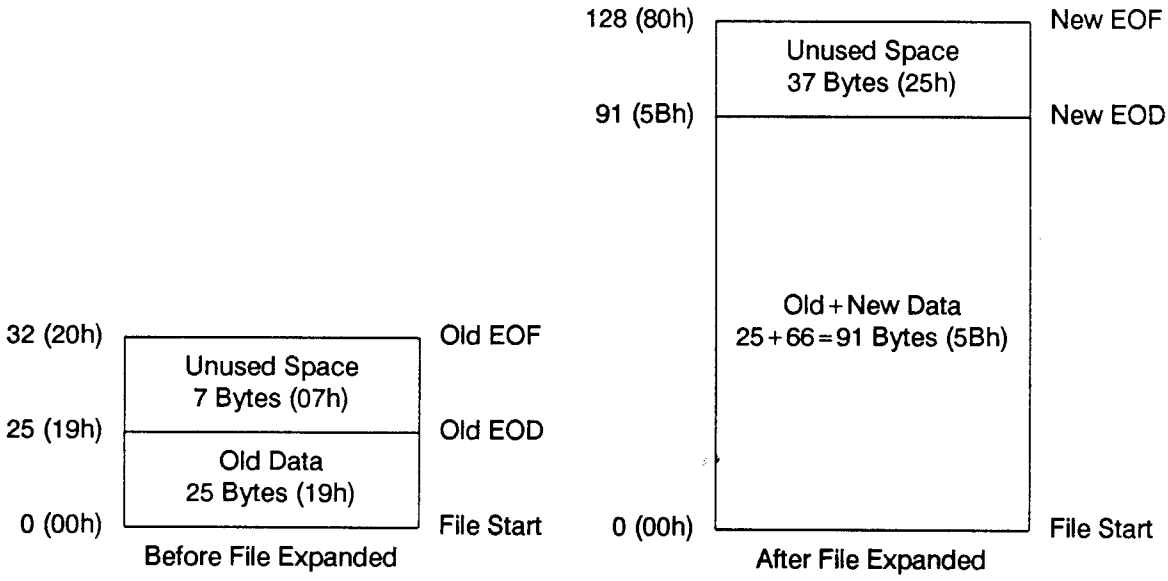


Figure 1-8. Example of Data File Expansion

If the file access pointer had been at the start of the file before the write operation, only a single 3-paragraph expansion would have been needed to accommodate 66 - 32 = 34 bytes.

Free Space

Free space is the pool of available memory from which data files are created and expanded in RAM (main memory and 40K RAM card) and scratch areas are allocated (main memory only). Free space is not available for any use in a ROM or EPROM. It starts at the start of free space pointer in any directory, which is the end of all data files, and ends at the end of free space pointer, which will be at the end of the directory (for main memory only, it could also be at the start of the scratch areas).

In any directory, data files are created and expand by allocating the required memory from the bottom of free space, expanding toward higher addresses. In main memory, scratch areas are created by allocating the required memory from the top of free space, building toward lower addresses, as shown below.

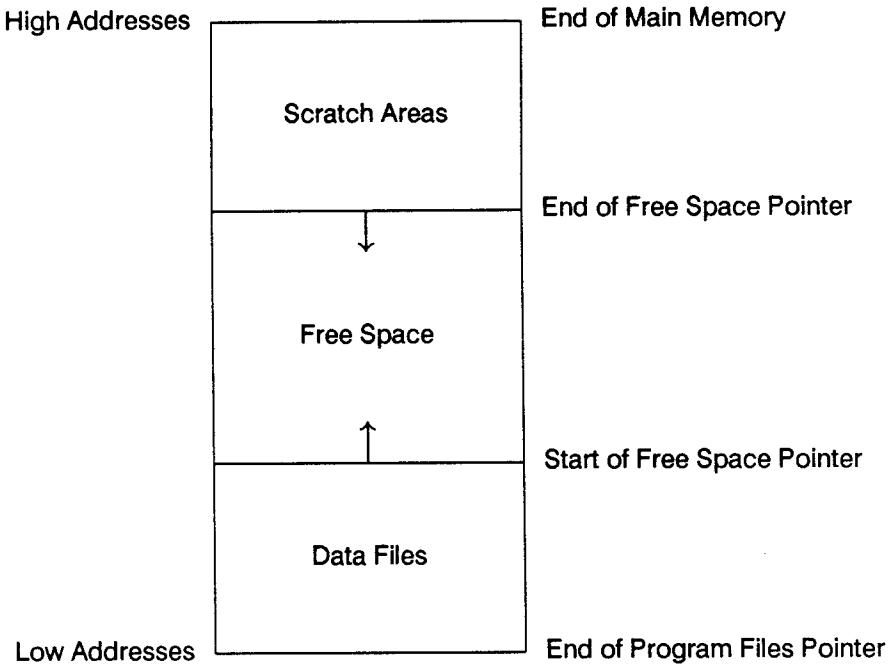


Figure 1-9. Use of Free Space in Main Memory

When the free space goes to zero from either direction, the 94 is out of memory. No data files can be created or expanded, and no more scratch areas can be allocated. The ROOM function (0Eh) reports the amount of free space in any directory; in main memory, it will take into account any existing scratch areas.

Usage in Command Mode

Whenever the operating system enters command mode, all scratch areas in main memory are eliminated, allowing the free space in directory 0 to extend to the end of main memory. The available memory for all directories is then just the size of the free space.

When any RAM file is erased with the E (*erase*) command, the space occupied by that file is returned to free space, and all files higher in memory, regardless of type, are moved down to fill in the hole. When a new file is loaded using the C (*copy*) command, a previously existing file with the same name is erased, and the memory it occupied is reclaimed. Then space for the new file is allocated from free space, and the new file is loaded. If the file loaded is a program file, all files above the end of program files pointer are moved up to make room for the program. If the file loaded is a data file, it is added at the end of the existing data files, and other files do not need to move.

Usage at Run Time

During a running program, there may be scratch areas allocated in main memory, so free space in directory 0 extends only up to the start of the scratch areas. The available memory for other directories is still just the size of the free space.

At run time, program files do not move — only data files and scratch areas interact with free space at run time. When a RAM data file is deleted programmatically, the space occupied by that file is returned to free space, and all data files higher in memory are moved down to fill in the hole. When a new data file is created programmatically, its memory is allocated from free space at the end of the existing data files. When a data file expands because of a write past its end-of-file, the expansion space is allocated from free space, and all data files higher in memory are moved up to make room for the expanded file.

When a scratch area is created, its memory is allocated from free space. When scratch areas are released, their memory is returned to free space only if the area is adjacent to the top of free space. See "Releasing Scratch Areas" for more details.

Scratch Areas

Scratch areas are blocks of memory that a program can reserve for its own use. The built-in BASIC interpreter allocates scratch areas to hold BASIC program variables and subprogram calling information. User-written assembly language programs and user-defined handlers can allocate scratch areas for parameters, status, configuration information, buffer space, space for data returned by operating system functions, or whatever other purpose is required.

Allocating Scratch Areas

The operating system GET_MEM function (0Bh) provides the ability to allocate scratch areas in sizes from 0001h to FFFFh paragraphs (although the maximum expansion is limited by available memory), and returns the segment address of the scratch area. Scratch areas are allocated in main memory only, regardless of which directory contains the program requesting the scratch area: directories 0-4, RAM or ROM. Scratch areas start at the end of main memory and use the space required from free space, building down toward lower addresses. They can also use previously-released scratch areas that have not been returned to free space. This will be discussed later.

Scratch areas are automatically initialized to all nulls. They are all released at cold start, but are preserved at warm start.

When a handler allocates a scratch area during its OPEN routine, the operating system saves the scratch area address in a table based on the channel number of the handler. When the other routines in the handler are called (such as READ, WRITE, etc.), the operating system passes the scratch area address to the routine. (The handler must save this address in the handler information table if it will be needed for an interrupt service routine.)

If a handler allocates more than one scratch area, only the address of the last one allocated will be saved and automatically passed to handler routines. Therefore, when multiple scratch areas are allocated by a handler, the allocation order is important. A handler can allocate scratch areas so that the last one allocated is the one whose address should be passed to handler routines. Alternatively, the

handler can call GET_MEM with the channel number set to 0, and the operating system will not save that scratch area address or pass it to handler routines.

When an assembly language program allocates scratch areas, it is responsible for keeping track of the locations of its scratch areas. The operating system saves scratch area addresses only for user-defined handlers.

The assembler provides the ability to define the offsets within an external scratch area using the SEGMENT AT directive, as shown below.

```
SCR_AREA        segment at 0            ;Addresses start at 0
PARAM1          db      6 dup(?)        ;First parameter needs 6 bytes
PARAM2          db      00              ;Second parameter needs a byte
PARAM3          dw      0000            ;Third parameter needs a word
SCR_AREA        ends
```

Figure 1-10. Defining Scratch Area Data Structure

The SEGMENT AT directive provides an address template that can be imposed on the scratch area. SEGMENT AT causes no code to be generated for the uninitialized data defined within that program segment (in this case, the SCR_AREA segment).

Releasing Scratch Areas

Scratch areas are released using the REL_MEM function (0Ch). The program supplies the address of the scratch area to be released. An error will occur if the program tries to release a scratch area that does not exist by supplying an address that does not point to any defined scratch area.

When a scratch area is released, the operating system will attempt to return the area to free space. This can only occur if the scratch area is adjacent to free space. Consequently, it may not be possible to return a scratch area to free space because of the order that the scratch areas were allocated.

For example, if a handler is opened in a BASIC subprogram, and allocates a scratch area, the area will be adjacent to free space, and will be lower in memory than the scratch area allocated by the subprogram for its variables. When the subprogram ends, the scratch area used for its variables will be released, but will not be returned to free space. It is blocked from being adjacent to free space because of the handler's scratch area. This area is flagged as a free block, available for scratch area allocation, but not for data file creation or expansion since it is not part of free space.

In the diagram below, scratch area 3 was allocated for variables for a BASIC subprogram, and scratch area 4 by a handler.

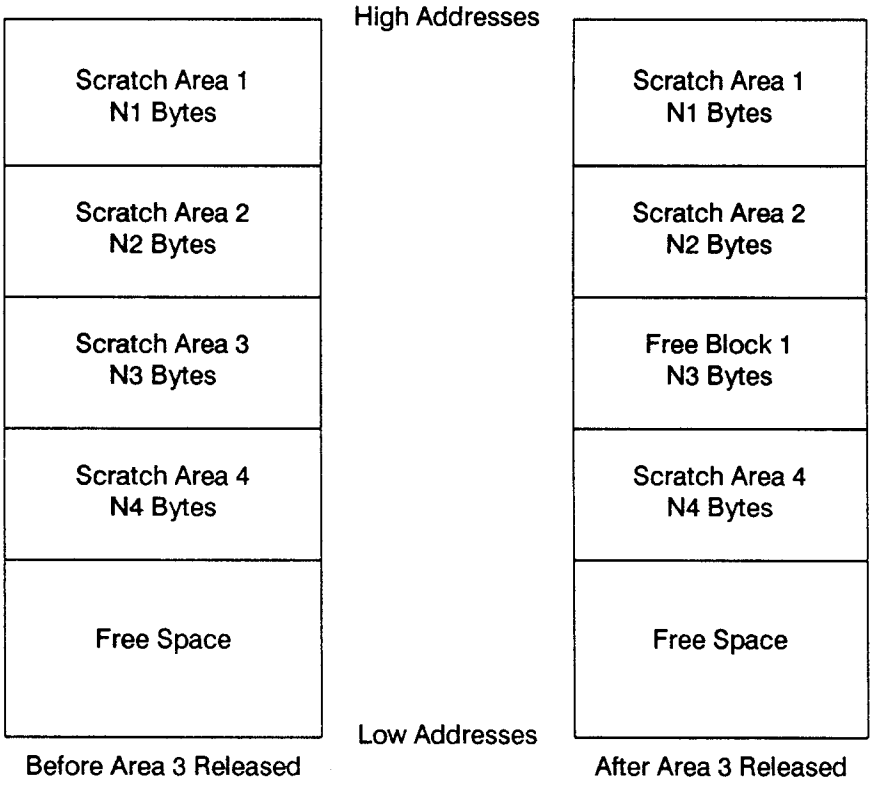


Figure 1-11. Blocking a Released Scratch Area

Scratch area 4 prevents released scratch area 3 from being returned to free space. Scratch area 3 becomes the first free block. It will not be returned to free space until scratch area 4 is released.

To allow this newly-available free block to be reused, regardless of the order in which scratch areas were allocated and released, it will be combined with any adjacent free blocks formed when other trapped scratch areas were released. This coalescing process attempts to form a few large available free blocks, rather than many small ones. This is illustrated below.

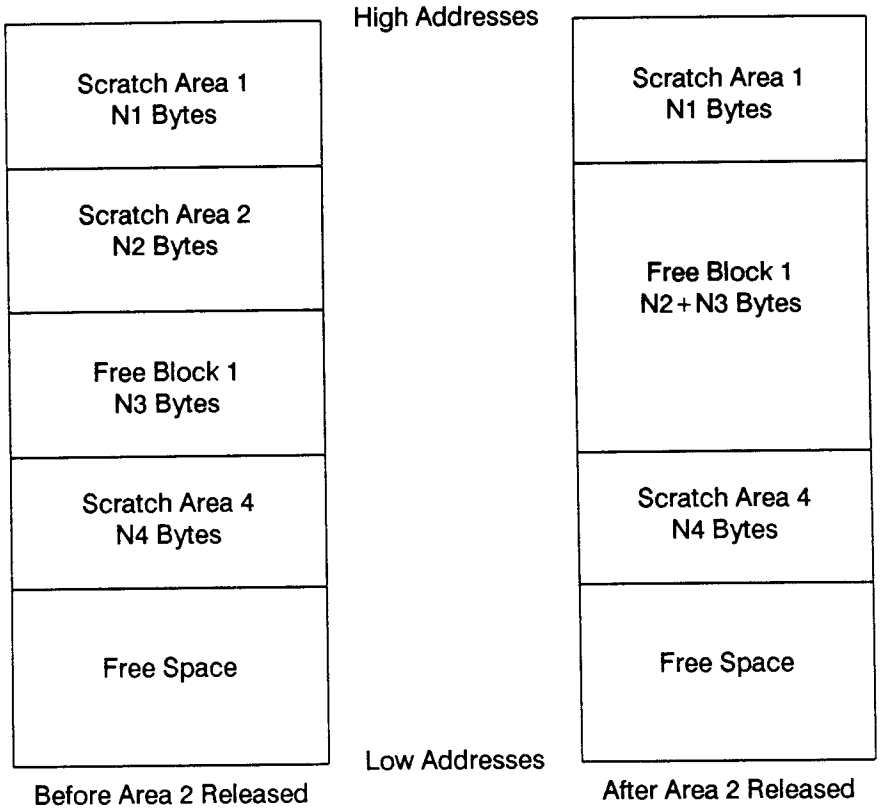


Figure 1-12. Coalescing Adjacent Released Scratch Areas

When scratch area 2 is released, it forms a new free block that cannot be returned to free space. The coalescing process combines this new block with free block 1 that already exists, forming a single free block whose size is the sum of the two smaller blocks. This keeps the number of free blocks to a minimum, since the operating system can only keep track of 20 free blocks.

Subsequent allocation of new scratch areas will use the first free block that is large enough among all those available before allocating additional memory from free space. Only as much of the free block will be used as is required. The remainder will be flagged as a smaller free block.

Data files cannot use free blocks until they are returned to free space — only scratch areas can reuse free blocks. Consequently, free space can go to zero and leave no room for data files creation or expansion, even though there may be free blocks available for reuse when allocating scratch areas.

There is no facility to pack the free blocks together, since many tables and handlers keep track of the segment address of the their scratch areas. Only allocation and release of scratch areas in careful order can help prevent fragmentation of free blocks.

After the coalescing has been completed, if there is an available free block adjacent to free space, it is returned to free space for other uses (data file allocation and expansion or new scratch area allocation when the available free blocks are not large enough).

When the 94 cold starts, all scratch areas and free blocks are automatically returned to free space. This will occur the next time the machine is turned on after a program calls the END_PROGRAM function (00h) and specifies a subsequent cold start. This also occurs whenever the operating system enters command mode, whether because of a program error or because of an explicit call to END_PROGRAM. If a program calls END_PROGRAM and specifies a subsequent warm start, all scratch areas and free blocks are preserved the next time the machine is turned on.

Number of Scratch Areas

A maximum of 34 scratch areas can be allocated in main memory. An error will occur when a scratch area is allocated if 34 scratch areas are already in use.

The BASIC interpreter allocates scratch areas for its own use, for BASIC variables, and for control information. In this sense, the BASIC interpreter can be thought of as another assembly language program, using the facilities within the operating system for scratch space management.

When a BASIC main program is run, two scratch areas are allocated immediately:

- One scratch area for the BASIC interpreter scratch space (2K long).
- One scratch area for the BASIC program variables. The length of this area is shown as "Variable Space Required" in the BMP file produced by HXC (although the length is rounded up to the nearest paragraph boundary). This area will not be allocated in the case of a BASIC main program with no variables.

This leaves a total of 32 scratch areas available for other uses. After that, every time a BASIC subprogram is called with the CALL statement, two scratch areas are allocated:

- One scratch area for the control information save area that contains information passed between programs (32 bytes).
- One scratch area for the BASIC subprogram variables (length shown in the BMP file, not allocated if no variables).

This is why BASIC subprograms can only be nested a maximum of 16 levels deep — scratch area allocation limits permit 32 scratch areas beyond those used for the main program.

Fewer scratch areas may actually be available for BASIC subprogram nesting, since user-defined handlers and assembly language programs can allocate scratch areas also. A high-level and low-level handler combination, for example, may have three scratch areas allocated between them: one for configuration passing and two for scratch and buffer space (one for each handler). Assembly language programs generally allocate one scratch area for scratch and buffer space, but may allocate a second one for configuration passing to handlers. Consequently, BASIC subprogram nesting may be restricted to less than 16 levels.

Optimum Memory Use With Scratch Areas

To allow the most efficient use of memory, scratch areas should be allocated and released in such a way that they do not block other scratch areas from being returned to free space. Long-term scratch areas that must remain in place throughout program execution (such as handler scratch areas) should be allocated when the program begins executing. Short-term scratch areas should be released as soon as they are not needed.

This is particularly important for BASIC programs. BASIC programs should attempt to do tasks that allocate long-term scratch areas in the main program, rather than in subprograms, where they will trap short-term subprogram-related scratch areas. Whenever possible, tasks requiring short-term scratch space should be isolated within a subprogram.

Logical ROMs

The HP 82412A ROM/EPROM card accommodates ROMs or EPROMs of different sizes: 32, 64, 96, or 128K. These different sizes are considered to be "logical ROMs" for two reasons:

- A logical ROM of size *N* does not have to contain *N* bytes of program and data files; it can contain less than *N* bytes. For example, a 64K logical ROM may only contain 44K of program and data files.
- A logical ROM of size *N* does not have to be placed in a ROM or EPROM integrated circuit (IC) of size *N*. For example, a 96K logical ROM can be contained in either three 32K ICs or two 64K ICs.

Logical Structure of the ROM/EPROM Card

Below is a memory map of the ROM/EPROM card.

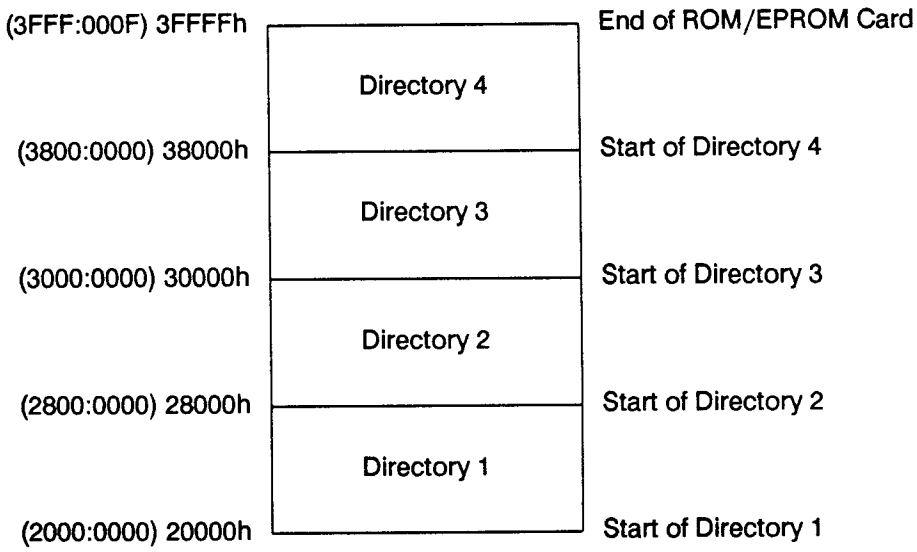


Figure 1-13. Memory Map of the HP 82412A ROM/EPROM Card

This memory map illustrates an important aspect of logical ROMs. Each directory begins on a 32K address boundary within the ROM/EPROM card address space (20000h to 3FFFFh). Each logical ROM is assigned a directory number corresponding to the 32K address boundary where the ROM will start. A logical ROM larger than 32K will span more than one 32K block of addresses. The pointers in

the directory table header created by HXC will reflect that the starting address is on a 32K boundary, and that the logical ROM space for large ROMs spans multiple 32K blocks. (For ROMs that span more than one directory, the directory number specified when the ROM is created is the starting directory number.)

For example, a 96K logical ROM starting at directory 1 will span directories 1, 2, and 3, leaving one 32K block of addresses, directory 4, available for a single 32K logical ROM. Similarly, a 64K logical ROM starting at directory 3 will span directories 3 and 4, leaving two 32K block of addresses, directories 1 and 2, available. These can be filled by either another 64K logical ROM starting at directory 1, or two 32K logical ROMs, one starting at directory 1, and the other starting at directory 2. A 96K logical ROM could not start at directory 3, nor could a 64K logical ROM start at directory 4, because they would have to span into a 32K block of addresses not available to the ROM/EPROM card.

Combining Logical ROMs of Different Sizes

Logical ROMs of different sizes can be combined in many different ways, subject to the following restrictions:

- The total number of logical ROMs cannot exceed four.
- The total number of directories spanned by all the logical ROMs cannot exceed four.
- The total space required by all logical ROMs, regardless of the amount of code they contain, cannot exceed 128K.

This is illustrated by the following diagram, which shows the possible logical ROM combinations for filling 128K of ROM space. Of course, a ROM/EPROM card does not have to be full — that is, it can contain fewer than four logical ROMs, span fewer than four directories, and contain less than 128K total ROM.

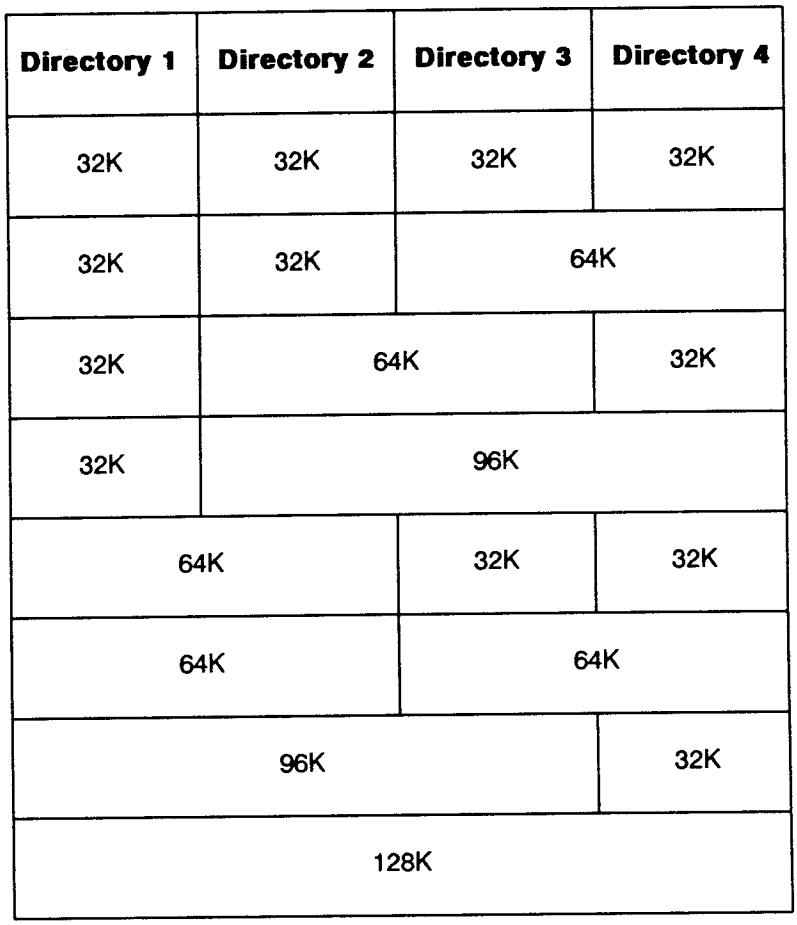


Figure 1-14. Possible Logical ROM Configurations

The memory map of an individual ROM within the ROM/EPROM card is essentially the same as for the 40K RAM card. The major difference is the values of the pointers — these can vary depending on the starting directory number, the directory table size, and the logical ROM size. Below is a memory map of a 32K logical ROM starting at directory 2.

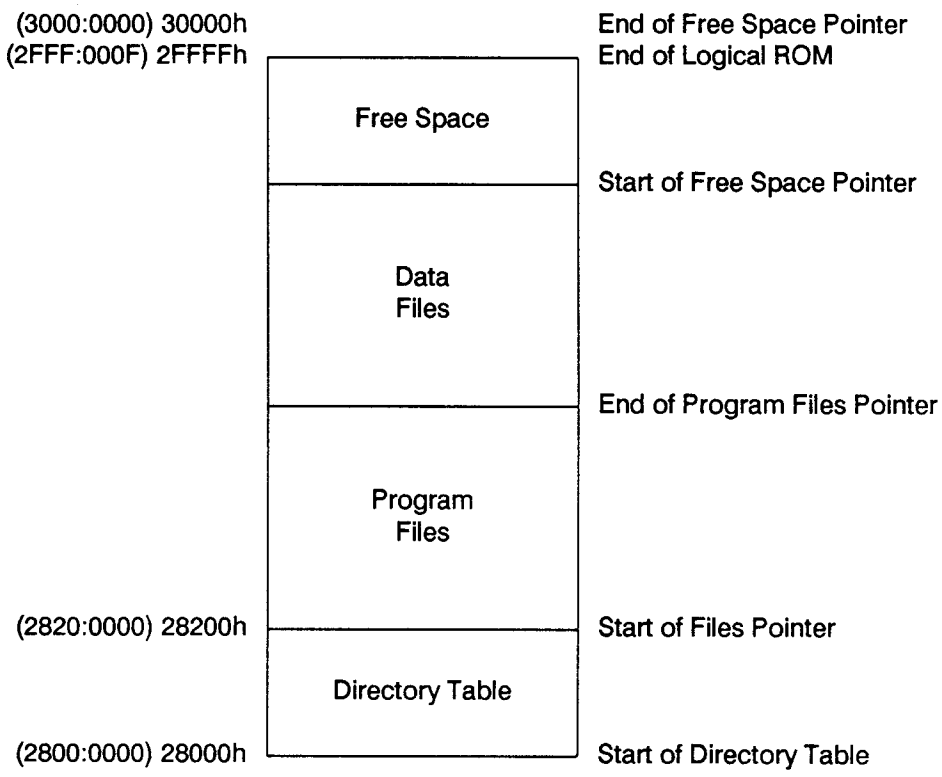


Figure 1-15. Memory Map of a 32K Logical ROM in Directory 2

Rather than provide memory maps for all the possible logical ROMs in directories 1-4, the addresses of the start and end of the logical ROM and for the start of program files (end of directory table) are shown in the following table.

Table 1-4. Addresses for All Logical ROM Sizes in Directories 1-4

Logical ROM Size	Directory Number	Start of Logical ROM	Start of Program Files Pointer	End of Free Space Pointer
32K	1	2000:0000	2020:0000	2800:0000
	2	2800:0000	2820:0000	3000:0000
	3	3000:0000	3020:0000	3800:0000
	4	3800:0000	3820:0000	4000:0000
64K	1	2000:0000	2020:0000	3000:0000
	2	2800:0000	2820:0000	3800:0000
	3	3000:0000	3020:0000	4000:0000
96K	1	2000:0000	2040:0000	3800:0000
	2	2800:0000	2840:0000	4000:0000
128K	1	2000:0000	2040:0000	4000:0000

Selecting a Logical ROM Size

From the different possible logical ROM sizes, select those best for a specific application based on its particular needs. Some of the items to consider are the total number of program and data files needed, maximum file size, total ROM space required for directory tables (which decreases available ROM for the application), and segmentation of code into blocks of different sizes. Below is a comparison of the differences in organizing a 96K application in three different ways: three 32K ROMs, one 64K ROM and one 32K ROM, or one 96K ROM.

Table 1-5. Different Organizations of a 96K Application

Logical ROM Sizes	Total Number of Files	Maximum File Size	Directory Table Overhead	Segmentation Required
3 32K ROMs	3 * 31 = 93	31.5K	3 * .5 = 1.5K	three separate groups of files that each fit in 32K
1 64K ROM + 1 32K ROM	31 + 31 = 62	63.5K, 31.5K	.5 + .5 = 1K	one group of files that fits in 64K and one group of files that fits in 32K
1 96K ROM	63	95K	1K	none

The same reasoning can be applied to other size applications and other logical ROM choices. The results of this analysis should be matched up against the requirements of the application to select the best way to organize it.

ROM and EPROM IC selection is another factor to consider, and will be discussed later.

Physical Layout of the ROM/EPROM Card

The ROM/EPROM card contains a circuit board with three sockets on it for ROM or EPROM ICs. The sockets can accommodate either 32K ICs or 64K ICs (a jumper on the board selects which IC size is being used). Different IC sizes cannot be mixed and matched — the board can hold either up to three 32K ICs or up to two 64K ICs. A diagram of the card is shown below.

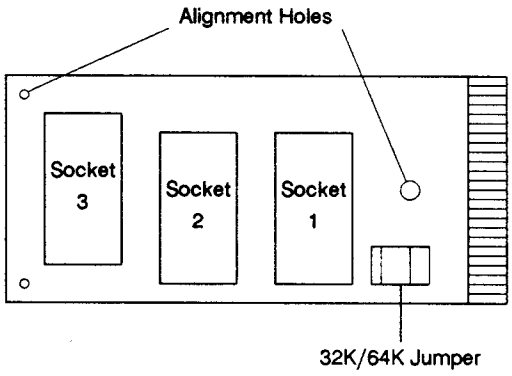


Figure 1-16. HP 82412A ROM/EPROM Card Circuit Board

The socketed jumper on the board selects between 32K ICs and 64K ICs. Underneath the jumper are the legends [256] and [512], meaning 256 Kbits (32 Kbytes) or 512 Kbits (64 Kbytes). To select the 32K ICs, insert the jumper so its solid metal strips connect jumper pins whose mating holes on the board are marked with the [256] symbol. (This is the configuration shown in the diagram.) To select 64K ICs, insert the jumper to use the holes marked with the [512] symbol.

Each socket on the board begins on a 32K address boundary within the ROM/EPROM card address space corresponding to the 32K blocks of address space in which logical ROMs reside. Socket 1 corresponds to directory 1, 2 to 2, and 3 to 3. A 32K IC can therefore be placed in any socket on the board (1, 2, or 3). A 64K IC will span more than one 32K block of addresses. Consequently, 64K ICs can be placed only in sockets 1 and 3. Placing a 64K IC in socket 3 gives access to the fourth 32K block of addresses — this is the "fourth" socket on the board for directory 4.

This means that using 32K ICs, 96K of physical ROM space is the maximum available, and using 64K ICs, the full 128K is available.

Selecting an IC Size

The directory numbers selected for the different logical ROMs will depend on where the logical ROMs will be placed on the board in the ROM/EPROM card. Some of that will depend on which IC size is chosen. The following items should be considered when making an IC size selection:

- Application size
- Price
- Availability
- Correct electrical specifications
- Supported by EPROM programmer (EPROMs only)

Refer to the "Hardware Specifications" for information about electrical and environmental specifications and manufacturers for the different IC sizes.

Placing Logical ROMs Into Physical ICs

In addition to the previous restrictions on combining logical ROMs, and the fact that IC sizes cannot be mixed, there is one more restriction that applies when placing logical ROMs into physical ICs: the physical IC must be placed in the socket on the board which corresponds to the directory number for the logical ROM contained in that IC.

Logical ROMs and physical ICs can both span 32K address boundaries, but this spanning is independent of each other (with the above restriction). This fact yields two important results. First, a logical ROM can cross physical IC boundaries; if it could not, logical ROMs larger than 32K would not be possible. Second, it does not matter what part of a logical ROM occupies a given physical IC as long as the logical ROM's starting directory number corresponds with the socket it occupies on the board, and the different pieces of the logical ROM are kept in the proper order.

Continuing the previous example of a 96K application, below are the ways that the logical ROMs could be placed in physical ICs. Each row of the tables represents a different way to place the particular logical ROM in the ICs.

Table 1-6. Placing a 96K Application Into Three 32K ICs

Logical ROM Sizes	Which Part of Logical ROM Put In 32K IC In Socket 1	Which Part of Logical ROM Put In 32K IC In Socket 2	Which Part of Logical ROM Put In 32K IC In Socket 3
3 32K ROMs	one entire 32K ROM	one entire 32K ROM	one entire 32K ROM
1 64K ROM + 1 32K ROM	first half of 64K ROM	last half of 64K ROM	entire 32K ROM
	entire 32K ROM	first half of 64K ROM	last half of 64K ROM
1 96K ROM	first third of 96K ROM	middle third of 96K ROM	last third of 96K ROM

Table 1-7. Placing a 96K Application Into Two 64K ICs

Logical ROM Sizes	Which Part of Logical ROM Put In 64K IC In Socket 1		Which Part of Logical ROM Put In 64K IC In Socket 3	
	First Half of IC	Last Half of IC	First Half of IC	Last Half of IC
3 32K ROMs	one entire 32K ROM	one entire 32K ROM	one entire 32K ROM	
		one entire 32K ROM	one entire 32K ROM	one entire 32K ROM
	one entire 32K ROM		one entire 32K ROM	one entire 32K ROM
	one entire 32K ROM	one entire 32K ROM		one entire 32K ROM
1 64K ROM + 1 32K ROM	first half of 64K ROM	last half of 64K ROM	entire 32K ROM	
	first half of 64K ROM	last half of 64K ROM		entire 32K ROM
	entire 32K ROM	first half of 64K ROM	last half of 64K ROM	
		first half of 64K ROM	last half of 64K ROM	entire 32K ROM
	entire 32K ROM		first half of 64K ROM	last half of 64K ROM
		entire 32K ROM	first half of 64K ROM	last half of 64K ROM
1 96K ROM	first third of 96K ROM	middle third of 96K ROM	last third of 96K ROM	
		first third of 96K ROM	middle third of 96K ROM	last third of 96K ROM

As the tables indicate, the segmentation of an application across logical ROM boundaries has no bearing on the way the ROMs are segmented to fit into physical ICs, as long as the starting directory number corresponds with the socket number, and the different pieces of the logical ROM are kept in the proper order.

The same reasoning can be applied to other size applications and other logical ROM choices. The results of this analysis should be matched up against the requirements of the application to select the best way to organize it.

System ROM

The system ROM is 32K of EPROM located in directory 5 in the upper 32K of the CPU address space. While this directory can be examined in command mode, it cannot be referenced by number or by any of its files during a running program. During a running program, the OPEN, FIND_FILE, and FIND_NEXT functions (0Fh, 16h, and 17h) will only find files in directories 0-4. The system ROM

contains four major blocks, shown in the memory map below.

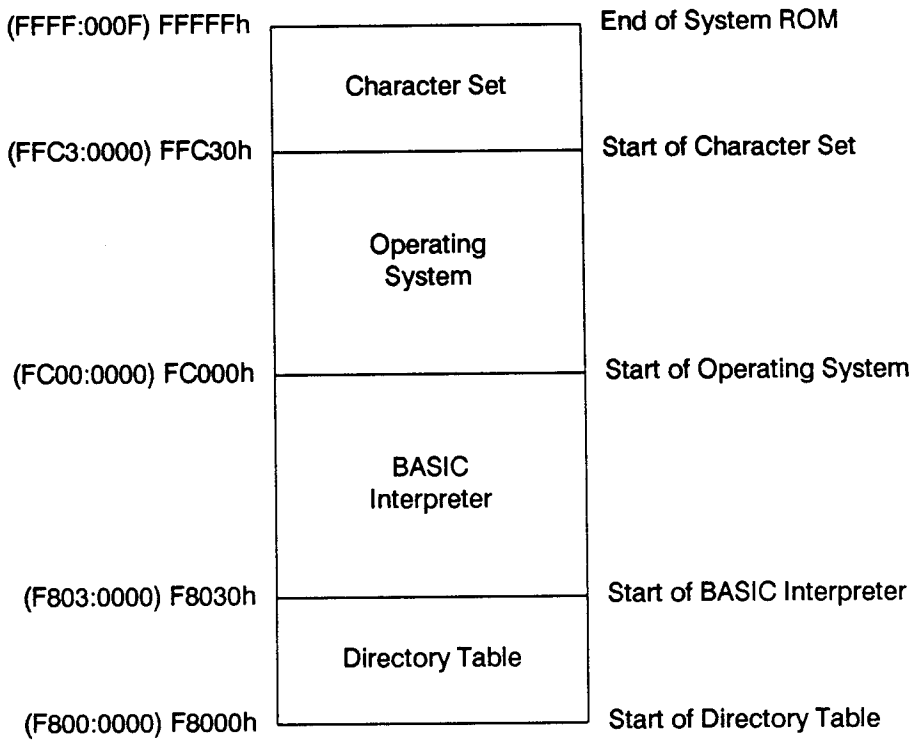


Figure 1-17. Memory Map of the System ROM

- Directory Table
  This contains only three entries: directory table header, BASIC interpreter file entry (SYBI), and operating system file entry (SYOS).
- BASIC Interpreter
  This is file SYBI.
- Operating System
  This is file SYOS.
- Character Set
  This is the dot pattern for the Roman-8 character set.

Memory Integrity Verification

The operating system computes and saves checksums of various areas of memory when the 94 is turned off. When the 94 is turned back on, the checksums are recomputed and compared with the saved values. Any changes indicate that memory integrity has not been preserved, and an error message is issued. Checksums are computed such that the sum of all words in the block being verified, plus the

checksum, will equal zero.

The major blocks of memory for which checksum errors are reported are directory tables, files, reserved scratch space, and free space. In addition, a checksum is made of the system ROM, and the reserved scratch space is tested extensively. These operations are discussed below.

Checksums Computed at Power Off

At power off, checksums for all RAM areas (main memory and 40K RAM card) are computed and saved. Checksums for ROM/EPROM card are not computed, since they are fixed in ROM, but they are saved in the reserved scratch space for comparison at power on. The system ROM checksum is also not computed.

Memory Integrity Tests at Power On

At power on, the operating system checks the main NiCd battery voltage. If it is below the low battery interrupt level, the machine is immediately turned off. If the voltage is OK, integrity tests are performed in the order shown in the following table. If any of the first three tests fail, the machine will not enter command mode. If any of the other tests fail, the machine will enter command mode and issue an error message. Any program run at that time will cold start.

Table 1-8. Memory Integrity Errors

Integrity Test Performed	Main Memory Error	40K RAM Card Error	ROM/EPROM Card Error
System ROM Checksum	low beep	—	—
Reserved Scratch Space Read/Write	high beep	—	—
Valid RAM Configuration	high beep and memory map	—	—
Directory Table Header Consistency	212 and require `I 0`	213 and require `I 1`	—
Reserved Scratch Space Checksum *	214	—	—
Free Space Checksum *	215	—	—
Directory Table Checksums *	212 and make type O	213 and make type O	213
File Checksums *	216 and set MSB of name	217 and set MSB of name	217

* Not computed at power off or power on if power turned off by pressing the reset switch or by automatic turn-off 2-5 minutes after the low battery interrupt.

These tests and their results are described below.

- System ROM Checksum
  If the stored checksum in the system ROM does not match the computed checksum, the operating system will issue a continuous low tone beep, and will not enter command mode.

- Reserved Scratch Space Read/Write
  If every byte in the reserved scratch space cannot be read and written, the operating system will issue a continuous high tone beep, and will not enter command mode.

- Valid RAM Configuration
  The RAM configuration is checked by reading and writing the first word of every RAM IC. If there is any other configuration of built-in RAM than 64K, 128K, 256K, or the RAM card has other than 40K, the operating system will issue a continuous high tone beep, and will not enter command mode.

  In addition, a memory configuration map will be displayed indicating the incorrect RAM ICs. The map is in the form "`Error `" followed by eight hex characters. The bits in each character represent individual RAM ICs. Reading from right to left, each bit will be a 1 if the IC was present, and a 0 if the IC was not present. For example,

  ```
  Error FFFFFFDF
  ```

  indicates that the sixth RAM IC was not present (the last 8 bits of the map are 11011111). Shown below is what the memory configuration map would be if the different configurations were correct. (These patterns will never appear, because only an incorrect pattern will be displayed.)

Table 1-9. Configuration Map for Valid Memory Configurations

Memory Configuration	Configuration Map if Configuration Correct
64K	`Error 000000FF`
128K	`Error 0000FFFF`
256K	`Error FFFFFFFF`
64K+40K RAM Card	`Error 001F00FF`
128K+40K RAM Card	`Error 001FFFFF`

After this test, the operating system will check the keyboard. If any keys are down other than CLEAR and ENTER, the machine will turn back off immediately. This is to prevent accidental turn on (while in a full briefcase, for example).

- Directory Table Header Consistency
  This verifies the consistency of the directory table headers for main memory and the 40K RAM card. The `*DIR*` directory identifier must be intact and the different pointers must point to successively higher addresses. If not, error 212 or 213 is issued, and the directory table is flagged such that the user must initialize the directory with the `I` (*initialize*) command (`I0` or `I1`). This also occurs if the size of main memory has changed (by adding or removing the 128K memory board).

- Reserved Scratch Space Checksum
  This is the checksum of the interrupt vector area and the operating system scratch space. If this checksum error occurred, error 214 will be issued.

- Free Space Checksum
  This is the checksum of the free space (and scratch areas, if any) — everything higher in main memory than the end of free space pointer. If this checksum error occurred, error 215 will be issued.

- Directory Table Checksums
  These are the checksums of the directory tables in any directory. If a directory table checksum error occurred for main memory or the 40K RAM card, error 212 or 213 will be issued, and the directory type in the directory table header will be changed to O (ROM directory). This makes the directory read-only, allowing the data to be retrieved, but not changed. To make the directory table type **M** or **A** again, the user must initialize the directory with the `I` command (`I0` or `I1`) after retrieving any desired data.

  For the ROM/EPROM card, only the error (213) will be issued — the directory type is already type O. Any checksum error in a ROM or EPROM (especially an EPROM) implies that the IC had one or more bits change state, and the IC should be replaced.

  The operating system recognizes that a card has been plugged in or removed, or that ROMs were changed on the ROM/EPROM card, because the number and contents of the directory tables has changed. When these conditions occur, they will not cause a checksum error, but will cause the machine to cold start.

- File Checksums
  These are the individual checksums for each file in any directory. If a file checksum error occurred for main memory or a RAM card, the MSB of the first character of the file name will be set. This will cause the file names to be displayed with a leading asterisk (*) when the D (*directory*) or M (*memory*) operating system commands are executed. If a file name has already been flagged as being corrupted, its checksum will not be computed at power on.

  If a file checksum error occurred in a ROM/EPROM card, the file name will not be altered, so no asterisk will appear when using the D or M commands. Any checksum error in a ROM or EPROM (especially an EPROM) implies that the IC has had one or more bits change state, and the IC should be replaced.

  Even with the MSB set in the file name, all normal file operations can still be performed: open, close, read, write, delete, find, execute, etc. All these operations are risky (especially running corrupted programs) because the state of the file is unknown. Unless the program or the user has the ability to reconstruct corrupted data, the safest action would be to erase the corrupted files and either replace them (program files) or recreate them (data files).

After all memory integrity tests have been performed, the operating system checks the lithium backup battery voltages. If the voltages are too low, the machine will enter command mode, and issue error 210 (main memory) and/or 211 (128K memory board or 40K RAM card).

2

Program Execution

Contents

Chapter 2 Program Execution

2-1 Running Programs
2-1 Autostart
2-1 In-Place Execution of Programs
2-1 Behavior at Run Time
2-2 Behavior of Reserved Files
2-2 Cold Start and Warm Start
2-4 When Cold Start Occurs
2-4 When Warm Start Occurs
2-4 Operating System Activities During Cold Start
2-5 Operating System Activities During Warm Start
2-5 Ending Programs
2-6 Operating System Activities When Entering Command Mode
2-8 Program Structure
2-8 Program Headers
2-9 BASIC Keyword Structure
2-10 Program Restrictions
2-10 Valid EXE Format
2-11 Use of Operating System Stack
2-11 Programs in ROM or EPROM

2

Program Execution

This chapter describes program execution in the HP-94: behavior at run time, cold start and warm start, program structure, and restrictions.

Running Programs

Program files are any of the non-data files — file types A, B, or H. They can reside in RAM or ROM/EPROM, and have some characteristics that are described here. Details on new BASIC keywords (type A) and user-defined handlers (type H) are in the BASIC interpreter and handler sections of this manual. BASIC programs are discussed in the BASIC interpreter section of this manual, as well as in the *BASIC Language Reference Manual.*

Autostart

When the HP-94 cold starts (discussed later), the operating system will automatically run the first file called `MAIN` that it finds. It searches directories 0-4 in ascending order, and if the first `MAIN` file encountered is type A or type B, it will be run; if not, an error will be issued. This search order allows a `MAIN` program in directory 0 (main memory) to override a `MAIN` file in directories 1-4 (40K RAM card or ROM/EPROM card).

Programs can also be run using the S (*start*) operating system command. Programs run with S will always cold start.

In-Place Execution of Programs

Program files are executed in place, regardless of where they are located in memory. Programs in ROM do not have to be copied into RAM before being executed. Space for BASIC program variables and scratch areas for assembly language programs and handlers are allocated from main memory, regardless of which directory the program resides in.

Behavior at Run Time

Program files always appear first in the file system for each directory, as illustrated in the memory maps. This placement occurs regardless of the order in which files are loaded. The C (*copy*) command ensures that all RAM-based program files are located before any data files. HXC ensures the same condition for ROM-based programs.

This is important because program files do not move at run time. All files lower in memory than the

end of program files pointer will not move at run time. However, because the order programs are loaded may vary, it is not known until run time exactly where each file may be located (and therefore what the initial CS will be). There is no segment fixup performed as is true for MS-DOS programs. Consequently, all references to addresses within program files must be relative to the start of the file — there can be no far calls or far jumps. This is particularly important for assembly language programs; HXBASIC and HXC handle this for BASIC programs.

Data files, however, can move at run time, since they can expand and be deleted. Since the operating system assumes that programs do not move at run time, data files must appear after all program files so that data file expansion and deletion will not change the location of programs.

Behavior of Reserved Files

There are four files with reserved names that must not be used for anything except their current use:

- SYBI — built-in BASIC interpreter
  If this file is run with the S (*start*) command, the operating system will immediately return to command mode.
- SYBD — BASIC debugger
  If this file is run with the S command, the operating system will immediately return to command mode (with the side effects shown in Table 2-3 for a FAR RET).
- SYFT — user-defined font
  If this file is run with the S command, the data in the file will be treated as code, which will have unpredictable (and possibly harmful) side effects.
- SYOS — built-in operating system
  If this file is run with the S command, the operating system will immediately turn the machine off.

When the BASIC interpreter searches for user-defined keywords with %CALL, the 12 built-in keywords starting with new keyword files of the same name SY will be *not* be overridden by new keyword files of the same name (SYAL, SYBP, SYEL, SYER, SYIN, SYLB, SYPO, SYPT, SYRS, SYRT, SYSW, and SYTO).

Cold Start and Warm Start

The HP-94 supports two methods of running programs when the machine is turned on: *cold start* and *warm start*. The fundamental difference is where the program starts running.

At cold start, the program starts running at the beginning. All conditions are reset to their default state. At warm start, the program continues running from the point at which it turned the power off. Most conditions are preserved in the state they were in while the program was previously running, although a few are reset to their default state. The warm start state is seen by user-defined handlers when their WARM routines are called.

The details of what state the machine is in at cold and warm start are described below. Notice that there are several items at the beginning of the table that behave identically, regardless of cold or warm start. This is particularly important for handlers. In the WARM routine of a handler, the handler must restore I/O devices to their required state (power, interrupt vector addresses, and interrupt enable/disable status) since they are always set to their default state, even at cold start.

Table 2-1. HP-94 Status at Cold and Warm Start

Item	Status at Cold Start	Status at Warm Start
Display	Cleared	Cleared
Input/Output	Halted	Halted
Interrupt Vector Addresses	Set to Default *	Set to Default *
Interrupt Enable/Disable Status	Set to Default †	Set to Default †
Copy of Main Control Register	00h	00h
Copy of Interrupt Control Register	31h †	31h †
Serial Port Power	Off	Off
Built-In Serial Port Buffer	Cleared	Cleared
Bar Code Port Power	Off	Off
Bar Code Port Transitions	Disabled	Disabled
Key Buffer	Cleared	Cleared
Beeper	Turned Off	Turned Off
User-Defined Characters	Available	Available
Access to Directory 5	Disabled	Disabled
MAIN Program	Starts at Beginning	—
Current Program	—	Restarts at Power Off Point
System Timeout Value	120 s	Unchanged
Display Backlight Timeout Value	120 s	Unchanged
Display Backlight	Turned Off	Unchanged
Cursor Status	On	Unchanged
Cursor Type	Underline	Unchanged
Keyboard Status	Unshifted	Unchanged
Low Battery Behavior	Halt Program With Error 200	Unchanged
Power Switch Behavior	Turn Off Machine	Unchanged
Timeout Behavior	Turn Off Machine	Unchanged
Allocated Scratch Areas	Returned to Free Space	Preserved
Available Free Blocks	Returned to Free Space	Preserved
BASIC Variable Contents	Lost	Preserved
Open Data Files	Closed	Left Open
File Access Pointers	Reset to Zero	Unchanged
Handler Information Table	Cleared	Unchanged
Open Channel 1-4 Handlers	Closed	Left Open ‡
Channel 1-4 Handler Configurations	Lost	Preserved ‡
Channel 1-4 Buffers	Lost	Preserved ‡
Open Built-In Serial Port Handler	Closed	Left Open, Serial Port On
Built-In Serial Port Configuration	Set to Default §	Unchanged
Stack Pointer	Points to OS Stack	Unchanged

* System timer (50h), serial port data (53h), low main battery voltage (54h), power switch (55h), operating system function (1Ah), user timer (1Ch), and dedicated (00h-03h) interrupt vectors all point to their operating system interrupt service routines. All others point to a dummy FAR RET.

† System timer, low main battery voltage, and power switch interrupts are enabled. All others are disabled.

‡ Exact warm start behavior depends on user-defined handler. The handler must restore the I/O device to its proper state (power, interrupt vector addresses, and interrupt enable/disable status).

§ 9600 baud, 7ES, XON/XOFF enabled, no terminate character, null strip disabled.

When Cold Start Occurs

The 94 will cold start a program under the following conditions:

- After default power off, either because the machine timed out or because the program turned it off with the END_PROGRAM function (00h) and specified cold start.
- After pressing the reset switch.
- After the automatic power off occurs 2-5 minutes after low battery interrupt.
- If any memory integrity error occurred at power on.
- After entering command mode, either when a program ends or by pressing [CLEAR] and [ENTER] at power on.
- If the program is run using the S (*start*) operating system command.
- If main memory size changes (128K memory board added or removed).
- If 40K RAM card changed to ROM/EPROM card, or vice-versa.
- If number or size of directories in ROM/EPROM card changed.

When Warm Start Occurs

The 94 will warm start the program if the program turned the machine off with the END_PROGRAM function and specified warm start, and none of the cold start conditions occurred.

Operating System Activities During Cold Start

When the 94 cold starts, it begins by performing the normal power-on initialization (check memory integrity, determine memory configuration, etc.). The operating system looks for a file called MAIN by searching directories 0-4 in ascending order. If MAIN exists, the status defined in the previous table is set. If no MAIN file is found, or if MAIN is not type A or B, the machine cannot autostart, so it enters command mode.

If MAIN is type A, the operating system does a FAR CALL to the main entry point of the program — the segment address of the start of the program and an offset of 6 (past the end of the program header). This implies that an assembly language program can end with a FAR RET — see the section on "Ending Programs" for further information.

If MAIN is type B, it will be executed by the BASIC interpreter. The operating system searches for a BASIC interpreter (SYBI) in directories 0-5 in ascending order. Error 100 is issued if none is found, or if the one found is not type A. Once the interpreter is found, control is transferred to it. It allocates and initializes its scratch area and the variable space required by the program, sets default values for various BASIC program conditions (shown below), and begins interpreting the program.

Table 2-2. Cold Start Status of BASIC Programs

Item	Initial Status
BASIC Numeric Variables and Arrays	Set to zero
BASIC String Variables and Arrays	Set to null string
SYEL Value	120 seconds
SYER Value	Error trapping disabled
SYLB Value	Default low battery behavior
SYRS Value *	9600 baud, 7ES, XON/XOFF enabled, no terminate character, null strip disabled
SYSW Value	Default power switch/timeout behavior
SYTO Value	120 seconds
* These values override any values specified by the B (*baud*) operating system command.	

Operating System Activities During Warm Start

When the 94 warm starts, it begins by performing the normal power-on initialization (check memory integrity, determine memory configuration, etc.) and executes the WARM routines of any open handlers. Then the operating system transfers control to where the program was running when the power was turned off, and the program continues running.

Ending Programs

Assembly language programs can end in one of two ways. They can either turn the power off, or they can leave the power on and enter command mode. Command mode is where the user can type operating system commands such as C (*copy*) or D (*directory*), and is usually reached by turning on the machine on while holding down the CLEAR and ENTER keys.

The END_PROGRAM function (00h) is used to end a program and turn the power off, specifying that the next power on be cold or warm start. For warm start, the CPU registers are saved on the operating system stack for use when the machine next turns on. If the program has used the operating system stack for its own data, the data will be destroyed when the CPU registers are saved. Therefore, a program cannot specify warm start unless it uses its own stack. If it specifies warm start while using the operating system stack, END_PROGRAM will issue error 219 and enter command mode.

There are two ways to enter command mode from a program. The first way is with a FAR RET, since the program was executed with a FAR CALL. The second way is to use the END_PROGRAM function, specifying to enter command mode. There are subtle differences in the operating system behavior with these two approaches, summarized below.

Table 2-3. Ending a Program With END_PROGRAM or FAR RET

Item	Behavior Using END_PROGRAM	Behavior Using FAR RET
CPU Interrupt Flag	Set (STI)	Unchanged
Access to Directory 5	Enabled	Disabled
Open Files	Closed	Not Closed
Handler CLOSE Routines	Called	Not Called *

* The handler will have no opportunity to restore interrupt vectors or status. Power will be continue to be supplied to the serial port, level converter, and bar code port if they were enabled.

Because of these differences, the END_PROGRAM function is the preferred method of ending a program and entering command mode.

Operating System Activities When Entering Command Mode

When the operating system enters command mode, it initializes certain things to their default values, as shown below.

Table 2-4. HP-94 Status in Command Mode

Item	Status
Input/Output	Halted *
Interrupt Vector Addresses	Unchanged *
Interrupt Enable/Disable Status	Unchanged *
Copy of Main Control Register	Unchanged *
Copy of Interrupt Control Register	Unchanged *
Serial Port Power	Off *
Built-In Serial Port Buffer	Cleared
Bar Code Port Power	Off *
Bar Code Port Transitions	Disabled *
Key Buffer	Unchanged
Beeper	Unchanged
User-Defined Characters	Not Available
Access to Directory 5	Enabled †
System Timeout Value	120 s
Display Backlight Timeout Value	120 s
Display Backlight	Turned Off
Cursor Status	On
Cursor Type	Block
Keyboard Status	Shifted
Low Battery Behavior	Halt Program With Error 200
Power Switch Behavior	Turn Off Machine
Timeout Behavior	Turn Off Machine
Allocated Scratch Areas	Returned to Free Space
Available Free Blocks	Returned to Free Space
BASIC Variable Contents	Lost
Open Data Files	Closed
File Access Pointers	Reset to Zero
Handler Information Table	Cleared
Open Channel 1-4 Handlers	Closed
Channel 1-4 Handler Configurations	Lost
Channel 1-4 Buffers	Lost
Open Built-In Serial Port Handler	Closed †
Built-In Serial Port Configuration	Set to Default ‡
Stack Pointer	Points to OS Stack

* Whether or not these conditions are true depends on the what the program does before it ends and the behavior of the CLOSE routines in any user-defined handlers in use (assuming the routines are called before the program ends). The CLOSE routines will be executed automatically when entering command mode with the END_PROGRAM function (rather than a FAR RET).

† Only if the END_PROGRAM function was used to enter command mode (rather than a FAR RET).

‡ 9600 baud, 7ES, XON/XOFF enabled, no terminate character, null strip disabled.

Program Structure

The three different types of programs (types A, B, and H) have a simple structure consisting of a program header followed by the code. Assembly language programs (type A) have a six-byte header, then the executable code. Handlers (type H programs) have a six-byte header, a jump vector table, then the code pointed to by each of the jump vectors. BASIC programs (type B) have a 16-byte header, then the program tokens.

Program Headers

Assembly language programs start with a six-byte header, shown below with hex offsets on the left side. Note that the order of this illustration is with the lowest offset at the top, which is the order the entries would be placed in the source code for the handler.

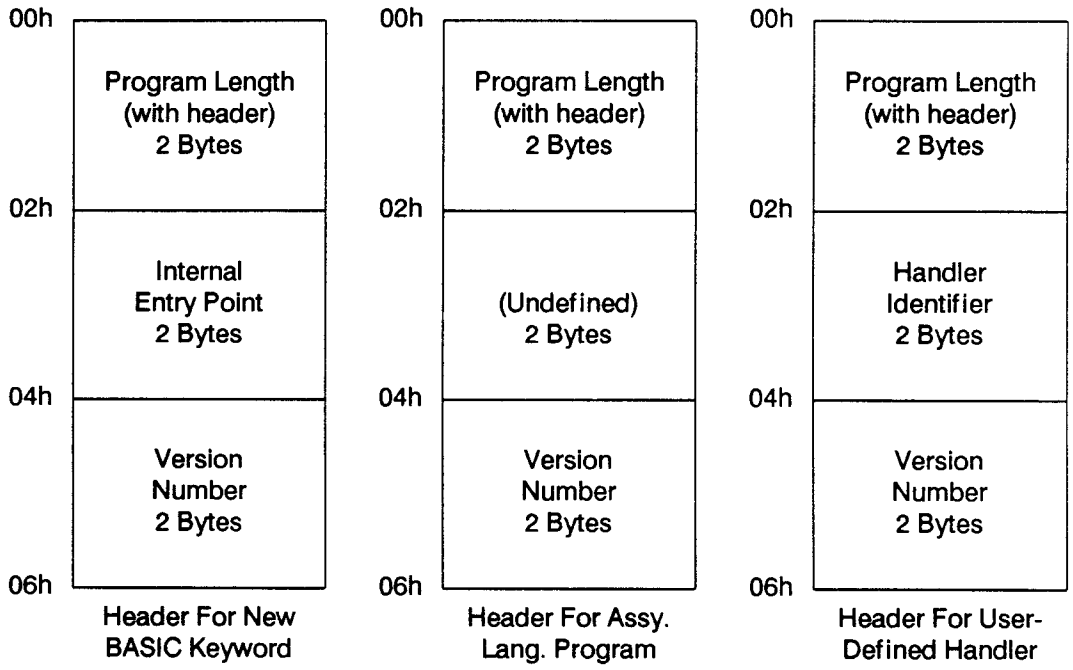


Figure 2-1. Program Headers

There are three fields in the header:

- Program Length
  This field is the length of the program, including the length of the header itself.
- Internal Entry Point
  For type A programs that are new BASIC keywords, this field is the offset of the processing block relative to the start of the program. This assumes a particular BASIC keyword structure which will be described shortly. If a BASIC keyword does not use this structure, this field can be set to point to the first byte after the header, to a dummy FAR RET instruction, or be used for other purposes.

- (Undefined)
  For type A programs that are not BASIC keywords, the place to start executing the program is immediately after the header, so the value of the internal entry point field does not matter — it will never be called by another program. It can therefore either be set to point to the first byte after the header, to a dummy FAR RET instruction, or be used for other purposes.

- Handler Identifier
  The second field in the header has a slightly different meaning for handlers. It contains a two-character identifier that is returned by the identify handler I/O control function (00h).

- Version Number
  This is used for revision control by the programmer. It is a two-byte binary number representing a decimal fraction of the form *II.FF*, where the *II* is the integer part of the version, and the *FF* is the fractional part of the version. The statement VERSION dw 0103h would designate a version number of 1.03, and the statement VERSION dw 0212h would define version 2.18 of the software. This can also be defined in decimal as db 18,2, where the fractional part precedes the integer part.

For type A programs, the program code starts after the header. For type H programs, the jump vector table that follows the header defines the locations of the executable code.

BASIC Keyword Structure

BASIC keywords can be written so that they are accessible from both BASIC and assembly language programs. This requires a keyword structure in which there are two distinct blocks: an I/O block in which all interaction with BASIC variables occurs, and a processing block in which the function of the keyword is implemented. Once the I/O block has read and validated the supplied variables, it calls the processing block. When the processing block is done, it returns its results to the I/O block, which then places them in BASIC variables as appropriate. This structure is shown below.

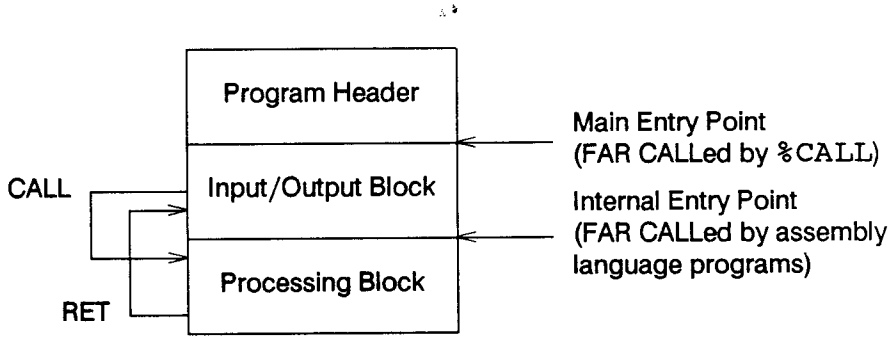


Figure 2-2. BASIC Keyword Structure

The internal entry point in the program header would point to the start of the processing block. This allows both BASIC and assembly language programs access to the functionality implemented by the keyword. BASIC programs execute new keywords with %CALL, which FAR CALLs the main entry point at the end of the header. Assembly language programs execute the processing block only via the internal entry point. They find the program, read the internal entry point from the header, set up appropriate parameters, and FAR CALL the processing block.

Errors should be reported differently depending on which entry point is called. If the main entry point is called (which implies the keyword was called by a BASIC program), non-numeric errors should be reported using the ERROR BASIC interpreter utility routine (offset 34h). This will cause a non-numeric error to be issued by the BASIC interpreter, and the BASIC program will halt. If the internal entry point is called (which implies the keyword was called by an assembly language program), numeric errors should be returned in the AL register (00h if no errors).

The main entry point of a BASIC keyword can also be called from command mode with the S command. This condition should be recognized by BASIC keywords. If the keyword was called from a BASIC program using %CALL, the CS register will be the same as the DS register. If the keyword was called from command mode with the S command, the CS register will be different than the DS register.

There are two possible ways to handle this condition. One approach is for the keyword to end immediately if the keyword is called from command mode. Another approach is to implement an input/output block for interacting with command mode, analogous to the input/output block for interacting with the BASIC interpreter.

Program Restrictions

Programs can start on any paragraph boundary, depending on where the program was loaded and what other files were loaded or deleted. Once they begin to run, they do not move — there is no run-time relocation. Consequently, there should be no far calls or jumps to absolute addresses in type A or H programs. (HXBASIC and HXC ensure this for type B programs.)

Valid EXE Format

When EXE files are created, they should not contain any MS-DOS-style relocation entries. HXC will reject any EXE file if it contains a relocation table. An EXE file, to be accepted by HXC, must have the following characteristics:

- EXE file size of 512 bytes or greater.
- Valid EXE identifier.
- 512-byte header.
- No relocation entries.
- Initial CS = 0000h.

It is recommended that source files use byte alignment by specifying SEGMENT BYTE at the beginning of each program segment. The assembler's default alignment is on paragraph boundaries, causing each object file to be padded with 1-15 bytes. Byte alignment eliminates this unused space. HXC will pad the entire EXE file only once, not once for each object file.

Use of Operating System Stack

A program can use the operating system stack for its own use. The stack varies in length, depending on how the program was called (from the operating system or from another program), up to a maximum of approximately 600 bytes. If a program turns off the machine and specifies a subsequent warm start (see "Cold Start and Warm Start"), it must not use the operating system stack. The END_PROGRAM function (00h) will issue error 219 if the program is using the operating system stack. Consequently, if a program wants to use the warm start option, it must put its stack in its own data space.

Programs in ROM or EPROM

Programs can be in RAM or ROM, and execute in place in either location. ROM programs have additional restrictions. There can be no data space in the code itself if the program is to have the option of running in ROM. The operating system provides scratch area allocation and release functions to allow ROM programs to get needed data space.

The assembler provides the ability to define the offsets within an external scratch area using the SEGMENT AT directive, as shown below.

```
SCR_AREA        segment at 0            ;Addresses start at 0
PARAM1          db      6 dup(?)        ;First parameter needs 6 bytes
PARAM2          db      00              ;Second parameter needs a byte
PARAM3          dw      0000            ;Third parameter needs a word
SCR_AREA        ends
```

Figure 2-3. Defining Scratch Area Data Structure

The SEGMENT AT directive provides an address template that can be imposed on the scratch area. SEGMENT AT causes no code to be generated for the uninitialized data defined within that program segment (in this case, the SCR_AREA segment).

3

User-Defined Handlers

Contents

Chapter 3 User-Defined Handlers

3-1 Handler Structure
3-1 Program Header
3-1 Jump Table
3-3 Channel Input and Output
3-4 File Search Order
3-4 Types of Handlers
3-4 Low-Level Handlers
3-4 High-Level Handlers
3-4 Who Calls Handler Routines
3-5 Handler Information Table
3-6 Table Usage While Handlers Are Closed
3-6 Table Usage While Handlers Are Open
3-6 Table Entry Offsets
3-7 Reading and Setting the Handler Information Table
3-7 Passing Parameters to Handlers
3-7 Passing Parameters in a Parameter Scratch Area
3-8 Verifying Parameter Area Existence
3-8 Validating the Contents of the Parameter Scratch Area
3-9 Passing Parameters After the Handler Name
3-10 Restrictions on In-Line Parameters
3-10 Handler Linkage Routines
3-12 Handler Routine Descriptions
3-12 Registers Passed to Handler Routines
3-13 High-Level Handler Behavior With Unused Registers
3-14 CLOSE
3-16 IOCTL
3-17 Reserved IOCTL Functions
3-20 OPEN
3-22 POWERON
3-23 HP-94 Status During POWERON Routine
3-25 READ
3-27 RSVD2
3-28 RSVD3
3-29 TERM
3-31 WARM
3-33 WRITE

3

User-Defined Handlers

User-defined handlers, or handlers for short, allow BASIC or assembly language programs simple access to the HP-94 I/O ports — the devices associated with channels 1-4. In particular, user-defined handlers can be written for the serial port (channel 1) and bar code port (channel 2); channels 3 and 4 are reserved, and currently have no I/O port associated with them. Handlers are assembly language program files that are assembled and linked into EXE files on the development system. Then they are processed by HXC and given file type H before being copied into the HP-94.

Handlers are similar in concept to UNIX or MS-DOS device drivers. They are a collection of routines to handle various activities associated with I/O devices, such as initializing the port for use, reading and writing data to it, and releasing control of the port. Handlers have a special structure that allows the individual routines to be called, either from BASIC or assembly language, solely by supplying the name of the handler being used when the channel is opened.

This chapter will discuss handler organization in general, how handlers interact with the channel-oriented input and output of the HP-94, the different types of handlers, passing configuration parameters and registers to handler routines, and what tasks handler routines perform.

Handler Structure

Handlers contain three major components: the program header, the jump table, and the executable code for each of the handler routines.

Program Header

Handlers, like all assembly language programs, start with a six-byte header. The first two bytes are the length of the handler, including the header. The next two bytes are a two-character handler identifier that is returned by handlers that implement function 00h of the IOCTL routine (discussed later). The last two bytes of the header are the software version number. It is a two-byte binary number representing a decimal fraction of the form *II.FF*, where the *II* is the integer part of the version, and the *FF* is the fractional part of the version. The statement `VERSION dw 0103h` would designate a version number of 1.03, and the statement `VERSION dw 0212h` would define version 2.18 of the software. This can also be defined in decimal as `db 18,2`, where the fractional part precedes the integer part.

Jump Table

Immediately following the header is a jump table with 10 entries of three bytes each. Each entry contains a JMP instruction to one of the handler routines. Each routine must end with a FAR RET. The header and jump table, showing the order in which the jump table must appear in the program, is shown below. The hex offsets from the start of the program are along the left side. Note that the order

of this illustration is with the lowest offset at the top, which is the order the entries would be placed in the source code for the handler.

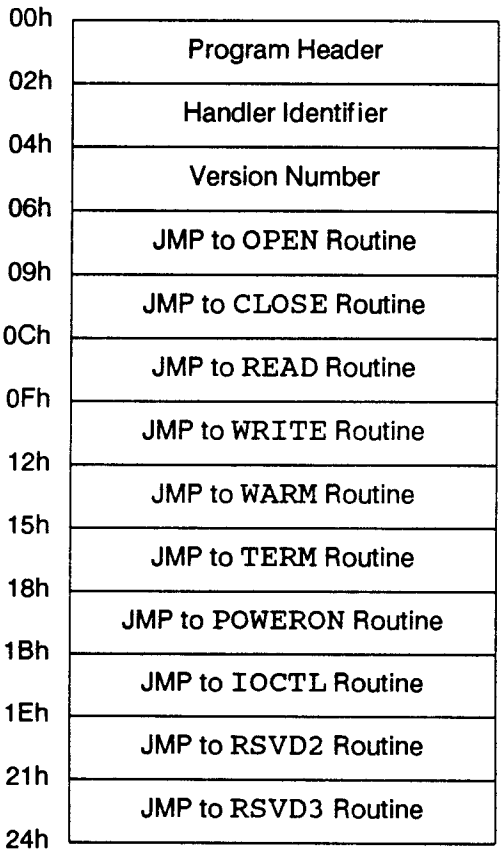


Figure 3-1. Handler Header and Jump Table

The purpose of the different handler routines are listed briefly below.

- OPEN Routine — initializes the port.
- CLOSE Routine — releases control of the port.
- READ Routine — reads data coming into the port.
- WRITE Routine — writes data to the port.
- WARM Routine — allows reinitialization of the port at warm start.
- TERM Routine — allows I/O to be terminated because of the power switch or low battery.
- POWERON Routine — allows initialization at machine power-on.
- IOCTL Routine — controls actions of handler.
- RSVD2 Routine — for future use.
- RSVD3 Routine — for future use.

Entries in the jump table are required for all handler routines. However, not all handlers will implement all routines. If a routine is not implemented, the jump table entry should just JMP to a dummy FAR RET.

There is no jump table entry for the handler's interrupt service routine. The address of that routine is placed in the appropriate interrupt vector in the reserved scratch space. For details on using interrupts, refer to the "Interrupt Controller" chapter.

The tasks performed by the different handler routines will be discussed later in this chapter. The next sections will describe general information relevant to all handlers and handler routines.

Channel Input and Output

The HP-94 operating system performs input and output through 16 different logical channels, each of which is associated with different physical devices. The channels being used for I/O are defined by opening them. From an assembly language program, this is done with the OPEN function (0Fh); from a BASIC program, this is done with the OPEN # statement (which calls the OPEN function). Both the OPEN function and the OPEN # statement take the channel number to open and a file name as their parameters. The table below summarizes the uses of the 16 logical channels, and the meaning of the file name for the different channels.

Table 3-1. Channel Number Assignments

Channel Number	Physical Device	File Name Meaning
0	Console *	Ignored
1	Serial Port	Name of User-Defined Handler (Type H)
2	Bar Code Port	Name of User-Defined Handler (Type H)
3-4	Reserved	Name of User-Defined Handler (Type H)
5-15	Data Files	Name of Data File (Type D)
* The console is the keyboard for input operations and the display for output operations.		

Below is more information about the different channels.

- Channel 0
  The console is always opened by the operating system. A program can specify a file name as a parameter when opening channel 0, but the name will be ignored — user-defined handlers for channel 0 are not allowed.
- Channel 1
  The built-in serial port handler is specified by supplying the null string ("") for the file name. If a user-defined device handler name is supplied and no such handler exists in memory, the default handler will be used.
- Channels 2-4
  There is no default handler for these channels. If the null string is used as the file name, or there is no handler in memory matching the file name supplied, an error will be reported.

- Channels 5-15
  When a data file is opened, the file access pointer is reset to the start of the file. Only one channel at a time can be assigned to a single file. Multiple channels cannot be open to the same file simultaneously.

Once a channel has been opened, an error will occur if it is reopened without first being closed.

File Search Order

The OPEN function will search for the specified file name in directories 0-4 in ascending order. If the file name includes a directory number (e.g., `"1:HNBC"`), only that directory will be searched. If the file name is found, but is an illegal type, (not type H for channels 1-4, or not type D for channels 5-15), an error will be issued. If it is a legal type, it will be opened.

Types of Handlers

There are two types of handlers: high-level and low-level. These support the concept of layered software, in which successively higher layers become more hardware-independent.

Low-Level Handlers

Low-level handlers interact only with the I/O port hardware. They take care of the characteristics of the I/O port on the HP-94 only. An example of this is HNBC, a low-level bar code port handler supplied with the *HP-94 Software Development System* that does low-level I/O with the bar code port. Low-level handlers usually include one or more interrupt service routines for the hardware interrupts associated with the I/O port.

High-Level Handlers

High-level handlers interact only with low-level handlers, not with the I/O port hardware. They take care of the characteristics of the external device connected to the port, but not of the port itself. An example of this is HNWN, a high-level handler that handles the device-specific features of Hewlett-Packard Smart Wands, but relies on the low-level handlers HNBC or HNSP to perform port-specific activities. High-level handlers do not have interrupt service routines because they do not interact directly with the hardware.

Who Calls Handler Routines

The routines in both types of handlers can be called by operating system functions, which in turn are called by BASIC I/O keywords, assembly language programs, or by the operating system itself. If a high- and low-level handler pair are being used, the operating system will think that only the high-level handler is open. All communication between the two handlers is performed by the high-level handler using *handler linkage routines*. These routines are described later in this chapter, and are available as an include file that can be included with the high-level handler source code (discussed in the appendixes).

The relationship between all the layers of software used for I/O is shown below.

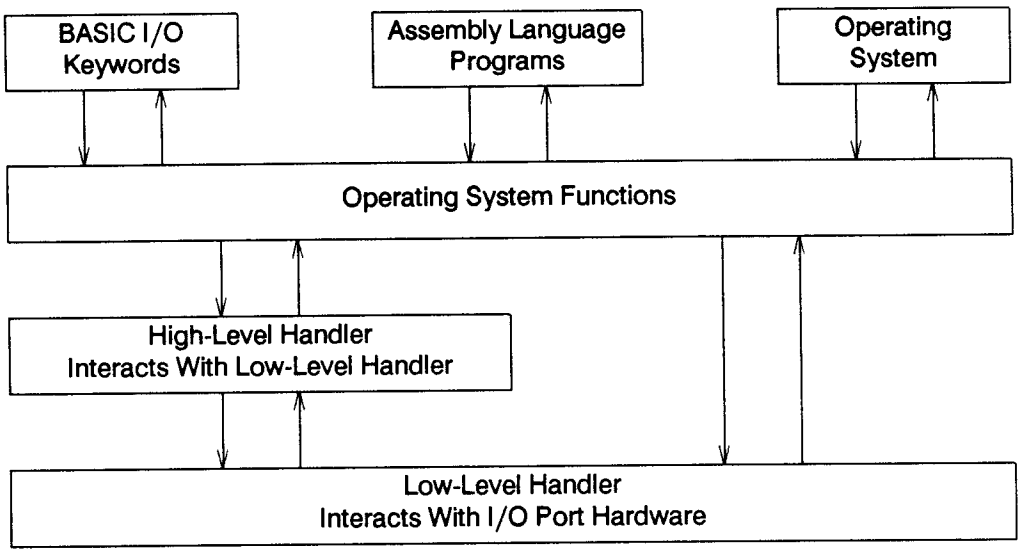


Figure 3-2. Relationship Between High- and Low-Level Handlers

As this diagram indicates, all that is required to perform I/O to a port is a low-level handler. It is not necessary to have or use a high-level handler. If external devices will be used with unique characteristics better accommodated on a driver level than an application level (so the application is more device-independent), then a high-level handler may also be necessary.

Because the high-level handler is totally dependent on the low-level handler to actually move data through the I/O port, high-level handlers cannot stand alone. A low-level handler can be used by itself, but a high-level handler must be used as part of a high- and low-level handler pair.

Handler Information Table

There is a table in the operating system scratch space where handlers keep information about scratch area locations. The table contains five two-byte entries, each of which is associated with a specific channel and has a different meaning depending on whether the handler is closed or open.

Table 3-2. Handler Information Table Entries

Entry Offset	Which Channel	Meaning While Handler Closed	Meaning While Handler Open	Used By Which Interrupt
00h	Bar Code Port	None	Low-Level Handler Scratch Area Address	Bar Code Timer (51h)
02h	Serial Port	Parameter Scratch Area Address	Low-Level Handler Scratch Area Address	Serial Port Data Received (53h)
04h	Bar Code Port	Parameter Scratch Area Address	Low-Level Handler Scratch Area Address	Bar Code Port Transition (52h)
06h	Channel 3	Parameter Scratch Area Address	Low-Level Handler Scratch Area Address	Reserved 1 (56h)
08h	Channel 4	Parameter Scratch Area Address	Low-Level Handler Scratch Area Address	Reserved 2 (57h)

Table Usage While Handlers Are Closed

When a handler is closed, the handler information table is used for the segment address of the parameter scratch area for that channel. When the OPEN routine in either a high- or low-level handler is called, it looks at the appropriate table entry to determine if the parameter scratch area exists and if the information it contains is valid. The procedure for doing this will be discussed later.

Table Usage While Handlers Are Open

Every time a routine in an open handler is called, the operating system automatically passes the segment address of the handler's scratch area to the routine in the DS register. However, the operating system cannot do this when an interrupt causes the handler's interrupt service routine to be executed. To allow the interrupt service routine to locate the scratch area, the handler information table is used for the address of the low-level handler's scratch area. This is done only when the handler is open, for this is the only time that interrupts will be enabled for the handler.

After verifying its parameters, the low-level handler's OPEN routine must save the parameter scratch area address in the handler's scratch area, and place the handler's scratch area address in that table entry. When the handler is closed, the low-level handler CLOSE routine must restore the original parameter scratch area address in that table entry.

Table Entry Offsets

The handler information table entry offsets for a particular handler are *2 * the handler channel number.* Once the handler is open, the entry is read during the handler interrupt service routine. This means that each handler can have one hardware interrupt associated with it. This is not true for the bar code port, since it has both a transition interrupt and a timer interrupt. The primary interrupt for the bar code port is the transition interrupt since it occurs on every transition, so it is associated with the entry for channel 2. The bar code port timer interrupt uses the first entry in the table at offset 0.

Reading and Setting the Handler Information Table

The handler information table is located in the first 10 bytes (5 words) of the operating system scratch space. Using the operating system pointer to locate the scratch space (described in the appendix), the following code will take the channel number in AL and load the table entry for that channel into ES:

```
mov     si,16h              ;get segment address of OS pointers
mov     ds,si               ;put in segment register
xor     ah,ah               ;clear ah
mov     si,ax               ;put channel number in si
shl     si,1                ;2 * channel number
mov     ds,ds:[0000h]       ;get the segment address of OS scratch space
mov     es,word ptr ds:[si] ;get this channel's table entry
```

Figure 3-3. Example of Reading Handler Information Table Entries

Passing Parameters to Handlers

Parameters are passed to a handler mainly to define its operating configuration (such as baud rate for the serial port). The handler uses them to set its configuration when its OPEN routine is called. Parameters can be passed in one of two ways when the handler is opened:

- The parameters can be placed in a *parameter scratch area*. This can be done from a BASIC program with a separate keyword (such as the SYBC keyword that defines parameters for HNBC), or from an assembly language program that allocates and initializes the parameter scratch area before opening the handler. This is the approach used for passing parameters to Hewlett-Packard handlers.
- The parameters can be placed after the handler name that is passed to the OPEN function or the OPEN # statement (e.g., "LLHN 9600,7ES"). The handler OPEN routine then parses the parameters from the name string.

Regardless of which approach is used to pass parameters, the low-level handler must save a copy of them in its scratch area. This is needed by the IOCTL routine of the handler.

Passing Parameters in a Parameter Scratch Area

A parameter scratch area is a one-paragraph scratch area. The upper 8 bytes (bytes 08h-0Fh) are reserved for high-level handler parameters, and the lower 8 bytes (bytes 00h-07h) are reserved for low-level handler parameters. The first byte of each half is used as a *valid data flag* (discussed shortly) to indicate the validity of the parameters. This leaves 7 bytes available for parameters for each high- and low-level handler.

Handlers verify two aspects of configuration parameters: first, that the parameter scratch area exists, and second, that it contains valid configuration information.

Verifying Parameter Area Existence

High- and low-level handlers determine if the parameter area exists by reading the handler information table entry for that channel. If the entry is zero, there is no parameter scratch area for the handler. The handler should then allocate a one-paragraph parameter scratch area and place its address in the table entry. If the entry is non-zero, the entry contains the segment address of a parameter scratch area that already exists.

It is important that the address of the parameter area put in the handler information table actually point to a scratch area. If an assembly language program opens a handler and passes it parameters, the address put in the table must not point to parameters on the program's stack, or to fixed parameters embedded in the program code. This is because if the stack vanishes or the program moves, the address in the handler information table will no longer point to valid parameters.

CAUTION When a handler is open, the entry in the handler information table will be the scratch area address of the handler, *not* of the parameter scratch area (see "Handler Information Table"). If a separate configuration program is run after the handler is open, it could misinterpret the handler information table entry, and modify the handler scratch area by mistake. Configuration programs should check if the handler is open before examining the handler information table. See the appendixes for a utility routine that determines if a channel is open or not.

Validating the Contents of the Parameter Scratch Area

High- and low-level handlers validate the contents of the parameter scratch area by looking at the first byte in their respective parts of the area (upper 8 bytes for high-level handlers, lower 8 bytes for low-level handlers). This first byte is a valid data flag that is unique for each handler associated with a particular channel. The valid data flag is set to zero when the scratch area is allocated because the operating system initializes all scratch areas to zero (00h). The flag is then set to a value either by a handler, by the program calling the handler, or by a configuration keyword. The action that a handler should take for different values of the valid data flag is shown below.

Table 3-3. Interpreting the Valid Data Flag

Value of Flag	High-Level Handler Action	Low-Level Handler Action
Zero	Put correct valid data flag and default high-level handler configuration in upper 8 bytes of parameter scratch area.	Put correct valid data flag and default low-level handler configuration in lower 8 bytes of parameter scratch area.
Correct for Handler	Use these parameters to define high-level handler configuration.	Use these parameters to define low-level handler configuration.
Any Other Value	Return an error, since the parameters are not valid for this handler.	Return an error, since the parameters are not valid for this handler.

Handlers should use values for the valid data flag in the range 01h-7Fh. Hewlett-Packard uses values in the range 80h-FFh for its handlers, and 00h is reserved because it indicates uninitialized parameters. Refer to the "Program Resource Allocation" appendix for information about reserving a valid data flag that will not conflict with any other flag in use.

Passing Parameters After the Handler Name

If parameters are passed in-line with the handler name, the handler's OPEN routine must parse and interpret the handler names and parameters. When the handler OPEN routine executes, `ES:BX` points to the start of the entire handler name string. The routine can skip past the handler name in the string to find the beginning of the parameters, and parse them into whatever internal form is required for the handler. The syntax of the name string is as follows:

- High-level handler name
- One or more spaces
- High-level handler parameters separated by commas
- Semicolon
- One or more spaces
- Low-level handler name
- One or more spaces
- Low-level handler parameters separated by commas
- Ending null (00h)

This results in handler and parameter strings that look like the following examples:

`"HNLL 7,2"`	Low-level handler with parameters
`"HNHL 1,3;HNLL 7,2"`	High- and low-level handlers with parameters
`"1:HNHL 1,3;1:HNLL 7,2"`	Same but with directory numbers
`"HNHL;HNLL"`	High- and low-level handlers with no parameters

Restrictions on In-Line Parameters

- If the OPEN # statement is used, the maximum length of the handler names and parameters is 255 characters.
- The OPEN # statement uppercases all characters in the name string, so the name string in OPEN `#1,"llhn 7es"` will be passed as `"LLHN 7ES"`. If a handler that accepts in-line parameters will be opened with the OPEN # statement, the parameters should not be case-sensitive.
- If a high-level handler that accepts in-line parameters calls a low-level handler that accepts parameters in a parameter scratch area (such as Hewlett-Packard handlers), the high-level handler must parse its in-line parameters and put them in the form expected by the low-level handler. Then it must create a parameter scratch area, place the parameters in it, and modify the handler information table before calling the low-level handler.

Handler Linkage Routines

If a high- and low-level handler pair are being used, the operating system will think that only the high-level handler is open. All communication between high- and low-level handlers is performed by the high-level handler using handler linkage routines. These routines are available as an include file that can be included with the high-level handler source code (discussed in the appendixes).

Each handler routine has a corresponding linkage routine that it uses to call the low-level handler. To use the linkage routines, load appropriate values into the registers, put the channel number in AL, and FAR CALL the routine by name. The activities of each high-level handler routine before and after calling the linkage routine will be discussed shortly.

The linkage routines are designed to mimic the way the operating system calls handler routines. A low-level handler will not be able to distinguish that it is being called by a high-level handler rather than by the operating system. Like the operating system, the caller's registers (in this case, the high-level handler's) are saved in a register save area on the stack when the low-level handler is called. Upon return, the registers are popped off in exactly the same manner. This means that low-level handlers must return the error code in AL (00h if no errors), and all other register values in the appropriate location in the register save area.

Below is a summary of the registers passed to and returned by the linkage routines.

Table 3-4. Register Usage By Handler Linkage Routines

Routine Name	Registers Passed		Registers Returned	
	Register	Contents	Register	Contents
LLH_CLOSE	AL	Channel number to close	AL	Error code
LLH_IOCTL	AL AH Others	Channel number IOCTL function code As defined by routine	AL Others	Error code As defined by routine
LLH_OPEN	AL ES BX	Channel number to open Segment address of low-level handler name to open Offset address of low-level handler name to open	AL	Error code
LLH_READ	AL CX ES BX	Channel number to read Number of bytes to read Segment address of read buffer Offset address of read buffer	AL CX	Error code Number of bytes actually read
LLH_RSVD2	AL Others	Channel number Not yet defined	AL Others	Error code Not yet defined
LLH_RSVD3	AL Others	Channel number Not yet defined	AL Others	Error code Not yet defined
LLH_TERM	AL AH *	Channel number Cause of termination 1 = power switch 0 = low battery	AL	Error code
LLH_WARM	AL	Channel number	AL	Error code
LLH_WRITE	AL CX ES BX	Channel number to write Number of bytes to write Segment address of write buffer Offset address of write buffer	AL CX	Error code Number of bytes actually written
All (supplied automatically)	DS † BP DI	Segment address of low-level handler scratch area Stack offset address of register save area Destroyed	BP	Unchanged from value passed to routine

* The TERM routine for high- and low-level handlers will receive the cause of the termination in AL. A high-level handler must move this value into AH and place the channel number in AL before calling LLH_TERM. LLH_TERM will swap them back, thereby passing the cause of the termination to the low-level handler in AL.

† Not passed to LLH_OPEN routine.

Handler Routine Descriptions

Handler routine descriptions consist of the following:

- A brief description of the routine.
- A summary of the parameters passed to the routine.
- A summary of the parameters that the routine must return.
- Details on when the routine is called.
- Supplementary notes and cautions on the use and behavior of the routine.

Registers Passed to Handler Routines

Handler routines are called by the analogous operating system functions. For example, the READ function will FAR CALL the READ routine in the handler that is open to the channel being read. When handler routines are called, either by the operating system or by handler linkage routines, all the registers values that were passed to the operating system function will be passed to the handler routine, with the following exceptions:

- The DS register contains the segment address of the handler scratch area (except for the OPEN routine).
- The BP register contains the offset on the stack where all the caller's registers were saved.
- The DI register is destroyed.

All the caller's original registers are saved in a register save area on the stack. When the handler routine ends (with a FAR RET), the caller (operating system function or handler linkage routine) will automatically pop all the saved registers off the stack *except* AL, which is used to return error codes, and BP, which must be unchanged from the value passed to the routine. Consequently, if a handler wants to return a value in a register other than AL or BP, it cannot just put the value in the register — the register will be lost when the saved register copies are popped off the stack. Instead, the handler routine must place values to be returned into the register save area on the stack.

The order that the registers are saved on the stack is shown below, with the hex offsets on the left.

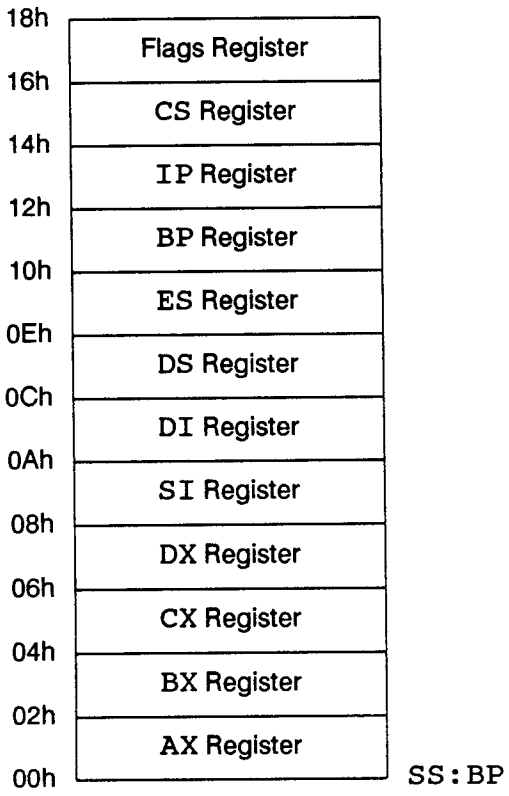


Figure 3-4. Register Save Area

CAUTION Do not alter values in the register save area except those that the handler routine is required to change. Some registers are critical to the proper operation of the calling routines, and changing them can have significant, detrimental side effects (including loss of data).

High-Level Handler Behavior With Unused Registers

Routines in high-level handlers must return to their callers all registers returned by the low-level handler, even if the high-level handler doesn't use or modify any of those registers. The reason is that even if the high-level handler doesn't care about the contents of a particular register, the register may be important to the caller.

This is particularly true of the IOCTL routine, in which the high-level handler may just pass through, unmodified, low-level handler IOCTL requests from an application. If the high-level handler does not similarly pass back the results from the low-level handler, the caller will not see them.

CLOSE

The CLOSE routine in a handler is where the I/O port and the external device are shut down, and control of the port is released by the handler.

Passed to routine:

AL	Channel number to close.

Routine must return:

AL=00h	Successful close.
>00h	Error code.
BP	Unchanged from value passed to routine.

When routine is called:

- By the CLOSE function (10h) if a high- or low-level handler name was specified when the handler was opened. The CLOSE function can be invoked either by the BASIC CLOSE # statement or by an assembly language program.
- By a high-level handler using the LLH_CLOSE linkage routine.
- When a program ends and returns to command mode by calling the END_PROGRAM function (00h), the operating system closes all open handlers by calling their CLOSE routines.

Notes:

- Registers specified by the caller of the CLOSE function or the LLH_CLOSE linkage routine are passed to the handler CLOSE routine with the following exceptions:

DS	Set to the segment address of the scratch area allocated by the handler.
BP	Points to the offset on the stack where all the caller's registers are saved and where all returned values except AL must be put.
DI	Destroyed.

Cautions:

- When returning to command mode, the operating system calls the CLOSE routines of all open handlers to close them, but does not set AL to the channel number being used. Make sure AL is set to the channel number before calling LLH_CLOSE, or the linkage routine will not call the low-level handler CLOSE routine properly.

  If the high-level handler is only valid for one channel, that valid channel number can be placed in AL before calling LLH_CLOSE. If the high-level handler can be used for more than one channel, the channel number being used should have been saved in the handler's scratch area by its OPEN routine.

Activities of routine:

High-Level Handler Activities	Low-Level Handler Activities
Perform device-specific shut down activities.	Disable hardware interrupts for the I/O port.
	Disable and power down the I/O port.
Call low-level handler with LLH_CLOSE linkage routine (see caution below).	Restore original hardware interrupt vectors for the I/O port.
Release high-level handler scratch area.	Restore parameter scratch area address from the low-level handler scratch area into the handler information table.
Return an error code if the routine failed (00h if no errors).	
	Deallocate low-level handler scratch area.
	Return an error code if the routine failed (00h if no errors).

IOCTL

The IOCTL (*I/O control*) routine in a low-level handler allows a program to control the handler operation after the handler has already been opened. This is in addition to providing the handler configuration parameters at open time. High-level handler IOCTL routines only call their low-level handler, since most external devices are controlled by command sequences embedded in data sent to them (via the WRITE function).

Passed to routine: *

AH	IOCTL function code.
AL	Channel number.

Routine must return: *

AL=00h	Successful.
>00h	Error code.
BP	Unchanged from value passed to routine.
AH †	As defined by routine (return in register save area, offset 00h)
BX †	As defined by routine (return in register save area, offset 02h)
CX †	As defined by routine (return in register save area, offset 04h)
DX †	As defined by routine (return in register save area, offset 06h)
SI †	As defined by routine (return in register save area, offset 08h)
DI †	As defined by routine (return in register save area, offset 0Ah)
ES †	As defined by routine (return in register save area, offset 0Eh)

When routine is called:

- By a high-level handler using the LLH_IOCTL linkage routine.
- By an assembly language program using the IOCTL utility routine (see the appendixes). If a high-level handler is called, it passes the call on to the low-level handler by calling LLH_IOCTL.
- Not called by the operating system. IOCTL is one of the three reserved handler routines whose use was not defined until after the operating system was developed; the others are RSVD2 and RSVD3.

Notes:

- Registers specified by the caller of the LLH_IOCTL linkage routine or the IOCTL utility routine are passed on to the low-level handler IOCTL routine with the following exceptions:

* Because each handler implements different handler control functions within its IOCTL routine, other register requirements are defined by the handler itself.

† Returned by high-level handler only.

DS Set to the segment address of the scratch area allocated by the handler.

BP Points to the offset on the stack where all the caller's registers are saved and where all returned values except AL must be put.

DI Destroyed.

- Routines in high-level handlers must return to their callers all registers returned by the low-level handler, even if the high-level handler doesn't use or modify any of those registers. The reason is that even if the high-level handler doesn't care about the contents of a particular register, the register may be important to the caller. This is particularly true of the IOCTL routine, in which the high-level handler may just pass through, unmodified, low-level handler IOCTL requests from an application. If the high-level handler does not similarly pass back the results from the low-level handler, the caller will not see them.

Cautions:

- The high-level handler must not change the DS register in the register save area from the caller. Doing so may cause the caller to use the wrong scratch area.

Activities of routine:

High-Level Handler Activities	Low-Level Handler Activities
Call low-level handler with LLH_IOCTL linkage routine.	Perform low-level handler control activities.
Return any registers (in the register save area) that may have been used by the low-level handler routine to the caller.	Return an error code if the routine failed or if a function code was passed that the handler does not implement (00h if no errors).
Return an error code if the routine failed (00h if no errors).	

Reserved IOCTL Functions

Certain I/O control functions have been assigned fixed function codes 00h-06h. Each handler may implement additional functions; refer to the documentation for the particular handler of interest for details. The "Program Resource Allocation" appendix indicates other function codes that have been reserved by other handlers. The fixed function codes are listed below in numeric order.

- IDENTIFY (Function 00h)
  The IDENTIFY function returns two pieces of information to identify handlers: the handler identifier (bytes 2 and 3 of the program header) in CX (byte 2 in CH, byte 3 in CL), and the version number in DX (DH=integer part, DL=fractional part). Hewlett-Packard handlers also return the characters "HP" in BX (BH="H", BL="P"),

- GET_CONFIG (Function 01h)
  The GET_CONFIG function returns the address of the current configuration in ES:DX. Refer to the documentation for each handler for details on the format of the configuration.

The configuration that is returned should be the one saved in the handler's scratch area during its OPEN routine. If the CHANGE_CONFIG function has changed the configuration, the changes would have been made to the saved copy, not the original configuration in the parameter scratch area.

- CHANGE_CONFIG (Function 02h)
  The CHANGE_CONFIG function changes the current handler (and possibly port) configuration while the handler is open. The address of the new configuration is passed in ES:DX. Refer to the documentation for each handler for details on the format of the configuration.

  The configuration that is altered should be the one saved in the handler's scratch area during its OPEN routine. The reason is that configuration changes while the handler is open should not affect the original status defined while it was closed. If a program has initialized a parameter scratch area with certain values prior to opening the handler, the program expects that set of parameters to be unchanged the next time the handler is opened.

- RECEIVE_STATUS (Function 03h)
  The RECEIVE_STATUS function returns the number of bytes in the receive buffer in CX.

- RECEIVE_FLUSH (Function 04h)
  The RECEIVE_FLUSH flushes the receive buffer.

- SEND_STATUS (Function 05h)
  The SEND_STATUS function returns the number of bytes in the send buffer in CX.

- SEND_FLUSH (Function 06h)
  The SEND_FLUSH flushes the send buffer.

The register usage for these functions is summarized below. The AH register is set to the function code, and the AL register is set to the channel number. Like all handler routines, all the registers returned by these functions must be placed in the register save area except AL (for error codes) and BP (which must be unchanged from the value passed to the routine).

Table 3-5. Reserved IOCTL Function Codes

Function Name	Registers Passed		Registers Returned	
	Register	Contents	Register	Contents
CHANGE_CONFIG	AH AL ES DX	02h Channel number Segment address of configuration Offset address of configuration	AL	Error code (00h if no errors)
GET_CONFIG	AH AL	01h Channel number	AL ES DX	00h Segment address of configuration Offset address of configuration
IDENTIFY	AH AL	00h Channel number	AL BX CX DX	00h "HP" * Handler identifier Version number
RECEIVE_FLUSH	AH AL	04h Channel number	AL —	Error code (00h if no errors) Receive buffer cleared
RECEIVE_STATUS	AH AL	03h Channel number	AL CX	00h Number of bytes in receive buffer
SEND_FLUSH	AH AL	06h Channel number	AL —	Error code (00h if no errors) Send buffer cleared
SEND_STATUS	AH AL	05h Channel number	AL CX	00h Number of bytes in send buffer

* Returned by Hewlett-Packard handlers.

OPEN

The OPEN routine in a handler is where the I/O port and the external device are initialized and readied for I/O.

Passed to routine:

AL	Channel number to open.
ES	Segment address of handler name string to open.
BX	Offset address of handler name string to open.
DS	Segment address of parameter area (built-in serial port handler only).
DX	Number of bytes to write.

Routine must return:

AL=00h	Successful open.
>00h	Error code.
BP	Unchanged from value passed to routine.

When routine is called:

- By the OPEN function if the caller invoking the function specifies a high- or low-level handler name. The OPEN function can be invoked either by the BASIC OPEN # statement or by an assembly language program.
- By a high-level handler using the LLH_OPEN linkage routine.

Notes:

- Registers specified by the caller of the OPEN function or the LLH_OPEN linkage routine are passed to the handler OPEN routine with the following exceptions:

  BP Points to the offset on the stack where all the caller's registers are saved and where all returned values except AL must be put.

  DI Destroyed.

- Handlers allocate one or two scratch areas in their OPEN routine: the parameter scratch area for parameter passing (if not already allocated), and the handler scratch area for its pointers, buffers, etc. The operating system saves the handler's scratch area address in an internal table based on the channel number of the handler (this is not the same as the handler information table). When the other routines in the handler are called (such as READ, WRITE, etc.), the operating system reads the appropriate scratch area address from this internal table, and passes it to the routine.

  If a handler allocates more than one scratch area, only the address of the last one allocated will be saved and automatically passed to handler routines. Therefore, when multiple scratch areas are allocated by a handler, the allocation order is important. A handler can allocate scratch areas so that the last one allocated is the one whose address should be passed to handler routines. Alternatively, the handler can call GET_MEM with the channel number set to 0, and the operating system will not save that scratch area address or pass it to handler routines.

Activities of routine:

High-Level Handler Activities	Low-Level Handler Activities
Verify that the channel being opened to is correct for this handler.	Read and verify configuration parameters. If passed in parameter scratch area, use the handler information table and the valid data flag. If passed in-line with the name string, parse the parameters from the string, and convert to the form required by the handler.
Read and verify configuration parameters. If passed in parameter scratch area, use the handler information table and the valid data flag. If passed in-line with the name string, parse the parameters from the string, and convert to the form required by the handler.	Allocate and initialize parameter scratch area if necessary.
Allocate and initialize parameter scratch area if necessary.	Allocate low-level handler scratch area for port-specific needs.
Allocate high-level handler scratch area for device-specific needs.	Save parameter scratch area address from the handler information table in the low-level handler scratch area. This will be needed by the CLOSE routine.
Save channel number the handler is opened to in the high-level handler scratch area. This will be needed by the CLOSE, TERM, and WARM routines. *	Save parameters from the parameter scratch area in the low-level handler scratch area. This will be needed by the IOCTL routine.
Change handler name pointer (ES:BX) to point to the start of the low-level handler name. Skip past the directory number and colon, if any, and any in-line parameters to find the low-level handler name. Return an error if there is no low-level handler name.	Save the low-level handler scratch area address in the handler information table. This will be needed by the interrupt service routine.
Call low-level handler with LLH_OPEN linkage routine. Return an error if no low-level handler with that name exists.	Take over hardware interrupt vectors for the I/O port, and save the previous vector address in the low-level handler scratch area.
Perform device-specific initialization activities.	Initialize the I/O port and provide power to it.
Return an error code if the routine failed (00h if no errors).	Enable hardware interrupts for the I/O port.
	Return an error code if the routine failed (00h if no errors).

* This is only necessary if the high-level handler can be used for more than one channel, such as HNWN. If the handler can be used for only one channel, that channel number need not be saved, since it will always be known.

POWERON

The POWERON routine allows a handler to perform device or port initialization when the machine is turned on, even if the handler is not open.

Passed to routine:

Nothing. *

Routine must return:

Nothing. *

When routine is called:

- Only when the HP-94 is turned on, after all memory integrity checks have been performed, and all battery voltages have been tested. All handlers, whether open or closed, will have their POWERON routine executed at that time. This includes the low-level handler of a high- and low-level handler pair, even though the operating system thinks that only the high-level handler of the pair is open. For this reason, there is no LLH_POWERON linkage routine.
- Not executed if the machine enters command mode because of a memory integrity error or because the [CLEAR] and [ENTER] keys are held down. (The latter prevents an erroneous POWERON routine from permanently preventing access to command mode.) This means that if the machine autostarts MAIN, POWERON will have been executed, but if MAIN is run from command mode using the S (*start*) command, POWERON will not have been executed. POWERON will not have been executed if the program is always started from command mode (e.g., SCOLL to start a program called COLL).

Notes:

- If a high-level handler wants to perform device-specific power-on initialization, it must be done after a low-level handler performs port-specific power-on initialization, or the I/O port may not allow access to the device. The POWERON routines are called in each handler, open or closed, in directories 0-4 in ascending order. Within each directory, handlers are called in ascending directory table entry order. This implies that the low-level handler's POWERON routine would have to be called before the high-level handler's. This will only occur if the low-level handler appears earlier in the same directory as the high-level handler, or in a lower-numbered directory than the high-level handler.
- HP-94 status during the POWERON routine is discussed later.

Cautions:

- Power-on initialization of the HP-94 can be completely altered, with significant, detrimental side effects (including loss or alteration of existing data) if the POWERON routine changes any of the registers that are passed to it. It is therefore imperative that the POWERON routine save and restore any registers that it uses.

* Unlike all other handler routines, no registers are saved before calling POWERON, or restored upon its exit. No registers are passed to the routine, nor are any values expected to be returned by it.

Activities of routine:

High-Level Handler Activities	Low-Level Handler Activities
Save any registers used by routine.	Save any registers used by routine.
Perform device-specific power-on initialization activities.	Perform port-specific power-on initialization activities.
Restore original registers.	Restore original registers.

HP-94 Status During POWERON Routine

The machine status when the POWERON routine is called is identical to the status at warm start, *even if the machine was turned off with cold start specified*, with the following exceptions:

- The status of user-defined characters is unchanged. If the machine was turned off during a running program, they will be available (assuming they were present in the machine) during the POWERON routine. If the machine was turned off by pressing the power switch in command mode, they will not be available during POWERON routine.
- Access to directory 5 is enabled. This is the only time when a program is running that directory 5 is accessible.
- The display backlight will be off.

After the POWERON routine is called, the operating system will perform either cold start or warm start initialization, depending on how the machine was turned off. If it should cold start, the cold start status is set, and program MAIN will be autostarted. User-defined characters are located and made available if they exist, access to directory 5 is disabled, and the backlight remains off. If it should warm start, the warm status is left unchanged, and the program continues running at its power-off point. User-defined characters are left in their warm start state, access to directory 5 is disabled, and the backlight is turned on if it was on when the machine turned off.

Because the cold or warm status is set after POWERON is called, some operating system functions cannot be used in the routine. For example, if the machine is going to cold start, all open data files will be closed. If a file was opened during the POWERON routine, it will be closed immediately during cold start initialization. Here is a list of the operating system functions that can be used in the POWERON routine:

Table 3-6. Functions Allowed in POWERON Routine

Function Name	Function Number
BEEP	07h
BUFFER_STATUS	06h
CURSOR	05h
DISPLAY_ERROR	18h
FIND_FILE	16h
FIND_NEXT	17h
GET_CHAR	01h
GET_LINE	02h
MEM_CONFIG	0Dh
PUT_CHAR	03h
PUT_LINE	04h
ROOM	0Eh
SET_INTR	0Ah
TIMEOUT	09h
TIME_DATE	08h

READ

The READ routine in a handler is where the data coming into the I/O port is read and returned to the caller.

Passed to routine:

AL	Channel number to read.
CX	Number of bytes to read.
ES	Segment address of read buffer.
BX	Offset address of read buffer.

Routine must return:

AL=00h	Successful read.
>00h	Error code.
BP	Unchanged from value passed to routine.
CX	The number of bytes actually read (return in register save area, offset 04h).

When routine is called:

- By the READ function if a high- or low-level handler name was specified when the handler was opened. The READ function can be invoked either by the BASIC GET #, INPUT # or INPUT$ statements, or by an assembly language program.
- By a high-level handler using the LLH_READ linkage routine.

Notes:

- Registers specified by the caller of the READ function or the LLH_READ linkage routine are passed to the handler READ routine with the following exceptions:

DS	Set to the segment address of the scratch area allocated by the handler.
BP	Points to the offset on the stack where all the caller's registers are saved and where all returned values except AL must be put.
DI	Destroyed.

Cautions:

- The number of bytes to read must not be greater than the actual read buffer length (although it can be less).

Activities of routine:

High-Level Handler Activities	Low-Level Handler Activities
Call low-level handler with LLH_READ linkage routine. The read buffer specified can be either the caller's buffer or one in the handler's scratch area.	Enable the system timeout. *
	Monitor system events (system timeout, power switch, and low battery) while waiting for incoming data. *
Perform device-specific read activities.	Read the data from the I/O port.
Transfer the data from the high-level handler's buffer (if any) into the caller's read buffer (but no more than the low-level handler returned).	Transfer the data from the low-level handler's buffer (in its scratch area) into the caller's read buffer (but no more than the caller requested).
Return the actual number of bytes read, and an error code if the routine failed (00h if no errors).	Disable the system timeout. *
	Return the actual number of bytes read, and an error code if the routine failed (00h if no errors).
* Refer to the appendixes for information about a utility routine to do this.	

RSVD2

The RSVD2 routine in a handler is the second routine reserved for future use, the first (with a use now assigned) being IOCTL, and the third being RSVD3.

Passed to routine: *

AL	Channel number.

Routine must return: *

AL=00h	Successful write.
>00h	Error code.
BP	Unchanged from value passed to routine.

When routine is called:

- By a high-level handler using the LLH_RSVD2 linkage routine.
- Not called by the operating system or by any utility routines.

Notes:

- Registers specified by the caller of the LLH_RSVD2 linkage routine are passed on to the handler RSVD2 routine with the following exceptions:

DS	Set to the segment address of the scratch area allocated by the handler.
BP	Points to the offset on the stack where all the caller's registers are saved and where all returned values except AL must be put.
DI	Destroyed.

Activities of routine: *

High-Level Handler Activities	Low-Level Handler Activities
Call low-level handler with LLH_RSVD2 linkage routine.	Return an error code if the routine failed (00h if no errors).
Return an error code if the routine failed (00h if no errors).	

* Because these routines have not yet been defined, other register requirements and activities may be defined at a later date.

RSVD3

The RSVD3 routine in a handler is the third routine reserved for future use, the first (with a use now assigned) being IOCTL, and the second being RSVD2.

Passed to routine: *

AL	Channel number.

Routine must return: *

AL=00h	Successful write.
>00h	Error code.
BP	Unchanged from value passed to routine.

When routine is called:

- By a high-level handler using the LLH_RSVD3 linkage routine.
- Not called by the operating system or by any utility routines.

Notes:

- Registers specified by the caller of the LLH_RSVD3 linkage routine are passed on to the handler RSVD3 routine with the following exceptions:

DS	Set to the segment address of the scratch area allocated by the handler.
BP	Points to the offset on the stack where all the caller's registers are saved and where all returned values except AL must be put.
DI	Destroyed.

Activities of routine: *

High-Level Handler Activities	Low-Level Handler Activities
Call low-level handler with LLH_RSVD2 linkage routine.	Return an error code if the routine failed (00h if no errors).
Return an error code if the routine failed (00h if no errors).	

* Because these routines have not yet been defined, other register requirements and activities may be defined at a later date.

The TERM routine in a handler is used to halt I/O in progress when low battery or power switch interrupts occur.

Passed to routine:

AL	Cause of termination (0=low battery, 1=power switch pressed).

Routine must return:

AL=00h	Successful.
>00h	Error code.
BP	Unchanged from value passed to routine.

When routine is called:

- By the operating system when low battery occurs.
- By the operating system when the power switch is pressed, unless the program disabled the power switch using the SET_INTR function (0Ah).
- By a high-level handler using the LLH_TERM linkage routine.
- *Not* called by the operating system when the system timeout occurs. Since each handler must monitor the system timeout itself, that handler will be the only one waiting on I/O when the timeout expires. Consequently, it is the only one that needs to terminate I/O.

Notes:

- Registers specified by the caller of the TERM function or the LLH_TERM linkage routine are passed on to the handler TERM routine with the following exceptions:

DS	Set to the segment address of the scratch area allocated by the handler.
BP	Points to the offset on the stack where all the caller's registers are saved and where all returned values except AL must be put.
DI	Destroyed.

Cautions:

- When low battery or power switch occurs, the operating system calls the TERM routines of all open handlers, but does not set AL to the channel number being used. Instead, it sets AL to the cause of the termination (0=low battery, 1=power switch). Place the cause of the termination into AH, and make sure AL is set to the channel number before calling LLH_TERM, or the linkage routine will not call the low-level handler TERM routine properly. LLH_TERM will swap the values so that the low-level handler's TERM routine will receive the cause of the termination in AL.

  If the high-level handler is only valid for one channel, that valid channel number can be placed in AL before calling LLH_TERM. If the high-level handler can be used for more than one channel, the channel number being used should have been saved in the handler's scratch area by its OPEN routine.

...TERM

Activities of routine:

High-Level Handler Activities	Low-Level Handler Activities
Call low-level handler with LLH_TERM linkage routine (see caution below).	Halt I/O in progress.
	Clean up incomplete data.
Perform device-specific termination activities.	Return an error code if the routine failed (00h if no errors).
Return an error code if the routine failed (00h if no errors).	

The WARM routine in a handler is where the I/O port and the external device are reinitialized to their open state and readied for I/O when the HP-94 warm starts.

Passed to routine:

AL	Channel number.

Routine must return:

AL=00h	Successful.
>00h	Error code.
BP	Unchanged from value passed to routine.

When routine is called:

- By the operating system when the HP-94 turns on with a warm start, after the POWERON routine has been called. The WARM routines of any high- or low-level handlers that were open at power off are called after the operating system performs all memory integrity tests and sets all warm start status, just before returning control to the program that turned the power off. Refer to the "Program Execution" chapter for details on machine status at warm start.
- By a high-level handler using the LLH_WARM linkage routine.

Notes:

- Registers specified by the caller of the WARM function or the LLH_WARM linkage routine are passed to the handler WARM routine with the following exceptions:

DS	Set to the segment address of the scratch area allocated by the handler.
BP	Points to the offset on the stack where all the caller's registers are saved and where all returned values except AL must be put.
DI	Destroyed.

Cautions:

- At warm start, the operating system calls the WARM routines of all open handlers, but does not set AL to the channel number being used. Make sure AL is set to the channel number before calling LLH_WARM, or the linkage routine will not call the low-level handler WARM routine properly.

  If the high-level handler is only valid for one channel, that valid channel number can be placed in AL before calling LLH_WARM. If the high-level handler can be used for more than one channel, the channel number being used should have been saved in the handler's scratch area by its OPEN routine.

Activities of routine:

High-Level Handler Activities	Low-Level Handler Activities *
Call low-level handler with LLH_WARM linkage routine (see caution below).	Take over hardware interrupt vectors for the I/O port, and save the previous vector addresses in the low-level handler scratch area.
Perform device-specific initialization activities.	
Return an error code if the routine failed (00h if no errors).	Initialize the I/O port and provide power to it.
	Enable hardware interrupts for the I/O port.
	Return an error code if the routine failed (00h if no errors).

* The status of I/O devices at warm start is the same as at cold start. It is the responsibility of the handler to restore I/O devices to their proper state (power, interrupt vector addresses, and interrupt enable/disable status).

WRITE

The WRITE routine in a handler is where the data is sent out the I/O port to the external device.

Passed to routine:

AL	Channel number to write.
CX	Number of bytes to write.
ES	Segment address of write buffer.
BX	Offset address of write buffer.

Routine must return:

AL=00h	Successful write.
>00h	Error code.
BP	Unchanged from value passed to routine.
CX	The number of bytes actually written (return in register save area, offset 04h).

When routine is called:

- By the WRITE function (13h) if a high- or low-level handler name was specified when the handler was opened. The WRITE function can be invoked either by the BASIC PRINT #, PRINT #...USING or PUT # statements, or by an assembly language program.
- By a high-level handler using the LLH_WRITE linkage routine.

Notes:

- Registers specified by the caller of the WRITE function or the LLH_WRITE linkage routine are passed to the handler WRITE routine with the following exceptions:

DS	Set to the segment address of the scratch area allocated by the handler.
BP	Points to the offset on the stack where all the caller's registers are saved and where all returned values except AL must be put.
DI	Destroyed.

Cautions:

- The number of bytes to write must not be greater than the actual write buffer length (although it can be less).

Activities of routine:

High-Level Handler Activities	Low-Level Handler Activities
Perform device-specific write activities.	Enable the system timeout. *
Call low-level handler with LLH_WRITE linkage routine. The write buffer specified can be either the caller's buffer or one in the handler's scratch area.	Monitor system events (system timeout, power switch, and low battery) while outputting data. *
	Write the data to the I/O port.
Return the actual number of bytes written, and an error code if the routine failed (00h if no errors).	Disable the system timeout. *
	Return the actual number of bytes written, and an error code if the routine failed (00h if no errors).

* Refer to the appendixes for information about a utility routine to do this.

4

Operating System Functions

Contents

Chapter 4 Operating System Functions

4-1 Operating System Function Usage
4-1 Operating System Function Descriptions
4-1 Registers Passed to Operating System Functions
4-2 BEEP
4-3 BUFFER_STATUS
4-4 CLOSE
4-6 CREATE
4-8 CURSOR
4-9 DELETE
4-11 DISPLAY_ERROR
4-12 END_PROGRAM
4-14 FIND_FILE
4-16 FIND_NEXT
4-19 GET_CHAR
4-21 GET_LINE
4-23 GET_MEM
4-25 MEM_CONFIG
4-27 OPEN
4-29 PUT_CHAR
4-30 PUT_LINE
4-32 READ
4-35 REL_MEM
4-36 ROOM
4-37 SEEK
4-39 SET_INTR
4-41 TIMEOUT
4-43 TIME_DATE
4-45 WRITE

4

Operating System Functions

This chapter describes the operating system functions. These functions allow assembly language programs to simplify the interaction between assembly language programs and the HP-94 hardware: memory, keyboard, display (and display backlight), serial port, bar code port, power switch, low battery detection, real-time clock, and beeper. The BASIC interpreter also uses these functions to provide analogous capability to BASIC language programs.

Operating System Function Usage

Operating system functions are called by the following procedure:

- Load function code into register AH.
- Load any other function parameters into the corresponding registers.
- Issue a software interrupt `1Ah`.

When functions end, they pass results back in the registers listed in the function descriptions.

Operating System Function Descriptions

Function descriptions consist of:

- A brief description of the operating system function.
- A summary of the function call parameters.
- A summary of the function return parameters including any possible returned error codes.
- Supplementary notes and cautions on the use and behavior of the function.
- A list of related operating system functions.
- An example of the use of the operating system function. These examples are provided only to illustrate typical use of the various functions. Several of the examples contain data scratch areas embedded in the code, and consequently will only work if executed in RAM — they will not run in ROM or EPROM.

Registers Passed to Operating System Functions

Each operating system function saves the contents of all the registers passed to it, and returns those values to the caller when the function ends. The only registers altered by the functions are those that explicitly return particular values to the caller — all other registers will retain their original values. AL is always used to return error codes.

BEEP

Beep a high or low tone for a specified duration.

Call with:

AH=07h	BEEP function code.
AL=00h	Low tone.
=01h	High tone.
BL	Length of tone in 0.1 second units (0.1 - 25.5 seconds).

Returns:

Nothing.

Notes:

- When AL is greater than 01h, no action is performed.

Cautions:

- As soon as BEEP starts the beeper, the application program will continue to run; that is, the program does not wait for the beep to finish before resuming execution.
- BEEP can be called while the beeper is beeping. If the tone specified is different than the tone in progress, beeping will continue at the high tone and duration - the high tone and its duration will always take precedence, regardless of the order in which the tones are specified. If the tone specified is the same as the tone in progress, beeping will continue at either the remaining duration or the new duration, whichever is longer.

Related functions:

None.

Example:

The following example will do a one-second low beep.

```
BEEP            equ     07h             ;BEEP function code
LOTONE          equ     00h
HITONE          equ     01h
                .
                .
                .
                mov     ah,BEEP         ;BEEP function code
                mov     al,LOTONE       ;low tone...
                mov     bl,10           ;for 1 second
                int     1Ah             ;beep it.
                .
                .
                .
```

BUFFER_STATUS

Get the number of bytes in or flush either the key buffer or the receive buffer for the built-in serial port handler.

Call with:

AH=06h	BUFFER_STATUS function code.
AL=00h	Flush key buffer.
=01h	Get the number of bytes in the key buffer.
=02h	Flush the receive buffer for the built-in serial port handler.
=03h	Get the number of bytes in the receive buffer for the built-in serial port handler.

Returns:

DL	Number of bytes in the key buffer (AL=01h) or the receive buffer for the built-in serial port handler (AL=03h).

Notes:

- The operations performed when AL is 02h or 03h only apply to the buffer for the built-in serial port handler. For user-defined serial port handlers with their own buffers, these operations will not work.
- When AL is greater than 03h, no action is performed.

Related functions:

GET_CHAR, GET_LINE, READ

Example:

The following example will flush any characters in the key buffer and serial port receive buffer.

```
BUFFER_STATUS       equ     06h             ;BUFFER_STATUS function code
KBD_FLUSH           equ     00h
KBD_STAT            equ     01h
SER_FLUSH           equ     02h
SER_STAT            equ     03h
                    .
                    .
                    .
;
;                   initialize the key buffer and serial port receive buffer
;
                    mov     ah,BUFFER_STATUS    ;BUFFER_STATUS function code
                    mov     al,KBD_FLUSH
                    int     1Ah                 ;flush keyboard buffer
                    mov     al,SER_FLUSH
                    int     1Ah                 ;flush serial port receive buffer
                    .
                    .
                    .
```

CLOSE

Close and release an open channel.

Call with:

AH=10h	CLOSE function code.
AL	Channel number to close.

Returns:

AL=00h	Successful close.
=65h (101)	Illegal parameter.
=69h (105)	Channel not open.

Notes:

When closing channels 1 - 4, CLOSE will transfer control to the CLOSE routine of the user-defined handler specified when the channel was opened. The same registers passed to the CLOSE function will be passed to the user-defined handler CLOSE routine with the following exceptions:

DS	Set to the segment address of the scratch area allocated by the handler.
BP	Points to the offset on the stack where all the caller's registers are saved and where all returned values except AL must be put.
DI	Destroyed.

Refer to the "User-Defined Handlers" chapter for details.

- When closing channels 1 - 4 and the user-defined handler has returned from its CLOSE routine, the handler will no longer be in control of the device.
- Once a channel is closed, it may not be accessed until it is reopened.

Cautions:

- This function may not be called from the POWERON routine of a handler.

Related functions:

CREATE, DELETE, OPEN, READ, SEEK, WRITE

Example:

The following example procedure will close a file.

```
CLOSE           equ     10h             ;CLOSE function code
                .
                .
                .
;               fclose -- close an open file
;
;               call with:
;                       al = channel #
;
fclose          proc    near
                mov     ah,CLOSE        ;CLOSE function code
                int     1Ah             ;close the file
                or      al,al           ;set status for caller
                ret
fclose          endp
                .
                .
                .
```

CREATE

Allocate initial storage for a data file and build the directory table entry for the file.

Call with:

AH=11h	CREATE function code.
ES	Segment address of file name to create.
BX	Offset address of file name to create.
CX	Initial allocated size in paragraphs.
DX	Size increment in paragraphs.

Returns:

AL=00h	Successful create.
=65h (101)	Illegal parameter.
=66h (102)	Directory does not exist.
=68h (104)	Too many files. The directory is full.
=6Ch (108)	File already exists.
=6Dh (109)	Read-only access.
=6Fh (111)	No room for file.
CX	Start segment address of the file.
BX	Available free space in paragraphs (when AL=6Fh).

Notes:

- The file name must be uppercase.
- The file name must be terminated by either a null (00h) or a space (20h).
- Wild card characters are not allowed in the file name.
- If the file name is longer than 4 characters, only the first 4 characters (plus a leading directory number and colon, if any) will be used.
- If the file name contains a directory specifier, the file will be created in that directory. If the file name does not contain a directory specifier, the file will be created in directory 0 (main memory).
- The size increment is the expansion increment used when data is written past the end-of-file.
- CREATE will fail if an attempt is made to create a data file in a ROM or EPROM.
- A data file must be opened before it can be written to or read from.
- Allocated file space is automatically initialized to nulls (00h).
- If there is not enough free space in the directory to create a file with the specified number of paragraphs, CREATE will return 6Fh (111) in AL along with the number of available paragraphs in BX.

Cautions:

- This function may not be called from the POWERON routine of a handler.

Related functions:

CLOSE, DELETE, OPEN, READ, SEEK, WRITE

Example:

The following example procedure will create a file.

```
CREATE          equ     11h             ;CREATE function code
                .
                .
                .
;               fcreate -- create a file
;
;               call with:
;                       bx = offset address of file name buffer
;                       cx = initial size in paragraphs
;                       dx = size increment in paragraphs
;
fcreate         proc    near
                mov     ah,CREATE       ;CREATE function code
                push    cs              ;segment address of file name to ES
                pop     es
                int     1Ah             ;create the file
                or      al,al           ;set status for caller
                ret
fcreate         endp
                .
                .
                .
```

CURSOR

Move the display cursor or obtain its current position.

Call with:

AH=05h	CURSOR function code.
AL=00h	Get the current display cursor position.
=01h	Move the display cursor.
CL	Cursor column position 0 - 19 (for move cursor).
CH	Cursor row position 0 - 3 (for move cursor).

Returns:

CL	Current cursor column position 0 - 20 (for get cursor).
CH	Current cursor row position 0 - 3 (for get cursor).

Notes:

- When AL is greater then 01h, no action is performed.
- When an attempt is made to move the cursor outside the range of the display window, no action is performed.
- When a character is displayed in the last column of the display, the cursor will remain in that column until another character is displayed. In this case, it is considered to be in column position 20.

Related functions:

PUT_CHAR, PUT_LINE

Example:

The following example will move the cursor to column 0 of the current line.

```
CURSOR          equ     05h             ;CURSOR function code
                .
                .
                .
                mov     ah,CURSOR       ;CURSOR function code
                mov     al,00h          ;to get current cursor position
                int     1Ah             ;get cursor position
                mov     cl,00h          ;set to column 0
                mov     al,01h          ;to set cursor position
                int     1Ah             ;and set the new position
                .
                .
                .
```

DELETE

Delete a currently open data file. The area occupied by the file will be returned to the free space in the directory containing the file.

Call with:

AH=14h	DELETE function code.
AL	Channel number of the open file to delete.

Returns:

AL=00h	Successful delete.
=65h (101)	Illegal parameter. This error will also occur for deletes of channels 0 - 4.
=69h (105)	Channel not open.
=6Dh (109)	Read-only file.
=6Eh (110)	Access restricted.

Notes:

- The released area is merged with the other free space in the directory each time a file is deleted.
- An automatic CLOSE occurs after a DELETE.

Cautions:

- This function may not be called from the POWERON routine of a handler.

Related functions:

CLOSE, CREATE, OPEN, READ, SEEK, WRITE

Example:

The following example will delete a file by first opening and then deleting it.

```
DELETE          equ     14h             ;DELETE function code
OPEN            equ     0Fh             ;OPEN function code
                .
                .
                .
                mov     ah,OPEN         ;OPEN function code
                mov     al,5            ;use channel 5
                push    ds              ;put fname segment address into ES
                pop     es
                                        ;fname address offset into BX
                mov     bx,offset byte ptr fname
                int     1Ah             ;open the file
                or      al,al           ;successful open?
                jne     open_err        ;no -- handle the open error
                mov     ah,DELETE       ;DELETE function code
                mov     al,5            ;the open channel
                int     1Ah             ;delete the file
                or      al,al           ;successful delete?
                jne     del_err         ;no -- handle the delete error
                .
                .
                .
```

DISPLAY_ERROR

Display the specified numeric error code. The displayed message will be of the form `Error nnn`, where `nnn` is the decimal value of the error code.

Call with:

AH=18h	DISPLAY_ERROR function code.
AL	Error code.

Returns:

Nothing.

Notes:

- The error number in the displayed message will always be three decimal digits. For error codes less than 64h (100), leading zeroes will be added.
- Before displaying an error message the cursor is moved to the first column of the next line in the display. After displaying the message, the cursor will be placed in the first column to the right of the error message.
- A beep will occur when the error message is displayed.

Related functions:

None.

Example:

The following example will display error 101 (illegal parameter).

```
DISPLAY_ERROR       equ     18h             ;DISPLAY_ERROR function code
                    .
                    .
                    .
                    mov     ah,DISPLAY_ERROR    ;DISPLAY_ERROR function code
                    mov     al,101          ;illegal parameter error code
                    int     1Ah             ;and display the error message
                    .
                    .
                    .
```

END_PROGRAM

Terminate an application program. The application program can turn off the HP-94, specifying either cold or warm start at the next power on, or return to command mode.

Call with:

AH=00h	END_PROGRAM function code.
AL=00h	Cold start.
=01h	Warm start.
>01h	End application program and enter command mode.

Returns:

Nothing.

Notes:

- When cold start is selected, the HP-94 will be turned off. When power is turned back on, the program MAIN will be run if it exists, or command mode will be entered.
- When warm start is selected, all registers and flags will be stored by the HP-94 before turning off power. When power is turned back on, the registers and flags will be restored and program execution resumed.

Cautions:

- Warm start may not be used by an application program which uses the HP-94 operating system's stack. An application which is initialized from command mode must allocate its own stack area in order to do a warm start END_PROGRAM. If a warm start is attempted using the operating system's stack, error 219 will be generated and control will return to command mode.
- This function may not be called from the POWERON routine of a handler.
- A FAR RET can also be used to end a program, but there are some subtle side effects. Refer to "Program Execution" for details.

Related functions:

None.

Example:

The following example will end the program and return to command mode.

```
END_PROGRAM     equ     00h             ;END_PROGRAM function code
COLDSTART       equ     00h
WARMSTART       equ     01h
CMDMODE         equ     02h
                .
                .
                .
                mov     ah,END_PROGRAM  ;END_PROGRAM function code
                mov     al,CMDMODE      ;to enter command mode
                int     1Ah
                .
                .
                .
```

FIND_FILE

Find the first file to match a file name pattern. Wild card characters may be included in the file name.

Call with:

AH=16h	FIND_FILE function code.
ES	Segment address of the search file name.
BX	Offset address of the search file name.
DS	Segment address of the file information buffer.
DX	Offset address of the file information buffer.

Returns:

AL=00h	Successful FIND_FILE.
=65h (101)	Illegal parameter.
=66h (102)	Directory does not exist.
=67h (103)	File not found.
CX	Segment address of the directory table entry for the matched file.
DX	Offset address of the directory table entry for the matched file.

Notes:

- The search file name must be uppercase.
- There is one wild card character, "*", which matches any character at that position and all subsequent positions in the file name.
- If the search file name contains a directory specifier, only that directory will be searched. If it does not contain a directory specifier, directories 0 - 4 will be searched in ascending order. The system directory (directory 5) will not be searched by FIND_FILE.
- If the file name is longer than 4 characters, only the first 4 characters will be used.
- The file information buffer consists of 14 bytes formatted as follows:

Bytes	Data
00h - 06h	Name of first file found which matches the search file name.
07h	File type
08h - 09h	Start segment address of the file
0Ah - 0Bh	Low word of the end-of-data address
0Ch	High byte of the end-of-data address
0Dh	NUL (00h)

File names are of the form "d:name" where "d" is the directory number and "name" is the 1 - 4 byte file name terminated by a null (00h).

Related functions:

FIND_NEXT

Example:

The following example will search for the first file which matches a specific file name. It is part of the example included with the FIND_NEXT function. Since the code contains a scratch buffer and the file information buffer, it will not work in ROM.

```
FIND_FILE       equ     16h             ;FIND_FILE function code
                .
                .
                .
                mov     ah,FIND_FILE    ;FIND_FILE function code
                push    cs              ;ES:BX address of scratch buffer
                pop     es
                mov     bx,offset buffer
                push    cs              ;DS:DX address of file info buffer
                pop     ds
                mov     dx,offset fbuffer
                int     1Ah             ;find the first file
                call    errchk          ;check for error
                .
                .
                .
buffer          db      BUFSIZ+1 dup (?)    ;read buffer
                                            ;must be in RAM
fbuffer         db      14 dup (?)          ;find filename dest buffer
                                            ;must be in RAM
                .
                .
                .
```

FIND_NEXT

Find the next file to match a file name pattern set up by a FIND_FILE function call.

Call with:

AH=17h	FIND_NEXT function code.

Returns:

AL=00h	Successful FIND_NEXT.
=67h (103)	File not found.
CX	Segment address of the directory table entry for the matched file.
DX	Offset address of the directory table entry for the matched file.

Notes:

- The format of the file information buffer is the same as that of the FIND_FILE function.
- The FIND_FILE function must be executed before FIND_NEXT.
- FIND_NEXT will return data in the area specified by the last FIND_FILE function call.

Cautions:

- FIND_NEXT will search only in the directory in which FIND_FILE found the first matching file name.

Related functions:

FIND_FILE

Example:

The following example will prompt for a file name and use FIND_FILE and FIND_NEXT to find and display all the files which match that file name. Since the code contains a scratch buffer and the file information buffer, it will not work in ROM.

```
GET_CHAR        equ     01h             ;GET_CHAR function code
ECHO            equ     00h
NOECHO          equ     01h
GET_LINE        equ     02h             ;GET_LINE function code
PUT_LINE        equ     04h             ;PUT_LINE function code
FIND_FILE       equ     16h             ;FIND_FILE function code
FIND_NEXT       equ     17h             ;FIND_NEXT function code
DISPLAY_ERROR   equ     18h             ;DISPLAY_ERROR function code
BUFSIZ          equ     80

code            segment
                assume cs:code,ds:code
prog            proc    far
start:

                dw      prgmend-start
                dw      0006h           ;offset of internal entry point
                db      0100h           ;version 1.00

                push    cs              ;set DS to CS
                pop     ds
```

```
                        ; display "DIR> " prompt
                        mov         bx,offset DIR
                        call        puts
                        ; get name of file
                        mov         bx,offset buffer
                        mov         al,BUFSIZ
                        call        gets
                        call        errchk
                        ; null out the terminating CR
                        sub         dh,dh               ;clear out dh
                        mov         di,dx               ;and put it into di
                        mov         byte ptr buffer[di],00h;null out last byte
                        ; go through buffer and substitute ":" for "."
                        sub         dx,dx               ;clear out dx
                        mov         di,dx               ;and put it into di
dots01:
                        mov         al,byte ptr buffer[di];get the next character
                        cmp         al,"."              ;is it a dot?
                        jne         nodot               ;no, don't swap it
                        mov         al,":"              ;get a ":"
                        mov         byte ptr buffer[di],al;and save it
nodot:
                        inc         di                  ;increment di
                        or          al,al               ;is al null?
                        jne         dots01              ;no, check next byte

                        mov         ah,FIND_FILE        ;FIND_FILE function code
                        push        cs                  ;ES:BX address of buffer
                        pop         es
                        mov         bx,offset buffer
                        mov         dx,offset fbuffer   ;DS:DX address of file info buffer
                        int         1Ah                 ;find the first file
                        call        errchk              ;check for error
                        mov         byte ptr lcnt,5     ;reset display line counter
loop01:
                        dec         byte ptr lcnt       ; decrement line counter
                        jne         pause01             ;not 0 -- don't pause
                        mov         ah,GET_CHAR         ;GET_CHAR function code
                        mov         al,NOECHO           ;don't echo
                        int         1Ah                 ;get a character
                        mov         byte ptr lcnt,4     ;reset the counter (less this line)
                        cmp         dl,"."              ;was it a dot?
                        jne         pause01             ;no -- leave the counter as is
                        mov         byte ptr lcnt,1     ;only 1 line
pause01:
                        ; display file name from fbuffer
                        mov         bx,offset CRLF      ;display a cr/lf
                        call        puts
                        mov         bx,offset fbuffer
                        call        puts
                        mov         ah,FIND_NEXT        ;FIND_NEXT function code
                        int         1Ah                 ;find the next file
                        or          al,al               ;returned a 0?
                        je          loop01              ;yes -- display the file (if found)
                        mov         al,GET_CHAR         ;GET_CHAR function code
                        mov         ah,NOECHO           ;don't echo
                        int         1Ah                 ;wait for a key
exit:
                        ret                             ;if no more files, done.
prog                    endp
```

```
puts            proc    near
                mov     ah,PUT_LINE         ;PUT_LINE function code
                push    cs                  ;segment address of buffer to ES
                pop     es
                int     1Ah                 ;display it
                ret
puts            endp

gets            proc    near
                mov     ah,GET_LINE         ;GET_LINE function code
                push    cs                  ;CS to ES
                pop     es
                int     1Ah                 ;get a line
                ret
gets            endp

errchk          proc    near
                or      al,al               ;return code 0?
                je      errret              ;yes -- just return
                mov     ah,DISPLAY_ERROR    ;DISPLAY_ERROR function code
                int     1Ah                 ;display the error message
                add     sp,2                ;pull off the near return address
                jmp     exit                ;terminate program
errret:
                ret
errchk          endp

lcnt            db      ?
DIR             db      "DIR> ",0Fh,00h     ;prompt, alpha mode
CRLF            db      0Dh,0Ah,00h         ;cr/lf
buffer          db      BUFSIZ+1 dup (?)    ;read buffer
                                            ;must be in RAM
fbuffer         db      14 dup (?)          ;find filename dest buffer
                                            ;must be in RAM

prgmend:
code            ends
                end
```

GET_CHAR

Get one character from the key buffer and optionally echo it to the display.

Call with:

AH=01h	GET_CHAR function code.
AL=00h	Echo the character being read.
>00h	Do not echo the character being read.

Returns:

AL=00h	Successful read.
=76h (118)	Timeout. A timeout occurred before a key was pressed.
=77h (119)	Power switch pressed.
=C8h (200)	Low battery.
DL	Character read from the key buffer and optionally echoed to the display.

Notes:

- When the key buffer is empty, GET_CHAR will wait for a key.
- The [SHIFT] key cannot be read.
- The keys [CLEAR], [←], and [ENTER], return the codes 18h, 7Fh, and 0Dh respectively. They are never echoed to the display, nor are any control codes (00h-1Fh) or user-defined characters (80h-8Fh), the first 16 of which correspond to user-defined keys.
- The following behavior only applies when echoing to the display: When a character is echoed to the last column of a line in the display, the cursor will remain over that character. The display will be scrolled, if necessary, and the cursor moved to the first column of the next line before echoing the next character.

Related functions:

BUFFER_STATUS, GET_LINE

...GET_CHAR

Example:

The following example will wait for the ENTER key to be pressed.

```
GET_CHAR        equ     01h             ;GET_CHAR function code
ECHO            equ     00h
NOECHO          equ     01h
                .
                .
                .
entwait:
                mov     ah,GET_CHAR     ;GET_CHAR function code
                mov     al,NOECHO       ;don't echo the character read
                int     1Ah             ;read a character
                or      al,al           ;read error (al <> 0)?
                jne     rd_err          ;yes -- process read error
                cmp     dl,0Dh          ;[ENTER] key?
                jne     entwait         ;no -- wait for another key
                .
                .
                .
```

GET_LINE

Get a character string from the keyboard buffer and echo it to the display.

Call with:

AH=02h	GET_LINE function code.
AL	Maximum number of bytes to read (1 - 255 bytes).
ES	Segment address of the read buffer.
BX	Offset address of the read buffer.

Returns:

AL=00h	Successful read.
=76h (118)	Timeout. A timeout occurred before a key was pressed.
=77h (119)	Power switch pressed.
=C8h (200)	Low battery.
DL	Number of characters read from keyboard buffer.

Notes:

- If the key buffer does not contain an [ENTER], GET_LINE will wait until [ENTER] is pressed.
- The terminating [ENTER] will not be echoed to the display. The cursor will be left at the column to the right of the character before the [ENTER].
- The buffer must contain enough space for the maximum number of bytes to read (register AL), as well as one byte for the terminating [ENTER].
- The character count returned in DL does not include the terminating [ENTER].
- The buffer returned by GET_LINE will contain the terminating [ENTER] (0Dh).
- When AL characters have been read and [ENTER] has not been pressed, subsequent characters will be discarded, and a low beep issued, until [ENTER] is pressed.
- GET_LINE processes [←] and [CLEAR] separately. If there are any characters in the buffer, [←] will remove the last character from the input buffer, erase the character from the display and move the display cursor back one character position. [CLEAR] will clear the entire input buffer and erase the entire input line from the display.
- When a character is displayed in the last column of a line in the display, the cursor will remain over that character. The display will be scrolled, if necessary, and the cursor move to the first column of the next line before displaying the next character.
- If timeout, power switch, or low battery interrupts occur, the buffer will contain any characters already entered. The DL register will contain the number of characters actually read.

Cautions:

- GET_LINE will not check for wraparound of the read buffer's offset address.

...GET_LINE

Related functions:

BUFFER_STATUS, GET_CHAR

Example:

The following example will read in a 20-character string. Since the code contains the read buffer for GET_LINE, it will not work in ROM.

```
GET_LINE        equ     02h             ;GET_LINE function code
BUFSIZ          equ     20
buffer          db      BUFSIZ+1 dup (?)    ;must be in RAM
                .
                .
                .
                mov     ah,GET_LINE     ;GET_LINE function code
                mov     al,BUFSIZ       ;size of buffer
                push    ds              ;set ES to DS
                pop     es
                mov     bx,offset buffer    ;buffer offset to BX
                int     1Ah             ;read string
                or      al,al           ;read error (al <> 0)?
                jne     rd_err          ;yes -- process read error
                .
                .
                .
```

GET_MEM

Allocate a scratch area of memory.

Call with:

AH=0Bh	GET_MEM function code.
AL	Channel number if the request is being made by a handler.
=00h	If the request is not being made by a handler.
BX	Size of the requested area in paragraphs.

Returns:

AL=00h	Successful allocation.
=65h (101)	Illegal parameter. Invalid channel number.
=6Eh (110)	Access restricted. No scratch areas available or main memory not initialized.
=71h (113)	No room for scratch area.
CX	Segment address of scratch area.
DX	Length in paragraphs of scratch area.

Notes:

- A maximum of 34 scratch areas may be allocated.
- Scratch memory is automatically initialized to nulls (00h).
- Handlers should set AL to the channel number to which they are open, or to zero, depending on whether or not they want the operating system to pass this scratch area address to all their routines. See the "User-Defined Handlers" chapter for details.

Cautions:

- This function may not be called from the POWERON routine of a handler.

Related functions:

REL_MEM

Example:

The following example will allocate a 10-paragraph (160-byte) scratch area.

```
GET_MEM         equ     0Bh             ;GET_MEM function code
SCRSIZ          equ     0Ah             ;scratch area size (10 paragraphs)
                .
                .
                .
                mov     ah,GET_MEM      ;GET_MEM function code
                mov     al,00h          ;called by an application (not a handler)
                mov     bx,SCRSIZ       ;size of scratch area
                int     1Ah
                or      al,al           ;error?
                jne     get_mem_err     ;yes -- handle it
                ;
                ;       CX=segment address of scratch area initialized to nulls
                ;       DX=length of allocated scratch area (0Ah)
                ;
                .
                .
                .
```

MEM_CONFIG

Get the current memory configuration of the HP-94.

MEM_CONFIG returns 5 bytes of configuration information. Bytes 0 - 4 describe the contents of directories 0 - 4 respectively as follows.

Value		Meaning
Hex	ASCII	
00h	NUL	No memory installed
4Dh	M	Main memory
41h	A	40K RAM card
4Fh	O	ROM/EPROM card

Call with:

AH=0Dh	MEM_CONFIG function code.
ES	Segment address of the 5-byte configuration buffer.
BX	Offset address of the 5-byte configuration buffer.

Returns:

AL	Number of directories with memory installed. This value is the same as the number of bytes in the configuration buffer which contain a non-zero value.

Related functions:

ROOM

...MEM_CONFIG

Example:

The following program will display the number of installed directories followed by the type of each directory. Since the code contains the configuration buffer, it will not work in ROM.

```
MEM_CONFIG      equ     0Dh             ;MEM_CONFIG function code
PUT_CHAR        equ     03h             ;PUT_CHAR function code
END_PROGRAM     equ     00h             ;END_PROGRAM function code
CMDMODE         equ     02h
code            segment
                assume  cs:code,ds:code
mem             proc    far
start:
                dw      prgmend-start
                dw      0006h           ;offset of internal entry point
                dw      0100h           ;version 1.00

                mov     ax,cs           ;set DS to CS
                mov     ds,ax
                mov     es,ax           ;set ES to segment addr of membuf
                mov     bx,offset membuf ;set BX to offset addr of membuf
                mov     ah,MEM_CONFIG   ;MEM_CONFIG function code
                int     1Ah             ;get it
                add     al,"0"          ;turn al into a number
                mov     ah,PUT_CHAR     ;PUT_CHAR function code
                int     1Ah             ;display it
                mov     al,":"          ;display a ":"
                mov     ah,PUT_CHAR
                int     1Ah
                mov     ax,offset membuf ;set DI to offset addr of membuf
                mov     di,ax
                mov     cx,5            ;number of bytes to check
mem01:
                mov     al,byte ptr [di] ;get it
                cmp     al,00h          ;is it 00h?
                jne     notzero         ;no, leave it alone
                mov     al,"-"          ;change it to a "-"
notzero:
                mov     ah,03h          ;PUT_CHAR function code
                int     1Ah             ;display it
                inc     di              ;increment offset (DI)
                loop    mem01           ;and do the next character
                mov     ah,END_PROGRAM  ;enter command mode
                mov     al,CMDMODE
                int     1Ah

membuf          db      5 dup (?)       ;must be in RAM

prgmend:
mem             endp
code            ends
                end
```

OPEN

Open a data file or handler and assign it to a specific channel.

Call with:

AH=0Fh	OPEN function code.
AL	Channel number to open.
ES	Segment address of file or handler name string to open.
BX	Offset address of file or handler name string to open.
DS	Segment address of parameter area (built-in serial port handler only).
DX	Offset address of parameter area (built-in serial port handler only).

Returns:

AL=00h	Successful open.
=65h (101)	Illegal parameter.
=66h (102)	Directory does not exist.
=67h (103)	File or handler not found.
=6Ah (106)	Channel already open.
=6Bh (107)	File or handler already open.
=6Eh (110)	Access restricted. The specified file is not a data file or handler.
CX	Segment address of the data file or handler.

Notes:

- The OPEN function will search for the file (type D) or handler (type H) with the specified name in directories 0 - 4 in ascending order, or only in a specified directory (e.g., "2:ABCD").
- Channel 0 (keyboard for read operations, display for write operations) is always open. If channel 0 is opened, AL will always return zero. The handler name string is ignored when opening channel 0.
- When opening channels 1 - 4, if the handler is not found, or if a null string was specified as the handler name, the default handler will be used. For channel 1, the default handler is the built-in serial port handler. For channels 2 - 4, there is no built-in handler, and the OPEN function will report error 65h.

  Once the handler is found (either user-defined or built-in), the OPEN function will transfer control to the OPEN routine of the handler. The handler will then become associated with the device, and the CLOSE, READ, and WRITE functions will transfer control to the CLOSE, READ, and WRITE routines of the handler. The same registers passed to the OPEN function will be passed to the user-defined handler OPEN routine with the following exceptions:

BP	Points to the offset on the stack where all the caller's registers are saved and where all returned values except AL must be put.
DI	Destroyed.

  Refer to the "User-Defined Handlers" chapter for details.

- If the name string is longer than 4 characters, only the first 4 characters (plus a leading directory number and colon, if any) will be used by the OPEN function. The entire handler name string (pointed to by ES : BX) will be passed to the OPEN routine of user-defined handlers. This name string can include a high- and low-level handler pair (such as "HNWN;HNBC"), or in-line parameters if the handler allows them (e.g., "RSHN 9600,7ES").
- Alphabetic characters in the name string must be uppercase.
- The file or handler name part of the name string must be terminated by either a null (00h) or a space (20h).
- The wild card character "*" is not allowed in the file or handler name part of the name string.
- The parameter area address (DS : DX) is only used when the built-in serial port handler is opened. The meanings of the parameters are defined in the "Serial Port" chapter. Refer to the "User-Defined Handlers" chapter for a discussion of passing configuration parameters to user-defined handlers using the handler information table.

Cautions:

- This function may not be called from the POWERON routine of a handler.

Related functions:

CLOSE, CREATE, DELETE, READ, SEEK, WRITE

Example:

The following example will open the serial port (channel 1) with the built-in handler.

```
OPEN        equ     0Eh             ;OPEN function code
            .
            .
            .
spmode      db      1               ;9600 baud
            db      00001101b       ;XON/XOFF,7 bits, even parity,
                                    ;1 stop, null strip disabled
            db      0Dh             ;terminate on CR
null        db      00h             ;the null string

            mov     ah,OPEN         ;OPEN function code
            mov     al,1            ;serial port channel
            push    cs              ;use default handler (ES:BX = null string)
            pop     es
            mov     bx,offset null
            push    cs              ;DS:DX = port config buffer
            pop     ds
            mov     dx,offset spmode
            int     1Ah             ;open the port
            or      al,al           ;error?
            jne     open_err        ;yes -- process the error
            .
            .
            .
```

PUT_CHAR

Display one character on the display and move the cursor one column to the right.

Call with:

AH=03h	PUT_CHAR function code.
AL	Character to display.

Returns:

Nothing.

Notes:

- When a character is written to the last column of a line in the display, the cursor will remain over that character. The display will be scrolled, if necessary, and the cursor moved to the first column of the next line before writing the next character.

Cautions:

- While processing the display control character that homes the cursor and clears the screen (0Ch), interrupts are disabled for ~45 ms. This time may be important to serial and bar code port handlers.

Related functions:

CURSOR, PUT_LINE

Example:

The following example will turn on the backlight, change the keyboard into alpha mode and display a prompt character.

```
PUT_CHAR        equ     03h             ;PUT_CHAR function code
ELON            equ     1Eh
ELOFF           equ     1Fh
ALPHMODE        equ     0Fh
NUMMODE         equ     0Eh
                .
                .
                .
                mov     ah,PUT_CHAR     ;PUT_CHAR function code
                mov     al,ELON         ;turn on backlight
                int     1Ah
                mov     al,ALPHMODE     ;alpha mode keyboard
                int     1Ah
                mov     al,">"          ;prompt character
                int     1Ah
                .
                .
                .
```

PUT_LINE

Display a character string on the display.

Call with:

AH=04h	PUT_LINE function code.
ES	Segment address of the write string.
BX	Offset address of the write string.

Returns:

Nothing.

Notes:

- The write string must be terminated with a null character (00h); the null will not be displayed. Any other ASCII character, including display control characters, may be embedded in the string.
- When a character is written to the last column of a line in the display, the cursor will remain over that character. The display will be scrolled, if necessary, and the cursor moved to the first column of the next line before writing the next character.

Cautions:

- While processing the display control character that homes the cursor and clears the screen (0Ch), interrupts are disabled for ~45 ms. This time may be important to serial and bar code port handlers.
- PUT_LINE will not check for wraparound of the write buffer's offset address.

Related functions:

CURSOR, PUT_CHAR

Example:

The following program will display the message "Hello world".

```
PUT_LINE        equ     04h             ;PUT_LINE function code
END_PROGRAM     equ     00h             ;END_PROGRAM function code
CMDMODE         equ     02h
code            segment
                assume  cs:code,ds:code
hiworld         proc    far
start:
                dw      prgmend-start
                dw      0006h           ;offset of internal entry point
                dw      0100h           ;version 1.00

                push    cs              ;set DS to CS
                pop     ds
                .
                .
                .
                mov     ah,PUT_LINE     ;PUT_LINE function code
                push    ds              ;set ES to DS
                pop     es
                mov     bx,offset msg   ;buffer offset to BX
                int     1Ah             ;write string to LCD
                .
                .
                .
                mov     ah,END_PROGRAM  ;enter command mode
                mov     al,CMDMODE
                int     1Ah
prgmend:
hiworld         endp
msg             db      "Hello world",0Dh,0Ah,00h
code            ends
                end
```

READ

Read data from an open channel.

Call with:

AH=12h	READ function code.
AL	Channel number to read.
CX	Number of bytes to read.
ES	Segment address of read buffer.
BX	Offset address of read buffer.

Returns:

AL=00h	Successful read.
=65h (101)	Illegal parameter.
=69h (105)	Channel not open.
=73h (115)	Short record detected.
=74h (116) *	Terminate character detected.
=75h (117)	End of data.
=76h (118)	Timeout. A timeout occurred before the read was completed.
=77h (119) †	Power switch pressed.
=C8h (200) †	Low battery.
=C9h (201) †	Receive buffer overflow.
=CAh (202) *	Parity error.
=CBh (203) *	Overrun error.
=CCh (204) *	Parity and overrun error.
=CDh (205) *	Framing error.
=CEh (206) *	Framing and parity error.
=CFh (207) *	Framing and overrun error.
=D0h (208) *	Framing, overrun and parity error.
CX	The number of bytes actually read.

* Can only occur when reading from channels 1 - 4. Whether these errors occur for a user-defined handler depends on the handler.

† Can only occur when reading from channels 0 - 4. Whether these errors occur for a user-defined handler depends on the handler.

Notes:

- When reading data from channels 1 - 4, READ will transfer control to the READ routine of the user-defined handler specified when the channel was opened. The same registers passed to the READ function will be passed to the user-defined handler READ routine with the following exceptions:

  DS Set to the segment address of the scratch area allocated by the handler.

  BP Points to the offset on the stack where all the caller's registers are saved and where all returned values except AL must be put.

  DI Destroyed.

  Refer to the "User-Defined Handlers" chapter for details.

- Timeout, power switch and low battery will cause reads from channels 0 - 4 to be aborted, but will not interrupt reads from channels 5 - 15. Device I/O will be halted by these conditions, but file I/O will always be completed (unless the reset switch is pressed or the machine turns off automatically because of very low battery).
- When reading data from the keyboard (channel 0), no echoing to the display will occur. All keys pressed (except [SHIFT]) will be returned, unlike the GET_CHAR and GET_LINE functions. The number of bytes to read determines when READ will end, whether or not the [ENTER] key was pressed.
- When reading data from the built-in serial port handler (channel 1), if a terminate character was specified when the channel was opened, READ will stop if the terminate character is received even though the full read count has not been reached. The terminate character will be placed in the read buffer, but it will not be included in the returned read length, and error 74h will be reported.
- When reading data from a file (channels 5 - 15), data is read from the current file access pointer position. After the read is complete, the file access pointer is advanced by the size of the data read.
- Error 65h will occur if the number of bytes to read would cause the read buffer's offset address to wraparound.

Related functions:

CLOSE, CREATE, DELETE, OPEN, SEEK, WRITE

Cautions:

- This function may not be called from the POWERON routine of a handler.
- The number of bytes to read must not be greater than the actual read buffer length (although it can be less).

Example:

The following example will read from a channel.

```
READ                    equ         12h                 ;READ function code
                        .
                        .
                        .
;                       fread -- read a channel into a buffer
;
;                       call with:
;                                   al = channel #
;                                   cx = number of bytes to read
;                                   es = segment address of read buffer
;                                   bx = offset address of read buffer
;
fread                   proc        near
                        mov         ah,READ             ;READ function code
                        int         1Ah                 ;read the channel
                        or          al,al               ;set status for caller
                        ret
fread                   endp
                        .
                        .
                        .
```

REL_MEM

Release a scratch area obtained via GET_MEM.

Call with:

AH=0Ch	REL_MEM function code.
CX	Segment address of scratch area to release.

Returns:

AL=00h	Successful release.
=6Eh (110)	Access restricted. No free blocks available.
=72h (114)	Scratch area does not exist. Scratch area address does not correspond to a currently allocated scratch area.

Cautions:

- This function may not be called from the POWERON routine of a handler.

Related functions:

GET_MEM

Example:

The following example will free a the scratch area addressed by the current extra data segment (ES).

```
REL_MEM         equ     0Bh             ;REL_MEM function code
                .
                .
                .
                mov     cx,es           ;segment address of scratch area into cx
                mov     ah,REL_MEM      ;REL_MEM function code
                int     1Ah
                or      al,al           ;error?
                jne     rel_mem_err     ;yes -- handle it
                .
                .
                .
```

ROOM

Identify available room in a directory.

Call with:

AH=0Eh	ROOM function code.
AL	Directory number (0 - 4).

Returns:

AL=00h	Successful request.
=65h (101)	Illegal parameter. Invalid directory number.
=66h (102)	Directory does not exist.
BX	The available directory free space in paragraphs.
CX	Segment address of directory table.
DX	Total memory in directory in paragraphs, not including directory table.

Notes:

Related functions:

MEM_CONFIG

Example:

The following example will get the remaining space in main memory.

```
ROOM            equ     0Dh             ;ROOM function code
                .
                .
                .
                mov     al,0            ;directory 0
                mov     ah,ROOM         ;ROOM function code
                int     1Ah
                ;
                ;       BX=available free space in paragraphs
                ;       CX=segment address of directory table
                ;       DX=total memory in directory 0 in paragraphs
                ;
                .
                .
                .
```

SEEK

Move the file access pointer of an open file, or get the current pointer position.

Call with:

AH=15h	SEEK function code.
AL	Channel number.
BL=00h	Read the current file access pointer position.
=01h	Seek relative to the start of the file.
=02h	Set the file access pointer to EOD.
CX	High byte of 24-bit seek offset — CH ignored (for BL=00h or 01h).
DX	Low word of 24-bit seek offset (for BL=00h or 01h).

Returns:

AL=00h	Successful seek.
=65h (101)	Illegal parameter. This error will also occur for seeks on channels 0 - 4.
=69h (105)	Channel not open.
CX	High byte of the current 24-bit file access pointer (CH always set to zero).
DX	Low word of the current 24-bit file access pointer.

Notes:

- Seeks past EOD will generate error 65h.
- The 24-bit seek offset and file access pointer are relative to the start of the file. The first byte of the file has a seek offset and file access pointer position of 0.
- The file access pointer is set to 0 when the file is opened.

Cautions:

- This function may not be called from the POWERON routine of a handler.

Related functions:

CLOSE, CREATE, DELETE, OPEN, READ, WRITE

...SEEK

Example:

The following example will seek to EOD to get the current true file size.

```
SEEK            equ     15h             ;SEEK function code
channel         db      5               ;channel to seek on
                .
                .
                .
                mov     ah,SEEK
                mov     al,channel      ;channel to seek on
                mov     bl,02h          ;SEEK to EOD
                int     1Ah             ;SEEK...
                or      al,al           ;SEEK error?
                jne     seek_err        ;yes, process it.
;
;               CX,DX now contain the exact number of bytes in the file,
;               regardless of padding to the nearest paragraph boundary.
;
                .
                .
                .
```

SET_INTR

Two system interrupts may put under program control — the power switch/system timeout and low battery interrupt. In addition, the power switch interrupt may be disabled or enabled.

Call with:

AH=0Ah	SET_INTR function code.
AL=00h	Define a power switch/system timeout interrupt routine.
=01h	Define a low battery interrupt routine.
=02h	Disable the power switch interrupt.
>02h	Enable the power switch interrupt.
BX	The data segment to be used for the interrupt routine. This value will be loaded into DS before the interrupt routine is activated (for AL=00h or 01h).
CX	Segment address of interrupt routine (for AL=00h or 01h).
DX	Offset address of interrupt routine (for AL=00h or 01h).

Returns:

Nothing.

Notes:

- An interrupt can be restored to its default behavior by calling SET_INTR with both CX=00h and DX=00h.
- When the power switch/timeout interrupt routine is called, the AL register will be set to 76h (118) if a timeout occurred, or 77h (119) if the power switch was pressed.

Cautions:

- The offset address specified in DX *must be non-zero* for the operating system to properly interpret the existence of user-defined interrupt routines.

Related functions:

None.

...SET_INTR

Example:

The following example will set up a power switch interrupt routine.

```
SET_INTR        equ     0Ah             ;SET_INTR function code
                .
                .
                .
                mov     ax,ds           ;put DS in BX
                mov     bx,ax
                mov     ax,cs           ;put CS in CX
                mov     cx,ax
                mov     dx,offset psint ;put routine offset into DX
                mov     ah,SET_INTR     ;SET_INTR function code
                mov     al,00h          ;set power switch interrupt routine
                int     1Ah             ;set it
                .
                .
                .
psint           proc    far             ;power switch interrupt routine
                .                       ;  interrupt routine for
                .                       ;  power switch goes here
                .
                ret                     ;return from interrupt
psint           endp
                .
                .
                .
```

TIMEOUT

Set the display backlight timeout and system timeout intervals. The timeouts may be as short as 1 second, as long as 1800 seconds, or disabled.

Call with:

AH=09h	TIMEOUT function code.
AL=00h	Set the display backlight timeout interval.
=01h	Set the system timeout interval.
BX	Number of seconds to set timeout to (1 - 1800). A value of 0 may be used to disable the timeout completely, in which case the backlight will never turn off or the system will never timeout (turn itself off).

Returns:

Nothing.

Notes:

- The initial value at cold start for both timeout intervals is 120 seconds.
- If AL is greater then 01h, no action is performed.
- If BX is greater than 1800 (0708h), no action is performed.
- Setting the display backlight timeout only sets the interval — it does not turn on the backlight. The backlight is turned on programmatically by writing the display control character 1Eh to the display.

Cautions:

- Leaving the backlight on continuously or for long periods of time (greater than 5 minutes) will reduce the life of the backlight.
- If the backlight is on and a new timeout interval is set, the backlight must be turned off (either programmatically or by timeout) before the new timeout interval will be in effect.

Related functions:

None.

...TIMEOUT

Example:

The following example will set the display backlight timeout interval to 3 minutes, and disable the system timeout.

```
TIMEOUT         equ     09h             ;TIMEOUT function code
                .
                .
                .
                mov     ah,TIMEOUT      ;TIMEOUT function code
                mov     al,00h          ;display backlight timeout
                mov     dx,180          ;3 minutes (180 seconds)
                int     1Ah             ;set it
                mov     al,01h          ;system timeout interval
                mov     dx,00h          ;disable timeout
                int     1Ah             ;set it
                .
                .
                .
```

TIME_DATE

Read or set the time and date of the real-time clock. The time and date is read into or set from a 17-byte fixed-length buffer. The format of the buffer is:

```
MM/DD/YY,hh:mm:ss
```

Symbol	Value	Range	Symbol	Value	Range
MM	Month	01-12	hh	Hour	00-23
DD	Date	01-31	mm	Minute	00-59
YY	Year	00-99	ss	Second	00-59

Call with:

AH=08h	TIME_DATE function code.
AL=00h	Set time and date.
=01h	Read time and date.
ES	Segment address of the time and date buffer.
BX	Offset address of the time and date buffer.

Returns:

Nothing.

Notes:

- When AL is greater than 01h, no action is performed.

Cautions:

- The validity of the time and date is not checked. If times and dates are set outside the above ranges, the clock wil be set to unpredictable values.

Related functions:

None.

Example:

The following program will read the current time and date and write it to the display. Since the code contains the read buffer for TIME_DATE, it will not work in ROM.

```
TIME_DATE       equ     08h             ;TIME_DATE function code
PUT_LINE        equ     04h             ;PUT_LINE function code
END_PROGRAM     equ     00h             ;END_PROGRAM function code
CMDMODE         equ     02h
TDBUFLEN        equ     17
code            segment
                assume  cs:code,ds:code
mem             proc    far
start:
                dw      prgmend-start
                dw      0006h           ;offset of internal entry point
                dw      0100h           ;version 1.00

                mov     ax,cs           ;set DS to CS
                mov     ds,ax
                mov     ah,TIME_DATE    ;TIME_DATE function code
                mov     al,01h          ;get date
                push    ds              ;set ES to DS
                pop     es
                mov     bx,offset buffer
                int     1Ah             ;get the time and date
                mov     ah,PUT_LINE     ;PUT_LINE function code
                int     1Ah             ;display it
                .
                .
                .
                mov     ah,END_PROGRAM  ;enter command mode
                mov     al,CMDMODE
                int     1Ah

buffer          db      TDBUFLEN dup (?)    ;must be in RAM
                db      0Dh,0Ah,00h
prgmend:
mem             endp
code            ends
                end
```

Write data to an open channel.

Call with:

AH=13h	WRITE function code.
AL	Channel number to write.
CX	Number of bytes to write.
ES	Segment address of write buffer.
BX	Offset address of write buffer.

Returns:

AL=00h	Successful write.
=65h (101)	Illegal parameter.
=69h (105)	Channel not open.
=6Dh (109)	Read-only access.
=70h (112)	No room to expand file.
=76h (118) *	Timeout. A timeout occurred before the write was completed.
=77h (119) *	Power switch pressed.
=C8h (200) *	Low battery.
=DAh (218) *	Lost connection while transmitting. The Clear to Send (CTS) control line was lowered.
CX	The number of bytes actually written.

Notes:

- When writing data to channels 1 - 4, WRITE will transfer control to the WRITE routine of the user-defined handler specified when the channel was opened. The same registers passed to the WRITE function will be passed to the user-defined handler WRITE routine with the following exceptions:

  DS — Set to the segment address of the scratch area allocated by the handler.

  BP — Points to the offset on the stack where all the caller's registers are saved and where all returned values except AL must be put.

  DI — Destroyed.

  Refer to the "User-Defined Handlers" chapter for details.

* Can only occur when writing to channels 1 - 4. Whether these errors occur for a user-defined handler depends on the handler.

- Timeout, power switch and low battery will cause writes to channels 1 - 4 to be aborted, but will not interrupt writes to channels 5 - 15. Device I/O will be halted by these conditions, but file I/O will always be completed (unless the reset switch is pressed or the machine turns off automatically because of very low battery).
- When writing data to the built-in serial port handler (channel 1), if a terminate character was specified, the terminate character will be written after writing the data in the write buffer.
- When writing data to a file (channels 5 - 15), data is written from the current file access pointer position. After the write is complete, the file access pointer is advanced by the size of the buffer written.
- A write of 0 bytes to a data file will cause the EOD to be set equal to the current file access pointer. This has the effect of truncating the data in the file to the current pointer position, even though the file size will remain unchanged.
- Error 65h will occur if the number of bytes to write would cause the write buffer's offset address to wraparound.

Cautions:

- This function may not be called from the POWERON routine of a handler.
- The number of bytes to write must not be greater than the actual write buffer length (although it can be less).

Related functions:

CLOSE, CREATE, DELETE, OPEN, READ, SEEK

Example:

The following program will append one data file to another. If the destination file does not exist, it will be created. The program illustrates the use of the OPEN, CLOSE, CREATE, READ, WRITE and SEEK functions. Since the code contains the buffer address and length, it will not work in ROM.

```
GET_LINE            equ     02h             ;GET_LINE function code
PUT_CHAR            equ     03h             ;PUT_CHAR function code
PUT_LINE            equ     04h             ;PUT_LINE function code
GET_MEM             equ     0Bh             ;GET_MEM function code
REL_MEM             equ     0Ch             ;REL_MEM function code
OPEN                equ     0Fh             ;OPEN function code
CLOSE               equ     10h             ;CLOSE function code
CREATE              equ     11h             ;CREATE function code
READ                equ     12h             ;READ function code
WRITE               equ     13h             ;WRITE function code
SEEK                equ     15h             ;SEEK function code
DISPLAY_ERROR       equ     18h             ;DISPLAY_ERROR function code

BUFSIZ              equ     4               ;4 paragraphs

code                segment
                    assume  cs:code,ds:code
prog                proc    far
start:
                    dw      prgmend-start
```

```
                dw      0006h           ;offset of internal entry point
                dw      0100h           ;version 1.00

                push    cs              ;set DS to CS
                pop     ds

                ; initialize storage
                sub     ax,ax           ;clear AX
                mov     word ptr bufaddr,ax ;and clear buffer address

                ; allocate buffer memory
                mov     bx,BUFSIZ       ;size of buffer to allocate
                mov     al,0            ;not associated with a handler
                call    alloc           ;allocate memory
                call    errchk          ;call error check
                mov     word ptr bufaddr,cx ;store away buffer address
                mov     cl,4            ;turn buffer size from paragraphs to bytes
                shl     dx,cl
                mov     word ptr buflen,dx  ;store away buffer length

                push    cs              ;ES:BX address of "From" message
                pop     es
                mov     bx,offset FROM
                call    puts
                call    gets
                call    errchk
                mov     al,15           ;channel number for input file
                call    fopen           ;open the file
                call    errchk

                push    cs              ;ES:BX address of "To" message
                pop     es
                mov     bx,offset TO
                call    puts
                call    gets
                call    errchk

                mov     ah,SEEK         ;SEEK function code
                mov     al,15           ;infile channel
                mov     bl,02h          ;seek to EOD
                int     1Ah             ;seek to EOD to find file size
                                        ;(could also use FIND_FILE)
                or      cx,cx           ;is the file big (> 64k) (cx > 0)?
                jnz     crbig           ;yes -- use a default big size
                add     dx,15           ;to round up # of paragraphs
                jc      crbig           ;if we had a carry it is big file
                mov     cl,4            ;shift dx 4 bits right
                shr     dx,cl           ;to turn from bytes to paragraphs
                mov     cx,dx           ;initial size allocation to dx
                mov     dx,1            ;size increment
                jmp     crfile          ;and create the file
crbig:
                mov     cx,1000h        ;file is >= 10000h bytes (64K) long
                mov     dx,10h          ;use a large increment
crfile:
                mov     ax,word ptr bufaddr ;segment address of buffer
                mov     es,ax           ;to ES
                sub     bx,bx           ;offset address of buffer
                call    fcreate         ;create the file if it does not exist
                cmp     al,6Ch          ;file already exist?
```

```
                        je      nocrerr         ;ignore the error
                        or      al,al           ;set status for caller
                        call    errchk          ;check for any other create errors
nocrerr:
                        sub     bx,bx           ;offset address of buffer
                                                ;(clobbered by CREATE)
                        mov     al,14           ;channel number for outfile
                        call    fopen           ;open the file
                        jnz     errexit1        ;error -- close infile
                        mov     ah,SEEK         ;SEEK function code
                        mov     al,15           ;channel number for infile
                        mov     bl,01h          ;seek absolute
                        sub     cx,cx           ;to start of file (000000h)
                        sub     dx,dx
                        int     1Ah             ;seek to start of infile
                        mov     al,14           ;channel number for outfile
                        mov     bl,02h          ;seek to EOD
                        int     1Ah             ;seek to EOD to append to outfile
                        push    cs              ;ES:BX address of CRLF
                        pop     es
                        mov     bx,offset CRLF
                        call    puts

loop:
                        mov     ax,word ptr bufaddr ;ES:BX buffer address
                        mov     es,ax
                        sub     bx,bx           ;offset of buffer is 0
                        mov     cx,word ptr buflen  ;buffer size
                        mov     al,15           ;infile channel #
                        call    fread           ;read infile
                        cmp     al,73h          ;short record error?
                        je      norderr         ;not an error
                        cmp     al,75h          ;EOD?
                        je      closefiles      ;yes -- finish up
                        or      al,al           ;any other read error?
                        jnz     errexit         ;yes -- exit
norderr:
                        mov     al,14           ;outfile channel #
                        call    fwrite          ;write outfile (rest already set up)
                        jnz     errexit         ;error? -- exit
                        mov     ah,PUT_CHAR     ;PUT_CHAR function code
                        mov     al,"."
                        int     1Ah             ;display a "."
                        jmp     loop
closefiles:
                        sub     al,al           ;no error (al=0)
errexit:                                        ;close both infile and outfile
                        push    ax              ;save error code
                        mov     al,14           ;outfile channel #
                        call    fclose          ;close the file (ignore error code)
                        pop     ax              ;restore error code
errexit1:                                       ;only close infile
                        push    ax              ;save error code
                        mov     al,15           ;infile channel #
                        call    fclose          ;close the file (ignore error code)
                        pop     ax              ;restore error code
dsperr:
                        or      al,al           ;error?
                        jz      noerr           ;no -- don't display an error message
                        mov     ah,DISPLAY_ERROR    ;DISPLAY_ERROR function code
```

```
                int     1Ah             ;display it
noerr:
                mov     cx,word ptr bufaddr ;get segment address of buffer
                or      cx,cx           ;=0?
                jz      nofree          ;yes -- no buffer to free
                call    free            ;free the buffer
nofree:
                ret                     ;exit program
prog            endp

;               puts -- write a line to the LCD
;
;               call with:
;                       es = segment address of string
;                       bx = offset address of string
;
puts            proc    near
                mov     ah,PUT_LINE     ;PUT_LINE function code
                int     1Ah             ;display it
                ret
puts            endp

;               gets -- Read a line from the keyboard into buffer.
;                       Turn trailing CR into a NUL.
;                       Turn '.' into ':'
;
;               call with:
;                       nothing.
;
gets            proc    near
                mov     ax,word ptr bufaddr ;get segment address of buffer
                mov     es,ax           ;set ES to it
                sub     bx,bx           ;offset address to buffer
                mov     ax,word ptr buflen  ;size in bytes
                or      ah,ah           ;bigger than 256?
                jnz     okbuffer        ;no -- we will use the actual size
                mov     al,255
okbuffer:
                dec     al              ;leave room for the CR
                mov     ah,GET_LINE     ;GET_LINE function code
                int     1Ah             ;get string

                or      al,al           ;set status for caller
                jnz     getsret         ;error? -- return now
                ; save away registers
                push    ax
                push    cx
                push    dx
                push    di
                sub     dh,dh           ;clear high byte of dh
                add     dx,bx           ;address of last byte
                mov     di,dx           ;length to index register
                mov     byte ptr es:[di],00h;null out CR

                ; change '.' to ':'
                mov     di,bx           ;offset of string into offset register
dotloop:
                mov     ah,es:[di]      ;get the next character
                cmp     ah,"."          ;is it a dot?
                jne     nodot           ;no -- don't change it.
```

```
                        mov     ah,":"          ;replace it with a ":"
                        mov     es:[di],ah
nodot:
                        inc     di
                        or      ah,ah           ;is it a NUL (end of string)
                        jnz     dotloop
                        ; restore registers
                        pop     di
                        pop     dx
                        pop     cx
                        pop     ax
getsret:
                        or      al,al           ;set status for caller
                        ret
gets                    endp

errchk                  proc    near
                        or      al,al           ;return code 0?
                        jnz     err01           ;yes -- display an error
                        ret                     ;no -- just return
err01:
                        add     sp,2            ;pull off the near return address
                        jmp     dsperr          ;display the error & exit program
errchk                  endp

;                       alloc -- allocate a scratch area
;
;                       call with:
;                                 al = channel number for handler, 0 for others
;                                 bx = size of area in paragraphs
;
alloc                   proc    near
                        mov     ah,GET_MEM      ;GET_MEM function code
                        int     1Ah             ;allocate scratch area
                        or      al,al           ;set status for caller
                        ret
alloc                   endp

;                       free -- free a scratch area
;
;                       call with:
;                                 cx = segment address of scratch area
;
free                    proc    near
                        mov     ah,REL_MEM      ;REL_MEM function code
                        int     1Ah             ;release scratch area
                        or      al,al           ;set status for caller
                        ret
free                    endp

;                       fopen -- open a file
;
;                       call with:
;                                 al = channel #
;                                 es = segment address of file name buffer
;                                 bx = offset address of file name buffer
;                                 dx = offset address of parameter area
;                                      (built-in serial port only)
;
fopen                   proc    near
```

```
                        mov     ah,OPEN         ;OPEN function code
                        int     1Ah             ;open the file
                        or      al,al           ;set status for caller
                        ret
fopen                   endp

;                       fclose -- close an open file
;
;                       call with:
;                               al = channel #
;
fclose                  proc    near
                        mov     ah,CLOSE        ;CLOSE function code
                        int     1Ah             ;close the file
                        or      al,al           ;set status for caller
                        ret
fclose                  endp

;                       fcreate -- create a file
;
;                       call with:
;                               es = segment address of file name buffer
;                               bx = offset address of file name buffer
;                               cx = initial size in paragraphs
;                               dx = size increment in paragraphs
;
fcreate                 proc    near
                        mov     ah,CREATE       ;CREATE function code
                        int     1Ah             ;create the file
                        or      al,al           ;set status for caller
                        ret
fcreate                 endp

;                       fread -- read a channel into a buffer
;
;                       call with:
;                               al = channel #
;                               cx = number of bytes to read
;                               es = segment address of read buffer
;                               bx = offset address of read buffer
;
fread                   proc    near
                        mov     ah,READ         ;READ function code
                        int     1Ah             ;read the channel
                        or      al,al           ;set status for caller
                        ret
fread                   endp

;                       fwrite -- write a buffer into a channel
;
;                       call with:
;                               al = channel #
;                               cx = number of bytes to write
;                               es = segment address of write buffer
;                               bx = offset address of write buffer
;
fwrite                  proc    near
                        mov     ah,WRITE        ;WRITE function code
                        int     1Ah             ;write the buffer
                        or      al,al           ;set status for caller
```

```
                        ret
fwrite                  endp

FROM                    db              "From: ",0Fh,00h
TO                      db          0Dh,0Ah,"To:   ",0Fh,00h
CRLF                    db          0Dh,0Ah,00h
bufaddr                 dw          ?                   ;must be in RAM
buflen                  dw          ?                   ;must be in RAM

prgmend:
code                    ends
                        end
```

5

Hardware Control and Status Registers

Contents

Chapter 5 **Hardware Control and Status Registers**

5-2 Main Control and Status Registers
5-3 Interrupt Control and Status Registers
5-5 Copies of Write-Only Control Registers

5

Hardware Control and Status Registers

The HP-94 has control and status registers that allow a program to control the various hardware devices and determine their status. The control and status registers are in the CPU I/O space, so programs interact with them using the IN and OUT instructions. The details of these registers are discussed in the appropriate device chapters. The table below summarizes the I/O addresses for all the control and status registers.

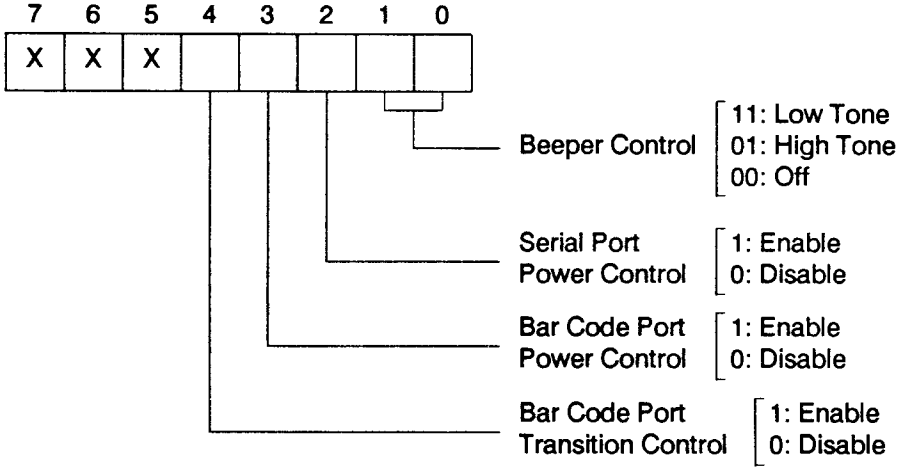
Table 5-1. I/O Addresses for Control and Status Registers

I/O Address	Register Name	Read/ Write
00h	Interrupt Control	W
00h	Interrupt Status	R
01h	Interrupt Clear	W
01h	End of Interrupt	R
02h	System Timer Data	R/W
03h	System Timer Control	W
04h	Bar Code Timer Data (lower 8 bits)	R/W
05h	Bar Code Timer Data (upper 4 bits)	R/W
06h	Bar Code Timer Control	W
07h	Bar Code Timer Value Capture	W
08h	Bar Code Timer Clear	W
0Ah	Baud Rate Clock Value	W
0Bh	Main Control	W
0Bh	Main Status	R
0Ch	Real-Time Clock Control	W
0Ch	Real-Time Clock Status/Data	R
0Eh	Keyboard Control	W
0Eh	Keyboard Status	R
10h	Serial Port Data	R/W
11h	Serial Port Control	W
11h	Serial Port Status	R
12h	Right LCD Driver Control	W
12h	Right LCD Driver Status	R
13h	Right LCD Driver Data	R/W
14h	Left LCD Driver Control	W
14h	Left LCD Driver Status	R
15h	Left LCD Driver Data	R/W
1Bh	Power Control	W

Two primary control registers are particularly important to programs: the main control register (0Bh) and the interrupt control register (00h),

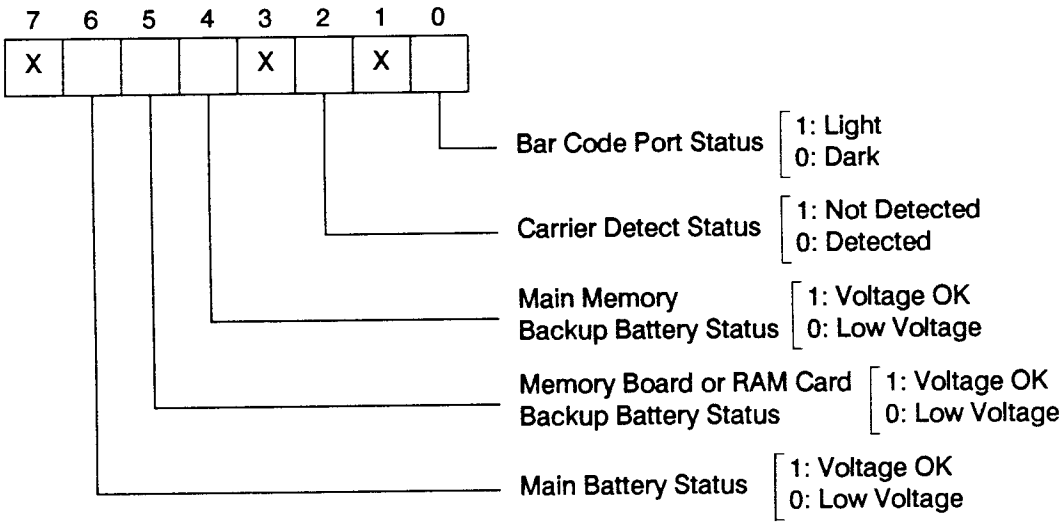
Main Control and Status Registers

The main control and status registers are at I/O address 0Bh. The uses of these registers to control specific hardware devices and determine their status are discussed in the appropriate device chapters. All the uses of these registers are summarized below.



X = don't care

Figure 5-1. Main Control Register (I/O Address 0Bh, Write)



X = ignore

Figure 5-2. Main Status Register (I/O Address 0Bh, Read)

Interrupt Control and Status Registers

The interrupt control and status registers are at I/O address 00h. The uses of these registers to enable specific hardware interrupts and determine which interrupts occurred are discussed in the "Interrupt Controller" chapter. All the uses of these registers are summarized below.

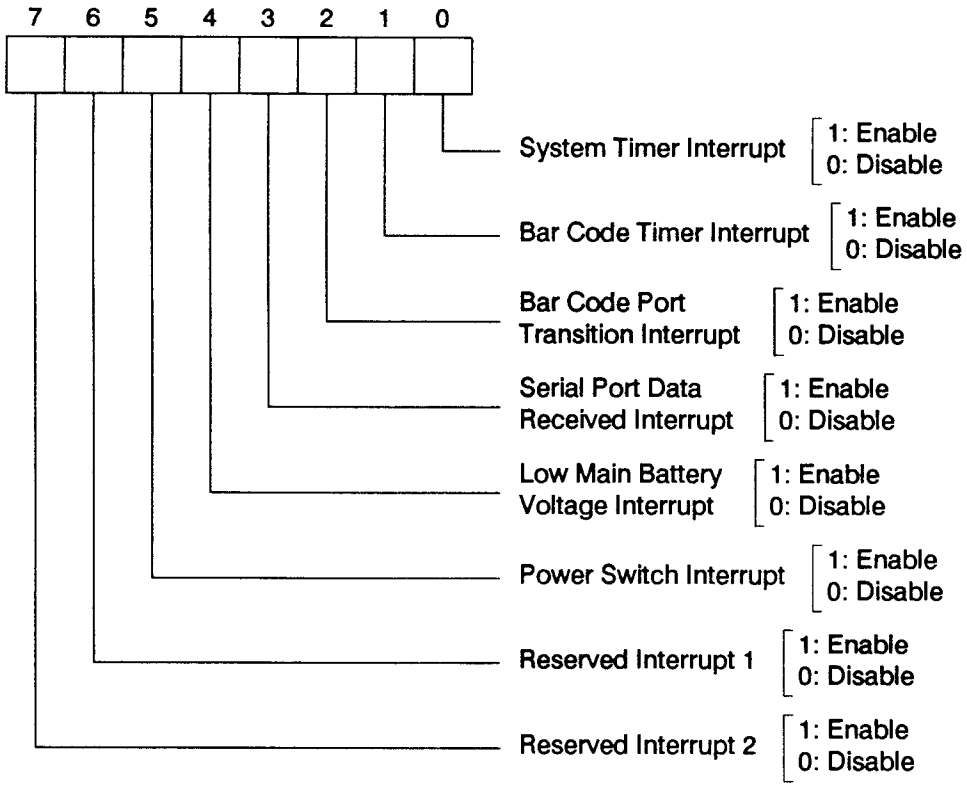


Figure 5-3. Interrupt Control Register (I/O Address 00h, Write)

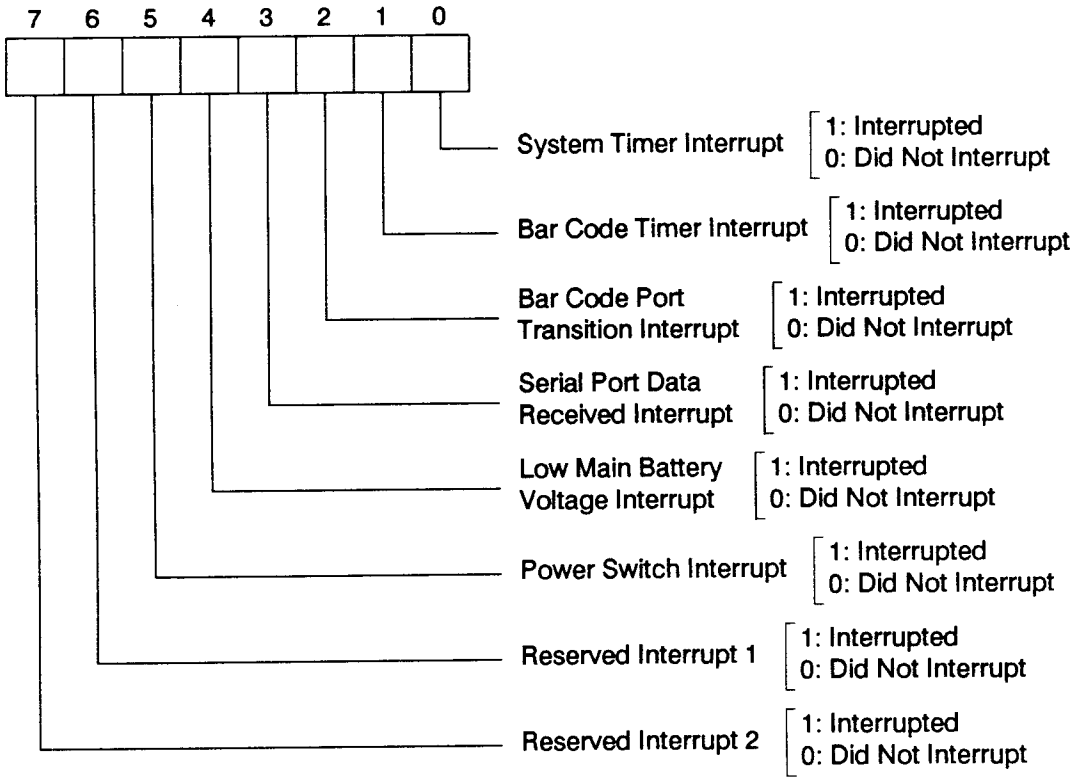


Figure 5-4. Interrupt Status Register (I/O Address 00h, Read)

Copies of Write-Only Control Registers

Control of the HP-94 I/O devices and interrupts is accomplished by using two primary control registers: the main control register and the interrupt control register. These are both write-only as far as controlling the devices and interrupts is concerned, and reading them back yields different results. Reading the main control register obtains other hardware status, and reading the interrupt control register indicates which interrupt occurred.

To allow the operating system and assembly language programs to know what status was set using these two registers, the operating system writes a copy of the register values to two locations in the operating system scratch space. When hardware or interrupt status is changed, the operating system uses the following procedure to ensure that hardware devices or interrupts unaffected by the change remain in their current state:

- Read the copy of the register being changed.
- Change the bits needed to cause the status to change.
- Write the updated value back to its original location.
- Output the updated value to the control register.

When a program uses the operating system functions and utility routines, these copies will be updated automatically. If a program changes the device or interrupt status independent of the operating system,

it is the program's responsibility to mimic the operating system action. That is, the program must make the change correctly while preserving the state of unaffected devices, and must update the copies of the control registers for use by the operating system and other programs.

The status of these registers at cold and warm start is shown below. Refer to appendix L for information about the utility subroutines for reading and saving copies of the control registers.

Table 5-2. Copies of Primary Control Registers

Control Register Name	I/O Address	Initial Value	Meaning of Initial Value	Utility Subroutines
Main Control	0Bh	00h	Beeper off, serial port power off, bar code port power off, bar code port transitions disabled	READCTRL.ASM SETCTRL.ASM
Interrupt Control	00h	31h	System timer, low battery, and power switch interrupts enabled	READINTR.ASM SETINTR.ASM

6

CPU

6

CPU

The HP-94 CPU is the NEC μPD70108 (V20) microprocessor. This is a CMOS microprocessor that is compatible with the Intel 8088 and provides a standby mode for reduced power consumption. Programs written for the 8088 can be run on the V20 with no modifications.

The V20 provides a superset of the 8088 instruction set. Some 8088 instructions have been enhanced, and new instructions have been added. All the changes are described in the CPU data sheet in the "Hardware Specifications". The enhancements and additions are only available if NEC assembly language development tools are used. Contact NEC for information on these if using the V20 features is important to your applications.

The HP-94 CPU runs at an operating frequency of 3.6864 MHz, which is 0.27 μs/clock cycle. Note, however, that the V20 instruction timing is different than the 8088 instruction timing. The V20 timing should be used whenever determining the number of clock cycles for specific operations. The instruction timing is shown in the CPU data sheet using NEC mnemonics. These are similar but not identical to 8088 mnemonics, as shown in the next table.

Table 6-1. Intel 8088 and NEC V20 Instruction Mnemonics

Intel 8088	NEC V20	Intel 8088	NEC V20	Intel 8088	NEC V20	Intel 8088	NEC V20
AAA	ADJBA	JA	BH	JZ	BE,BZ	REPE	REPE
AAD	CVTDB	JAE	BNC,BNL	LAHF	MOV	REPZ	REPZ
AAM	CVTBD	JB	BC,BL	LDS	MOV	REPNE	REPNE
AAS	ADJBS	JBE	BNH	LEA	LDEA	REPNZ	REPNZ
ADC	ADDC	JC	BC,BL	LES	MOV	RET	RET
ADD	ADD	JCXZ	BCWZ	LOCK	BUSLOCK	ROL	ROL
AND	AND	JE	BE,BZ	LODS	LDM	ROR	ROR
CALL	CALL	JG	BGT	LODSB	LDM	SAHF	MOV
CBW	CVTBW	JGE	BGE	LODSW	LDM	SAL	SHL
CLC	CLR1	JL	BLT	LOOP	DBNZ	SAR	SHRA
CLD	CLR1	JLE	BLE	LOOPE	DBNZE	SBB	SUBC
CLI	DI	JMP	BR	LOOPNE	DBNZNE	SCAS	CMPM
CMC	NOT1	JNA	BNH	LOOPNZ	DBNZNE	SCASB	CMPM
CMP	CMP	JNAE	BC,BL	LOOPZ	DBNZE	SCASW	CMPM
CMPS	CMPBK	JNB	BNC,BNL	MOV	MOV	SHL	SHL
CMPSB	CMPBK	JNBE	BH	MOVS	MOVBK	SHR	SHR
CMPSW	CMPBK	JNC	BNC,BNL	MOVSB	MOVBK	STC	SET1
CWD	CVTWL	JNE	BNE,BNZ	MOVSW	MOVBK	STD	SET1
DAA	ADJ4A	JNG	BLE	MUL	MULU	STI	EI
DAS	ADJ4S	JNGE	BLT	NEG	NEG	STOS	STM
DEC	DEC	JNL	BGE	NOP	NOP	STOSB	STM
DIV	DIVU	JNLE	BGT	NOT	NOT	STOSW	STM
ESC	FPO1	JNO	BNV	OR	OR	SUB	SUB
HLT	HALT	JNP	BPO	OUT	OUT	TEST	TEST
IDIV	DIV	JNS	BP	POP	POP	WAIT	POLL
IMUL	MUL	JNZ	BNE,BNZ	POPF	POP	XCHG	XCH
IN	IN	JO	BV	PUSH	PUSH	XLAT	TRANS
INC	INC	JP	BPE	PUSHF	PUSH	XOR	XOR
INT	BRK	JPE	BPE	RCL	ROLC		
INTO	BRKV	JPO	BPO	RCR	RORC		
IRET	RETI	JS	BN	REP	REP		

7

Interrupt Controller

Contents

Chapter 7 **Interrupt Controller**

7-1 Procedure for Using a Hardware Interrupt
7-3 Interrupt Control and Status Registers
7-5 When the Operating System Disables Interrupts
7-6 Operating System Functions

7

Interrupt Controller

The HP-94 interrupt controller receives interrupt requests from eight different HP-94 hardware devices. It prioritizes these interrupts, and informs the CPU of the highest priority interrupt. The CPU then locates the interrupt vector for that interrupt and transfers control to the interrupt service routine. The hardware interrupts and their priority are shown below:

Table 7-1. HP-94 Hardware Interrupts

Interrupt Type	Interrupt Name
50h	System Timer
51h	Bar Code Timer
52h	Bar Code Port Transition
53h	Serial Port Data Received
54h	Low Main Battery Voltage
55h	Power Switch
56h	Reserved Interrupt 1
57h	Reserved Interrupt 2

Highest

↓

Interrupt Priority

↓

Lowest

Information about the behavior of interrupt service routines for the different hardware devices are in the appropriate device chapters.

At both cold and warm start, the system timer, serial port data received, low main battery voltage, and power switch interrupt vectors all point to their operating system interrupt service routines. They are all enabled except for the serial port data received interrupt. The other hardware interrupt vectors point to a dummy interrupt service routine which clears the interrupt, reads the end of interrupt register, and returns (with an IRET). Reserved interrupts 1 and 2 are for future use.

Procedure for Using a Hardware Interrupt

There are four control registers available for controlling interrupt behavior:

- Interrupt Control Register
  This is used to enable or disable any of the hardware interrupts.
- Interrupt Status Register
  This indicates which hardware devices have issued interrupt requests. The interrupt status register will indicate that an interrupt request occurred *even if the interrupt was disabled.* This is useful for polling device status.

- Interrupt Clear Register
  Once a hardware interrupt has occurred, another interrupt of the same type will not be processed by the interrupt controller until that interrupt has been cleared.
- End of Interrupt Register
  This is read at the end of an interrupt service routine to allow the interrupt controller to generate new interrupts of any type.

There are several things that must be done to use a hardware interrupt. Some must be done when the interrupt is initialized, and others during an interrupt service routine. These are summarized below:

Table 7-2. Using Hardware Interrupts

Action	Control or Status Register Used	Required During Initialization	Required In Service Routine
Disable Interrupt	Interrupt Control	No *	No
Take Over Interrupt Vector	—	Yes	No
Enable Interrupt	Interrupt Control	Yes	No
Set CPU Interrupt Flag (STI)	—	Yes	No †
Verify Interrupt Source	Interrupt Status	No	No *
Clear Interrupt	Interrupt Clear	No *	Yes
Read End of Interrupt Register	End of Interrupt	No *	Yes
Return from Interrupt (IRET)	—	No	Yes

* Not required, but can be done as defensive programming. For example, it is unlikely when enabling an interrupt that a previous interrupt request of the same type is present, requiring that the interrupt be cleared before it can occur again. The same reasoning can be applied to the other items that reference this footnote.

† Set automatically by IRET.

When taking over an interrupt, the interrupt vector location is the two words starting at address *T* * 4, where *T* is the interrupt type. This is at addresses 00140h-0015Ch for the hardware interrupts. The instruction pointer (IP) offset of the interrupt service routine should be stored at the first word, and the code segment (CS) address of the routine should be stored at the second word.

The existing interrupt vector should be saved when the interrupt is taken over, then restored when the program gives up the interrupt.

If the interrupt service routine is in a user-defined handler, the program should save the segment address of the handler scratch area in the handler information table. See the "User-Defined Handlers" chapter for details.

Software interrupt 1Ah for calling operating system functions is discussed in the "Operating System Functions" chapter, and software interrupt 1Ch for the background timer is discussed in the "Timers" chapter.

Interrupt Control and Status Registers

The interrupt control and status registers are shown below. A copy of the main interrupt control register is maintained in the operating system scratch space for reference. Refer to the "Hardware Control and Status Registers" chapter for further information.

Table 7-3. Interrupt Control and Status Registers

Register Name	I/O Address	Bits Used	Read/ Write
Interrupt Control	00h	0-7	W
Interrupt Status	00h	0-7	R
Interrupt Clear	01h	0-7	W
End of Interrupt	01h	None	R

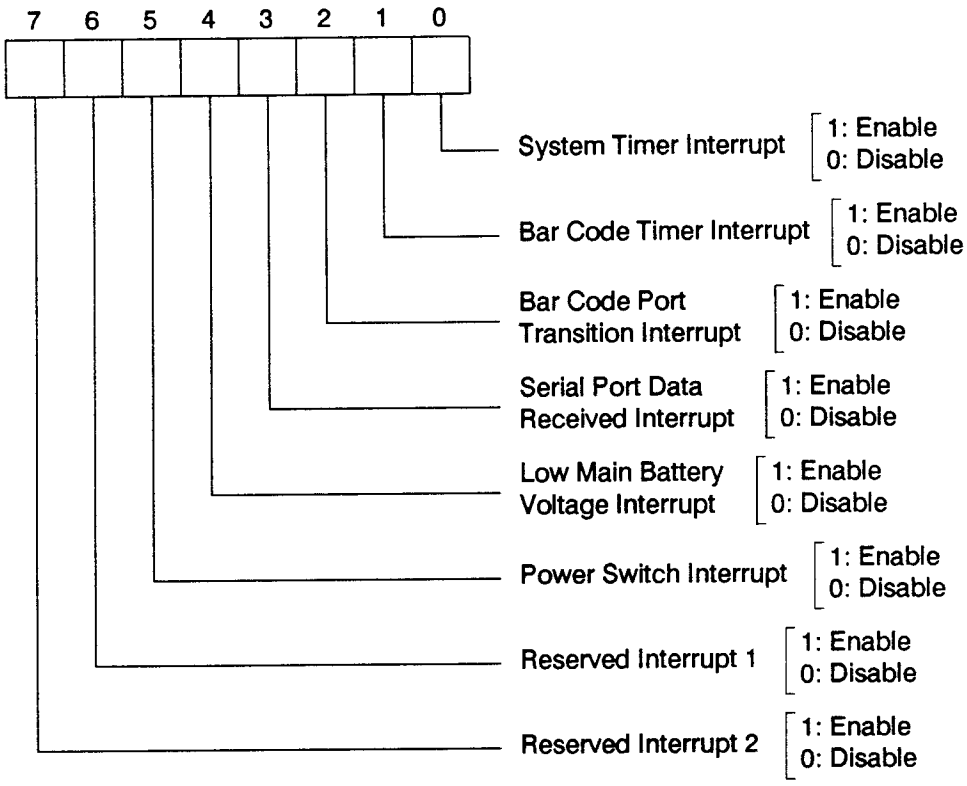


Figure 7-1. Interrupt Control Register (I/O Address 00h, Write)

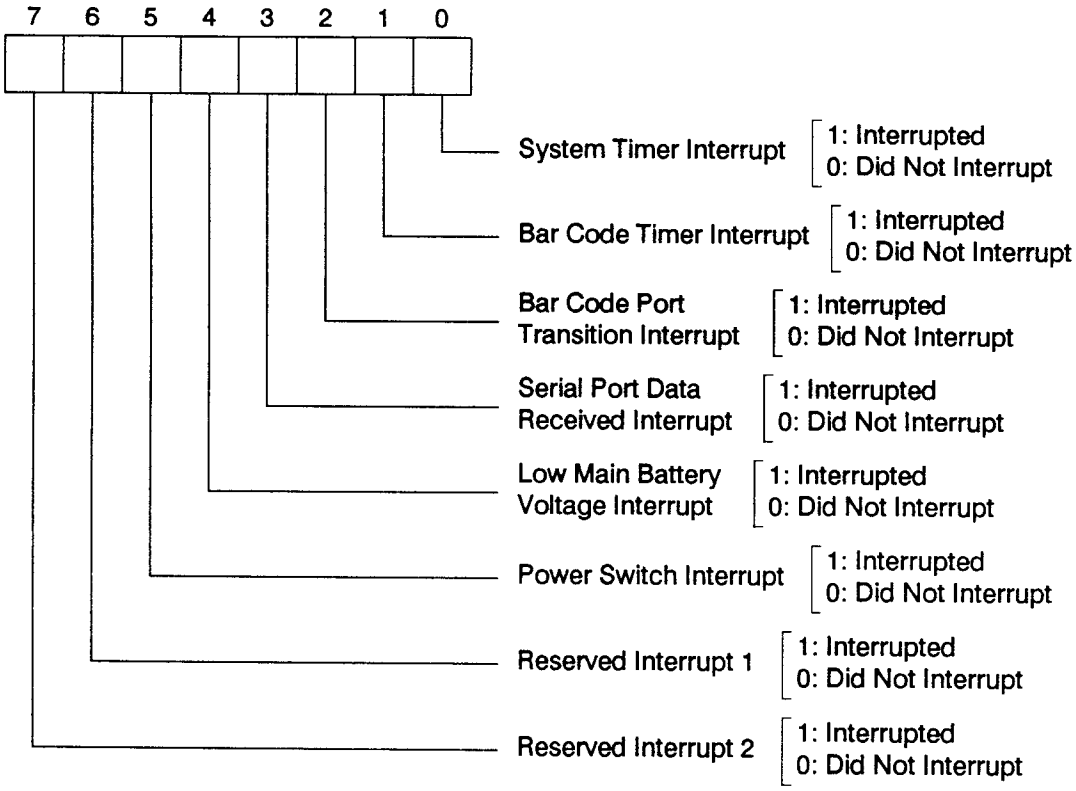


Figure 7-2. Interrupt Status Register (I/O Address 00h, Read)

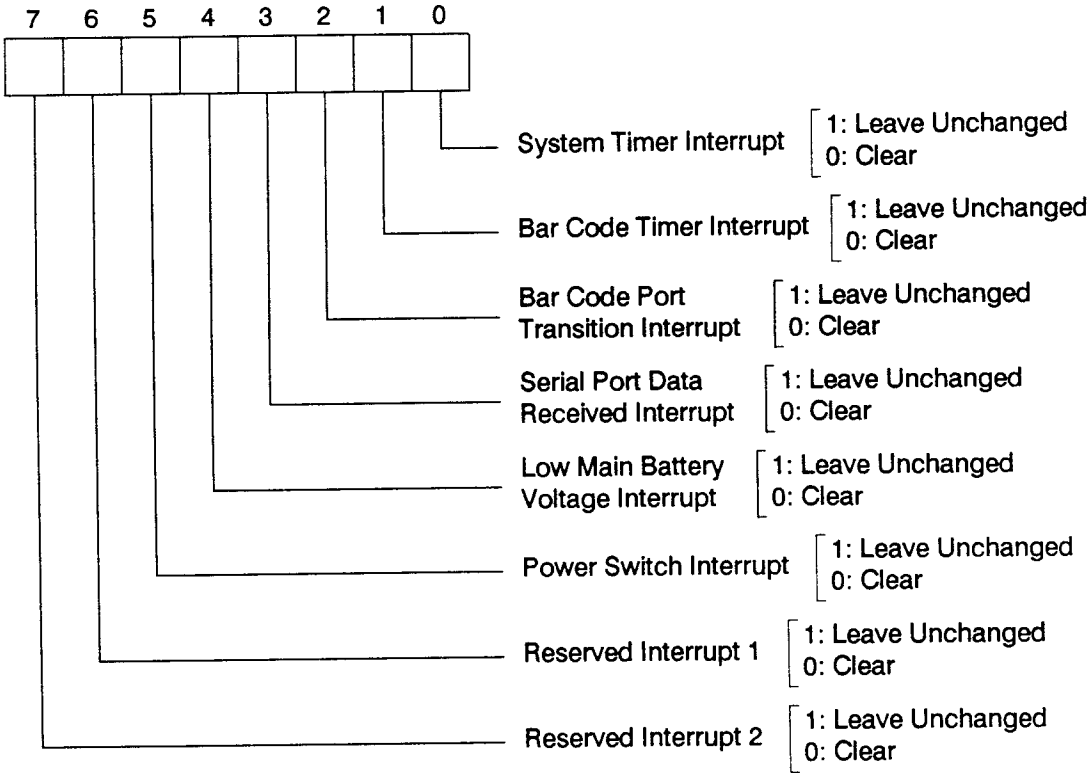


Figure 7-3. Interrupt Clear Register (I/O Address 01h, Write)

7	6	5	4	3	2	1	0
X	X	X	X	X	X	X	X

X = ignore

Figure 7-4. End of Interrupt Register (I/O Address 01h, Read)

When the Operating System Disables Interrupts

The operating system disables interrupts by clearing the CPU interrupt flag (CLI) at two times that may be important to time-critical interrupt service routines:

- While processing the display control character that homes the cursor and clears the screen (0Ch), interrupts are disabled for ~45 ms. This may be important for serial and bar code port handlers.
- While checking to see if the beeper needs to be turned off, interrupts are disabled for ~50 μs. This may be important for bar code port handlers.

Operating System Functions

The interrupt software implements the following operating system functions:

Table 7-4. Interrupt-Related Operating System Functions

Function Name	Function Code
TIMEOUT	09h
SET_INTR	0Ah

8

Keyboard

Contents

Chapter 8 Keyboard

8-1 Keyboard Shift Status
8-2 Display Backlight Control
8-2 Key Buffer
8-2 Waiting for a Key
8-3 Keyboard Scanning
8-5 Keyboard Scanning at Turn On
8-5 Keyboard Control and Status Registers
8-6 Operating System Functions

8

Keyboard

The HP-94 keyboard has 34 keys, arranged as shown below.

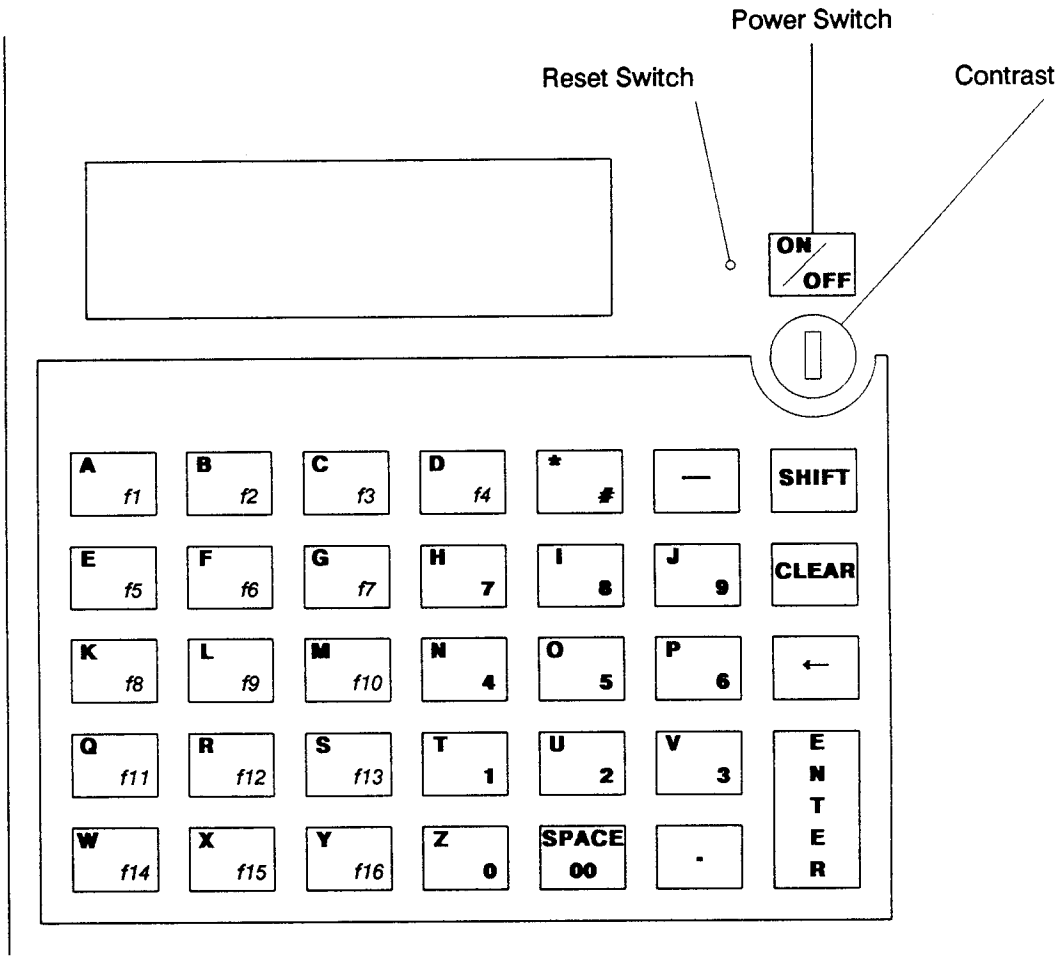


Figure 8-1. HP-94 Keyboard

Keyboard Shift Status

The symbols on the upper left corner of each key are in orange and can be entered when the keyboard is shifted. The symbols on the lower right corner of each key are in white and can be entered when the keyboard is unshifted. Keys with only one centered symbol are in white and can be entered whether the keyboard is shifted or not.

The keys labelled *f1* through *f16* are the user-defined keys, and have no predefined action associated with them. When the keyboard is unshifted, they return ASCII 80h-8Fh which corresponds to the first 16 user-defined characters (see the "Display" chapter for details).

The [SHIFT] key toggles between unshifted and shifted keys. The keyboard shift status is indicated by the shape of the cursor. An underscore cursor indicates unshifted (white keys), and a block cursor indicates shifted (orange keys).

Display Backlight Control

The [SHIFT] key controls the display backlight. If the [SHIFT] key is held down for one second, the display backlight will be turned on (or off if it was already on). When the backlight is toggled by holding down [SHIFT] for one second, the keyboard status and cursor type will be unchanged.

The backlight will turn off automatically after two minutes (120 seconds). This timeout can be set under program control between 0 (never turn off) and 1800 seconds. The display backlight can be turned on or off from a program by writing the appropriate display control character to the display: 1Eh turns on the backlight, and 1Fh turns off the backlight. The keyboard control register has a bit to turn on and off the backlight.

CAUTION Leaving the display backlight on continuously or for long periods of time (greater than 5 minutes) will reduce the life of the backlight.

Key Buffer

There is an eight-character key buffer where the ASCII equivalents of each scanned key (not the keycodes) are placed. A short, low tone beep will be issued when a key is placed in the key buffer (note that this beep cannot be disabled). A long, high tone beep will be issued when a key is pressed after the buffer is full — the key will be discarded. When the [SHIFT] key is pressed, it is processed for changing keyboard shift status and the backlight control, but is not placed in the key buffer.

Waiting for a Key

While waiting for a key to be pressed, the keyboard software puts the CPU into its standby mode to save power, and monitors the system timeout. The timeout is restarted every time a key is pressed. When the timeout expires, the default behavior is to turn the machine off. If a program has defined a power switch/timeout interrupt routine using the SET_INTR function (0Ah), that routine will be executed with a FAR CALL when the timeout expires. This will only occur in a running program, not in command mode.

Keyboard Scanning

The keyboard is scanned by the operating system software every 5 ms. Keys are debounced for 25 ms. When a key has been held down for 675 ms, it begins to repeat every 115 ms until it is released.

The keyboard control register has a bit for each column to be scanned. The keyboard is scanned by clearing the bit corresponding to the column to be scanned, and reading the keyboard status register to see which row(s) have a key down. If a key is down, the bit corresponding to that row will be set. The correspondence between the keyboard and the bits in the keyboard control and status registers are shown below.

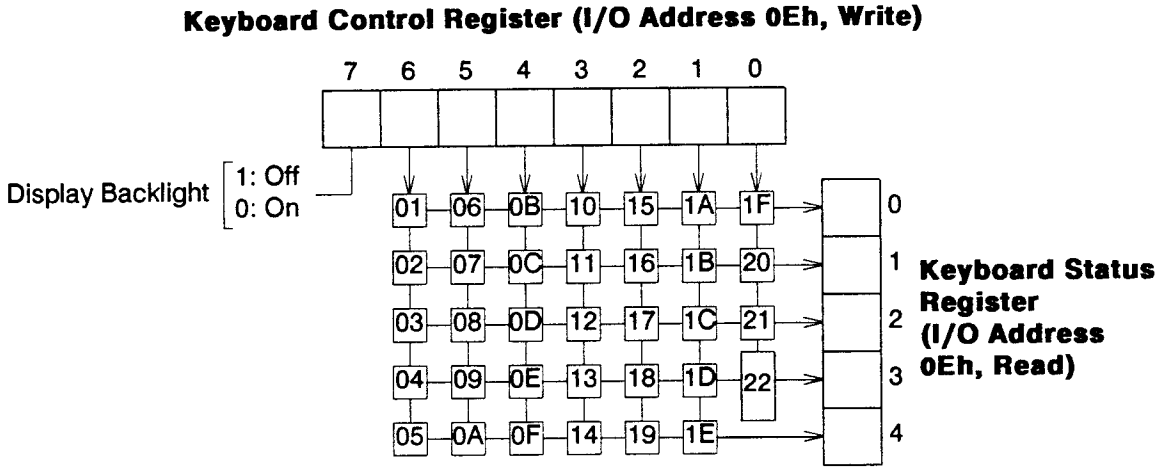


Figure 8-2. HP-94 Keycodes

If multiple columns are selected for scanning, the program will not be able to distinguish which key was pressed. It will only be able to identify that a key in a particular row was held down.

The operating system scans the keyboard columns from right to left, and checks the rows from top to bottom. The first key found down in that scanning sequence will be reported as a keycode (shown above in hex). Other keys to the left or below the first key found will be ignored (the [CLEAR] and [ENTER] sequence to enter command mode is scanned as a special case). The keycode will be translated into an ASCII character according to the keyboard shift status and the following keyboard map.

Table 8-1. ASCII Characters and Keycodes for Each Key

Shifted Key (orange)	Shifted Character	Unshifted Key (white)	Unshifted Character	Keycode
A	A (41h)	(unmarked)	user-defined (80h)	01h
B	B (42h)	(unmarked)	user-defined (81h)	06h
C	C (43h)	(unmarked)	user-defined (82h)	0Bh
D	D (44h)	(unmarked)	user-defined (83h)	10h
E	E (45h)	(unmarked)	user-defined (84h)	02h
F	F (46h)	(unmarked)	user-defined (85h)	07h
G	G (47h)	(unmarked)	user-defined (86h)	0Ch
H	H (48h)	7	7 (37h)	11h
I	I (49h)	8	8 (38h)	16h
J	J (4Ah)	9	9 (39h)	1Bh
K	K (4Bh)	(unmarked)	user-defined (87h)	03h
L	L (4Ch)	(unmarked)	user-defined (88h)	08h
M	M (4Dh)	(unmarked)	user-defined (89h)	0Dh
N	N (4Eh)	4	4 (34h)	12h
O	O (4Fh)	5	5 (35h)	17h
P	P (50h)	6	6 (36h)	1Ch
Q	Q (51h)	(unmarked)	user-defined (8Ah)	04h
R	R (52h)	(unmarked)	user-defined (8Bh)	09h
S	S (53h)	(unmarked)	user-defined (8Ch)	0Eh
T	T (54h)	1	1 (31h)	13h
U	U (55h)	2	2 (32h)	18h
V	V (56h)	3	3 (33h)	1Dh
W	W (57h)	(unmarked)	user-defined (8Dh)	05h
X	X (58h)	(unmarked)	user-defined (8Eh)	0Ah
Y	Y (59h)	(unmarked)	user-defined (8Fh)	0Fh
Z	Z (5Ah)	0	0 (30h)	14h
*	* (2Ah)	#	# (23h)	15h
SPACE	(space) (20h)	00	00 (30h 30h)	19h
—	— (2Dh)	—	— (2Dh)	1Ah
.	. (2Eh)	.	. (2Eh)	1Eh
SHIFT	(none)	SHIFT	(none)	1Fh
CLEAR	(CAN) (18h)	CLEAR	(CAN) (18h)	20h
←	(DEL) (7Fh)	←	(DEL) (7Fh)	21h
ENTER	(CR) (0Dh)	ENTER	(CR) (0Dh)	22h

Refer to the appendixes for a utility routine that scans the keyboard and returns the keycode of the first key found down.

Keyboard Scanning at Turn On

When the machine turns on, the operating system checks the keyboard after performing the first three memory integrity checks (system ROM checksum, reserved scratch space read/write, and valid RAM configuration). If any keys are down other than [CLEAR] and [ENTER], the machine will turn back off immediately. This is to prevent accidental turn on (while in a full briefcase, for example).

Keyboard Control and Status Registers

The keyboard control and status registers are summarized below.

Table 8-2. Keyboard Control and Status Registers

Register Name	I/O Address	Bits Used	Read/ Write
Keyboard Control	0Eh	0-7	W
Keyboard Status	0Eh	0-4	R

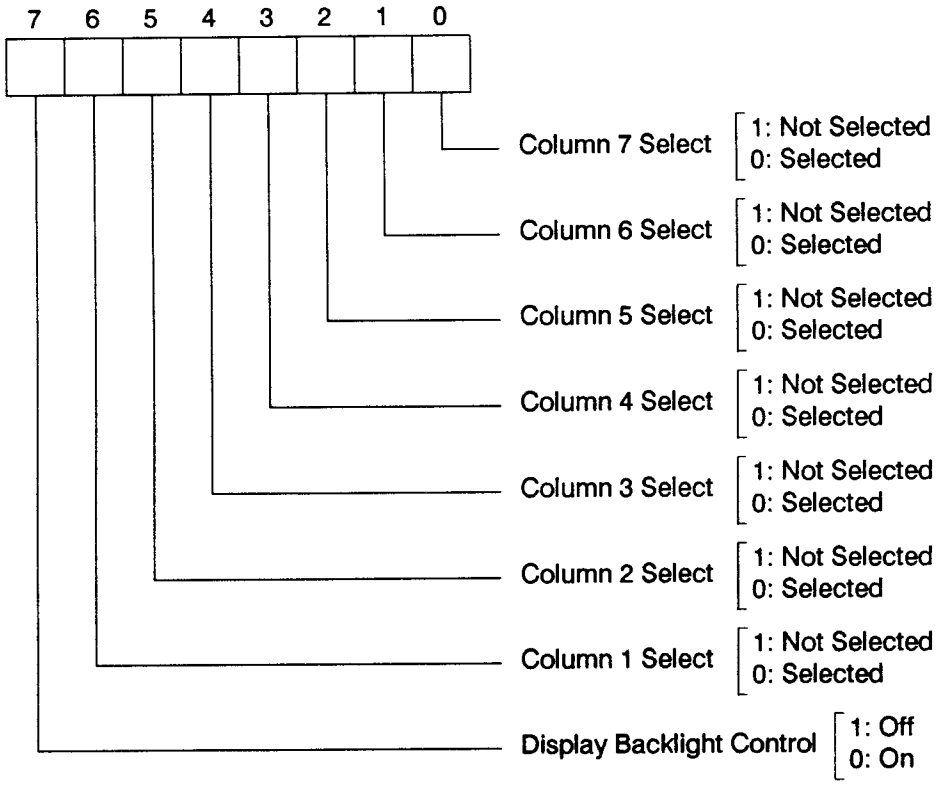
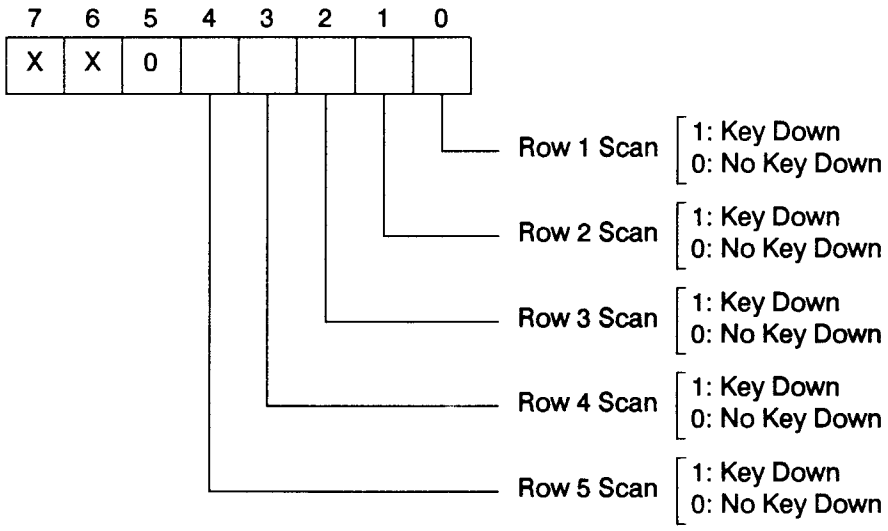


Figure 8-3. Keyboard Control Register (I/O Address 0Eh, Write)



X = ignore

Figure 8-4. Keyboard Status Register (I/O Address 0Eh, Read)

Operating System Functions

The keyboard software implements the following operating system functions:

Table 8-3. Keyboard-Related Operating System Functions

Function Name	Function Code
GET_CHAR	01h
GET_LINE	02h
PUT_CHAR	03h
PUT_LINE	04h
BUFFER_STATUS	06h
READ	12h

9

Display

Contents

Chapter 9 Display

9-1 Display Backlight Control
9-2 LCD Controllers
9-2 Writing Dots to the Display
9-2 Display Control and Status Registers
9-3 Writing Characters to the Display
9-4 Operating System Functions
9-5 User-Defined Characters
9-5 Structure of SYFT Font Definition File
9-6 Relationship to User-Defined Keys

9

Display

The HP-94 has a liquid crystal display (LCD) with an electroluminescent backlight. The display is a continuous dot-matrix of 120 columns and 32 rows, yielding 4 lines of 20 characters each, where each character is in a 6 × 8 character cell. The built-in Roman-8 character set places characters in a 5 × 8 cell, leaving the right column of the 6 × 8 cell blank. It uses the eighth dot for descenders only. The orientation of a character cell is shown below. The filled-in boxes are the dot positions used by the built-in character set.

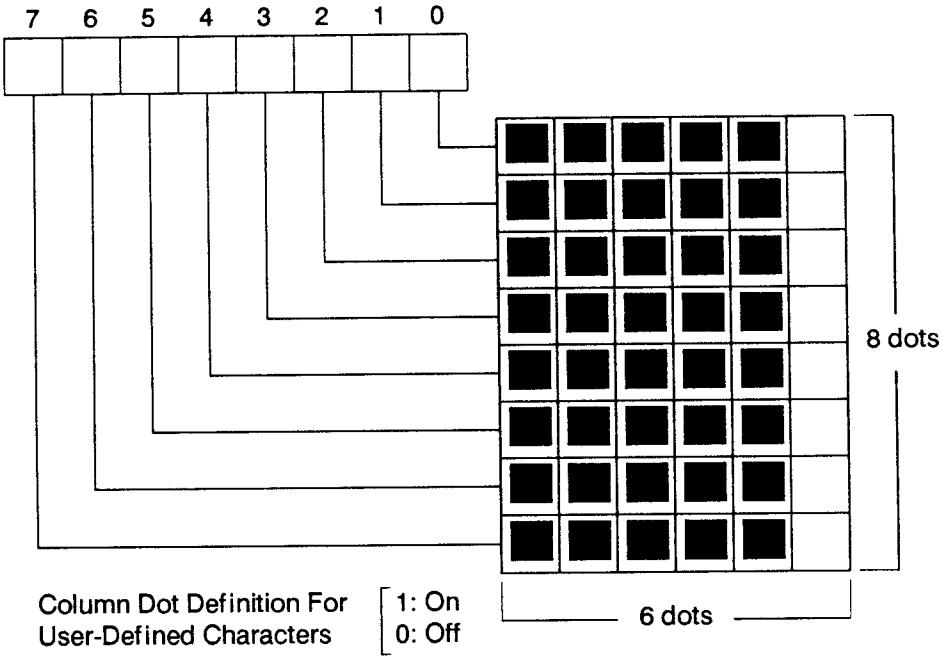


Figure 9-1. 6 × 8 Character Cell

All characters are mapped upside-down. The upper dot of a column of a character is bit 0 of the byte containing the bit pattern for that column. There are 6 bytes per character, one per column from left to right.

Display Backlight Control

The [SHIFT] key controls the display backlight. If the [SHIFT] key is held down for one second, the display backlight will be turned on (or off if it was already on). When the backlight is toggled by holding down [SHIFT] for one second, the keyboard status and cursor type will be unchanged.

The backlight will turn off automatically after two minutes (120 seconds). This timeout can be set under program control between 0 (never turn off) and 1800 seconds. The display backlight can be turned on or off from a program by writing the appropriate display control character to the display: 1Eh turns on the backlight, and 1Fh turns off the backlight. The keyboard control register has a bit to turn on and off the backlight.

CAUTION Leaving the display backlight on continuously or for long periods of time (greater than 5 minutes) will reduce the life of the backlight.

LCD Controllers

There are three LCD controllers. The row driver is a Hitachi HD61103A. It is not accessible to software — the rows are driven automatically by the hardware.

The column driver is a Hitachi HD61102A. Since the column driver can only support 64 columns, two are used. The left half driver controls columns 0-63 (counting from the left), and the right half driver controls column 64-119. Columns 120-127 are ignored. The details of the column driver hardware, operation, and usage are described in the Hitachi HD61102A data sheet in the "Hardware Specifications".

Writing Dots to the Display

Programs writing directly to the display hardware can write an 8-dot pattern to any column in the LCD. As with characters, the dots in the column being written are represented upside-down in the byte containing that dot pattern. A program cannot write individual dots to the display — the display control registers only allow writing columns of data. (Since a program can read individual columns of data, it could read a column, change a dot, and write the column back. This would have the effect of writing an individual dot.)

Display Control and Status Registers

The display control and status registers are shown below.

Table 9-1. Display Control and Status Registers

Register Name	I/O Address	Bits Used	Read/ Write
Keyboard Control	0Eh	7	W
Right LCD Driver Control	12h	0-7 *	W
Right LCD Driver Status	12h	0-7 *	R
Right LCD Driver Data	13h	0-7	R/W
Left LCD Driver Control	14h	0-7 *	W
Left LCD Driver Status	14h	0-7 *	R
Left LCD Driver Data	15h	0-7	R/W

* For the meaning of the bits in these registers, refer to the Hitachi HD61102A data sheet in the "Hardware Specifications".

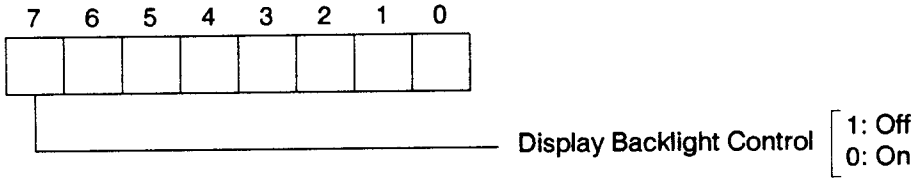


Figure 9-2. Keyboard Control Register (I/O Address 0Eh, Write)

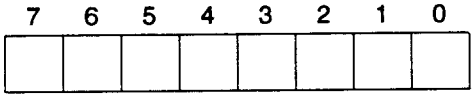


Figure 9-3. Right LCD Driver Data Register (I/O Address 13h, Read/Write)

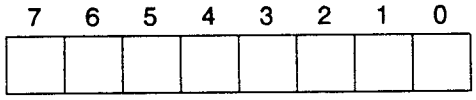


Figure 9-4. Left LCD Driver Data Register (I/O Address 15h, Read/Write)

Writing Characters to the Display

The display software performs the generation of characters from the built-in Roman-8 character set. The first half of the character set (characters 00h-7Fh) consists of standard U.S. ASCII characters. The second half (80h-FFh) contains special characters, including those used by other languages. The display software also displays user-defined characters in the range 80h-9Fh. These will be discussed shortly.

Cursor shape, status, movement, and blinking is also controlled by the display software. The cursor shape is a block to represent shifted keyboard status and an underline to represent unshifted status. The cursor can be either on or off. When on, it is blinked every 500 ms (0.5 s).

Power switch and low battery interrupts can occur while writing data to the display using operating system functions. The system timeout does not occur when writing to the display (channel 0).

The display software processes display control codes for the following actions:

Table 9-2. Display Control Characters

Hex Value	Meaning
01h (SOH)	Turn on cursor.
02h (STX)	Turn off cursor.
06h (ACK)	High tone beep for 0.5 second.
07h (BEL)	Low tone beep for 0.5 second.
08h (BS)	Move cursor left one column. When the cursor reaches the left end of the line, it will back up to the right end of the previous line. When the cursor reaches the top left corner, backspace will have no effect.
0Ah (LF)	Move cursor down one line. If the cursor is on the bottom line, the display contents will scroll up one line.
0Bh (VT)	Clear every character from the cursor position to the end of the current line. The cursor position will be unchanged.
0Ch (FF)	Move cursor to upper left corner and clear the display.
0Dh (CR)	Move cursor to left end of current line.
0Eh (SO)	Change keyboard to numeric mode (underline cursor).
0Fh (SI)	Change keyboard to alpha mode (block cursor).
1Eh (RS)	Turn on display backlight.
1Fh (US)	Turn off display backlight.

Control codes not listed in this table are ignored — that is, no character is displayed for those codes.

NOTE While processing the display control character that homes the cursor and clears the screen (0Ch), interrupts are disabled for ~45 ms. This time may be important to serial and bar code port handlers.

Operating System Functions

The display software implements the following operating system functions:

Table 9-3. Display-Related Operating System Functions

Function Name	Function Code
PUT_CHAR	03h
PUT_LINE	04h
CURSOR	05h
WRITE	13h
DISPLAY_ERROR	18h

User-Defined Characters

The HP-94 allows the font for 32 characters to be redefined: character codes 80h-9Fh, the control codes for the upper 128 characters of the built-in Roman-8 character set. The operating system will use these redefined characters only when a program is running — they will not be used in command mode. When a program is executed (either with the S (start) command or by autostarting), the operating system searches for a type A font definition file named SYFT. If this file is found, and is the correct type, then the dot pattern for characters 80h-9Fh will be taken from it. If SYFT does not exist, characters in that range will be displayed as blanks.

Character mapping will occur whenever characters 80h-9Fh are displayed on the LCD using the PUT_CHAR and PUT_LINE functions, or the WRITE function for channel 0 (functions 03h, 04h, and 13h). PUT_CHAR and PUT_LINE are used by the BASIC I/O keywords PRINT, PRINT USING, PRINT #, PRINT #...USING, and PUT #.

Structure of SYFT Font Definition File

The SYFT font file must contain definitions for 32 characters. If it does not, some characters will be constructed from the contents of the file immediately following SYFT (higher in memory). While this will not have any harmful side effects, it is unlikely to provide useful characters. Unlike type A program files, SYFT does not require a program header.

There are six bytes per character in SYFT, one for each of the six columns of data to be defined in the character's 6 × 8 character cell. All six bytes can be used for dot information. The built-in Roman-8 character set leaves the rightmost column of each character blank to provide intercharacter spacing, but that is not required.

All characters are mapped upside-down. The upper dot of a column of a character is bit 0 of the byte containing the byte for that column. This is illustrated in the earlier picture of a 6 × 8 character cell.

To create SYFT, enter the dot patterns (upside-down) into an assembly language source file, then assemble and link the file. Run HXC on the resulting EXE file, specifying file type A and handheld file name SYFT.

Relationship to User-Defined Keys

The HP-94 has 16 keys which have no predefined use: the alphabetic keys whose unshifted keycaps (lower right corner) are unmarked. These are shown as *f1-f16* on the keyboard layout in the "Keyboard" chapter, and correspond to character codes 80h-8Fh, half of the control codes for the upper 128 characters of the built-in Roman-8 character set.

Whether or not these keys cause the corresponding user-defined character to be echoed to the display depends on which operating system function was used to read the keyboard. GET_CHAR and READ for channel 0 (functions 01h and 12h) do not echo user-defined characters, while GET_LINE (02h) does. The only BASIC I/O statements that echo to the display while accepting keyboard input are INPUT and INPUT #, and they both use GET_LINE.

Even when echoing of keyboard input occurs, it will still track the behavior of user-defined characters — that is, echoed as blanks if no SYFT exists or if the machine is in command mode, and echoed as user-defined characters if SYFT exists and a program is running.

10

Serial Port

Contents

Chapter 10 Serial Port

10-1 Signal Levels
10-1 Enabling or Disabling the Serial Port
10-2 Initializing the Serial Port
10-2 Processing the Serial Port Data Received Interrupt
10-2 Serial Port Control and Status Registers
10-5 Built-in Serial Port Handler
10-5 Built-in Serial Port Handler Capabilities
10-7 Parameters at OPEN Time
10-8 Control Line Behavior
10-9 Operating System Functions

10

Serial Port

The HP-94 serial port is a read/write port controlled by an OKI MSM82C51A Universal Asynchronous Receiver Transmitter (UART). This is a CMOS UART compatible with the Intel 8251A. (It is actually a USART, but the 94 does not provide the additional hardware needed for synchronous operation.) The details of the UART hardware, operation, and usage are described in the Oki MSM82C51A data sheet in the "Hardware Specifications" elsewhere in this manual.

Signal Levels

The serial port signal levels are 0 to V_{cc} (~0-5) volts. Not all devices can operate at those levels, and may require the HP 82470A RS-232-C Level Converter. The converter changes the 0 to V_{cc} signal levels into +9 to -9 volts for those devices that require it. Refer to the "Hardware Specifications" for details on the signal levels as well as the connector pinouts for the serial port and the level converter.

Enabling or Disabling the Serial Port

The 82C51 can be enabled or disabled under software control. Power is supplied to the level converter only when it is enabled; it is only at this time that serial port has any power consumption. When the 82C51 is enabled, the 94 provides a baud rate clock at 16 times the desired baud rate. Before a program transmits or receives with the 82C51, the UART must be set in 16× mode. When the 82C51 has received an entire byte of serial data (including the start and stop bits) and checked for errors (parity, framing, and UART overrun), the serial port data received interrupt (type 53h) will be issued.

Initializing the Serial Port

Below are the things that must be done to initialize the serial port in the OPEN routine of a user-defined serial port handler.

- Take over the existing serial port interrupt vector.
- Set the baud rate clock value.
- Turn on power to the serial port, and wait 60 ms to allow the level converter to power up. This turn-on delay may not accommodate the turn-on or reset time required by the RS-232 device connected to the serial port.

  (Note: when turning off the serial port, the CLOSE routine should wait 60 ms after the 82C51 is disabled to allow signals to stabilize.)
- Reset the 82C51, and set it to the desired initial state.

- Enable the serial port interrupt.

Processing the Serial Port Data Received Interrupt

When the data received interrupt occurs, the following actions should be taken by the interrupt service routine. These are in addition to whatever data processing is done in the routine, and to normal interrupt routine overhead such as reading the end of interrupt register.

- Check if an 82C51 error occurred. If so, clear it.
- Read the data from the serial port data register.

NOTE While processing the display control character that homes the cursor and clears the screen (0Ch), interrupts are disabled for ~45 ms. This time may be important to serial port handlers.

Serial Port Control and Status Registers

The serial port control and status registers are summarized below.

Table 10-1. Serial Port Control and Status Registers

Register Name	I/O Address	Bits Used	Read/ Write
Interrupt Control	00h	3	W
Interrupt Status	00h	3	R
Interrupt Clear	01h	3	W
Baud Rate Clock Value	0Ah	0-2	W
Main Control	0Bh	2	W
Main Status	0Bh	2	R
Serial Port Data	10h	0-7	R/W
Serial Port Control	11h	0-7 *	W
Serial Port Status	11h	0-7 *	R

* For the meaning of the bits in these registers, refer to the Oki MSM82C51A data sheet in the "Hardware Specifications".

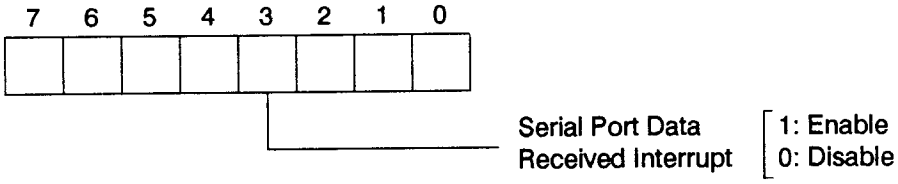


Figure 10-1. Interrupt Control Register (I/O Address 00h, Write)

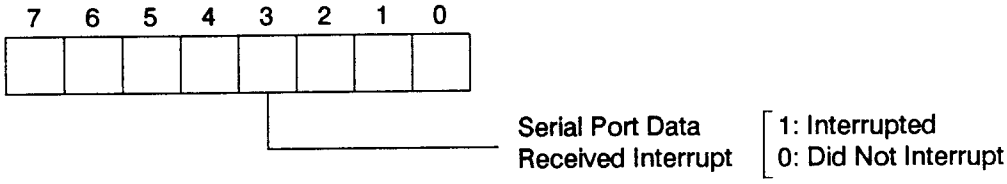


Figure 10-2. Interrupt Status Register (I/O Address 00h, Read)

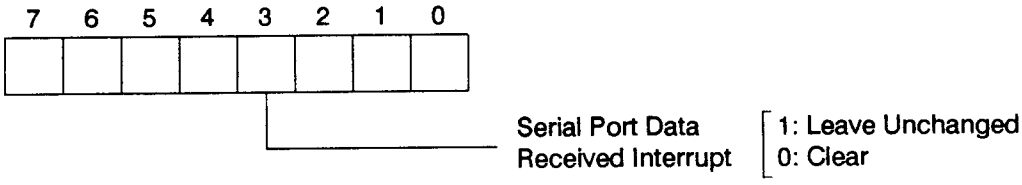
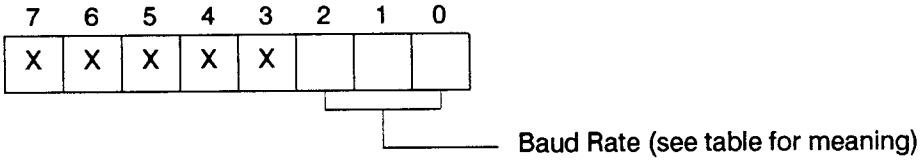


Figure 10-3. Interrupt Clear Register (I/O Address 01h, Write)



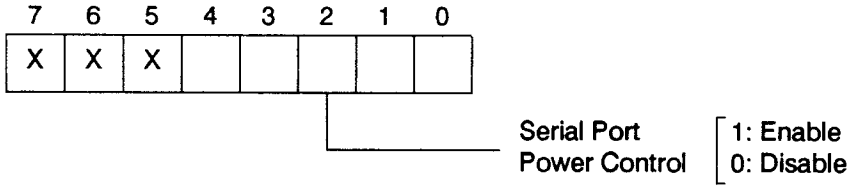
X = don't care

Figure 10-4. Baud Rate Clock Value Register (I/O Address 0Ah, Write)

Table 10-2. Baud Rate Clock Values

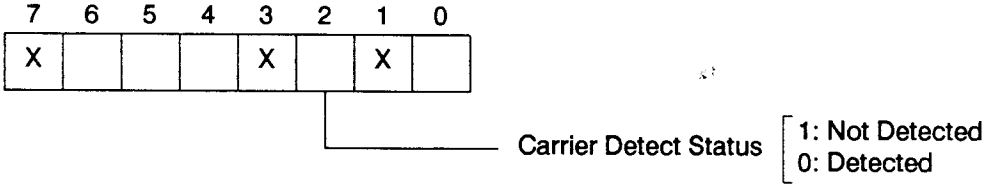
Baud Rate Clock Value	Baud Rate	Frequency (kHz) *
0	19200 †	307.2
1	9600	153.6
2	4800	76.8
3	2400	38.4
4	1200	19.2
5	600	9.6
6	300	4.8
7	150	2.4

* The actual clock frequency is 16 times the desired baud rate.
† Available but not supported.



X = don't care

Figure 10-5. Main Control Register (I/O Address 0Bh, Write)



X = ignore

Figure 10-6. Main Status Register (I/O Address 0Bh, Read)

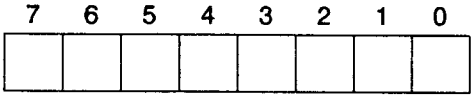


Figure 10-7. Serial Port Data Register (I/O Address 10h, Read/Write)

Built-In Serial Port Handler

The built-in serial port handler is the one used when the serial port is opened and the null string ("") is provided as the handler name. This handler is always used by the C (*copy*) operating system command and by the resident debugger when using the serial port as the console, even when user-defined handlers are available. The handler is designed for use with general serial devices that do not perform hardware handshaking.

Built-In Serial Port Handler Capabilities

The built-in serial port handler provides the following capabilities:

- Full Duplex Communications
  Two-way simultaneous communications.
- Received-Data Buffering
  Received data is placed in a 64-byte buffer. There is no transmit buffer.
- Speeds
  Speeds can be set from 150 to 9600 baud (19200 baud is available but not supported).
- Data Bits
  Seven or eight.
- Parity
  Odd, even, or no parity.
- Stop Bits
  One or two stop bits.
- XON/XOFF Software Handshaking
  When enabled, this option allows received XON (11h) and XOFF (13h) characters to start and stop HP-94 transmissions, and causes XON and XOFF characters to be sent to start and stop host transmissions.
- Null Stripping
  When enabled, this option causes any received NUL characters (00h) to be stripped from and not counted as received data, and not placed in the receive buffer.
- Terminate Character Control
  When defined, a received terminate character will end the wait for a fixed-length block of data, even if all the data has not been received. A terminate character will be sent after sending every block of data.
- Control Lines
  RTS and DTR are raised when the serial port handler is opened, and lowered when the handler is closed. CTS is monitored indirectly by checking if the TxRDY status bit in the 82C51 goes high within three byte-times after attempting to transmit a byte. In addition, V_{rs} (switched V_{cc}) is supplied to power the level converter when the handler is opened, and not supplied when the handler is closed.

The table below describes how the built-in serial port handler behaves. It shows the action taken by the handler routines as well as during its interrupt service routine, not including normal handler activities described in the "User-Defined Handlers" chapter. Note that certain actions, such as sending an XON or responding to a received terminate character, will only occur if the appropriate options were enabled when the handler was opened.

Table 10-3. Behavior of Built-In Serial Port Handler

Routine	Activities
CLOSE	Complete transmission of current byte Disable interrupt 53h Flush receive buffer Lower RTS and DTR Wait 60 ms for signals to stabilize Disable 82C51 and turn off power to serial port
IOCTL	Do nothing
OPEN	Flush receive buffer Enable 82C51 and supply power to level converter Wait 60 ms for level converter turn on Initialize operating configuration * Raise RTS and DTR Enable interrupt type 53h Send single XON Ignore parity, framing, overrun, and receive buffer overflow errors
POWERON	Do nothing
READ	Monitor and report low battery, power switch, and timeout errors † Report errors detected in interrupt service routine Send XON when receive buffer emptied End and report error 74h (116) if terminate character detected Return data from receive buffer
RSVD2	Do nothing
RSVD3	Do nothing
TERM	Do nothing
WARM	Perform all OPEN routine activities except sending XON
WRITE	Monitor and report low battery, power switch, and timeout errors † Monitor CTS indirectly and report error DAh (218) if lost Write data to 82C51 Send terminate character at end of data
Interrupt Service	Monitor parity, framing, overrun, and receive buffer overflow errors Read data from 82C51 and accumulate data into receive buffer Disable transmission when XOFF received Enable transmission when XON received Send XOFF for each byte when buffer 3/4 full Strip nulls (00h)

* Baud rate, data format, XON/XOFF handshaking, null stripping, and terminate character.

† System timeout restarts after each byte received or transmitted.

The errors reported by the built-in serial port handler are shown in the following table.

Table 10-4. Errors Reported by Built-In Serial Port Handler

Routine	Errors
CLOSE	None
IOCTL	None
OPEN	65h
POWERON	None
READ	74h,76h,77h,C8h,C9h,CAh,CBh,CCh,CDh,CEh,CFh,D0h
RSVD2	None
RSVD3	None
TERM	None
WARM	None
WRITE	76h,77h,C8h,DAh
Interrupt Service *	C9h,CAh,CBh,CCh,CDh,CEh,CFh,D0h
† Detected by interrupt service routine, but reported by READ routine.	

Parameters at OPEN Time

When the built-in serial port handler is opened, DS : DX must point to a three-byte parameter area. The meanings of the parameters are shown below. In these figures, the offsets are from DS : DX.

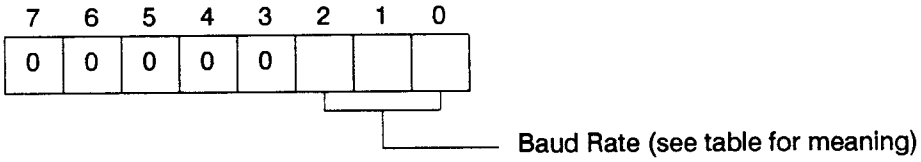
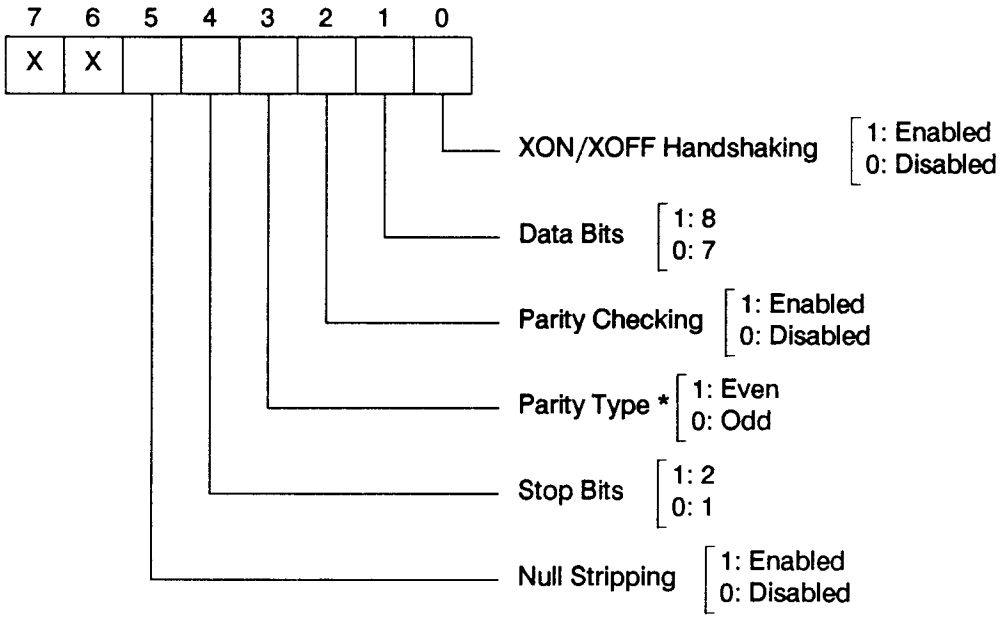


Figure 10-8. Baud Rate — Parameter Byte 1 (Offset 00h)

Table 10-5. Built-In Serial Port Handler Baud Rate Values

Value	Baud Rate
0	19200 *
1	9600
2	4800
3	2400
4	1200
5	600
6	300
7	150
* Available but not supported.	



X = don't care

Figure 10-9. Data Format — Parameter Byte 2 (Offset 01h)

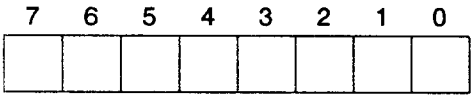


Figure 10-10. Terminate Character † — Parameter Byte 3 (Offset 02h)

The default values for the parameters are 01h (9600 baud), 0Dh (XON/XOFF enabled, 7 data bits, parity checking enabled, even parity, one stop bit, and null stripping disabled), and 00h (no terminate character).

Control Line Behavior

The 82C51 can monitor or control only a subset of the standard RS-232 control lines. Of those lines not monitored, one can be monitored indirectly, and one can be monitored using other HP-94 hardware. Not all these hardware capabilities are actually used by the built-in serial port handler. The usage is summarized below.

---

* The parity type is ignored if parity checking is disabled.

† To disable use of the terminate character, set it to zero.

Table 10-6. Control Line Behavior

Control Line		Monitored or Controlled By Hardware	Monitored or Controlled By Built-In Handler
Symbol	**Name**		
CTS	clear to send	monitored *	monitored *
DSR	data set ready	monitored	not monitored
DCD	data carrier detect	monitored	not monitored
RTS	request to send	controlled	controlled
DTR	data terminal ready	controlled	controlled

* Monitored indirectly by checking if the TxRDY status bit in the 82C51 goes high within three byte-times after attempting to transmit a byte.

A user-defined serial port handler could use all the lines supported by the hardware. Refer to the "User-Defined Handlers" chapter for details on how to write a user-defined serial port handler.

When the serial port is disabled, the control lines are turned off (set to 0 volts).

Operating System Functions

The serial port software implements the following operating system functions:

Table 10-7. Serial Port-Related Operating System Functions

Function Name	Function Code
BUFFER_STATUS	06h
OPEN	0Fh
CLOSE	10h
READ	12h
WRITE	13h

11

Bar Code Port

Contents

Chapter 11 Bar Code Port

11-1 Bar Code Port Power and Transition Detection
11-1 Bar Code Timer
11-1 Initializing the Bar Code Port
11-2 Processing the Bar Code Port Transition Interrupt
11-2 Bar Code Port Timing Constraints
11-3 Bar Code Port Control and Status Registers

11

Bar Code Port

The HP-94 bar code port is a read-only port designed to connect to bar code scanning devices such as wands. The port provides power to the external device. Interrupt control, timing for light and dark transitions, and light or dark state is available to programs reading bar code data.

Bar Code Port Power and Transition Detection

The main control register is used to enable power to the bar code port (and to the device attached to it) and, independently, to enable transition detection at the port. Once the port is powered and detecting transitions, interrupt type 52h will be issued whenever a transition occurs at the port — either light-to-dark or dark-to-light. When the interrupt occurs, the light or dark state is indicated by reading the main status register.

Bar Code Timer

The bar code timer is a 12-bit count-up timer with a 26 μs interval. This resolution allows timing intervals from 26 μs to 106.7 ms. Because it is a count-up timer, it must be set using the complement of the desired number of intervals. When the timer overflows (counts up to zero), interrupt type 51h is generated. This is usually used to indicate the end of a scan.

When the timer reaches zero, it is automatically reset to its starting value and restarted. If the count value has to be set to a specific value, the timer must be stopped first. Unlike the system timer, the bar code timer can be reset to zero while it is still running.

When the bar code port transition interrupt occurs, the timer value can be captured (i.e., placed in the timer data registers where it can be read) to indicate how long the bar code port has been at the current state. Then the timer can be reset to zero to continue counting up for the next transition. The value can be captured while the timer is still running.

Initializing the Bar Code Port

Below are the things that must be done to initialize the bar code port.

- Take over the existing bar code port transition and timer interrupt vectors.
- Turn on power to the bar code port, and enable transition detection.
- Set the bar code timer to the desired initial value (or clear it), and start the timer.
- Enable the bar code port transition and timer interrupts.

Some of the initialization activities will be done in the OPEN routine of a bar code port handler, while

others will be done in the READ routine. This will be discussed shortly.

Processing the Bar Code Port Transition Interrupt

When the transition interrupt occurs, the following actions should be taken by the interrupt service routine. These are in addition to whatever data processing is done in the routine and to normal interrupt routine overhead such as reading the end of interrupt register.

- Capture the current timer value into the timer data registers (04h and 05h) by writing to the timer value capture register (07h).
- Read the captured timer data from the timer data registers.
- Reset the timer to the desired value. If it is a specific value, stop the timer with the timer control register (06h), set the values, and restart it. If it is only necessary to clear the timer, do so by writing to the timer clear register (08h).
- Determine if the state after the transition is light or dark by reading the main status register (0Bh).

Bar Code Port Timing Constraints

The bar code port transition interrupt occurs on every transition. This requires an order of magnitude more processing time than the serial port, since its interrupt occurs only after the 82C51 has received 10-12 transitions (bits) of serial data. Experience has shown that it is unlikely that a bar code port handler can be run "in the background" to simply fill a receive buffer. When other interrupts occur, the CPU interrupt flag will be cleared while the corresponding interrupt service routine executes. This results in periods of time when bar code port transition interrupts occur but cannot be processed, and therefore may be missed.

To deal effectively with these timing constraints, a bar code port handler should only process bar code data during its READ routine. The transition and timer interrupts should only be enabled then, and certain other interrupts should be disabled to prevent transitions from being missed. The machine should essentially become dedicated to the sole task of reading bar code transitions for the duration of the READ operation. This is in contrast to a serial port handler, which can run "in the background", save data in its receive buffer when interrupts occur, and return the data in the buffer when its READ routine is called.

The particular interrupts that should be disabled are the system timer (50h) and serial port data received (53h). The latter has the side effect that data cannot be received by the serial port while bar code labels are being scanned. The former has the side effect that the events paced by the system timer will not occur for the period of time that the timer interrupt is disabled. Refer to the "Timers" chapter for details. There are utility routines available to perform some of these tasks (scan keyboard and blink cursor) without clearing the CPU interrupt flag. Refer to the appendixes for details.

The low main battery voltage (54h) and power switch (55h) interrupts should remain enabled, since those events need to be monitored by the handler to determine if it should abort a read operation.

NOTE While processing the display control character that homes the cursor and clears the screen (0Ch), interrupts are disabled for ~45 ms. While checking to see if the beeper needs to be turned off, interrupts are disabled for ~50 μs. These times may be important to bar code port handlers.

Bar Code Port Control and Status Registers

The bar code port control and status registers are shown below.

Table 11-1. Bar Code Port Control and Status Registers

Register Name	I/O Address	Bits Used	Read/ Write
Interrupt Control	00h	1-2	W
Interrupt Status	00h	1-2	R
Interrupt Clear	01h	1-2	W
Bar Code Timer Data	04h	0-7	R/W
Bar Code Timer Data	05h	0-3	R/W
Bar Code Timer Control	06h	0	W
Bar Code Timer Value Capture	07h	None	W
Bar Code Timer Clear	08h	None	W
Main Control	0Bh	3-4	W
Main Status	0Bh	0	R

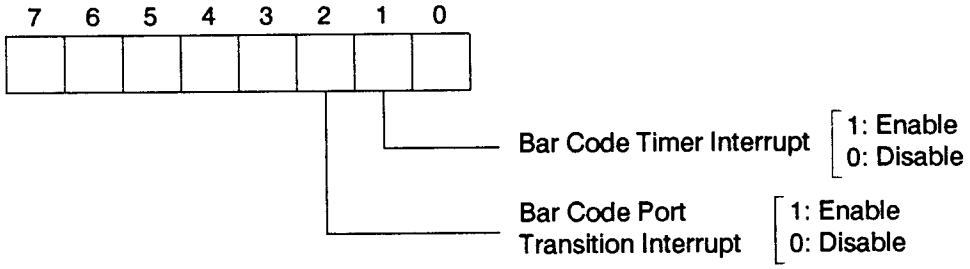


Figure 11-1. Interrupt Control Register (I/O Address 00h, Write)

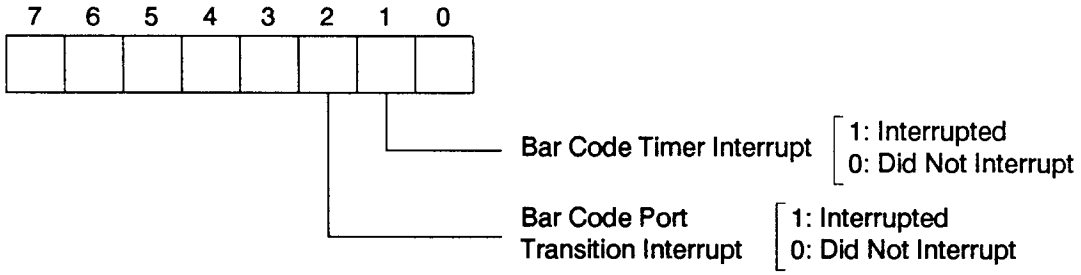


Figure 11-2. Interrupt Status Register (I/O Address 00h, Read)

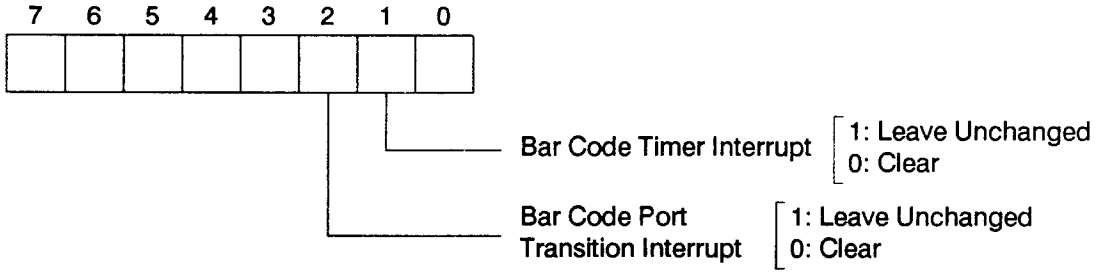


Figure 11-3. Interrupt Clear Register (I/O Address 01h, Write)

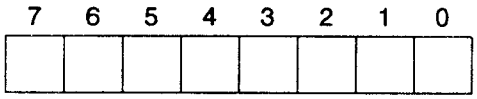
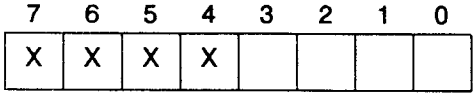


Figure 11-4. Bar Code Timer Data Register * (I/O Address 04h, Read/Write)

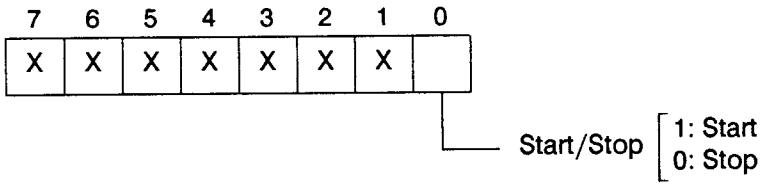


X = ignore

Figure 11-5. Bar Code Timer Data Register † (I/O Address 05h, Read/Write)

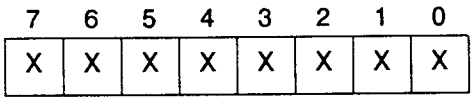
* Lower 8 bits of the 12-bit timer value.

† Upper 4 bits of 12-bit timer value.



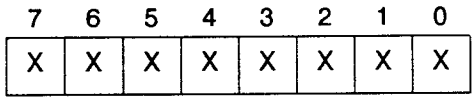
X = don't care

Figure 11-6. Bar Code Timer Control Register (I/O Address 06h, Write)



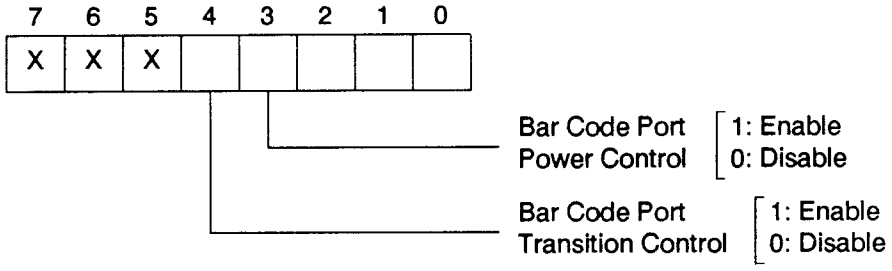
X = don't care

Figure 11-7. Bar Code Timer Value Capture Register (I/O Address 07h, Write)



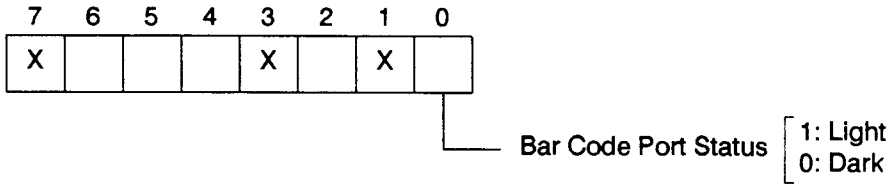
X = don't care

Figure 11-8. Bar Code Timer Clear Register (I/O Address 08h, Write)



X = don't care

Figure 11-9. Main Control Register (I/O Address 0Bh, Write)



X = ignore

Figure 11-10. Main Status Register (I/O Address 0Bh, Read)

12

Timers

Contents

Chapter 12 Timers

12-1 System Timer
12-2 System Timeout
12-2 Display Backlight Timeout
12-3 Background Timer
12-3 Bar Code Timer
12-4 Timer Control and Status Registers
12-7 Operating System Functions

12

Timers

The HP-94 has two timers available other than the real-time clock: the system timer and the bar code timer. These use a different time base than the real-time clock, and their accuracy is ±0.1%.

Table 12-1. HP-94 Timers

Timer Name	No. of Bits	Time Interval	Timer Type	Overflow Interval	Overflow Interrupt	Maximum Time
System	8	0.417 ms	up	5 ms	50h	106.7 ms
Bar Code	12	26 μs	up	— *	51h	106.7 ms
* Not defined by the operating system. Defined only by bar code port handler.						

System Timer

The system timer is an 8-bit count-up timer with an interval of 0.417 ms. It is initialized to -12 (-0Ch), so it overflows (counts up to zero) every 5 ms (12 * 0.417 = 5 ms, complemented because it is a count-up timer). When the system timer overflows, interrupt type 50h is generated. This interrupt is used to pace six different events in the operating system, shown below. While these events are checked and appropriate action is taken, interrupts are enabled except during the beeper event.

Table 12-2. Events Checked By System Timer Interrupt Routine

Timing Event	How Often Event Checked	How Often Action Taken
Scan Keyboard	5 ms	Put key into key buffer after 25 ms debounce Start key repeat if key still down after 675 ms Repeat key every 115 ms
Turn Off Beeper	10 ms	Turn beeper off after current beep time expires
Blink Cursor	100 ms	Blink cursor every 500 ms
System Timeout	1 s	Turn off machine or execute user-defined power switch/timeout routine after current system timeout expires
Display Backlight Timeout	1 s	Turn off backlight after current backlight timeout expires
Background Timer	1 s	Execute background timer interrupt routine every 1 s

NOTE While the beeper is checked to see if it needs to be turned off, interrupts are disabled for ~50 μs. This time may be important to bar code port handlers.

System Timeout

The system timeout is the time after which the machine will automatically turn off. It can be set from 0-1800 seconds using the TIMEOUT function (09h). The timeout is in effect while the machine is waiting for keyboard input or for data to be received at the serial or bar code ports. It will abort read operations from channels 0-4 and write operations to channels 1-4. It will not abort create, read, write, or delete operations for channels 5-15. The operating system will take one of the following actions when the system timeout expires:

- Turn off the HP-94.
  This is the default behavior if the program has not defined a power switch/timeout routine using the SET_INTR function (0Ah). The next time the machine is turned on, it will cold start.
- Execute the user-defined power switch/timeout interrupt routine.
  If the program has defined a power switch/timeout routine with SET_INTR, that routine will be executed with a FAR CALL (and therefore must end with a FAR RET). The AL register will be set to 76h, the timeout error, and the DS register will be set to the value specified when SET_INTR was called. This will only occur during a running program, not in command mode. When timeouts are monitored during I/O by a user-defined handler, the handler must execute the user-defined interrupt routine.
- Ignore the system timeout.
  If the program has disabled the system timeout by setting the timeout value to 0 with TIMEOUT, the operating system will ignore the system timeout.

The TERM routine of any open user-defined handlers will *not* be executed. Since each handler must monitor the system timeout itself, that handler will be the only one waiting on I/O when the timeout expires. Consequently, it is the only one that needs to terminate I/O.

Display Backlight Timeout

The display backlight timeout is the time after which the machine will automatically turn off the display backlight. This timeout is in effect whenever the backlight is on. It can be set from 0-1800 seconds using the TIMEOUT function.

When the display control code is processed to turn on or off the display backlight, the operating system controls the backlight state when keyboard scanning is done. If the system timer is disabled, no scanning is done, so the backlight will not be controlled. If a program disables the system timer, it must turn on the backlight explicitly using the keyboard control register, and then turn the backlight off explicitly after the timeout expires.

CAUTION Leaving the display backlight on continuously or for long periods of time (greater than 5 minutes) will reduce the life of the backlight.

Background Timer

The background timer is a one-second heartbeat timer that the machine provides for assembly language programs to use. Once a second, the operating system will issue a FAR CALL to the address in interrupt vector 1Ch, the background timer interrupt.

To take over the background timer interrupt, the program must do the following:

- Read the background timer interrupt vector (address 1Ch * 4 = 00070h), and save it in the program's scratch area.
- Write the address of the program's background timer interrupt routine into the vector location. The instruction pointer (IP) offset should be stored at the first word, and the code segment (CS) address should be stored at the second word.

To use the background timer interrupt, the program must do the following:

- When the interrupt routine is called, perform whatever processing is necessary.
- At the end of the routine, execute a FAR JMP to the address of the previous background timer interrupt routine.

The FAR JMP has the effect of daisy-chaining all the background timer interrupt routines together, allowing different programs to share the same interrupt. The last routine in the chain is the default routine, which is simply a FAR RET to end the aggregate background timer interrupt.

If the background timer does not provide enough resolution (1 second) for the program, the program can take over the system timer interrupt (vector at address 50h * 4 = 00140h) in the same manner (save the current interrupt vector, and FAR JMP to it at the end of the interrupt routine). This will provide a 5 ms timing resolution.

CAUTION The background timer routine must not clear the CPU interrupt flag (CLI). Doing so may cause interrupts from hardware devices to be delayed long enough that time-critical interrupt service routines (for open user-defined handlers) may miss their data.

Bar Code Timer

The second timer is the bar code timer, a 12-bit count-up timer with an interval of 26 μs. It is reserved for use by bar code port handlers, so it is never initialized to any value by the operating system. Like the system timer, it must be set using the complement of the desired number of intervals. When it overflows, interrupt type 51h is generated.

When either timer reaches zero, the timer is automatically reset to its starting value and restarted. If

the count value has to be set to a specific value, the timer must be stopped first. The bar code timer can be reset to zero or have its current value captured while it is still running.

Timer Control and Status Registers

The timer control and status registers are shown below.

Table 12-3. Timer Control and Status Registers

Register Name	I/O Address	Bits Used	Read/ Write
Interrupt Control	00h	0-1	W
Interrupt Status	00h	0-1	R
Interrupt Clear	01h	0-1	W
System Timer Data	02h	0-7	R/W
System Timer Control	03h	0	W
Bar Code Timer Data	04h	0-7	R/W
Bar Code Timer Data	05h	0-3	R/W
Bar Code Timer Control	06h	0	W
Bar Code Timer Value Capture	07h	None	W
Bar Code Timer Clear	08h	None	W

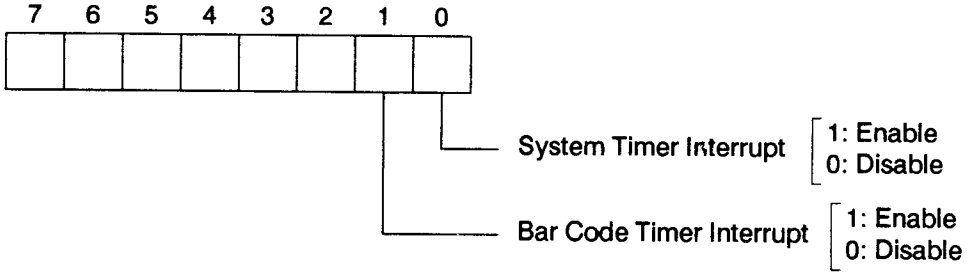


Figure 12-1. Interrupt Control Register (I/O Address 00h, Write)

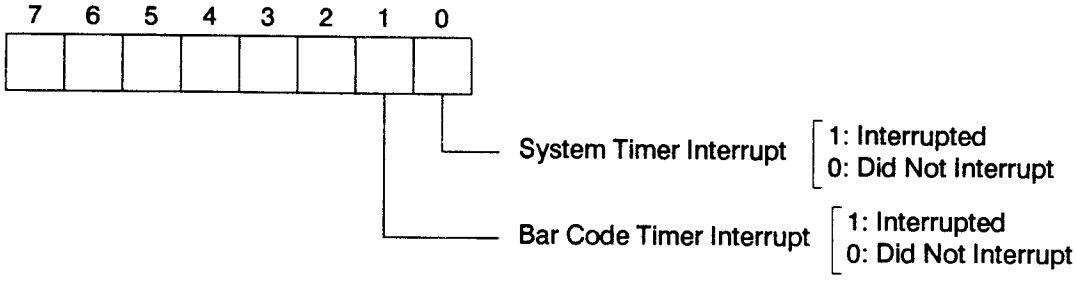


Figure 12-2. Interrupt Status Register (I/O Address 00h, Read)

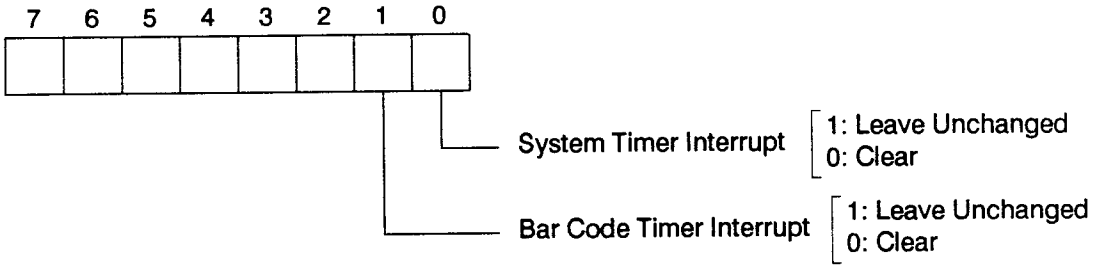


Figure 12-3. Interrupt Clear Register (I/O Address 01h, Write)

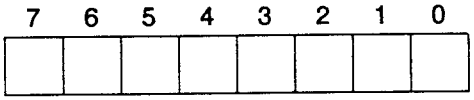
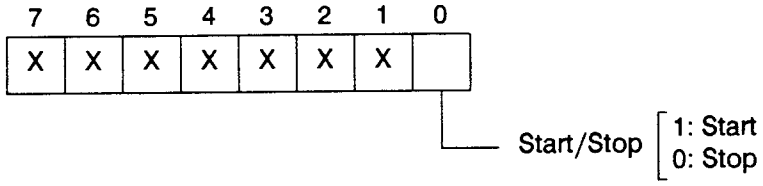


Figure 12-4. System Timer Data Register (I/O Address 02h, Read/Write)



X = don't care

Figure 12-5. System Timer Control Register (I/O Address 03h, Write)

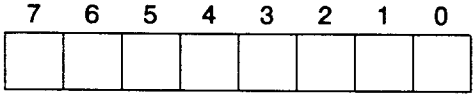
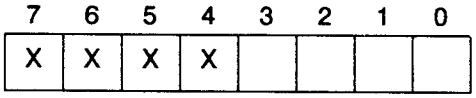
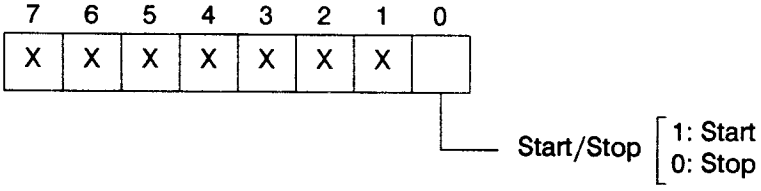


Figure 12-6. Bar Code Timer Data Register * (I/O Address 04h, Read/Write)



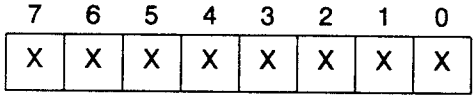
X = ignore

Figure 12-7. Bar Code Timer Data Register † (I/O Address 05h, Read/Write)



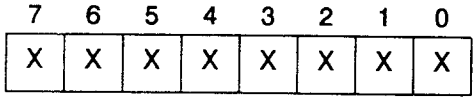
X = don't care

Figure 12-8. Bar Code Timer Control Register (I/O Address 06h, Write)



X = don't care

Figure 12-9. Bar Code Timer Value Capture Register (I/O Address 07h, Write)



X = don't care

Figure 12-10. Bar Code Timer Clear Register (I/O Address 08h, Write)

* Lower 8 bits of the 12-bit timer value.

† Upper 4 bits of 12-bit timer value.

Operating System Functions

The timer software implements the following operating system functions:

Table 12-4. Timer-Related Operating System Functions

Function Name	Function Code
TIMEOUT	09h
SET_INTR	0Ah

13

Power Switch

Contents

Chapter 13 **Power Switch**

13-1 Power Control and Status Registers
13-2 Operating System Functions

13

Power Switch

The HP-94 power switch provides software control for turning the machine off. When the HP-94 is off, pressing the power switch turns the machine on. When the machine is on, pressing the power switch generates interrupt type 55h. The power switch interrupt will abort read operations from channels 0-4 and write operations to channels 1-4. It will not abort create, read, write, or delete operations for channels 5-15. The operating system will take one of the following actions in response to this interrupt:

- Turn off the HP-94.
  This is the default behavior if the program has not defined a power switch/timeout interrupt routine using the SET_INTR function (0Ah). The next time the machine turns on, it will cold start.
- Execute the user-defined power switch/timeout routine.
  If the program has defined a power switch/timeout interrupt routine with SET_INTR, that routine will be executed with a FAR CALL (and therefore must end with a FAR RET). The AL register will be set to 77h, the power switch error, and the DS register will be set to the value specified when SET_INTR was called. This only occurs when the power switch is pressed during a running program, not in command mode.
- Ignore the power switch.
  If the program has disabled the power switch with SET_INTR, the operating system will respond to the interrupt but take no action, thereby ignoring the power switch.

In the first two cases, the TERM routine of any open user-defined handlers will be executed before the action is taken.

To turn the machine off, the operating system writes to the power control register.

Power Control and Status Registers

The power control and status registers are shown below.

Table 13-1. Power Control and Status Registers

Register Name	I/O Address	Bits Used	Read/ Write
Interrupt Control	00h	5	W
Interrupt Status	00h	5	R
Interrupt Clear	01h	5	W
Power Control	1Bh	None	W

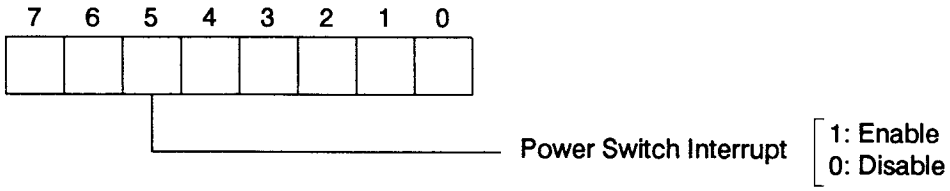


Figure 13-1. Interrupt Control Register (I/O Address 00h, Write)

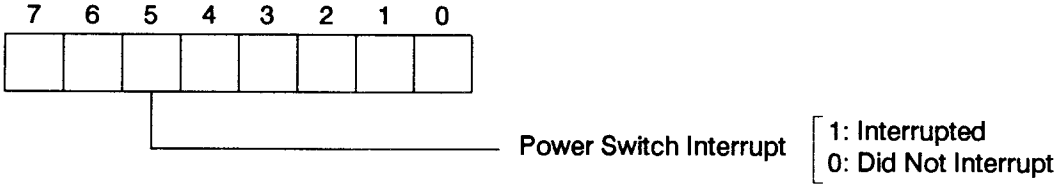


Figure 13-2. Interrupt Status Register (I/O Address 00h, Read)

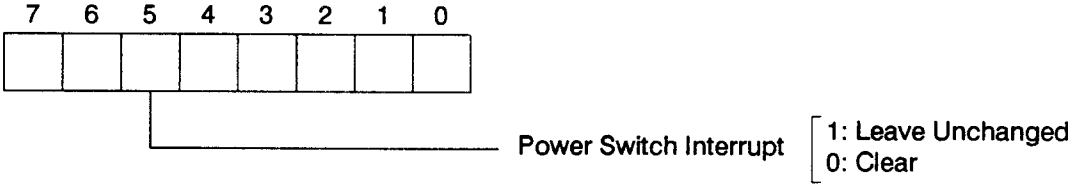
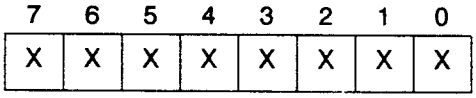


Figure 13-3. Interrupt Clear Register (I/O Address 01h, Write)



X = don't care

Figure 13-4. Power Control Register (I/O Address 1Bh, Write)

Operating System Functions

The power switch software implements the following operating system functions:

Table 13-2. Power Switch-Related Operating System Functions

Function Name	Function Code
END_PROGRAM	00h
SET_INTR	0Ah

14

Batteries

Contents

Chapter 14 **Batteries**

14-1 Main Nickel-Cadmium Battery Pack
14-2 Backup Lithium Batteries
14-2 Battery Control and Status Registers
14-4 Operating System Functions

14

Batteries

The HP-94 contains two types of batteries: nickel-cadmium batteries as the main power source, and lithium batteries for memory backup. Details about the characteristics of these batteries is in the "Hardware Specifications" elsewhere in this manual.

Main Nickel-Cadmium Battery Pack

The main power source for the machine is a rechargeable nickel-cadmium (NiCd) battery pack with a nominal capacity of 900 mAh. The machine operating voltage (which is slightly below the battery pack voltage) is continuously checked by the low battery detection circuitry whenever the machine is on. When the operating voltage drops to 4.6 ± 0.05 volts or below, interrupt type 54h is generated. The low battery interrupt will abort read operations from channels 0-4 and write operations to channels 1-4. It will not abort create, read, write, or delete operations for channels 5-15. The operating system will take one of the following actions in response to this interrupt:

- Halt all machine activities, issue error 200, and wait for the user to press the power switch to turn the machine off.
  This is the default behavior if the program has not defined a low battery interrupt routine using the SET_INTR function (0Ah). The following activities are halted:

Table 14-1. Activities Halted During Default Low Battery Behavior

Activity or Device	Action Taken
Cursor	Turned Off
Interrupts	Disabled
System Timer	Turned Off
Bar Code Timer	Turned Off
Beeper	Turned Off
Keyboard	Disabled
Display Backlight	Turned Off
Serial Port	Disabled
Serial Port Power	Turned Off
Bar Code Power	Turned Off
Bar Code Transitions	Disabled

  The next time the machine is turned on, it will cold start.

- Execute the user-defined low battery routine.
  If the program has defined a low battery interrupt routine with SET_INTR, that routine will be executed with a FAR CALL (and therefore must end with a FAR RET). The DS register will be set to the value specified when SET_INTR was called. This only occurs during a running program,

not in command mode.

In both cases, the TERM routine of any open user-defined handlers will be executed before the action is taken. SET_INTR does not allow disabling the low battery interrupt.

The low battery interrupt only occurs once, when the main battery voltage drops below 4.6 volts. At that point, the program has 2-5 minutes left before the battery voltage drops so low that the machine turns itself off automatically without warning. The low battery interrupt will not occur again until the machine has been turned off and back on. If the battery remains below 4.6 volts while the 94 is off, the machine will not turn back on again until the battery has been recharged enough to bring its voltage above that level (~4.8 volts). (The machine actually turns on, but the operating system turns it off before any memory integrity tests are performed if the voltage is too low.)

The actual amount of time available depends on what is happening when the low battery condition occurs. For example, the display backlight takes more power, as does the HP 82470A RS-232-C Level Converter (if one is connected to the serial port), so less operating time will be available if these are on. The time also depends on how much the battery was charged during its last charging cycle, the ambient temperature, and many other factors. Because the remaining operating time is variable, the program should respond to the low battery interrupt as rapidly as possible by ending its activities (shut off I/O and powered devices, complete file updates that were in progress, etc.), notifying the user that it is necessary to recharge the main battery, and turning the power off.

If the program continues operating until the machine turns itself off automatically, the effect is as if the reset switch was pressed. No data in data files will be lost, since the backup batteries will keep memory intact, but the machine will cold start the next time it is turned on. This means that any data in program variables or scratch areas that did not get saved in a data file will be lost.

Backup Lithium Batteries

The backup power source is user-replaceable 3-volt lithium backup batteries, CR-2032 or equivalent. There is one lithium battery for each major block of RAM: one for the first 64 or 128K (which also backs up the real-time clock), one for the 128K memory board, and one for the 40K RAM card. The mainframe lithium batteries are accessible through the back cover, and the RAM card battery is under a cover on the card. These batteries are only used to preserve the contents of memory when the main NiCd battery pack is completely discharged or disconnected (and there is no recharger connected). They are not used when other power sources are available to preserve memory.

Their state is checked and reported only when the machine is turned on, after all memory integrity tests are performed. Error 210 is reported at power on to indicate low voltage (2.7 volts) of the battery for the first 64 or 128K, while error 211 is reported for the memory board or RAM card battery. Both errors will be reported if both batteries need replacing.

Battery Control and Status Registers

The battery control and status registers are shown below.

Table 14-2. Battery Control and Status Registers

Register Name	I/O Address	Bits Used	Read/ Write
Interrupt Control	00h	4	W
Interrupt Status	00h	4	R
Interrupt Clear	01h	4	W
Main Status	0Bh	4-6	R

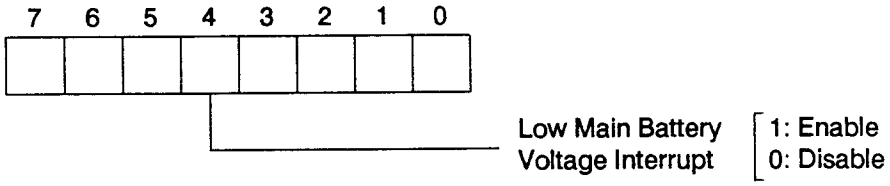


Figure 14-1. Interrupt Control Register (I/O Address 00h, Write)

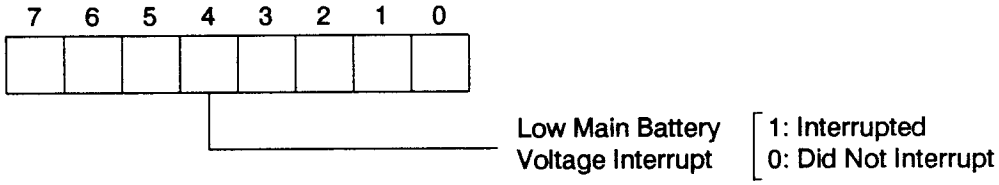


Figure 14-2. Interrupt Status Register (I/O Address 00h, Read)

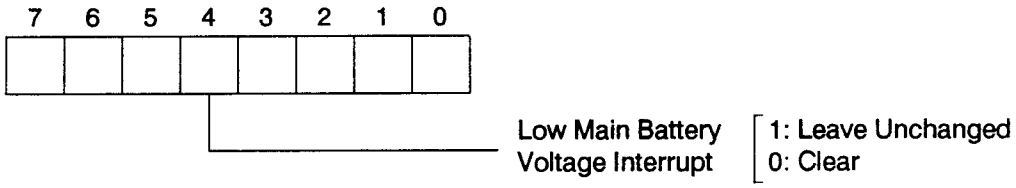
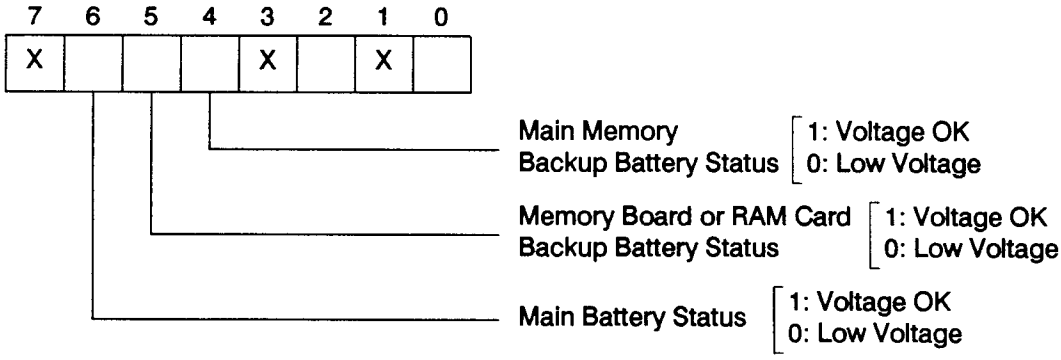


Figure 14-3. Interrupt Clear Register (I/O Address 01h, Write)



X = ignore

Figure 14-4. Main Status Register (I/O Address 0Bh, Read)

Operating System Functions

The battery software implements the following operating system functions:

Table 14-3. Battery-Related Operating System Functions

Function Name	Function Code
SET_INTR	0Ah

15

Real-Time Clock

Contents

Chapter 15 **Real-Time Clock**

15-1 Real-Time Clock Control and Status Registers
15-1 Operating System Functions

15

Real-Time Clock

The HP-94 contains an Epson RTC-58321 real-time clock. Its quartz crystal operates at 32768 Hz, and is backed up by the main memory lithium backup battery if the main NiCd battery is completely discharged or removed. The clock has a one-second resolution, and is accurate to ±50 ppm (~2 minutes/month). The clock supports time, date, and day-of-week functions, but the clock software in the operating system only supports time and date, as well as the T (*time*) operating system command. Leap years are accommodated automatically. The details of the real-time clock hardware, operation, and usage are described in the Epson RTC-58321 data sheet in the "Hardware Specifications".

The operating system provides the TIME_DATE function (08h) to set or read the time and date. No syntax checking is performed on the time and date when they are set. It is the responsibility of the application program to ensure that the time and date are in the proper format when they are set.

Real-Time Clock Control and Status Registers

The real-time clock control and status registers are shown below.

Table 15-1. Real-Time Clock Control and Status Registers

Register Name	I/O Address	Bits Used	Read/ Write
Real-Time Clock Control/Data	0Ch	0-7 *	W
Real-Time Clock Status/Data	0Ch	0-4 *	R

* For the meaning of the bits in these registers, refer to the Epson RTC-58321 data sheet in the "Hardware Specifications".

Operating System Functions

The real-time clock software implements the following operating system functions:

Table 15-2. Real-Time Clock-Related Operating System Functions

Function Name	Function Code
TIME_DATE	08h

16

Beeper

Contents

Chapter 16 **Beeper**

16-1 Beeper Control and Status Registers
16-2 Operating System Functions

16

Beeper

The HP-94 beeper is a piezoelectric buzzer that is turned on and off using the main control register. If a program turns the beeper on explicitly, it is responsible for turning it off as well after the appropriate duration. If a program uses the operating system BEEP function (07h), the operating system will turn the beeper off automatically after the specified time has elapsed.

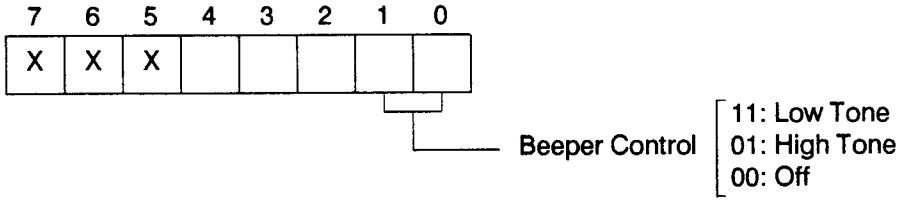
The BEEP function allows specifying beep durations from 0.1 to 25.5 seconds, and either high or low tones. It can be called while the beeper is beeping. If the tone specified is different than the tone in progress, beeping will continue at the high tone and duration — the high tone and its duration will take precedence regardless of the order in which the tones were specified. If the tone specified is the same as the tone in progress, beeping will continue at either the remaining duration or the new duration, whichever is longer.

Beeper Control and Status Registers

The beeper control and status registers are shown below.

Table 16-1. Beeper Control and Status Registers

Register Name	I/O Address	Bits Used	Read/ Write
Main Control	0Bh	0-1	W



X = don't care

Figure 16-1. Main Control Register (I/O Address 0Bh, Write)

Operating System Functions

The beeper software implements the following operating system functions:

Table 16-2. Beeper-Related Operating System Functions

Function Name	Function Code
BEEP	07h

17

Reset Switch

17

Reset Switch

The HP-94 has a small reset switch to the left of the power switch. Since the power switch is under program control, it is possible for a program to inadvertently prevent the user from turning off the machine. The reset switch is provided to accommodate this situation.

The reset switch is a hardware power off, not a software power off. When the reset switch is pressed, the machine is turned off immediately. No data in data files will be lost, since the backup batteries will keep memory intact, but the machine will cold start the next time it is turned on. This means that any data in program variables or scratch areas that did not get saved in a data file will be lost.

The TERM routine of any open user-defined handlers will not be executed, and no power-off checksums will be computed. The next time the machine is turned on, it will not compute power-on checksums (although the other memory integrity tests will be performed).

18

Other Hardware

Contents

Chapter 18 **Other Hardware**

18-1 Read/Write Memory (RAM)
18-1 System ROM
18-1 Custom Gate Array
18-2 Earphone Jack
18-2 External Bus Connector

18

Other Hardware

The HP-94 has some other hardware elements that will be discussed here: read/write memory (RAM), system ROM, custom gate array, earphone jack and external bus connector.

Read/Write Memory (RAM)

HP-94 read/write memory is Toshiba TC5565FL-15L CMOS static RAM (8K × 8). Refer to the "Memory Management" chapter for a detailed description of the memory organization. Major hardware blocks of memory are backed up by user-replaceable lithium backup batteries; refer to the "Batteries" chapter for details.

System ROM

The HP-94 has 32K of EPROM located in the upper 32K of the CPU address space. The system ROM contains all the HP-94 built-in software. Refer to the "Memory Management" chapter for a detailed description of the system ROM organization.

Custom Gate Array

The HP-94 contains a proprietary Hitachi 61L224 custom gate array that combines what would otherwise be several separate integrated circuits (ICs). The following is a list of the major hardware facilities provided by the gate array:

- Interrupt controller for HP-94 hardware interrupts.
- Hardware control registers (except for keyboard, display, and 82C51).
- Power off control.
- System timer.
- Serial port power and baud rate clock.
- Bar code port power control, transition detection, and timer.
- Real-time clock control.
- Beeper tone.
- Chip select address decoding.
- Address/data bus latches.
- Status of data carrier detect (DCD) control line.

Earphone Jack

The earphone jack accepts any standard earphone with a 3.5 mm plug. It allows the user of the machine to hear the beeper (particularly for applications using bar code) in noisy environments.

External Bus Connector

The external bus connector is located on the underside of the HP-94 behind a hard plastic port cover. It brings out all lines from the internal system bus. Details about the external bus connector (pin assignments, voltages, currents, and logic levels are described in the "Hardware Specifications".

Part 2

BASIC Interpreter

1

BASIC Program and Data Structure

Contents

Chapter 1 BASIC Program and Data Structure

1-1 BASIC Program Organization
1-2 BASIC Program Outline
1-4 Intermediate Code
1-4 Operand Codes
1-6 Explanation of Operand Codes
1-9 Variable Area
1-13 Data Structure
1-13 Real Numeric Data
1-13 Integer Numeric Data
1-14 Character Data
1-14 Array Data
1-15 Array Examples
1-16 Control Information Save Area

1

BASIC Program and Data Structure

BASIC application programs (type B) are interpreted by the HP-94 BASIC interpreter (SYBI). The BASIC application program may be in either RAM or ROM.

BASIC Program Organization

The following figure shows the organization of a BASIC program.

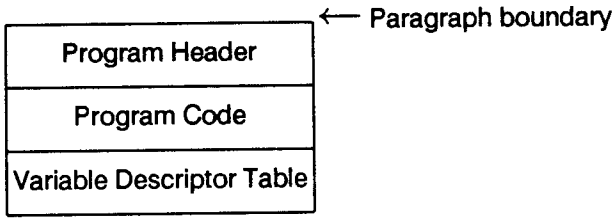


Figure 1-1. BASIC Program Organization

A variable area is necessary to execute a BASIC program. A control information save area is necessary when a CALL statement or an interrupt process routine is executed. The variable area and the control information save area are dynamically allocated in main memory.

BASIC Program Outline

BASIC programs start with a 10h byte program header. The contents of the header are shown below with hex offsets listed on the left side and a brief description on the right. The program code and variable descriptor table are also shown to illustrate their location and size.

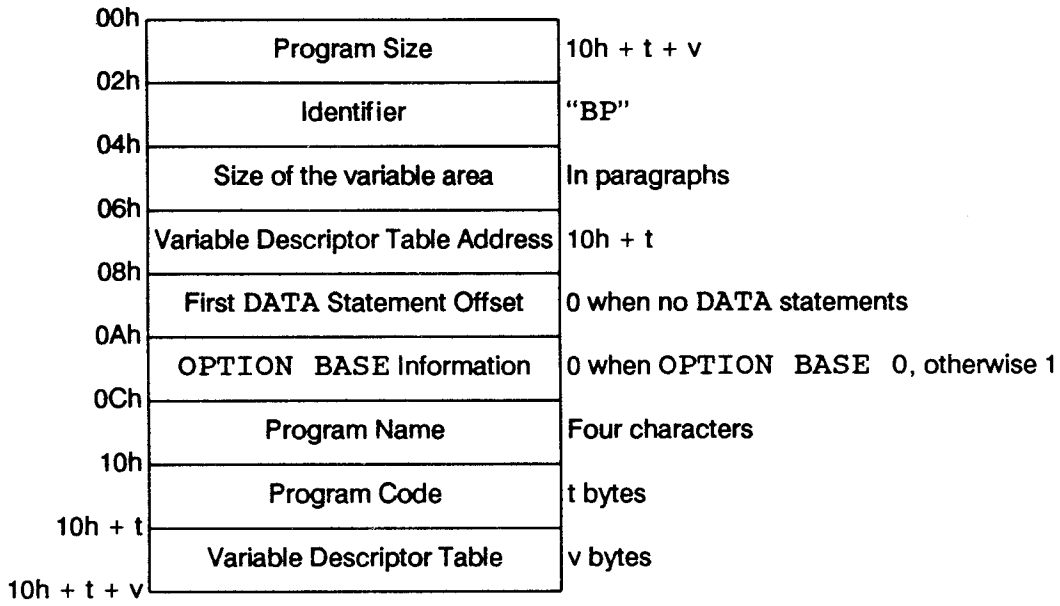


Figure 1-2. Program Header

The following figure shows the organization of the BASIC program code.

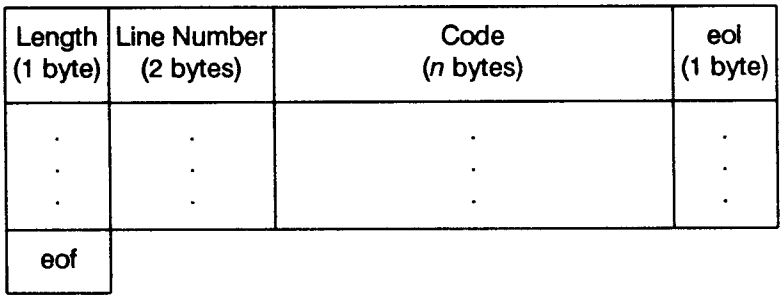


Figure 1-3. Program Code

The information contained in a line of program code is:

- Length: number of bytes in a line = 1 + 2 + *n* + 1 (must be less than 256).
- Line Number: 0 through 32767 (0000h through 7FFFh, least significant 8 bits first).
- eol (end of line) and eof (end of file) are the NUL character (00h).
- Some lines, such as comments, generate no program code.

Type (1 byte)	Length (1 byte)	Segment Address (2 bytes)	Offset Address (1 byte)

Figure 1-4. Variable Descriptor Table

The variable descriptor table contains information about the type, length, and address of each variable. The figure above illustrates the table organization. The meanings of the fields are as follows:

- Type:

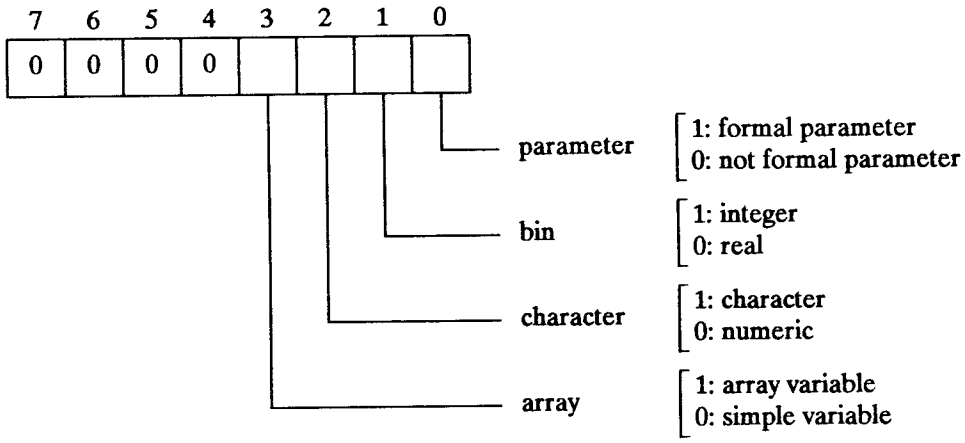


Figure 1-5. Variable Descriptor Type Byte

- Length:

Table 1-1. Variable Descriptor Length Byte

Type	Length in Bytes
Integer	2
Real	8
Character	Dimensioned size (default 8)
Parameter	5 *
Array	Size of one array element

* This entry points to another descriptor entry which contains the actual information for the variable.

- Segment Address, Offset Address:
  The segment address and offset address are a pointer to the variable data in the user variable area. They are relative to the start of the variable area. The first byte of the segment address field contains the least significant 8 bits of the segment address. The offset address contains values in the range 00h through 0Fh. When the parameter bit is 1, the segment and offset addresses are the address of the variable descriptor entry for the parameter.

Intermediate Code

Codes interpreted by the HP-94 BASIC interpreter are called intermediate code.

Table 1-2. Intermediate Code

	0x	1x	2x	3x	4x	5x	6x	7x	8x	9x	Ax	Bx	Cx	Dx	Ex	Fx
x0	eol		> =	LET	GET				SQR	FIX5	DMS					
x1			< =	GOTO	PUT				EXP	FIX9	ARD					
x2	bin		< >	GOSUB	PARAM				LOG	MAX	ADS					
x3	ral		>	RETURN	%CALL			TAB	LGT	MIN	FIXE					
x4	chr		<	FOR	DEF			XOR	SGN	RND	TIM					
x5	var		,	NEXT	READ			%CURSOR	ABS	EOF	PI					
x6	prm		;	IF	DATA			%HOME	INT	INPUT$	VER					
x7	fnc		:	ON	RESTORE			%DEL	LEN	TOD$	KEY					
x8	ext	(	#	DIM				AND	IDX	SIN	HEX$					
x9	lin	)		INPUT				OR	NUM	COS	SIZE					
xA	adr	+		PRINT				NOT	COD	TAN						
xB		-		CALL				TO	STR$	ASN						
xC	rem	**		END				STEP	CHR$	ACS						
xD		*		FORMAT				USING	ASC$	ATN						
xE		/		OPEN				MSG	MOD	FRC						
xF		=		CLOSE				SPACE	FIX0	RAD						

Note: A blank entry in the table indicates an unused code.

Table 1-3. Intermediate Code Groups

Code Group	Range of Codes
Operand	00h —— 17h
Delimiter	18h —— 2Fh
Statement	30h —— 47h
Optional Word	73h —— 7Fh
Function	80h —— A9h

Operand Codes

The symbols and formats for the various members of the operand code group are listed below.

Table 1-4. Operand Codes

Code	Symbol	Format	Comments
00h	eol	NUL	End of line
02h	bln	bln | VALUE (L), (H)	Integer constant
03h	ral	ral | real char string | NUL	Real type constant
04h	chr	chr | char string | NUL	Character constant
05h	var	var | ADRS (L), (H)	Variable
06h	prm	prm | ADRS (L), (H)	User-defined function parameter
07h	fnc	fnc | ADRS (L), (H) The ADRS after var, prm, and fnc is the appropriate position in the variable descriptor table	User-defined function
08h	ext	ext | external procedure name | NUL	Entry name (CALL and %CALL)
09h	lin	lin | ADDR (L), (H) lin is used by FORMAT, GOTO, GOSUB, and USING. The ADDR after lin is the relative offset to another line.	Text line address reference
0Ah	adr	adr | ADDR (L), (H) The ADDR after adr is the relative position from the start of the program.	Address of next DATA statement
0Ch	rem	rem | char string | NUL rem is used for the data in DATA statements or the format information in FORMAT statements.	Skipped during execution

Explanation of Operand Codes

The meaning of each operand code is as follows:

- eol (end of line)

  This indicates the end of a line in a program. Multiple statements within the line are separated with a colon (:), character code 27h.

- bin (integer constant)

  This indicates the following two bytes are an integer constant (-32768 through 32767). The first byte is the least significant 8 bits.

- ral (real constant)

  This indicates a real constant which is stored as a character string. Only positive numbers are stored in this format. Negative numbers are expressed as a unary expression.

  e.g. `-123.4` ⟶ - ral **1 2 3 . 4** NUL

- chr (character constant)

  This indicates a character string. It does not include the double quotation marks which specify the beginning and end of a character string. Two successive double quotation marks ("") indicate a double quote ("). Two successive ampersands (&&) indicate an ampersand (&). An ampersand (&) followed by two hexadecimal digits represents a single byte with that hexadecimal value.

- var (variable)

  The two bytes following var are the offset from the start of the variable descriptor table to the variable descriptor table entry. The first byte is the least significant 8 bits.

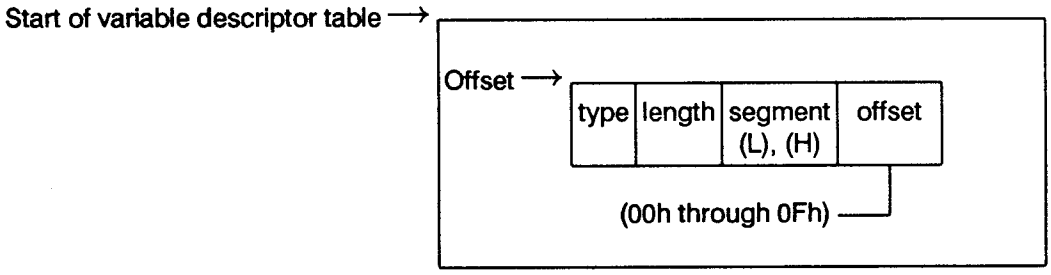


Figure 1-6. Variable Reference

- prm (user-defined function parameter name)

  The prm operand code is used for parameters in subprograms and user-defined functions.

  The two bytes following prm are the offset from the start of the variable descriptor table to the variable descriptor table entry. The first byte is the least significant 8 bits.

  The variable descriptor table entry has a type = 01h ('parameter') and length = 05h. The segment and offset values are relative to the start of the variable area. The variable area indicated by the variable descriptor table entry contains a variable descriptor table entry which has the correct type and length for the parameter. The segment and offset values in the latter entry are set to the actual address of the variable (*not* an offset from the start of the variable area).

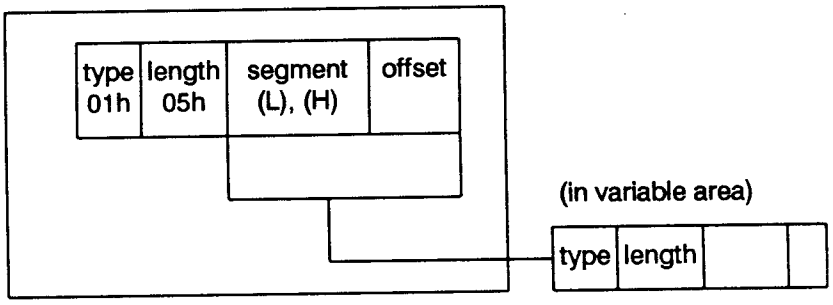


Figure 1-7. Parameters in the Variable Descriptor Table

- fnc (user-defined function)

  The two bytes following fnc are the offset from the start of the variable descriptor table to the variable descriptor table entry. The first byte is the least significant 8 bits.

  The variable descriptor table entry segment address field is the offset from the start of the program to the user-defined function definition. The final byte of this entry (offset) is 00h.

  If the definition contains one or more arguments, the segment address field points to the first argument. If the definition does not contain an argument, the segment address field points to the equals sign (=) which follows the definition.

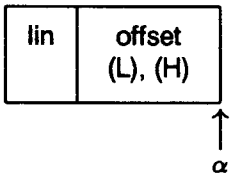
```
DEF FNA=
        ↑

DEF FNA(X,Y)=
        ↑
```

- ext (external program name)

  The subprogram name for CALL and %CALL is indicated with ext.

- lin (line reference)

  The two bytes following lin are an offset to the start of a line. The first byte is the least significant 8 bits. The offset is relative to the byte following the offset (the third byte following lin).



The start of the referenced line is (α) + offset

Figure 1-8. Line Reference

- adr (address reference)

  The adr operand code is used in DATA statements.

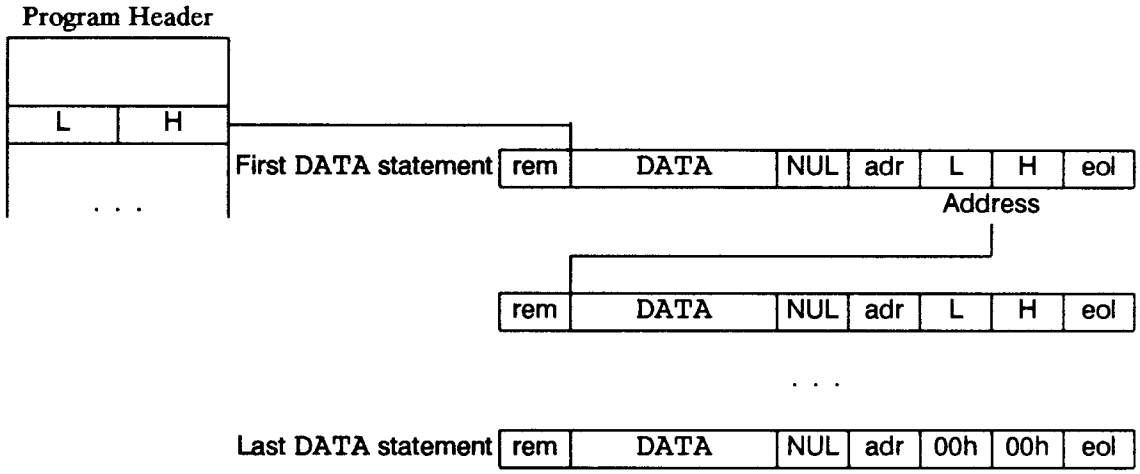


Figure 1-9. DATA Statement Linking

An adr of 0000h indicates the end of the DATA chain.

- rem (non-executed statement)

  The rem operand code indicates character strings for the FORMAT and DATA statements.

  A line with the rem operand code is not executed.

Variable Area

The variable area is allocated in main memory when BASIC program execution begins. It is released when execution ends.

The variable area is allocated or released as a block. The size of the variable area to be allocated is available in the variable area size field of the BASIC program header. The variable area is not allocated if the variable area size field is zero.

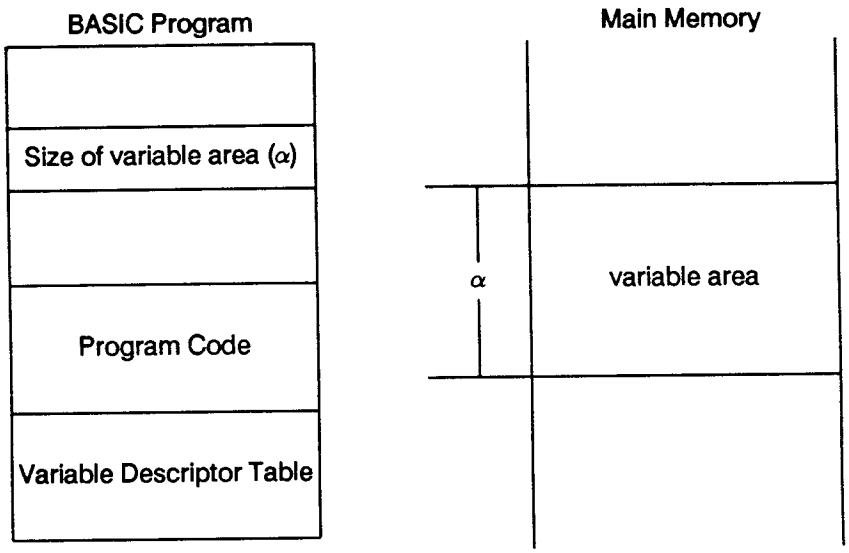


Figure 1-10. Variable Area Allocation

An example of the process of allocating and releasing variable areas is shown in the following figure. The example illustrates the main program (MAIN) calling a second program (program B), which in turn calls a third program (program C). Program C ends, returning control to program B. Program B also ends, returning control to MAIN. MAIN then ends, returning to command mode.

The control information save areas which are allocated between each variable area are omitted in this figure.

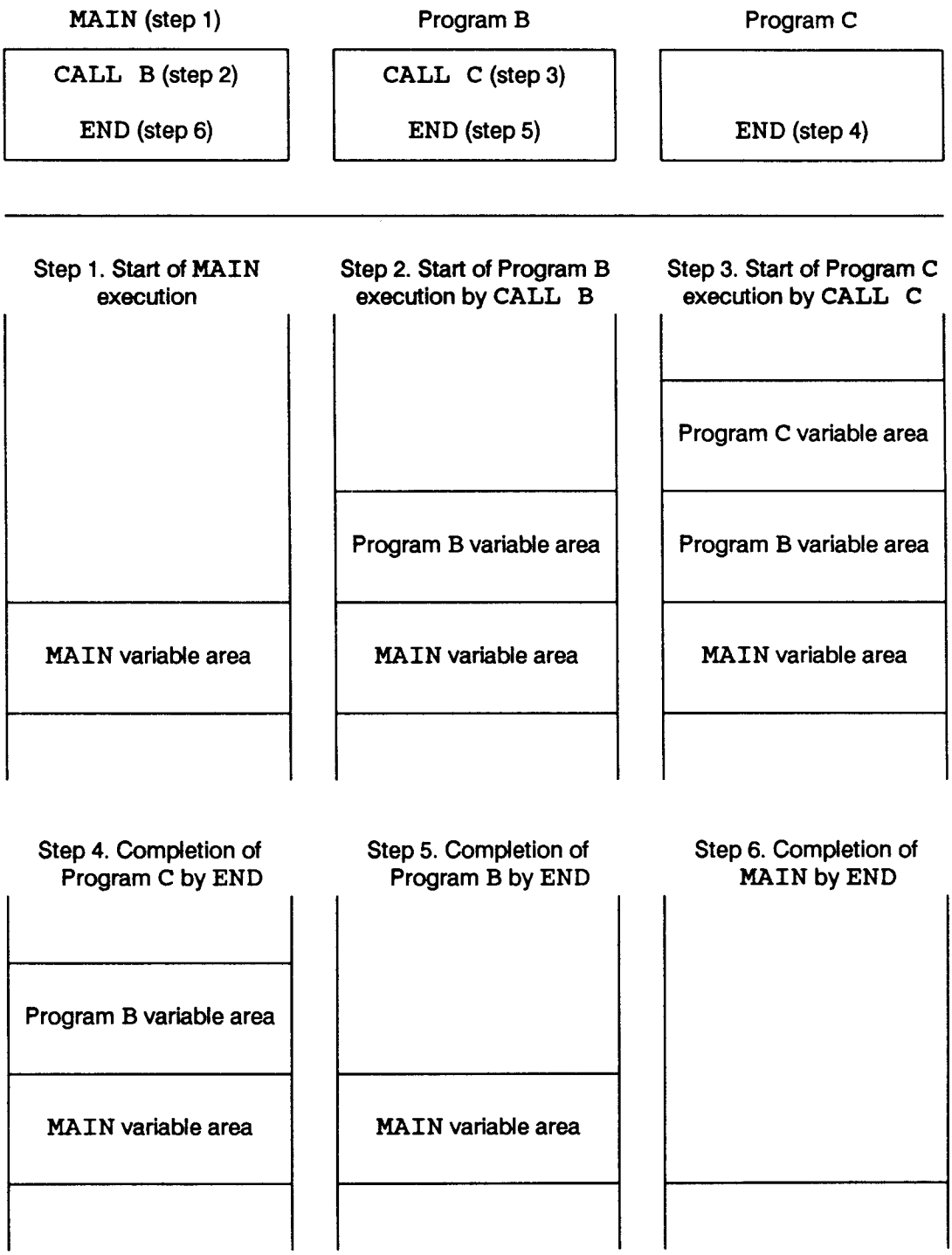


Figure 1-11. Allocating and Releasing Variable Areas

The relationship between the program code, variable descriptor table, and variable area is shown in the following figure.

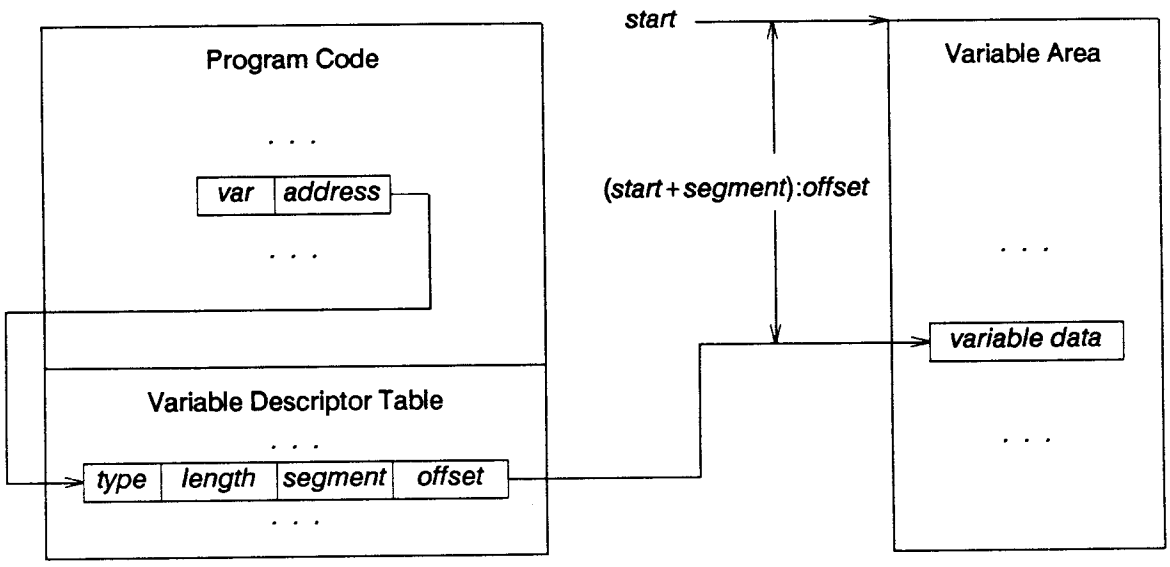


Figure 1-12. Program Code and Variables

The meaning of the items in italics is listed below.

- *var*
  Operand code for a variable
- *address*
  Relative address in the variable descriptor table
- *type*
  variable type
- *length*
  variable or array element length
- *segment*
  variable segment address relative to *start*
- *offset*
  variable offset address relative to *start*
- *start*
  Start of variable area (determined at CALL time)
- *variable data*
  Current value of the variable

An example showing several statements in a BASIC program helps clarify the relationship between program code and variables.

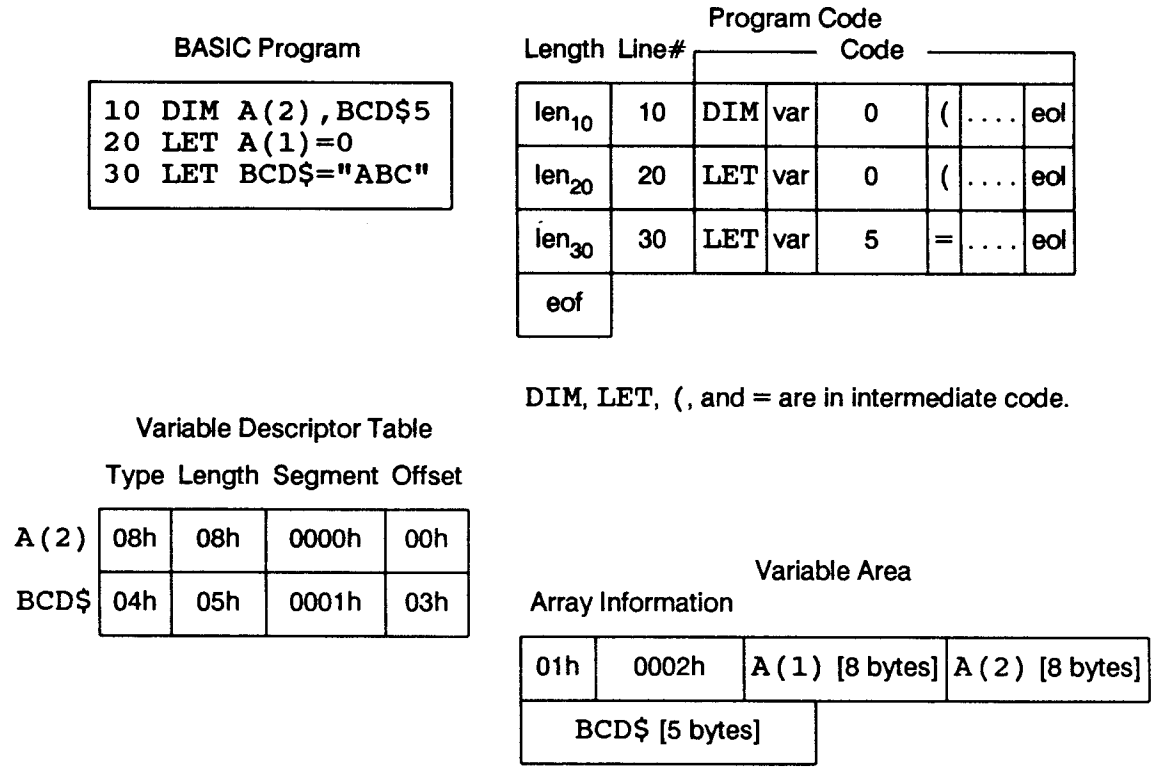


Figure 1-13. BASIC Program and Variable Relationships

Data Structure

There are three data types — real numeric data, integer numeric data, and character data. In addition, each of the data types can be collected into an array. Information about the array is stored preceding the elements in the array. The data in an array is stored consecutively.

Real Numeric Data

The format for real numeric data in the variable area is shown below.

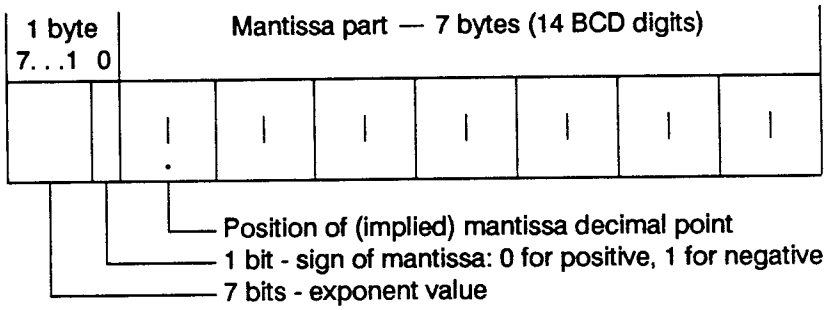


Figure 1-14. Real Numeric Data in the Variable Area

The exponent is in two's complement (binary). Exponent values -64 through 63 indicate 10^{-64} through 10^{63}.

Integer Numeric Data

Integers are stored as two bytes in both the variable area and data files; the first byte contains the most significant 8 bits, and the second byte contains the least significant 8 bits. The range of an integer is -32768 through 32767.

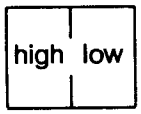


Figure 1-15. Integer Numeric Data in the Variable Area

Character Data

The format for character data in the variable area is shown below.

Figure 1-16. Character Data in the Variable Area

The default value for *n* is 8. A DIM statement can be used to assign values 1 through 255 to *n*. The value of *n* is in the variable descriptor table.

If the character string has fewer than *n* bytes, a NUL (00h) is stored following the last character of the string.

Only the first *n* characters assigned to a character string are stored — excess characters are discarded.

Array Data

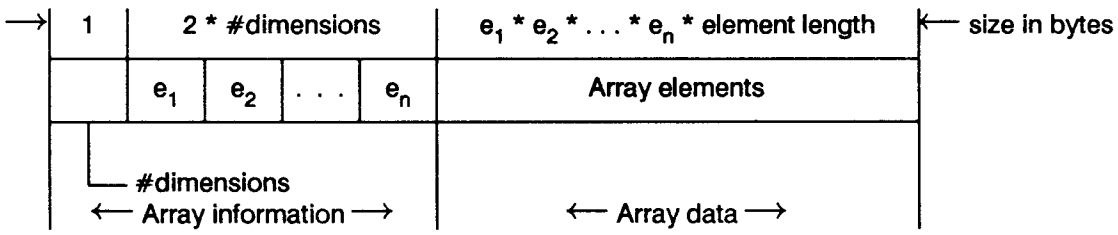


Figure 1-17. Array Data in the Variable Area

The maximum size of an array is 65535 (FFFFh) bytes, including both the array information and the array data. The number of dimensions must be in the range 1 through 255.

In the array information, e_n is the number of elements in that dimension. For `OPTION BASE 0`, the number of elements is the array's upper bound plus 1. Each e_n is stored with the least significant 8 bits in the first byte and the most significant 8 bits in the second byte.

Array elements are stored in row-major order (the right-most subscript varies most rapidly).

Array Examples

The following two examples show how the array information and data would be stored in memory.

Example: `DIM A(2,3)`

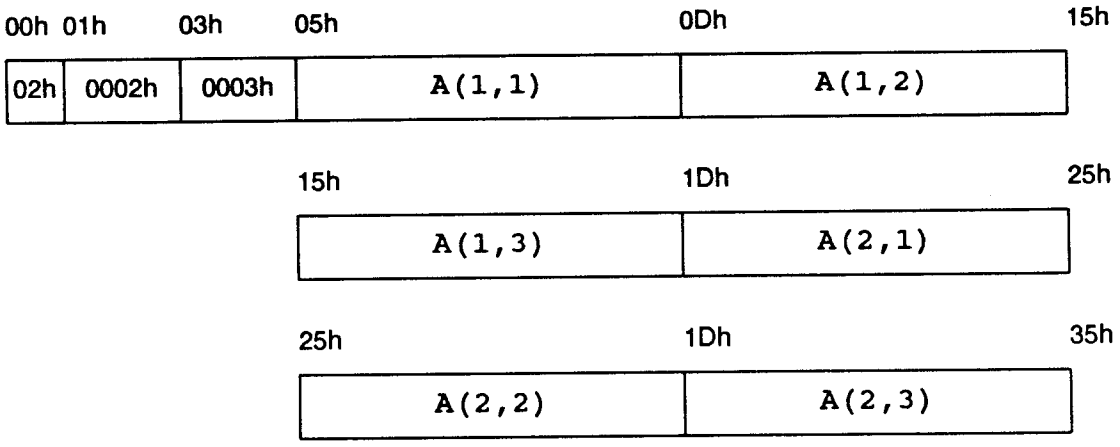


Figure 1-18. Array Data Example: DIM A(2,3)

Example: `OPTION BASE 0:DIM B$6(4)`

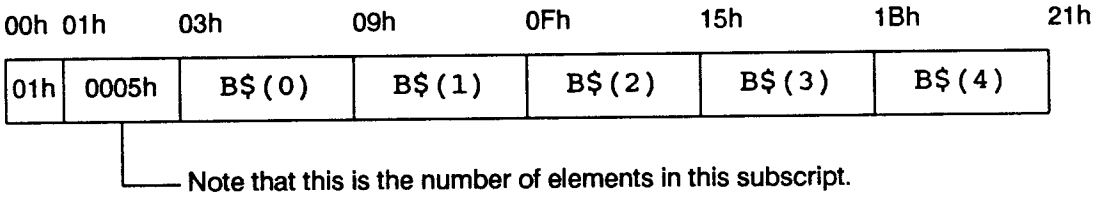


Figure 1-19. Array Data Example: OPTION BASE 0 : DIM B$6(4)

Control Information Save Area

The control information save area is used to save the control information of the currently executing program when a subprogram is called with the CALL statement or when an interrupt causes a jump to an interrupt routine.

The control information save area is allocated in main memory when a CALL statement or interrupt occurs. The control information for the currently executing program is saved in the save area. When the subprogram ends (END) or the interrupt routine ends (%CALL SYRT), the information is restored to the BASIC interpreter control area.

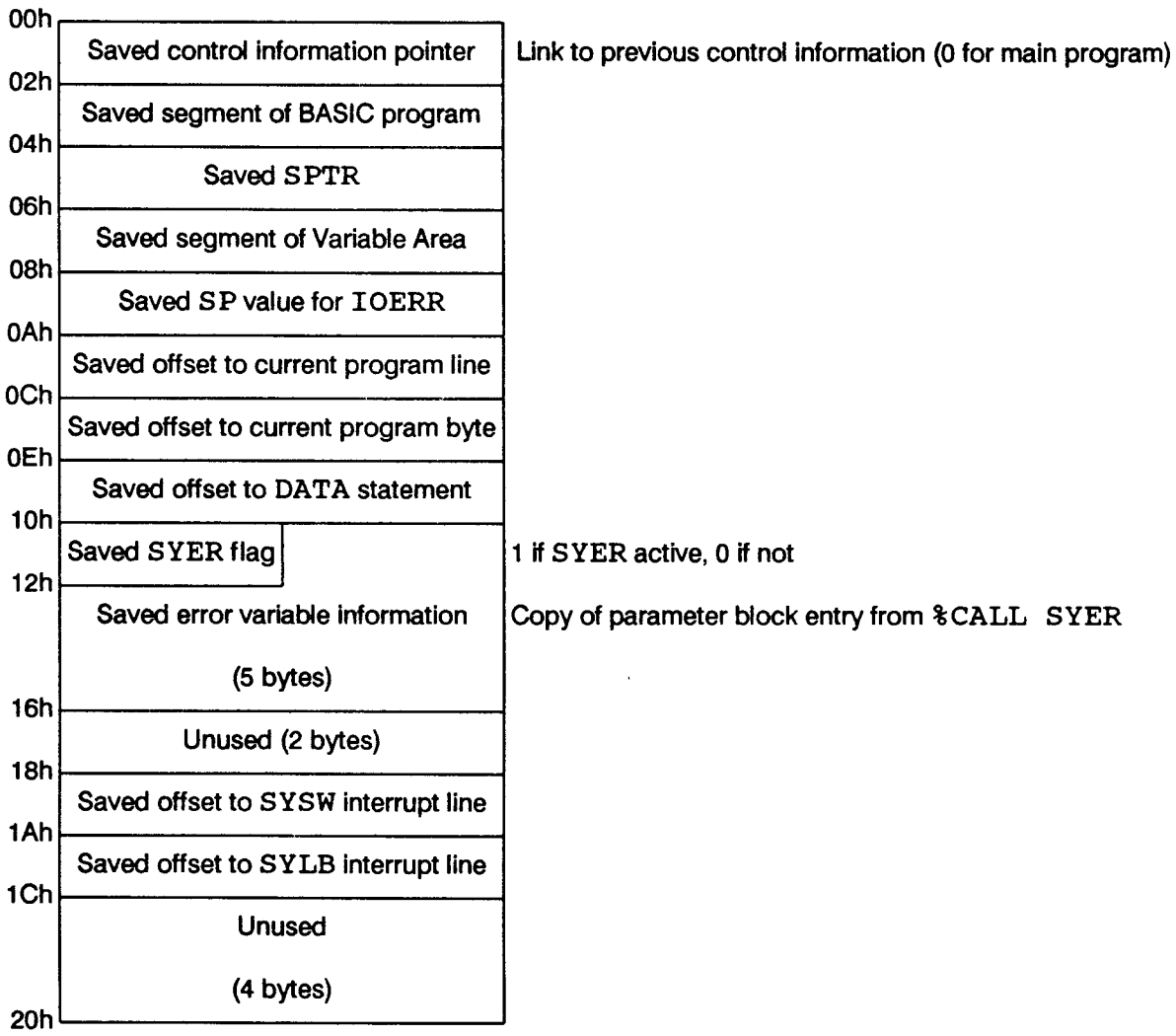


Figure 1-20. Format of the Control Information Save Area

2

Operation Stacks

Contents

Chapter 2 Operation Stacks

2-1 Operation Stack Area
2-2 Control Stack
2-3 GOSUB Control Element
2-4 FOR . . . NEXT Control Element
2-5 Numeric Operation Stack
2-5 Real Numeric Data
2-6 Integer Numeric Data
2-6 Numeric Operation Stack Example
2-7 Character Operation Stack
2-7 Character Operation Stack Example
2-8 Parameter Table (only for %CALL)

2

Operation Stacks

The operation stack area is used for:

- Control stack
- Numeric operation stack
- Character operation stack
- Parameter table entries (for %CALL)

Operation Stack Area

Parameters which are passed by value (constants and expressions) are evaluated, and the result of the expression is stored in the operation stack area.

The character operation stack pointer is CPTR, and the numeric operation stack pointer is SPTR.

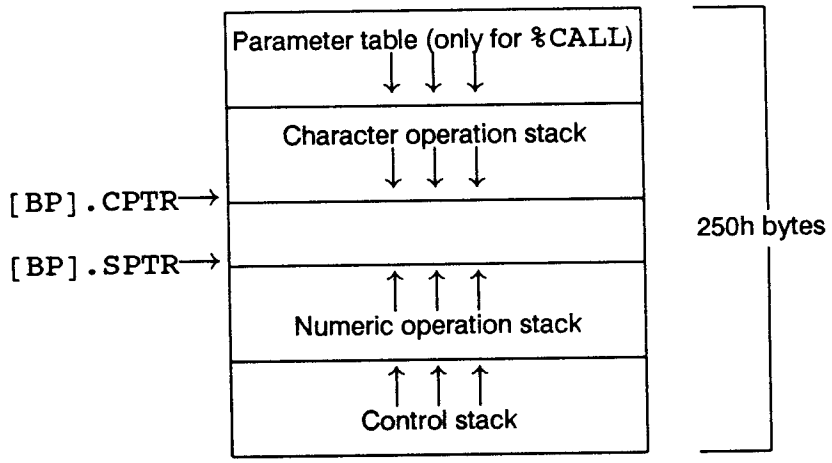


Figure 2-1. Operation Stack Area

Control Stack

The control stack is used to maintain address and variable information for GOSUB and FOR . . . NEXT loops.

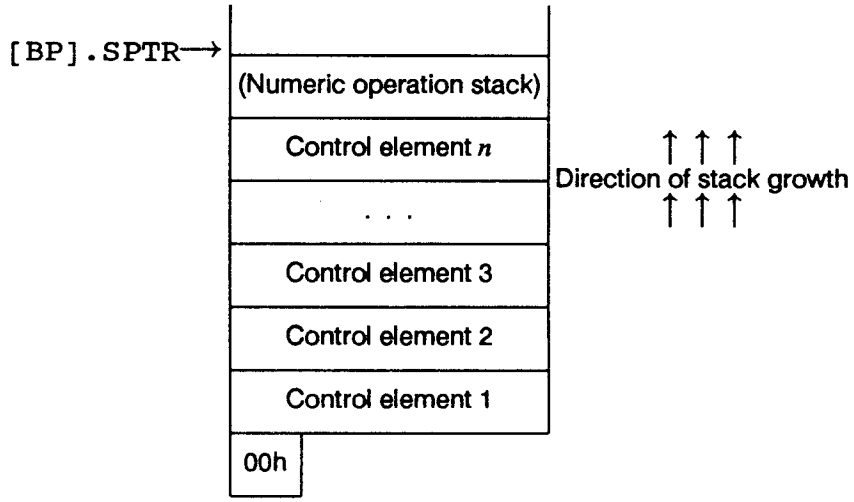


Figure 2-2. Control Stack Operation

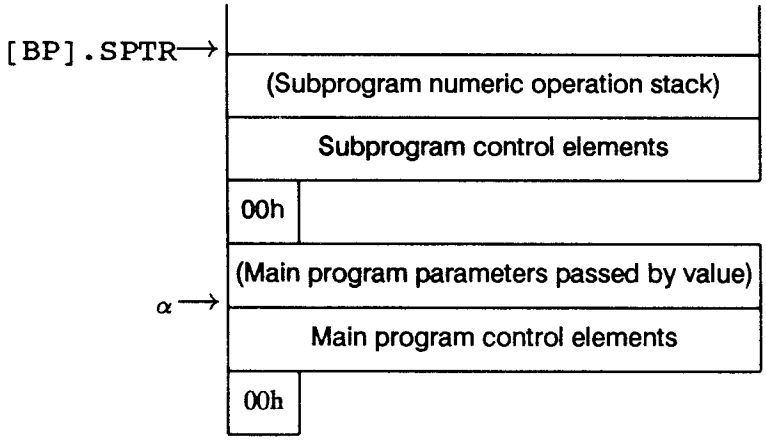


Figure 2-3. Control Stack During Subprogram Execution

Notes:

- α is the SPTR value saved in the control information save area.
- Control stack usage for an interrupt routine is the same as for a subprogram.
- Control elements consist of GOSUB return information and FOR . . . NEXT loop information.
- There is no pointer which separates the numeric operation stack from the control stack.

GOSUB Control Element

The GOSUB control element block size is 05h bytes.

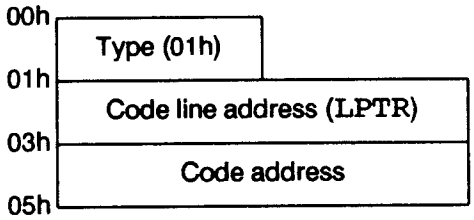


Figure 2-4. GOSUB Control Element

Code line address: The start of the code line containing the GOSUB statement.

Code address: The address of the eol or eos which follows the GOSUB statement.

FOR . . . NEXT Control Element

The FOR . . . NEXT control element block size is 18h bytes.

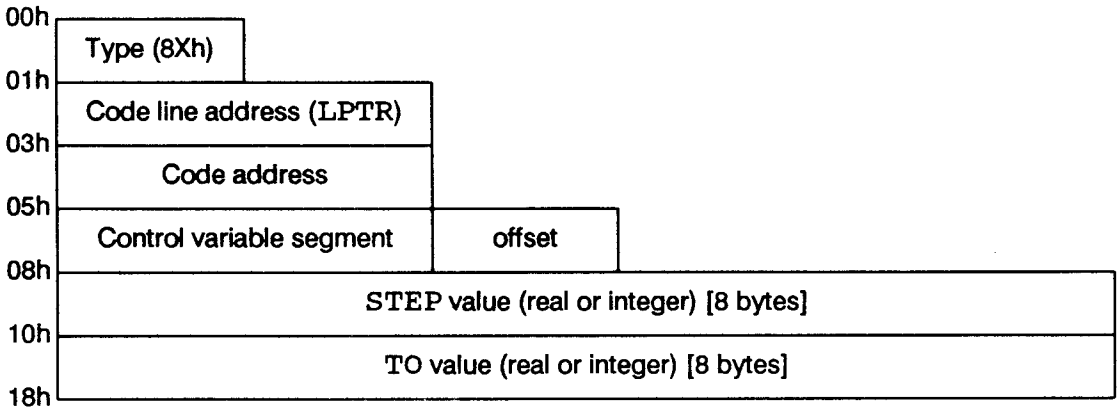


Figure 2-5. FOR . . . NEXT Control Element

Type (8Xh):	Indicates the control variable type (80h = real, 82h = integer).
Code line address:	The start of the code line containing the FOR statement.
Code address:	The address of the eol or eos which follows the FOR statement.
Control variable address:	The segment:offset address of the control variable for the FOR . . . NEXT loop. The offset is a single byte.
STEP value:	The value to be added to the control variable when the NEXT statement is executed. The type of the STEP value matches the type of the control variable (integer or real).
TO value:	The value to which the control variable is compared (after adding the STEP value) when the NEXT statement is executed. The type of the TO value matches the type of the control variable (integer or real).

The FOR . . . NEXT control element is removed from the control stack by the NEXT statement when the loop terminates. If the FOR loop is exited with a GOTO statement, the control element is left on the control stack. The FOR statement searches the control stack for FOR loop control elements before creating a new element. If there is a FOR loop control element with the same variable name, that control element is reused.

Numeric Operation Stack

Numeric parameters passed by value to subprograms are stored on the numeric operation stack (including any character values passed by value). The parameter table contains pointers to these values.

The SPTR and CPTR pointers are compared when pushing a value onto the stack. If SPTR ≤ CPTR, there is an overflow, and "Error MO" occurs.

Numeric values on the stack are always 8 bytes, whether real or integer type. An integer value on the stack starts with two bytes of 00h.

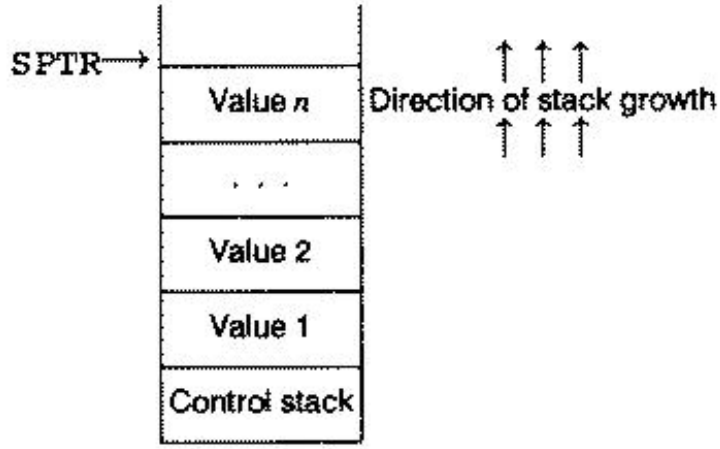


Figure 2-6. Numeric Operation Stack

Real Numeric Data

The format for real numeric data on the numeric operation stack is shown below.

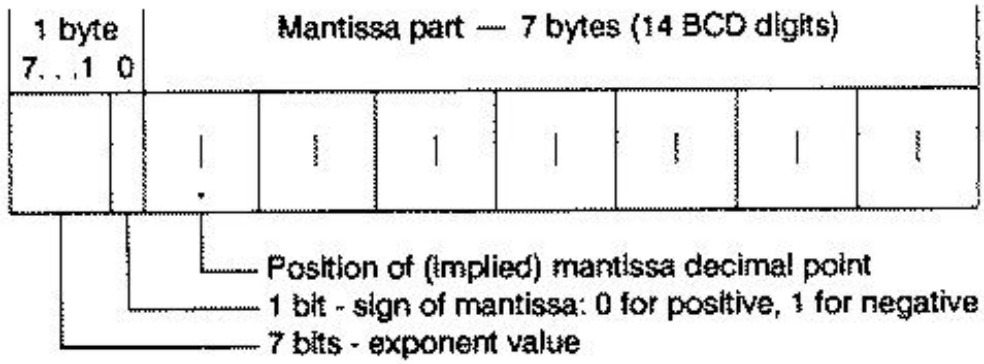


Figure 2-7. Real Numeric Data on the Numeric Operation Stack

The exponent is in two's complement (binary). Exponent values -64 through 63 indicate 10^{-64} through 10^{63}.

Integer Numeric Data

The range of an integer value is -32768 through 32767.

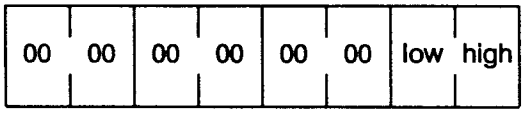


Figure 2-8. Integer Numeric Data on the Numeric Operation Stack

Numeric Operation Stack Example

A + B * C ⟶ D (S means SPTR)

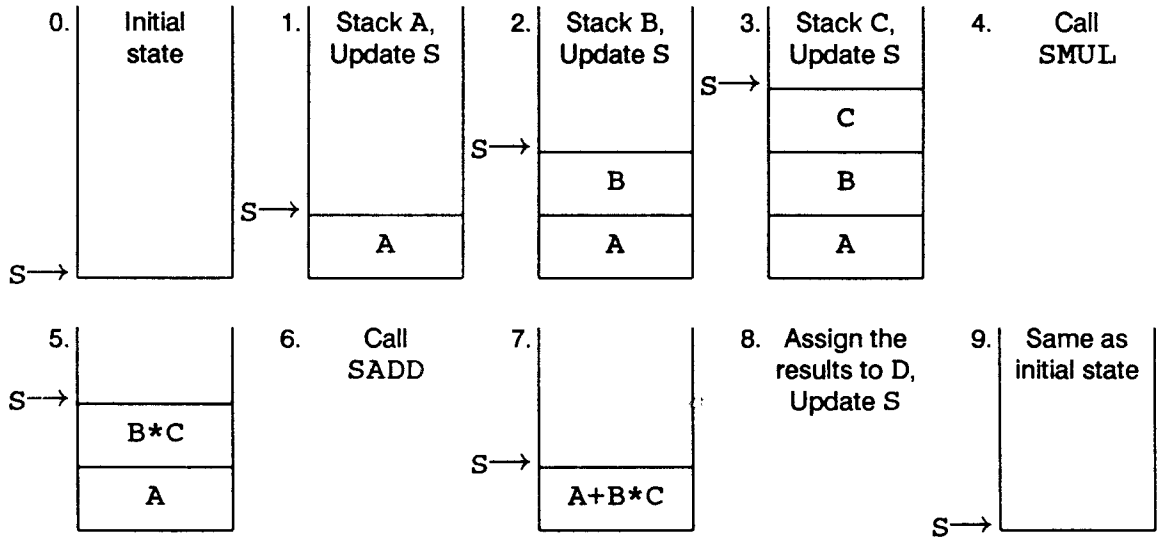


Figure 2-9. Numeric Operation Stack Example: A + B * C ⟶ D

Character Operation Stack

The character operation stack is used by character operators as a temporary storage area.

The SPTR and CPTR pointers are compared when pushing a value onto the stack. If CPTR ≥ SPTR, there is an overflow, and "Error MO" occurs.

A 00h byte must always be written at the byte pointed to by CPTR.

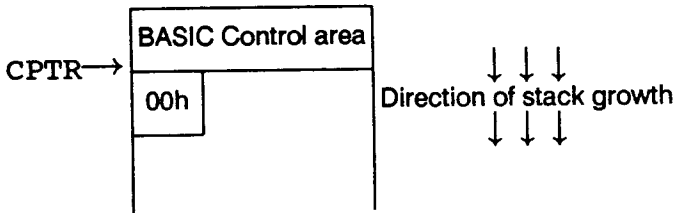


Figure 2-10. Character Operation Stack

Character Operation Stack Example

"ABC"+"DE"

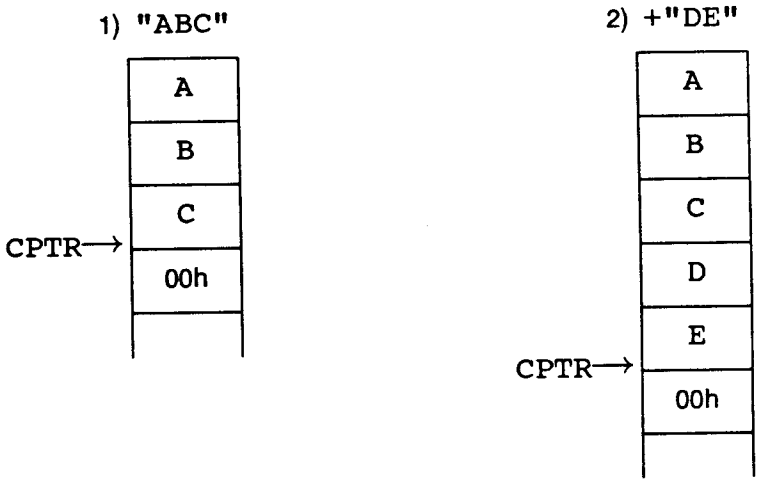


Figure 2-11. Character Operation Stack Example: "ABC" + "DE"

Parameter Table (only for %CALL)

The operation stack area is used by %CALL for the parameter table and for parameters passed by value.

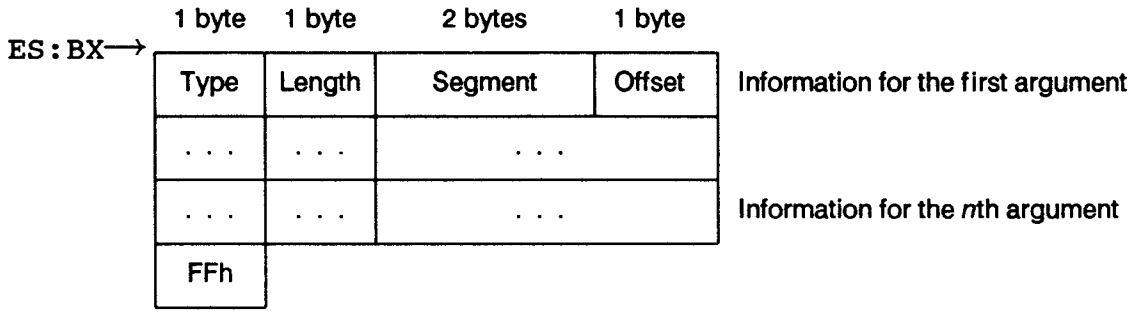


Figure 2-12. Parameter Table Format

The meanings of the fields in the parameter table are as follows:

- Type:

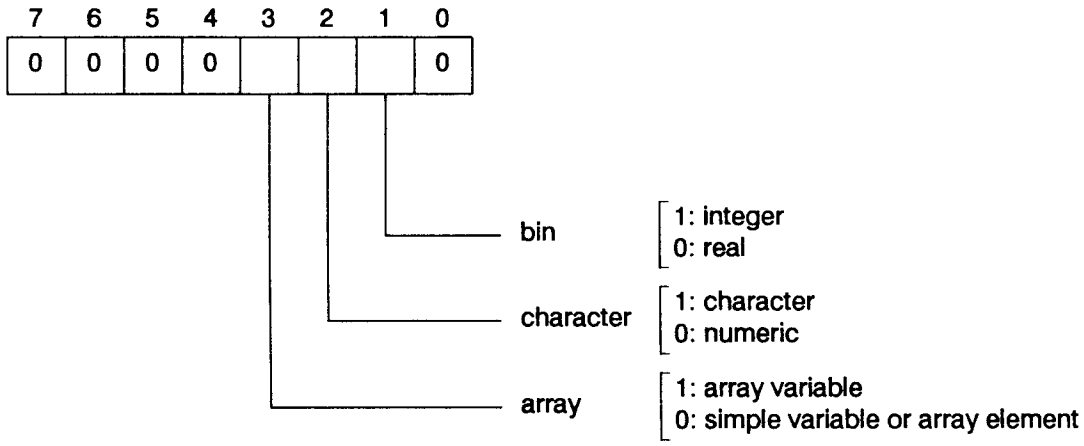


Figure 2-13. Parameter Table Type Byte

Arrays are passed to subprograms with subscript "*".

```
DIM XYZ(10)
  . . .
%CALL ABC(XYZ(*)) : REM pass the entire XYZ array
```

Numeric and string expressions (including constants) are evaluated by %CALL. Numeric values are put on the numeric operation stack as real numbers even if they could be expressed as an integer. String characters are moved from the character operation stack to the numeric operation stack before the subprogram is called.

- Length:

Type	Length in bytes
Integer	2
Real	8
Character	Dimensioned size (default is 8)
Array	Size of one array element

- Segment Address, Offset Address:

  The segment address and offset address contain the actual address of the variable's data area. This is different than in the variable descriptor table, where the address is relative to the start of the variable descriptor table.

  The segment address is a two-byte field; the offset address is a one-byte field with values 00h through 0Fh.

3

Assembly Language Subprograms (Keywords)

Contents

Chapter 3 Assembly Language Subprograms (Keywords)

3-1 Program Structure
3-2 BASIC Call and Return
3-2 BASIC Interpreter %CALL Procedure
3-3 Parameter Table Format
3-5 %CALL Example
3-6 Assembly Language Subprogram Return to BASIC
3-6 Access to BASIC Interpreter Utility Routines
3-8 Using a Utility from an Assembly Language Subprogram

3

Assembly Language Subprograms (Keywords)

An assembly language subprogram (also called a keyword) is called with the %CALL statement.

The following assembly language subprograms are built into the HP-94: SYAL, SYBP, SYEL, SYER, SYIN, SYLB, SYPO, SYPT, SYRS, SYRT, SYSW, and SYTO.

In addition, SYBD, SYBI, SYFT, and SYOS are reserved file names which must not be used for assembly language subprograms.

For assembly language subprograms which are not built into the HP-94, the file name is the subprogram name. In general, Hewlett-Packard uses SY as the first two characters of its assembly language files, and HN as the first two characters of its user-defined handlers. Names starting with SY and HN should not be used.

Assembly language subprograms must be written so that they can be executed in ROM.

This chapter assumes an understanding of HP-94 program structure. Refer to the "Program Execution" chapter in Part 1, "Operating System".

Program Structure

An assembly language program has a six-byte header followed by the program code. This structure is shown below with hex offsets indicated on the left side.

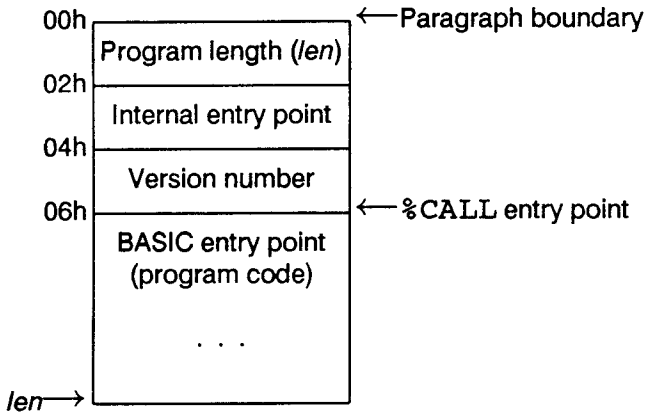


Figure 3-1. Assembly Language Subprogram Structure

See the "Program Execution" chapter in Part 1, "Operating System" for more information.

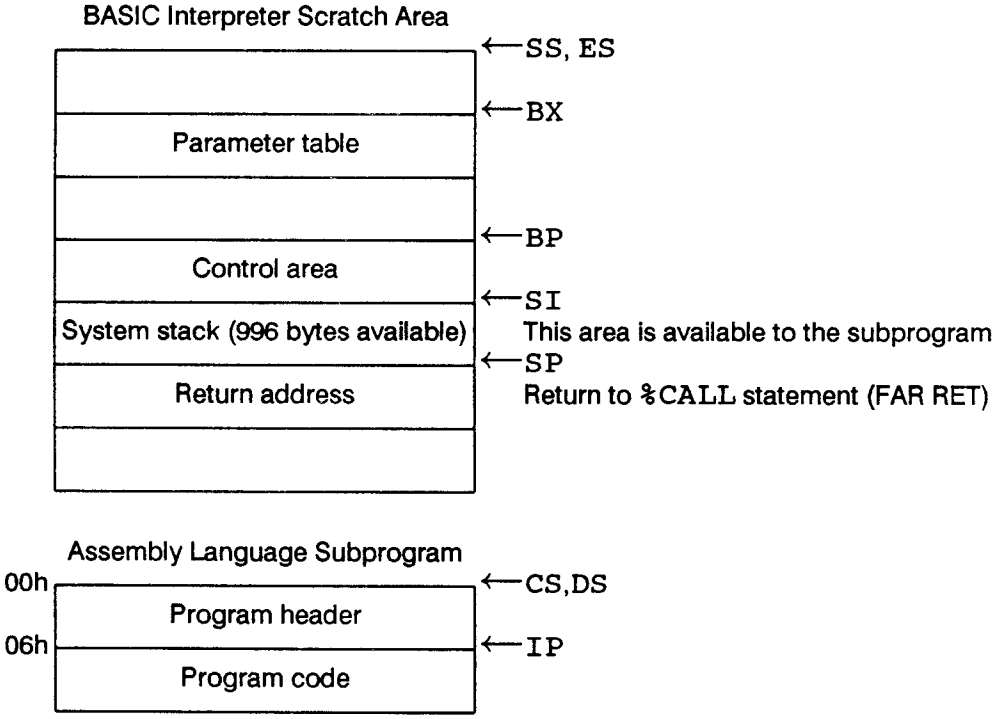
BASIC Call and Return

A BASIC program calls an assembly language program with the %CALL statement. When the assembly language routine finishes executing, a FAR RET is used to return to the BASIC interpreter.

BASIC Interpreter %CALL Procedure

The BASIC interpreter calls the assembly language subprogram at its entry point with a FAR CALL.

Contents of the CPU registers when an assembly language subprogram is called:



The direction flag is clear (CLD).

Interrupts are enabled (STI).

AX contains the value of SPTR before %CALL built the parameter table (not needed unless the subprogram uses IOERR; see IOERR for more information and an example).

The contents of registers which are not shown are not defined.

Parameter Table Format

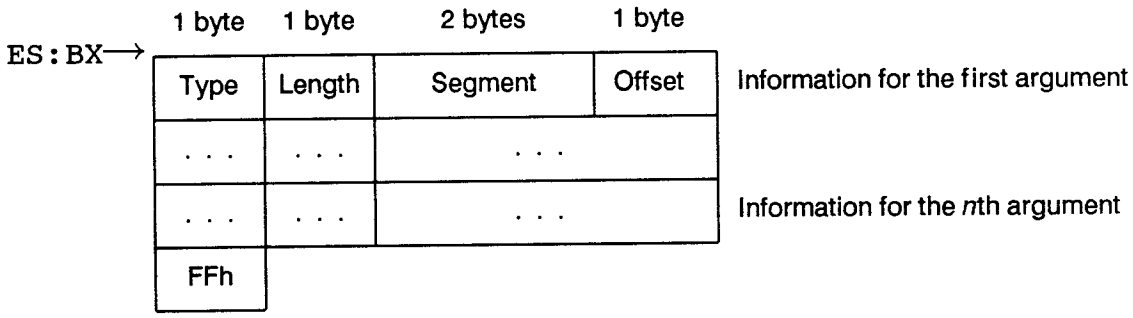


Figure 3-2. Parameter Table Format

The meanings of the fields in the parameter table are as follows:

- Type:

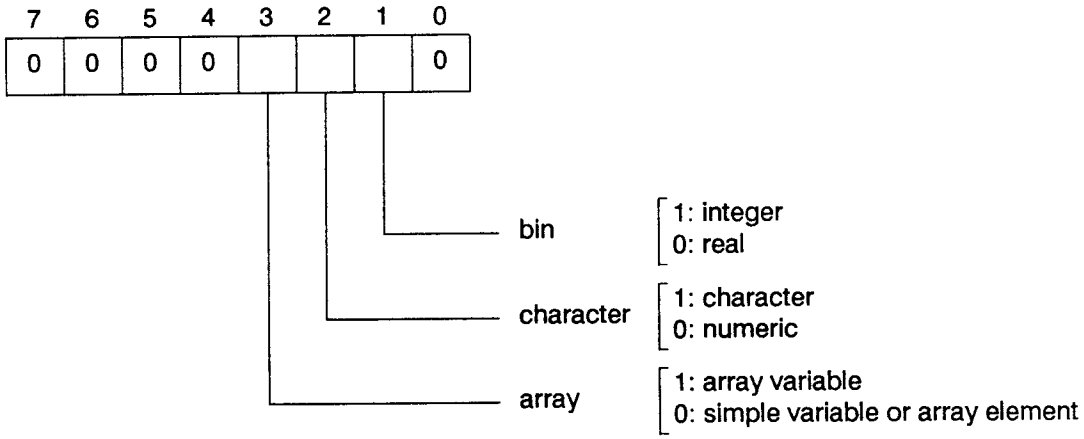


Figure 3-3. Parameter Table Type Byte

Arrays are passed to subprograms with subscript "*".

```
DIM XYZ(10)
  . . .
%CALL ABC(XYZ(*)) : REM pass the entire XYZ array
```

Numeric and string expressions (including constants) are evaluated by %CALL. Numeric values are put on the numeric operation stack as real numbers even if they could be expressed as an integer. String characters are moved from the character operation stack to the numeric operation stack before the subprogram is called.

- Length:

Type	Length in bytes
Integer	2
Real	8
Character	Dimensioned size (default is 8)
Array	Size of one array element

- Segment Address, Offset Address:

  The segment address and offset address contain the actual address of the variable's data area. This is different than in the variable descriptor table, where the address is relative to the start of the variable descriptor table.

  The segment address is a two-byte field; the offset address is a one-byte field with values 00h through 0Fh.

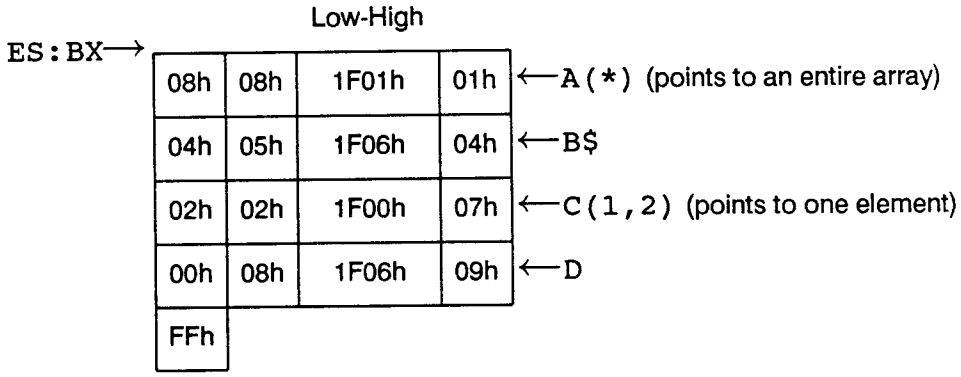
%CALL Example

```
 10 INTEGER C
 20 DIM A(10),B$5,C(3,2)
 30 D=1
    . . .
100 %CALL AB(A(*),B$,C(1,2),D)
```

When line 100 is executed, %CALL creates a parameter table (shown below) in the operation stack area and passes a pointer to it in ES : BX.

Assume that the BASIC variable area segment address is 1F00h.

Parameter Table



Variable Area

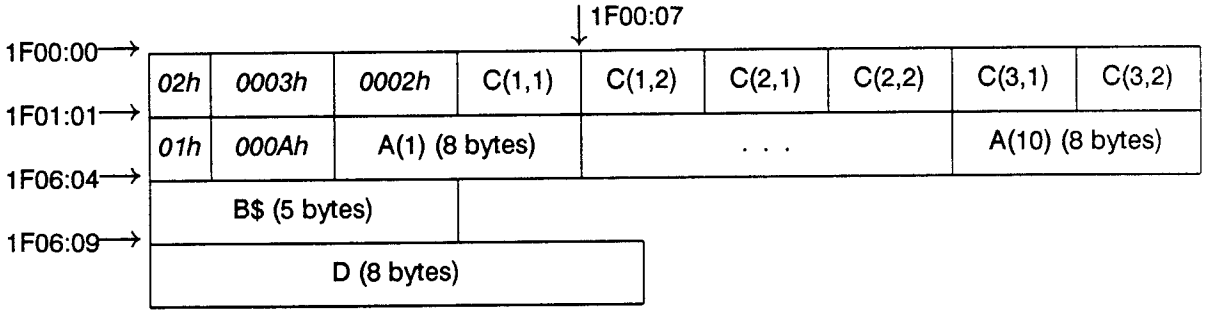


Figure 3-4. %CALL Example: Calling an Assembly Language Subprogram

Note: The values in *italics* are array information.

Assembly Language Subprogram Return to BASIC

When an assembly language subprogram returns to the BASIC program that called it, the following conditions should exist.

- The SS, BP, and SP registers must have the same value as when the assembly language subprogram was called.
- The direction flag must be clear (CLD instruction).
- Interrupts must be enabled (STI instruction).
- A FAR RET must be used to return to the BASIC Interpreter.

Access to BASIC Interpreter Utility Routines

This section describes how to access BASIC Interpreter utility routines for decimal math, stack manipulation, number conversion, and parameter processing from an assembly language program.

In the following table, CSEG is the segment address of the BASIC interpreter.

An assembly language subprogram can easily determine the value of CSEG by examining the return stack. The word at SS:SP+2 is the segment address of the BASIC interpreter.

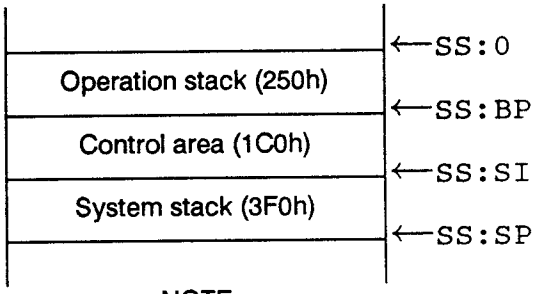
The regular entry point of the BASIC interpreter is CSEG:0. If the interpreter is called at CSEG:6 (as the operating system S command does), it immediately returns to the operating system.

If an error is detected by a BASIC interpreter utility routine, either the ERROR routine or the IOERR routine is called. The line number and the program name displayed in the error message point to the %CALL keyword.

BASIC Interpreter Code

CSEG:00h	JMP CSEG:51h (interpreter start)	
02h	Identifier "IP"	
04h	Release No.	Version No.
06h	JMP CSEG:44h (Exit to O.S.)	
08h	Data part size (paragraphs)	
0Ah	Operation stack size (bytes)	
0Ch	Control area size (bytes)	
0Eh	offset from SS:BP to SPTR	
10h	offset from SS:BP to CPTR	
12h	offset from SS:BP to SYSSTK	
14h	JMP SADD	(FAR RET)
18h	JMP SSUB	(FAR RET)
1Ch	JMP SMUL	(FAR RET)
20h	JMP SDIV	(FAR RET)
24h	JMP SPOW	(FAR RET)
28h	JMP SNEG	(FAR RET)
2Ch	JMP TOREAL	(FAR RET)
30h	JMP TOBIN	(FAR RET)
34h	JMP ERROR	(FAR RET)
38h	JMP IOERR	(FAR RET)
3Ch	JMP GETARG	(FAR RET)
40h	JMP SETARG	(FAR RET)
44h	EXIT (returns to the operating system)	
51h	BASIC Interpreter code	

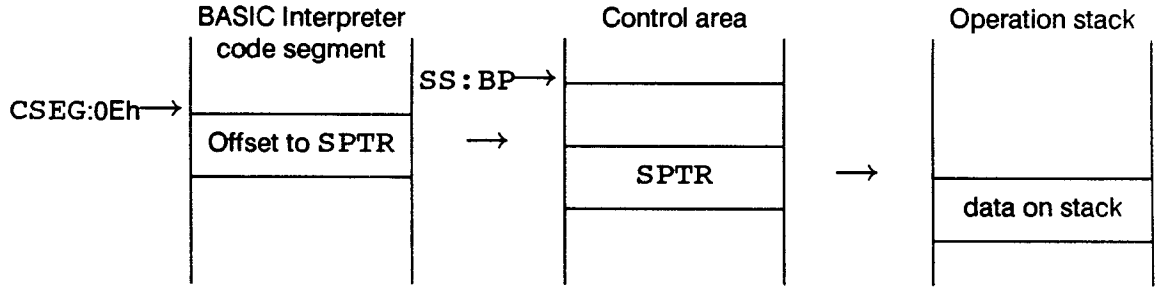
BASIC Interpreter Scratch Area



NOTE
The scratch area is allocated in main memory by the BASIC interpreter after a cold start.

Using a Utility from an Assembly Language Subprogram

Many of the utility routines require their data to be on the numeric operation stack. The numeric operation stack pointer (SPTR) must be set up to use these routines.



The SPTR address relative to BP is stored in the BASIC interpreter header at location CSEG:0Eh.

See the "Operation Stack" section for more information about using the numeric operation stack.

4

BASIC Interpreter Utility Routines

Contents

Chapter 4 BASIC Interpreter Utility Routines

4-1 BASIC Interpreter Utility Routine Descriptions
4-1 Registers Passed to BASIC Interpreter Utility Routines
4-2 ERROR
4-3 GETARG
4-5 IOERR
4-7 SADD
4-8 SDIV
4-9 SETARG
4-10 SMUL
4-11 SNEG
4-12 SPOW
4-13 SSUB
4-14 TOBIN
4-15 TOREAL

4

BASIC Interpreter Utility Routines

This chapter describes the BASIC interpreter utility routines. These utilities allow assembly language subprograms to use the decimal math routines in the BASIC interpreter and simplify the passing of parameters between BASIC programs and assembly language subprograms. Utility routines are also available for reporting errors detected in the assembly language subprograms and for converting between real and integer data.

BASIC Interpreter Utility Routine Descriptions

BASIC interpreter utility routine descriptions consist of the following:

- A brief description of the routine.
- The calling sequence for the routine.
- Notes on the use and behavior of the routine.
- A summary of the parameters passed to the routine and the parameters that the routine must return.

Registers Passed to BASIC Interpreter Utility Routines

The BASIC interpreter utility routines all expect BP to point to the BASIC interpreter control area. Other registers which are expected are mentioned in the "Input:" section for each routine.

ERROR

Display an error message and return to the operating system command mode.

Calling sequence:

```
FAR CALL CSEG:34h
```

Notes:

- If the code in AL is not in the table below, the code is displayed as three decimal digits.
- ERROR returns to command mode after displaying the message.

Input:

AL = code
(See table below)

Output:

```
Error CC nnnnn pppp
```
CC: Characters corresponding to the code
nnnnn: Error line number
pppp: Program name

Table 4-1. Codes for ERROR Utility Routine

Hex	Decimal	CC	Meaning
01h	1	SY	Syntax error
02h	2	TY	Data type mismatch
03h	3	CN	Conversion error
04h	4	RT	RETURN or SYRT error
05h	5	DT	Data error
06h	6	IL	Illegal argument
07h	7	BR	Branch destination error
08h	8	MO	Memory overflow
09h	9	NF	Program not found
0Ah	10	AR	Array subscript error
0Bh	11	CO	Conversion overflow
0Ch	12	EP	Missing END statement
0Dh	13	DO	Decimal overflow
0Eh	14	IR	Insufficient RAM
0Fh	15	FN	Illegal DEF FN statement
10h	16	UM	Unmatched number of arguments
11h	17	BM	BASIC interpreter malfunction
12h	18	LN	Nonexistent line
13h	19	IS	Illegal statement

GETARG

Convert a numeric parameter from %CALL into a binary value and return the value.

Calling sequence:

```
FAR CALL CSEG:3Ch
```

Notes:

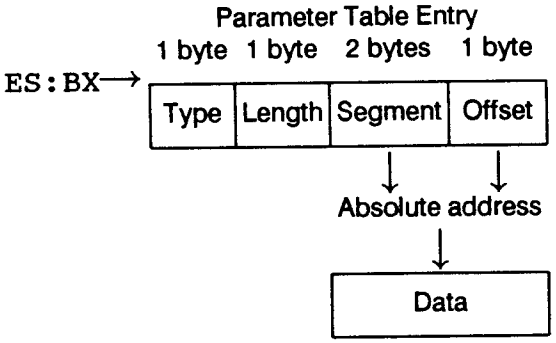
- If the parameter is an array or is of character type, `Error TY` occurs.
- If there is no parameter (type = FFh), `Error UM` occurs.
- If the parameter is negative and a negative number is not allowed, `Error IL` occurs.
- If the parameter is out of range, `Error IL` occurs. The valid range depends on the contents of register CL, as shown in this table:

Table 4-2. GETARG Result Flag (Register CL)

CL	Length	Positive/Negative	Range of values
0	word (16 bits)	positive or zero only	0 through 32767
1	double word (32 bits)	positive or zero only	0 through 2^{31}-1
2	word	negative allowed	-32768 through 32767
3	double word	negative allowed	-2^{31} through 2^{31}-1

Input:

ES:BX points to a parameter table entry
CL is the flag byte (see below)



Output:

If destination is a word:
AX = binary value
DX = (undefined)
If destination is double word:
AX = low word of binary value
DX = high word of binary value

Figure 4-1. GETARG Parameter Processing

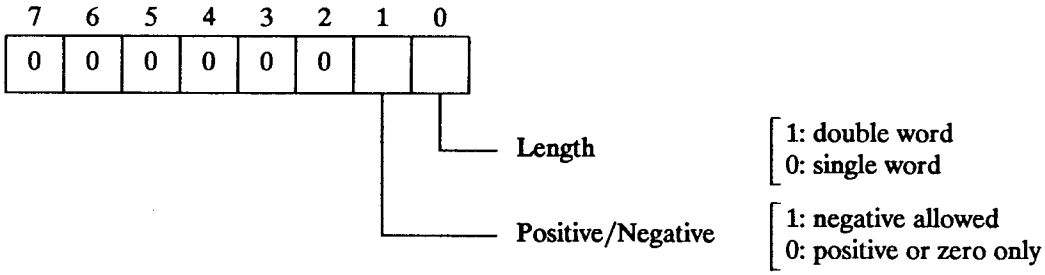


Figure 4-2. GETARG Result Flags (Register CL)

If error trapping (%CALL SYER) is not in effect, display an error message and return to the operating system command mode.

Calling sequence:

```
FAR CALL CSEG:38h
```

Notes:

- Assembly language subprograms must set up certain registers before calling IOERR. See the example program below to set up these registers.
- If error trapping (%CALL SYER) is in effect, the error number variable is set to the error code. BASIC execution resumes at the next line (not statement) of the BASIC program. IOERR does *not* return to the assembly language routine which called it.
- If error trapping (%CALL SYER) is not in effect, a call to IOERR has the same effect as a call to ERROR.

Input:

AL = code
(See Appendix B)
BP, SS, and SP unchanged from %CALL
SPTR restored (value was in AX after %CALL)

Output:

```
Error NNN nnnnn pppp
```

NNN: Error code (3 decimal digits)
nnnnn: Error line number
pppp: Program name

```
IOERR_OFFSET            equ     038h
SPTR_OFFSET             equ     0Eh

MYSEG                   segment public 'MYSEG'
                        assume cs:MYSEG
EXAMPLE                 proc    far
START:
PROG_SIZE               dw      FINISH-START
ASM_ENTRY_ADR           dw      offset START_ASM
VERSION                 dw      0100h           ; Version 1.00
;
; This is an outline of an assembly-language subprogram which shows how
; to save and restore the value of SPTR before a call to IOERR.
;
START_ASM:
                        push    ax              ; Save SPTR value on stack
;
;******* (user's code omitted here) *******
;
JMP_IOERR:                      ; AL contains the error code for IOERR
                        cld                     ; Needed only if code executed a STD
                        sti                     ; Needed only if code executed a CLI
                        pop     dx              ; Recall SPTR value to DX from stack
                        pop     cx              ; Drop BASIC interpreter offset
                        pop     es              ; Pop BASIC interpreter segment (CSEG)
                        push    es              ; Push CSEG for IOERR entry
                        mov     cx,IOERR_OFFSET
                        push    cx              ; Push offset for IOERR entry
                        mov     si,es:SPTR_OFFSET ; SI = offset of SPTR
```

```
                        mov     ss:[bp+si],dx   ; Restore SPTR
                        ret                     ; Jump to IOERR
;
NORMAL_RETURN:
                        pop     dx              ; Throw away (unused) SPTR value
                        ret                     ; Return to BASIC interpreter
;
EXAMPLE                 endp
FINISH:
MYSEG                   ends
                        end
```

SADD

Add two numbers on the operation stack.

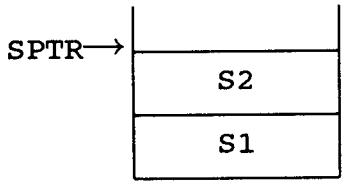
Calling sequence:

```
FAR CALL CSEG:14h
```

Notes:

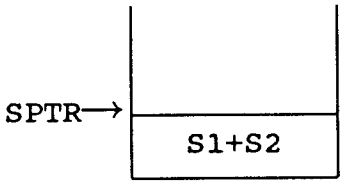
- The numbers can be either real numbers or integers. The result is an integer only if both numbers were integers, and the result fits in an integer.
- SADD does not use the operation stack as a scratch area.

Input:



S1, S2: numeric values

Output:



S1+S2: numeric value

SDIV

Divide two numbers on the operation stack.

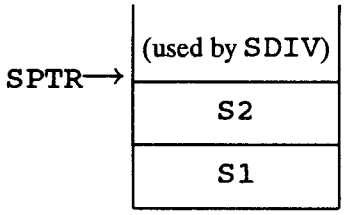
Calling sequence:

```
FAR CALL CSEG:20h
```

Notes:

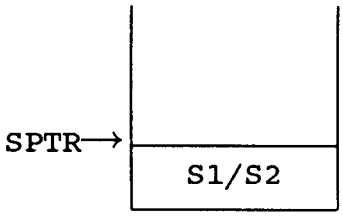
- The numbers can be either real numbers or integers. The result is always a real number.
- SDIV uses the operation stack as a scratch area.

Input:



S1, S2: numeric values

Output:



S1/S2: numeric real value

SETARG

SETARG converts a binary value into the type of a numeric parameter from %CALL (either real or integer) and stores the value into the parameter.

Calling sequence:

```
FAR CALL CSEG:40h
```

Notes:

- AX contains the binary value (-32768 through 32767).
- If there is no parameter (type = FFh), `Error UM` occurs.
- If the parameter is an array or is of character type, `Error TY` occurs.
- SETARG uses 8 bytes of the operation stack as a scratch area.

Input:

AX is a binary value
ES : BX points to a parameter table entry

Output:

Contents of AX placed in parameter.

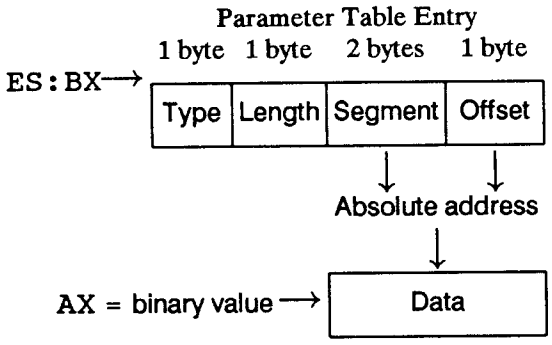


Figure 4-3. SETARG Parameter Processing

SMUL

Multiply two numbers on the operation stack.

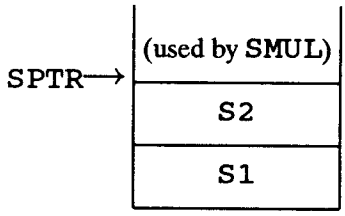
Calling sequence:

```
FAR CALL CSEG:1Ch
```

Notes:

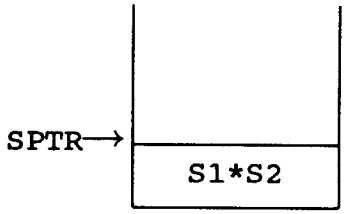
- The numbers can be either real numbers or integers. The result is an integer only if both numbers were integers, and the result fits in an integer.
- SMUL uses the operation stack as a scratch area.

Input:



S1, S2: numeric values

Output:



S1*S2: numeric value

SNEG

Change the sign of a number on the operation stack.

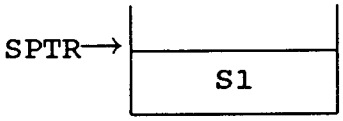
Calling sequence:

```
FAR CALL CSEG:28h
```

Notes:

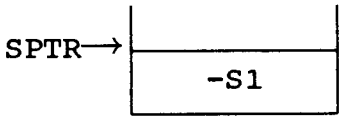
- The number can be either a real number or an integer. The result is an integer if the number was an integer.
- SNEG does not use the operation stack as a scratch area.

Input:



S1: numeric values

Output:



-S1: numeric value

SPOW

Exponential operation for two numbers on the operation stack.

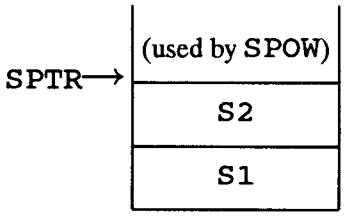
Calling sequence:

```
FAR CALL CSEG:24h
```

Notes:

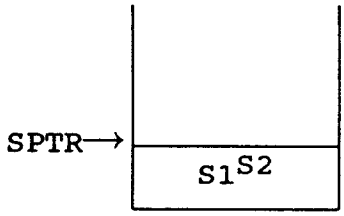
- The numbers can be either real numbers or integers. The result is always a real number.
- SPOW uses the operation stack as a scratch area.

Input:



S1, S2: numeric values

Output:



$S1^{S2}$: numeric real value

Subtract two numbers on the operation stack.

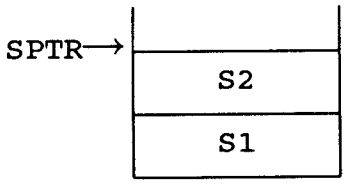
Calling sequence:

```
FAR CALL CSEG:18h
```

Notes:

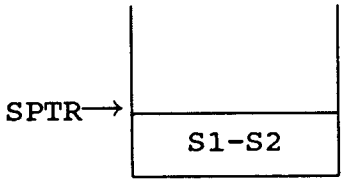
- The numbers can be either real numbers or integers. The result is an integer only if both numbers were integers, and the result fits in an integer.
- SSUB does not use the operation stack as a scratch area.

Input:



S1, S2: numeric values

Output:



S1-S2: numeric value

TOBIN

Convert a number at SS : BX to an integer.

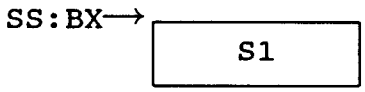
Calling sequence:

```
FAR CALL CSEG:30h
```

Notes:

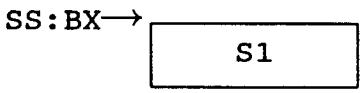
- The value is left unchanged if SS : BX points to an integer.
- The fractional part of the real number, if any, is truncated.
- An error occurs if the real number is not within the range -32768 through 32767.
- TOBIN does not use the operation stack as a scratch area.

Input:



S1: numeric real or integer data

Output:



S1: numeric integer data

TOREAL

Convert an integer or real number at SS:BX to a real number.

Calling sequence:

```
FAR CALL CSEG:2Ch
```

Notes:

- TOREAL does not use the operation stack as a scratch area.

Input:

SS:BX→ [S1]

S1: numeric real or integer data

Output:

SS:BX→ [S1]

S1: numeric real data

5

I/O Statement and Handlers

Contents

Chapter 5 **I/O Statement and Handlers**

5-1 Input Keywords (GET #, INPUT #, INPUT$)
5-4 Output Keywords (PRINT #, PRINT # . . . USING, PUT #)

5

I/O Statements and Handlers

The *BASIC Reference Manual* has tables associated with the BASIC I/O keywords (GET #, INPUT #, INPUT$, PRINT #, PRINT # ... USING, and PUT #) which describe the interaction between the keywords and the built-in handlers for channels 1 through 4. This chapter describes the interactions between these BASIC keywords and user-defined handlers for channels 1 through 4.

Input Keywords (GET #, INPUT #, INPUT$)

GET #, INPUT #, and INPUT$ all process incoming data in a different way.

- GET # reads data directly into the input variables.
- INPUT # reads data into a 256-byte internal buffer, then copies the data to the input variables.
- INPUT$ reads data and places it on the character stack. The data is then copied to the variable with the BASIC assignment operation.

The following table summarizes how each of the input keywords responds to conditions generated by user-defined handlers.

Table 5-1. Response of Input Keywords to Handler-Generated Errors

Condition	GET #	INPUT #	INPUT$
CR LF received.	N/A.	Characters received from the device (except the CR LF) are placed in the input variable. Input is aborted if no other characters were received.	N/A.
Character from terminate character string received.	N/A.	N/A.	Characters received from the device (including the terminate character) are placed in the input variable.
Short record detected (error 115).	Ends input for that variable.	The short read error generates a garbage byte, and the error is ignored (input operation for that variable not ended).	Ignored (input operation not ended).
Terminate character detected (error 116).	Ends input for that variable.	Ignored (input operation for that variable not ended).	Ignored (input operation not ended).
End of data (error 117).	Ends input for that variable. *	Characters read up to the EOD are placed in the input variable. Input is aborted if no characters were received before the EOD.	Ignored (input operation not ended). *
Timeout (error 118).	Input aborted.	Input aborted.	Input aborted.
Power switch pressed (error 119).	Input aborted.	Input aborted.	Input aborted.
Low battery (error 200).	Input aborted.	Input aborted.	Input aborted.
Errors 201-208.	Input aborted.	Input aborted.	Input aborted.

* The behavior of GET # and INPUT$ is altered if INPUT # has been used with the channel and the last INPUT # aborted input due to an EOD.

- GET # is not affected except that program execution continues on the next line of the program (not the next statement, if GET # is in a multistatement line).
- INPUT$ will abort input after reading one character. Program execution continues on the next line of the program (not the next statement, if INPUT$ is in a multistatement line).

When input is aborted, program execution continues on the next line of the program (not on the next statement, if the GET #, INPUT #, or INPUT$ statement is in a multistatement line).

"Input aborted" has different meanings for GET #, INPUT #, and INPUT$.

GET #: The input operation has been interrupted. When input is aborted, the input operation is ended, and any characters received up to that point are placed in the input variable. This may result in part of the previous value of the variable being overwritten. All subsequent variables in the input list are unchanged. This is in contrast to INPUT # and INPUT$, in which any received data for that variable is discarded.

When input is aborted because of a numeric error, the I/O length reported by SYIN is set to the number of bytes actually received up to that point, since that data has already been placed in the input variable.

INPUT #: No data has been received or the input operation has been interrupted. When input is aborted, the input operation is ended, and any characters received up to that point are discarded. The current input variable and all subsequent variables in the input list are left unchanged (note that variables prior to the one at which input was aborted will already have been changed). This is in contrast to GET #, in which any received data for that variable is saved.

When input is aborted because of a numeric error, the I/O length reported by SYIN is set to 0, since no data is placed in the input variable.

INPUT$: No data has been received or the input operation has been interrupted. When input is aborted, the input operation is ended, and any characters received up to that point are discarded. The input variable is left unchanged. This is in contrast to GET #, in which any received data is saved and the variables are set to 0 or the null string.

When input is aborted because of a numeric error, the I/O length reported by SYIN is set to 0, since no data is placed in the input variable.

Output Keywords (PRINT #, PRINT # . . . USING, PUT #)

This table summarizes how each of the output keywords (PRINT #, PRINT # ... USING, PUT #) responds to errors generated by user-defined handlers.

Table 5-2. Response of Output Keywords to Handler-Generated Errors

Condition	PRINT #	PRINT # . . . USING	PUT #
Timeout (error 118).	Output aborted.	Output aborted.	Output aborted.
Power switch pressed (error 119).	Output aborted.	Output aborted.	Output aborted.
Low battery (error 200).	Output aborted.	Output aborted.	Output aborted.
Errors 201-208.	Output aborted.	Output aborted.	Output aborted.
Lost connection while transmitting (error 218).	Output aborted.	Output aborted.	Output aborted.

"Output aborted" means that the output operation has been interrupted. When output is aborted, the output operation is ended. Subsequent variables in the output list are not output.

When output is aborted, program execution continues on the next line of the program (not on the next statement, if PRINT #, PRINT # ... USING, or PUT # is in a multistatement line).

When output is aborted because of a numeric error, the I/O length reported by SYIN is set to the number of bytes actually sent up to that point, since that data has already been written to the device.

Part 3

Hardware Specifications

Introduction to the Hardware Specifications

The purpose of this section of the *Technical Reference Manual* is to provide enough information for a developer to test a system configuration using the HP-94 and accessories in typical usage. There are four major topics:

- Electrical Specifications
  This provides voltage and current levels and specific integrated circuit (IC) information for some of the system ICs.
- Mechanical Specifications
  This includes HP-94 dimensions and information about connector types and pin assignments.
- Environmental Specifications
  This provides temperature, humidity, and other environmental information about the HP-94 operating environment.
- Accessory Specifications
  This discusses the principal accessories currently available for the HP-94.

In addition, for reference by developers, data sheets are provided for four of the ICs used in the machine.

Disclaimer

Hewlett-Packard makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Hewlett-Packard shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance, or use of this material.

The information contained in this document is subject to change without notice.

System Block Diagram

On the next page is a block diagram of the HP-94 hardware.

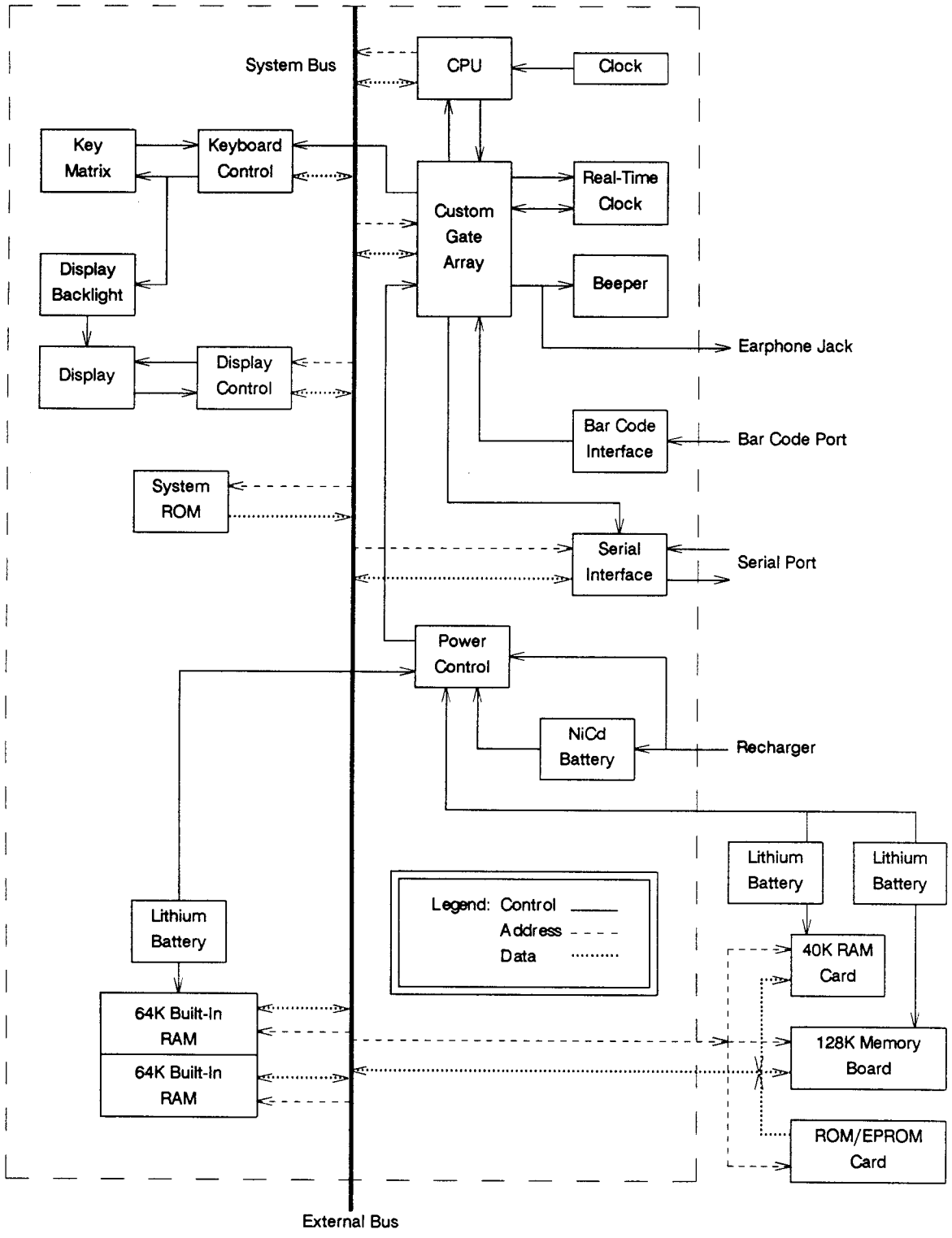


Figure 1. HP-94 Hardware Block Diagram

2 Introduction to the Hardware Specifications

1

Electrical Specifications

1

Electrical Specifications

This section provides the basic electrical specifications for the HP-94. Specific bus timing information is not provided. This information is available in the manufacturer's specifications for the individual components. The principal ICs used in the HP-94 are shown below.

Table 1-1. Principal Integrated Circuits

IC	Manufacturer	Part Number
Microprocessor *	NEC	μPD70108 (V20)
RAM	Toshiba	TC5565FL-15L
EPROM	Toshiba	TC57256D-20
LCD Column Driver *	Hitachi	HD61102A
LCD Row Driver	Hitachi	HD61103A
UART *	OKI	MSM82C51A
Real-Time Clock *	Epson	RTC-58321
Custom Gate Array	Hitachi	61L224
* Refer to the data sheets for these devices.		

Specific questions relating to the use of these ICs or their specifications should be directed to the IC manufacturer.

Table 1-2. Electrical Specifications

Parameter	Symbol	Min	Typical	Max	Units	Comments
Operating Voltage	V_{cc}	4.50	4.80	6.00	Vdc	Varies as batteries vary
Operating Current	I_{cc}		60 35	90 50	mA mA	Running Waiting for a key
Operating Frequency	F_{op}		3.6864		MHz	*
Display Backlight Current	I_{el}		20		mA	When backlight on
Standby Current	I_{sb}		30	100	μA	For HP-94F (256K RAM): T = 25 °C
Battery Pack Capacity			900		mAh	HP 82430A Rechargeable NiCd Battery Pack
Low Battery Detect Level — NiCd	V_{llNi}	4.55 4.70	4.60 4.75	4.65 4.80	Vdc Vdc	Discharging (voltage decreasing) Charging (voltage increasing)
Low Battery Detect Level — Lithium	V_{llLi}	2.65	2.70	2.75	Vdc	
Reset Detect Level	V_{rst}	4.35	4.40	4.45	Vdc	
V_{cc} Cutoff Level	V_{cu}	3.80		4.20	Vdc	
Serial Port Source Current	I_{rs}	250			mA	For $V_{rs} = V_{cc} - 0.1$ V
Serial Port Maximum Input Voltage	V_{inm}	-15		+15	Vdc	†
Serial Port Input Logic Levels	V_{ih} V_{il}	0.8 V_{cc}		 0.15 V_{cc}	Vdc Vdc	†
Serial Port Output Logic Levels	V_{oh1} V_{ol0}	V_{cc}-0.2 0		V_{cc} 0.2	Vdc Vdc	‡
Serial Port Input Current	I_{inm}			±10 ±3.5	μA mA	$V_{in} = V_{cc}$ or GND $V_{in} = \pm 15$ V
HP 82470A Level Converter Current	I_{lc}		25		mA	When active with a std. RS-232-C load at 9600 baud and $V_{cc} = 6.0$ V
Bar Code Port Source Current	I_{bcr}	175			mA	For $V_{bcr} = V_{cc} - 0.1$ V
Bar Code Port Input Logic Levels (V_{Obcr})	V_{ih} V_{il}	0.75 V_{cc} 0		V_{cc} 0.15 V_{cc}	Vdc Vdc	CMOS 40H004 inverting buffer
Bar Code Port Input Current	I_{inm}			+1 -6	μA mA	Input logic 1 § Input logic 0 §

* Voltage and timing specifications associated with the external bus and memory port are established by the CPU voltage and timing specifications. Please refer to the NEC μPD70108 data sheet.

† The HP-94 has a special input protection circuit using a 4.7 volt zener diode clamp that keeps the input voltage less than 4.7 volts and greater than -0.6 volts. Input signals between 0 and V_{cc} will not be modified.

‡ The output drivers are CMOS 40H004 inverting buffers that drive only between GND and V_{cc} voltage levels.

§ The bar code port input (V_{Obcr}) drives a 40H004 inverting buffer with a 1 kΩ pullup resistor.

2

Mechanical Specifications

Contents

Chapter 2 **Mechanical Specifications**

2-1 Physical Specifications
2-1 Serial Port Connector Specifications
2-2 Bar Code Port Connector Specifications
2-3 Memory Port Connector Specifications
2-5 External Bus Connector Specifications
2-7 Earphone Connector Specifications
2-7 Battery Pack Connector Specifications

2

Mechanical Specifications

This chapter describes mechanical specifications for the HP-94 and its connectors.

Physical Specifications

Below are the physical specifications for the HP-94.

Table 2-1. Physical Specifications

Parameter	Value	Units	Comments
Height	16.0	cm	6.3 in
Width	16.5	cm	6.5 in
Thickness	3.7	cm	1.4 in
Weight	686-745 *	g	1.5-1.6 lb

* The weight varies depending on the memory configuration. Minimum is for the HP-94D, and maximum is for the HP-94E with an HP 82412A ROM/EPROM Card containing three 27C256 EPROMs.

Serial Port Connector Specifications

The serial port connector is a 15-pin D-type female connector. The connector's attachment bolts use 4-40 × 1/4" slotted-head screws; note that it is not necessary to secure cables to the machine using these bolts. The following tables provide serial port connector pin assignments and information about mating connectors for the serial port.

Table 2-2. Serial Port Connector Pin Assignments

Pin Number	Symbol	Signal Name
Housing	FG	Frame Ground
1	NC	Not Connected
2	TxD	Transmitted Data
3	RxD	Received Data
4	RTS	Request To Send
5	CTS	Clear To Send
6	DSR	Data Set Ready
7	SG	Signal Ground
8	DCD	Data Carrier Detect
9	V_{rs}	Switched V_{cc}
10	V_{rch} *	Alternate Recharger Input
11	GND	Recharger Ground Return
12-14	NC	Not Connected
15	DTR	Data Terminal Ready

* The specifications for Vrch to allow charging of the NiCd battery pack using this pin are the same values as the output voltage and current specifications of the HP 82431A recharger.

Table 2-3. Serial Port Mating Connectors

Manufacturer	Part Number
TRW	DAM 15P
Amphenol	17-20150-1
JAE	DAC 15P
ITT Cannon	DA-15P

Bar Code Port Connector Specifications

The bar code port uses a 6-pin, 240° circular DIN connector. Either a 5-pin or 6-pin mating connector can be used on the bar code reader since pin 6 is not connected at the bar code port. The following tables provide bar code port connector pin assignments and information about mating connectors for the bar code port.

Table 2-4. Bar Code Port Connector Pin Assignments

Pin Number	Symbol	Signal Name
1	V_{bcr}	Switched V_{cc}
2	V_{Obcr}	Input From Barcode Reader
3	GND *	Signal Ground
4-6	NC	Not Connected

* The connector housing is attached to signal ground.

Table 2-5. Bar Code Port Mating Connectors

Manufacturer	Part Number
Switchcraft	12BL5M
Switchcraft	12BL6M
ITT Cannon	46005F
ITT Cannon	46006
TRW	014-00016-5
TRW	014-00024-1
SMK	DIN45322

Memory Port Connector Specifications

The memory port connector is inside the back cover of the HP-94, and is where the 40K RAM card, ROM/EPROM card, and 128K memory board connect to the machine. It is a Burndy PSE36C-2 36-pin PCB edge connector. The pin assignment is shown below.

Table 2-6. Memory Port Connector Pin Assignments

Pin	Symbol	Signal Name
1	GND	Ground
2	$\overline{RD}$	Read
3	$\overline{WR}$	Write
4	$\overline{CSV}$	System reset
5	A0	Address bit 0
6	A1	Address bit 1
7	A2	Address bit 2
8	A3	Address bit 3
9	A4	Address bit 4
10	A5	Address bit 5
11	A6	Address bit 6
12	A7	Address bit 7
13	A8	Address bit 8
14	A9	Address bit 9
15	A10	Address bit 10
16	A11	Address bit 11
17	A12	Address bit 12
18	A13	Address bit 13
19	A14	Address bit 14
20	A15	Address bit 15
21	$\overline{CSMC1}$ *	Memory card chip select 1
22	$\overline{CSMC2}$ †	Memory card chip select 2
23	AD0	Address/data bit 0
24	AD1	Address/data bit 1
25	AD2	Address/data bit 2
26	AD3	Address/data bit 3
27	AD4	Address/data bit 4
28	AD5	Address/data bit 5
29	AD6	Address/data bit 6
30	AD7	Address/data bit 7
31	$\overline{CSV}$	System reset
32	V_{ni}	NiCd battery voltage (output)
33	V_{mcs}	Lithium battery voltage (input)
34	V_{ad}	Recharger DC voltage (output)
35	V_{cc}	Supply voltage
36	GND	Ground

* $\overline{CSMC1}$ is active when A19, A18, and A16 = 0 and A17 = 1 ($\overline{CSMC1} = \overline{A19}\cdot\overline{A18}\cdot A17\cdot\overline{A16}$).

† $\overline{CSMC2}$ is active when A19 and A18 = 0, and A17 and A16 = 1 ($\overline{CSMC2} = \overline{A19}\cdot\overline{A18}\cdot A17\cdot A16$).

External Bus Connector Specifications

The external bus connector is located on the underside of the HP-94 behind a hard plastic port cover. The connector is a JAE PICL-40S-ST. Connection to the external bus connector can be made using a JAE PICL-40P-ST connector. The pin assignment is shown in the following table. Pin 1 is marked on the connector body. The odd-numbered pins are on the outer row (toward the outer edge of the HP-94 case), and the even-numbered pins are on the inner row.

Table 2-7. External Bus Connector Pin Assignments

Pin	Symbol	Signal Name
1	V_{ni}	NiCd battery voltage
2	V_{ni}	NiCd battery voltage
3	V_{cc}	Supply voltage
4	GND	Ground
5	NC	Not connected
6	NC	Not connected
7	NC	Not connected
8	DT/$\overline{R}$	Buffer read/write
9	DEBUG	Connected to ground
10	NC	No connection
11	IRQFK	Reserved interrupt request 2
12	IRQPR	Reserved interrupt request 1
13	IO/$\overline{M}$	IO/Memory
14	ALE	Address latch enable
15	CLK	CPU clock
16	AS16	Address/status bit 16
17	AS17	Address/status bit 17
18	AS18	Address/status bit 18
19	AS19	Address/status bit 19
20	RESET	System clocked reset
21	AD0	Address/data bit 0
22	AD1	Address/data bit 1
23	AD2	Address/data bit 2
24	AD3	Address/data bit 3
25	AD4	Address/data bit 4
26	AD5	Address/data bit 5
27	AD6	Address/data bit 6
28	AD7	Address/data bit 7
29	A15	Address bit 15
30	A14	Address bit 14
31	A13	Address bit 13
32	A12	Address bit 12
33	A11	Address bit 11
34	A10	Address bit 10
35	A9	Address bit 9
36	A8	Address bit 8
37	$\overline{WR}$	Write
38	$\overline{RD}$	Read
39	GND	Ground
40	GND	Ground

Earphone Connector Specifications

The earphone jack accepts a 3.5 mm miniature plug, with an overall length of less than 12 mm. A 0.125 inch diameter miniature plug will also fit, but will tend to have a lower insertion and removal force. Most standard earphones will connect properly to the HP-94 both mechanically and electrically. Some variation in audio output volume will occur between various earphone manufacturers.

Battery Pack Connector Specifications

The battery pack uses two AMP 42827-1 brass contacts. The HP-94 mates these contacts with custom nickel-plated brass pins that are 2.31 mm (0.091 in) nominal diameter, 6.3 mm (0.25 in) long, and 6.0 mm (0.24 in) center spacing.

3

Environmental Specifications

3

Environmental Specifications

Below are the environmental specifications for the HP-94.

Table 3-1. Environmental Specifications

Parameter	Min	Max	Units	Comments
Operating Temperature	0	55	°C	+32 to 131 °F
Storage Temperature	-40	65	°C	-40 to 149 °F
Operating Humidity	0	95	%RH	At 40 °C
Vibration	3.4 g rms, 5 to 500 Hz random vibration, 10 minutes per axis			
	Swept sine, 1 g, 5 to 500 Hz, 10 minutes dwell at resonance			
Shock	3 ms, 1/2 sine wave, 228 g, 6 axes			

4

Accessory Specifications

Contents

Chapter 4 Accessory Specifications

4-1 40K RAM Card Specifications
4-2 ROM/EPROM Card Specifications
4-3 Battery Pack Specifications
4-4 Guidelines for Using Rechargeable Batteries
4-4 Recharger Specifications
4-5 Level Converter Specifications
4-6 When to Use the Level Converter
4-7 Cables
4-7 Modem Cable
4-8 Printer Cable
4-8 Level Converter Cable
4-9 Vectra Cable
4-9 Vectra or IBM PC/AT to Level Converter Cable
4-10 IBM PC or PC/XT to Level Converter Cable
4-10 Bar Code Readers
4-10 Connecting the Serial Port to a Smart Wand

4

Accessory Specifications

The principal HP-94 hardware accessories (at the time of printing) are listed below. This accessory list does not include any software documentation, development tools, or utilities. For a complete list of all HP-94 accessories and support items, please refer to the current HP-94 price list, available at all HP sales offices. This chapter will describe only accessories listed in the table below.

Table 4-1. HP-94 Hardware Accessories

Model No.	Description
HP 82411A	40K RAM Card
HP 82412A	32K-128K ROM/EPROM Card
HP 82430A	Rechargeable NiCd Battery Pack
HP 82431A	U.S./Canada Recharger
HP 82431AB *	Europe Recharger
HP 82431AG *	Australia Recharger
HP 82431AU *	U.K. Recharger
HP 82470A	RS-232-C Level Converter
HP 82433A	HP-94 to Modem Cable
HP 82434A	HP-94 to Printer Cable
HP 82435A	HP-94 to Level Converter Cable
HP 82436A	HP-94 to Vectra Cable
HP 24542G	Vectra or IBM PC/AT to Level Converter Cable
HP 17255D	IBM PC or PC/XT to Level Converter Cable
HP 39961D	Smart Wand - Low Resolution
HP 39963D	Smart Wand - General Purpose
HP 39965D	Smart Wand - High Resolution

* The foreign versions of the recharger (Europe, Australia, and U.K.) are not available at the time this document was printed.

40K RAM Card Specifications

Pin assignments for the HP 82411A 40K RAM Card are described in the "Mechanical Specifications" chapter. The RAM card uses the same Toshiba RAM (TC5565FL-15L) as is used in the HP-94. A CR-2032 (or equivalent) lithium battery is required to provide battery backup for the RAM card.

ROM/EPROM Card Specifications

Pin assignments for the HP 82412A ROM/EPROM Card are described in the "Mechanical Specifications" chapter.

The ROM/EPROM Card has sockets for up to three 32 Kbyte (256 Kbit) ROMs or EPROMs, or up to two 64 Kbyte (512 Kbit) ROMs or EPROMs or their equivalents. There is a socketed jumper on the card that allows selection of the different sizes. The generic names for these ICs are 27C256 for the 32 Kbyte and 27C512 for the 64 Kbyte ICs. The CMOS version of the ROMs or EPROMs must be used. The NMOS versions require more current than is guaranteed by the HP-94. The EPROMs cannot be programmed while in the ROM/EPROM card, but must be programmed in an external EPROM programmer.

A list of the required specifications is provided below to assist in selecting the appropriate parts.

Table 4-2. ROM and EPROM Specifications

Parameter	Min	Max	Units	Comments
Operating Voltage	4.5	5.5	Vdc	6.0 V is recommended *
Operating Temperature	-10	+65	°C	+14 to 149 °F
Access Time		250	ns	All parts ≤ 250 ns
* Several manufacturers (including Intel) offer EPROMs with extended operating voltage range.				

The manufacturers that make correct size ROMs and EPROMs for use with the ROM/EPROM card and their part designations as of this printing are listed below. You should verify operating voltage, temperature, and speed with the manufacturer before making a final selection.

Table 4-3. ROM and EPROM Manufacturers

Manufacturer	32K IC		64K IC	
	EPROM	ROM	EPROM	ROM
Advanced Micro Devices	Am27C256	—	Am27C512	—
Fujitsu	MBM27C256	MB83256	MBM27C512	MB83512
Hitachi	HN27C256	HN613256P	—	—
Intel	27C256	—	27C512	—
Motorola	MCM67256	—	MCM67512	
National Semiconductor	NMC27C256	—	NMC27C512	—
NEC	μPD27C256	μPD23C256E	μPD27C512	μPD23C512
Texas Instruments	TMS27C256	TMS47C256	TMS27C512	TMS47C512
Toshiba	TC57256	TMM53257P	—	—

Battery Pack Specifications

The HP-94 uses the HP 82430A Rechargeable Battery Pack. When fully charged, the battery pack has approximately 900 milliamp-hours (mAh) of usable charge. The battery pack is charged whenever an HP 82431 recharger (HP 82431A/AB/AG/AU) is connected to the HP-94. Charging times and currents when charged using one of the HP 82431 rechargers are shown below.

Table 4-4. HP 82430A Rechargeable Battery Pack Specifications

Parameter	Symbol	Min	Typical	Max	Units	Comments
Capacity			900		mAh	
Charging Time	T_{ch}	6	10	14	hr	*
Charging Current	I_{ch}		150 †		mA	Pack attached to HP-94
Charging Current	I_{ch}			200 ‡	mA	Pack detached from HP-94

* The battery pack charging time is independent of HP-94 operating mode. Sufficient current is provided to operate the HP-94 and its principal accessories as well as fully charge the battery pack.

Charging times in excess of 18 hours are not recommended. Extended charging time may reduce the life of the battery pack. It is recommended that periodic "deep discharge - full recharge" cycles be performed to insure that maximum life and charge retention performance of the battery pack is maintained.

† The battery is connected to the recharger through a 2.7 Ω current-limiting resistor in series with a Schottky blocking diode. The actual charging current will vary as the battery pack voltage increases from the discharged state to the full charged state.

‡ Charging at currents greater than 200 mA for extended periods of time may damage the battery pack.

The battery pack contains four 2/3 C NiCd batteries completely enclosed in a detachable battery housing. All NiCd batteries are capable of extremely high short circuit currents. A thermal protector is built into the battery pack to prevent a constant short circuit condition. Since this circuit is temperature-sensitive, ambient conditions at or above its 75 °C temperature rating will cause a temporary open circuit in the battery pack. The HP-94 will then behave as if no battery pack is connected. When the short circuit or high temperature condition is removed, the battery pack short circuit protector will again close and the battery pack will continue normal operation.

WARNING **Never connect multiple battery packs in parallel while charging. Each individual pack should be blocked with a diode to prevent short circuit current from a failed cell from flowing into good cells of other packs.**

The battery pack connector specifications are described in the "Mechanical Specifications" chapter.

Guidelines for Using Rechargeable Batteries

The following is usage information and cautions about using rechargeable batteries.

CAUTION To avoid damage to the handheld computer, use only the batteries and recharger designated by Hewlett-Packard for the computer. Also, do not allow the batteries to discharge beyond their available capacity — recharge as soon as possible after the low battery indication appears. Allowing rechargeable batteries to discharge beyond their maximum limit can damage the batteries.

- Recharging batteries before they are low may eventually decrease their charging capacity.
- Do not overcharge the batteries by allowing them to recharge for longer than the recommended time. Shorter charging times will reduce the operating time before recharging is required, but will not harm the batteries.
- Do not leave the recharger permanently connected to the machine. Doing so decreases the useful life of the batteries.
- Do not use the recharger if it appears to have loose contacts, a cracked housing, or a damaged cord.
- Properly dispose of the batteries when they no longer adequately hold a charge or when they appear damaged.

WARNING **To prevent injury, keep all batteries out of the reach of children and properly dispose of exhausted batteries. Do not mutilate or puncture batteries, and do not dispose of them in fire. Exposure to excessive heat can cause release of toxic fumes or explosion.**

Recharger Specifications

The HP 82431 Recharger (HP 82431A/AB/AG/AU) supplies charging current to the HP-94's NiCd battery pack. The recharger is designed to supply sufficient current to charge the batteries even while the HP-94 is operating.

Table 4-5. HP 82431 Recharger Specifications

Parameter	Symbol	Min	Typical	Max	Units	Comments
Input Voltage	V_{ac}	108	120	132	Vac	HP 82431A
		198	220	242	Vac	HP 82431AB
		216	240	264	Vac	HP 82431AG/AU
Input Current	I_{ac}			80	mA	HP 82431A
				40	mA	HP 82431AB/AG/AU
Input Frequency	I_{fr}	57.5	60	62.5	Hz	HP 82431A
		47.5	50	55	Hz	HP 82431AB/AG/AU
Output Voltage	V_{rch}	6.2		6.7	Vdc	
Output Current	I_{rch}			400	mA	*
* Refer to "Battery Pack Specifications" for details about the charging current actually supplied to the battery pack.						

Level Converter Specifications

The HP-94 serial port outputs CMOS logic levels (refer to "Electrical Specifications"). Some RS-232 devices require that proper RS-232 voltage levels be provided for their serial interfaces to operate properly. These devices require the use of the HP 82470A RS-232-C Level Converter.

The level converter modifies the 0 to V_{cc} voltage level outputs from the HP-94 serial port to ±9 V EIA RS-232-C voltage levels. Additionally, the level converter's 25-pin connector inputs and outputs meet all RS-232 timing and load specifications. RS-232 voltage level inputs to the level converter's 25-pin connector are internally shifted to the 0 to V_{cc} range the HP-94 expects, and then are output to the HP-94 using the 15-pin connector.

When the serial port is disabled, the control lines are turned off (set to 0 volts). This is different than most AC-powered serial devices, in which the control lines are high (-3 volts or less) because the serial port is powered whenever the device is on.

Connection is made between the HP-94 and the level converter using an HP 82435A 1/4 meter cable. The level converter is powered by the HP-94 using pin 9 of the serial port (V_{rs}). The output voltage V_{rs} is activated under program control when the serial port is enabled (refer to the "Serial Port" chapter in the operating system section of this manual). Typical power consumption by the level converter is 25 mA when active with a standard RS-232-C load at 9600 baud and V_{cc} = 6.0 volts ($V_{rs}=V_{cc}$-0.1 V).

Below are the pin assignments for both the 15- and 25-pin connectors on the level converter.

Table 4-6. HP 82470A RS-232-C Level Converter Pin Assignments

25-Pin Female Connector			15-Pin Female Connector		
Signal Name	**Symbol**	**Pin No.**	**Pin No.**	**Symbol**	**HP-94 Signal Name**
Frame Ground	FG	Housing	Housing	FG	Frame Ground
Transmitted Data	TxD	2	2	TxD	Transmitted Data
Received Data	RxD	3	3	RxD	Received Data
Request To Send	RTS	4	4	RTS	Request To Send
Clear To Send	CTS	5	5	CTS	Clear To Send
Data Set Ready	DSR	6	6	DSR	Data Set Ready
Signal Ground	SG	7	7	SG	Signal Ground
Data Carrier Detect	DCD	8	8	DCD	Data Carrier Detect
Not Connected	NC	9	9	V_{rs}	Switched V_{cc} *
Data Terminal Ready	DTR	20	15	DTR	Data Terminal Ready

* HP 82470A level converter power.

When to Use the Level Converter

RS-232-C specifications require that input signal levels at the input of a device be greater than +3 or less than -3 volts. RS-232 output voltages experience a greater voltage swing to prevent signal degradation and line noise from interfering with communication signals. However, many available line receivers do not actually require voltage swings of these levels. The HP-94 system can take advantage of this by not requiring that the level converter be used when communicating with these devices.

The HP-94 will switch its RS-232 outputs between CMOS logic levels, where V_{cc} will be between 4.5 and 6.0 volts. This provides logic low levels of less than 0.2 volts and logic high levels of greater than V_{cc}-0.2 volts. Thus, any line receiver that will respond with high-to-low and low-to-high transitions in this range of logic 0/1 values will not need to have true RS-232 levels at its inputs to properly detect the logic level.

The line receivers that can communicate directly with the HP-94 (that is, no level converter required) are listed below. Certain parts listed must be operated in the specified mode or configuration, so special attention must be paid to the comments.

Table 4-7. Line Receivers That Do Not Require Level Converter

Part Number	Manufacturer	Comments
1489	National Semiconductor Motorola *	Response (threshold) control must be open
75189	Texas Instruments	Response (threshold) control must be open
75154	Texas Instruments	Response (threshold) control must be open
MAX232	Maxim	
MC145406	Motorola	
74HC14	many manufacturers	

* 1489-compatible parts from other manufacturers will also work.

CAUTION When using the HP-94 system without a level converter, special care must be taken to ensure that the interconnection cables are sufficiently short to prevent signal degradation. It is recommended that all communications cable for use with the HP-94 that do not use the level converter be less than 3 meters in length.

Cables

There are several cables available to allow configuration of the HP-94 in a system. The connections for each of these cables are provided in the tables that follow in this section. Cable lengths are 1 meter unless specified otherwise.

Modem Cable

The HP 82433A cable is used to connect the HP-94 to modems that do not require a level converter. It is specifically designed for use with Hayes Smartmodems, but is usable with many other modems as well.

Table 4-8. HP-94 to Modem Cable

HP-94 15-Pin Male Connector			Modem 25-Pin Male Connector		
Signal Name	**Symbol**	**Pin No.**	**Pin No.**	**Symbol**	**Direction**
Frame Ground	FG	Housing	Housing	AA	N/A
Transmitted Data	TxD	2	2	BA	To Modem
Received Data	RxD	3	3	BB	From Modem
Request To Send	RTS	4	4	CA *	To Modem
Clear To Send	CTS	5	5	CB	From Modem
Data Set Ready	DSR	6	6	CC	From Modem
Signal Ground	SG	7	7	AB	N/A
Data Carrier Detect	DCD	8	8	CF	From Modem
Data Terminal Ready	DTR	15	20	CD	To Modem

* Hayes Smartmodems do not implement this line.

The HP-94 has receive all the necessary approvals for connecting to modems in the U.S. Some countries require that the product and its interface cable be approved prior to connecting to a modem. Contact your local Hewlett-Packard sales office to verify that the HP-94 is approved for your specific location.

Printer Cable

The HP 82434A cable is used to connect the HP-94 to RS-232-C printers that do not require a level converter. It is specifically designed for use with Hewlett-Packard ThinkJet printers (HP 2225D), but is usable with many other printers as well.

Table 4-9. HP-94 to Printer Cable

HP-94 15-Pin Male Connector			Printer 25-Pin Male Connector		
Signal Name	**Symbol**	**Pin No.**	**Pin No.**	**Symbol**	**Signal Name**
Frame Ground	FG	Housing	Housing	AA	Protective Ground
Transmitted Data	TxD	2	3	BB	Received Data
Received Data	RxD	3	2	BA	Transmitted Data
Request To Send	RTS	4	5	CB *	Clear To Send
Clear To Send	CTS	5	4	CA	Request To Send
Data Set Ready	DSR	6	20	CD	Data Terminal Ready
Signal Ground	SG	7	7	AB	Signal Ground

* Many printers (including the Hewlett-Packard ThinkJet) do not implement this line.

Level Converter Cable

When using the level converter, an HP 82435A 1/4-meter cable is required. This cable provides a straight-through connection between the HP-94 and the level converter.

Table 4-10. HP-94 to Level Converter Cable

HP-94 15-Pin Male Connector			Level Converter 15-Pin Female Connector		
Signal Name	**Symbol**	**Pin No.**	**Pin No.**	**Symbol**	**Signal Name**
Frame Ground	FG	Housing	Housing	FG	Frame Ground
Transmitted Data	TxD	2	2	TxD	Transmitted Data
Received Data	RxD	3	3	RxD	Received Data
Request To Send	RTS	4	4	RTS	Request To Send
Clear To Send	CTS	5	5	CTS	Clear To Send
Data Set Ready	DSR	6	6	DSR	Data Set Ready
Signal Ground	SG	7	7	SG	Signal Ground
Data Carrier Detect	DCD	8	8	DCD	Data Carrier Detect
Switched V_{cc}	V_{rs} *	9	9	V_{rs} *	Switched V_{cc}
Data Terminal Ready	DTR	15	15	DTR	Data Terminal Ready

* HP 82470A level converter power.

Vectra Cable

The HP 82436A 2-meter cable is used whenever direct communication between the HP-94 and a 9-pin serial port on an HP Vectra personal computer. Each of the two Vectra serial interfaces has one 9-pin port: the HP 24540A Serial/Parallel Interface, and the HP 24541A Dual Serial Interface. HP supplies no cables that connect the HP-94 directly to the 25-pin port on the Vectra Dual Serial Interface.

Table 4-11. HP-94 to Vectra Cable

Vectra 9-Pin Female Connector			HP-94 15-Pin Male Connector		
Signal Name	**Symbol**	**Pin No.**	**Pin No.**	**Symbol**	**Signal Name**
Protective Ground	AA	Housing	Housing	FG	Frame Ground
Received Data	BB	2	2	TxD	Transmitted Data
Transmitted Data	BA	3	3	RxD	Received Data
Data Terminal Ready	CD	4	5 *	CTS	Clear to Send
Data Terminal Ready	CD	4	6 *	DSR	Data Set Ready
Signal Ground	AB	5	7	SG	Signal Ground
Data Set Ready	CC	6 *	15	DTR	Data Terminal Ready
Clear to Send	CB	8 *	15	DTR	Data Terminal Ready

* Pins 6 and 8 are tied together on the 9-pin connector, and pins 5 and 6 are tied together on the 15-pin connector.

Vectra or IBM PC/AT to Level Converter Cable

The HP-94 can communicate directly with the HP Vectra computer through the HP 82436A cable, without using a level converter. For applications that require extended cable lengths or desire the level converter option, the HP 24542G Serial Printer/Plotter Cable can be used. When communicating with the IBM PC/AT, a level converter is required, and this cable must be used. The level converter is then connected to the HP-94 using the HP 82435A cable.

Table 4-12. Vectra or IBM PC/AT to Level Converter Cable

Vectra or IBM PC/AT 9-Pin Female Connector			Level Converter 25-Pin Male Connector		
Signal Name	**Symbol**	**Pin No.**	**Pin No.**	**Symbol**	**Signal Name**
Protective Ground	AA	Housing	Housing	FG	Frame Ground
Data Carrier Detect	CF	1	4	RTS	Request To Send
Received Data	BB	2	2	TxD	Transmitted Data
Transmitted Data	BA	3	3	RxD	Received Data
Data Terminal Ready	CD	4	5 *	CTS	Clear To Send
Data Terminal Ready	CD	4	6 *	DSR	Data Set Ready
Signal Ground	AB	5	7	SG	Signal Ground
Data Set Ready	CC	6 *	20	DTR	Data Terminal Ready
Request To Send	CA	7	8	DCD	Data Carrier Detect
Clear To Send	CB	8 *	20	DTR	Data Terminal Ready

* Pins 6 and 8 are tied together on the 9-pin connector, and pins 5 and 6 are tied together on the 25-pin connector.

IBM PC or PC/XT to Level Converter Cable

When using an IBM PC or PC/XT to communicate with the HP-94, a level converter is required. The HP 17255D cable connects the 25-pin IBM serial port connector to the 25-pin connector on the level converter. The level converter is then connected to the HP-94 using the HP 82435A cable.

Table 4-13. IBM PC or PC/XT to Level Converter Cable

IBM PC or PC/XT 25-Pin Female Connector			Level Converter 25-Pin Male Connector		
Signal Name	**Symbol**	**Pin No.**	**Pin No.**	**Symbol**	**Signal Name**
Frame Ground	FG	Housing *	Housing *	FG	Frame Ground
Transmitted Data	TxD	2	3	RxD	Received Data
Received Data	RxD	3	2	TxD	Transmitted Data
Clear To Send	CTS	5 †	20	DTR	Data Terminal Ready
Data Set Ready	DSR	6 †	20	DTR	Data Terminal Ready
Signal Ground	SG	7	7	SG	Signal Ground
Data Terminal Ready	DTR	20	5 †	CTS	Clear To Send
Data Terminal Ready	DTR	20	6 †	DSR	Data Set Ready

* Pin 1 is connected to frame ground (housing) on both connectors.

† Pins 5 and 6 are tied together on both connectors.

Bar Code Readers

The primary bar code readers for the HP-94 are the three HP Smart Wands: HP 39961D (low resolution), HP 39963D (general purpose), and HP 39965D (high resolution). Contact your sales office for complete literature and specifications for these wands.

Connecting the Serial Port to a Smart Wand

HP Smart Wands can be configured in one of two ways:

- By scanning bar code configuration menus (optical configuration)
- By sending configuration escape sequences to the Smart Wand

When a Smart Wand is connected to the bar code port, only the first approach is available because the bar code port is read-only. The second approach is available if the Smart Wand can be connected to the serial port. To support this use, Hewlett-Packard supplies a low-level bar code handler with the *HP-94 Software Development System* called HNSP that allows "smart" bar code scanning devices to be connected to the serial port.

HP Smart Wands are supplied with a 5-pin, 240° circular DIN connector, but at this printing are not available with a 15-pin D-type connector that would connect to the serial port. Below are the connections for a cable that will connect the serial port to a Smart Wand. This cable is not available from Hewlett-Packard — the connections are provided to allow a developer to make the cable.

Table 4-14. HP-94 Serial Port to Smart Wand Cable

HP-94 15-Pin Male Connector			Smart Wand 5-Pin or 6-Pin Female Connector		
Signal Name	**Symbol**	**Pin No.**	**Pin No.**	**Symbol**	**Signal Name**
Frame Ground	FG	Housing *	Housing *	FG	Frame Ground
Transmitted Data	TxD	2	4	RxD	Received Data
Received Data	RxD	3	2	TxD	Transmitted Data
Request To Send	RTS	4 †	—	NC	Not Connected
Clear To Send	CTS	5 †	—	NC	Not Connected
Signal Ground	SG	7	3	SG	Signal Ground
Switched V_{cc}	V_{rs} ‡	9	1	V_{rs} ‡	Switched V_{cc}

* The shield or braid must be connected to frame ground (housing).

† Pins 4 and 5 are tied together on the 15-pin connector.

‡ HP Smart Wand power.

5

Data Sheets

5

Data Sheets

This chapter contains copies of manufacturer's data sheets for the following four ICs:

- NEC μPD70108 (V20) Microprocessor
- OKI MSM82C51A Universal Asynchronous Receiver Transmitter (UART)
- Hitachi HD61102A LCD Column Driver
- Epson RTC-58321 Real-Time Clock

These data sheets provide reference information for developers whose application interacts directly with the IC, independent of the HP-94 operating system. Refer to the appropriate chapters in the "Operating System" for information about how these ICs are used in the HP-94, what I/O control registers are associated with each IC, and what built-in software is available already to control them.

NEC μPD70108 (V20) Microprocessor Data Sheet

NEC

NEC Electronics Inc.

μPD70108 (V20) HIGH-PERFORMANCE 16-BIT MICROPROCESSOR

Revision 3

November 1985

Description

The μPD70108 (V20) is a CMOS 16-bit microprocessor with internal 16-bit architecture and an 8-bit external data bus. The μPD70108 instruction set is a superset of the μPD8086/8088; however, mnemonics and execution times are different. The μPD70108 additionally has a powerful instruction set including bit processing, packed BCD operations, and high-speed multiplication/division operations. The μPD70108 can also execute the entire 8080 instruction set and comes with a standby mode that significantly reduces power consumption. It is software-compatible with the μPD70116 16-bit microprocessor.

Features

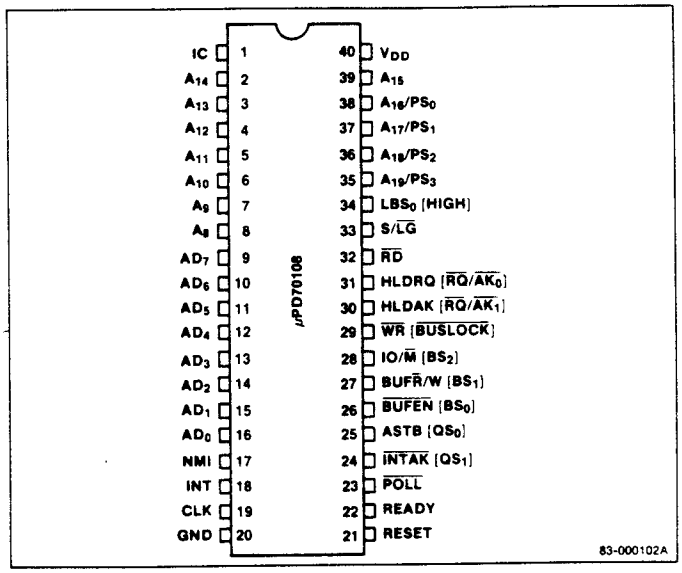
- ☐ Minimum instruction execution time: 250 ns (at 8 MHz)
- ☐ Maximum addressable memory: 1 Mbyte
- ☐ Abundant memory addressing modes
- ☐ 14 x 16-bit register set
- ☐ 101 instructions
- ☐ Instruction set is a superset of μPD8086/8088 instruction set
- ☐ Bit, byte, word, and block operations
- ☐ Bit field operation instructions
- ☐ Packed BCD instructions
- ☐ Multiplication/division instruction execution time: 4 μs to 6 μs (at 8 MHz)
- ☐ High-speed block transfer instructions: 1 Mbyte/s (at 8 MHz)
- ☐ High-speed calculation of effective addresses: 2 clock cycles in any addressing mode
- ☐ Maskable (INT) and nonmaskable (NMI) interrupt inputs
- ☐ IEEE-796 bus compatible interface
- ☐ 8080 emulation mode
- ☐ CMOS technology
- ☐ Low-power consumption
- ☐ Low-power standby mode
- ☐ Single power supply
- ☐ 5 MHz, 8 MHz or 10 MHz clock

Ordering Information

Part Number	Package Type	Max Frequency of Operation
μPD70108C-5	40-pin plastic DIP	5 MHz
μPD70108C-8	40-pin plastic DIP	8 MHz
μPD70108D-5	40-pin ceramic DIP	5 MHz
μPD70108D-8	40-pin ceramic DIP	8 MHz
μPD70108D-10	40-pin ceramic DIP	10 MHz
μPD70108G-5	52-pin flat pack	5 MHz
μPD70108G-8	52-pin flat pack	8 MHz
μPD70108L-5	44-pin PLCC	5 MHz
μPD70108L-8	44-pin PLCC	8 MHz

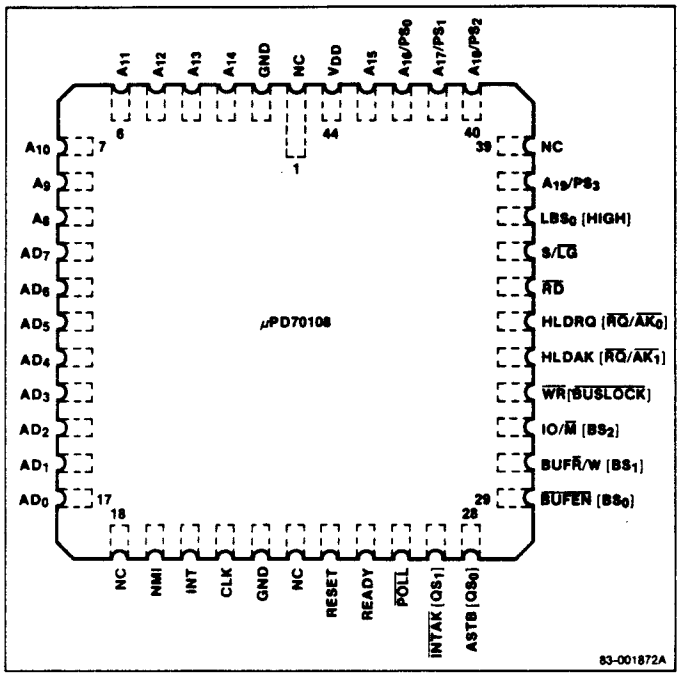
Pin Configurations

40-Pin Plastic DIP/Cerdip

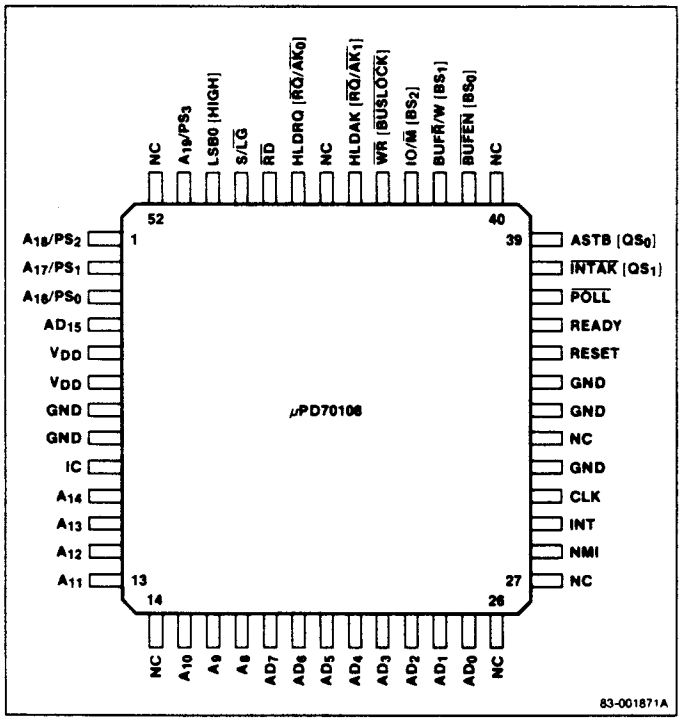


Pin Configurations (cont)

44-Pin Plastic Leadless Chip Carrier (PLCC)



52-Pin Plastic Miniflat



Pin Identification

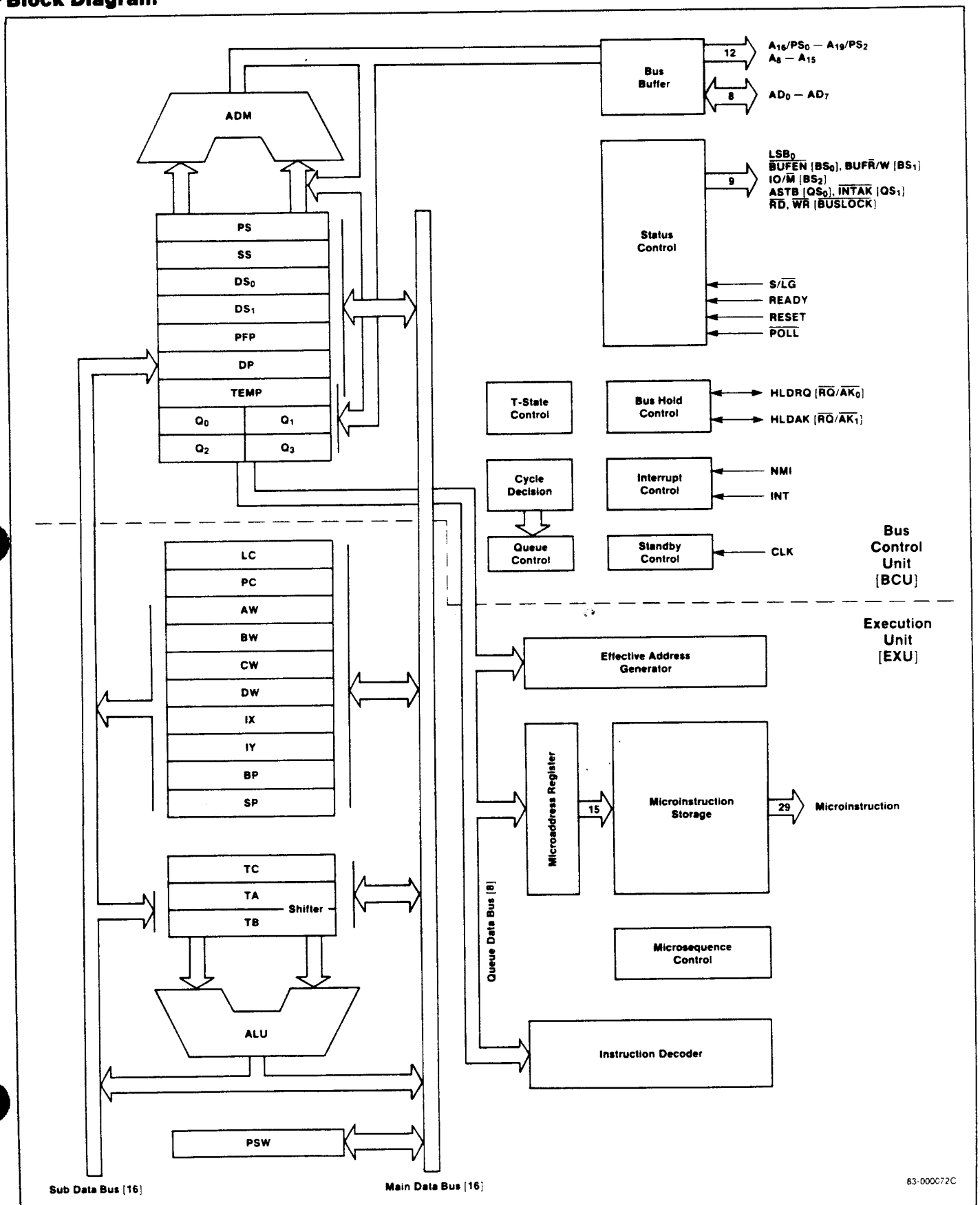
No.	Symbol	Direction	Function
1	IC*		Internally connected
2 - 8	A_{14} - A_8	Out	Address bus, middle bits
9 - 16	AD_7 - AD_0	In/Out	Address/data bus
17	NMI	In	Nonmaskable interrupt input
18	INT	In	Maskable interrupt input
19	CLK	In	Clock input
20	GND		Ground potential
21	RESET	In	Reset input
22	READY	In	Ready input
23	$\overline{POLL}$	In	Poll input
24	$\overline{INTAK}$ (QS_1)	Out	Interrupt acknowledge output (queue status bit 1 output)
25	ASTB (QS_0)	Out	Address strobe output (queue status bit 0 output)
26	$\overline{BUFEN}$ (BS_0)	Out	Buffer enable output (bus status bit 0 output)
27	BUF$\overline{R}$/W (BS_1)	Out	Buffer read/write output (bus status bit 1 output)
28	IO/$\overline{M}$ (BS_2)	Out	Access is I/O or memory (bus status bit 2 output)
29	$\overline{WR}$ ($\overline{BUSLOCK}$)	Out	Write strobe output (bus lock output)
30	HLDAK ($\overline{RQ}$/$\overline{AK_1}$)	Out (In/Out)	Hold acknowledge output, (bus hold request input/ acknowledge output 1)
31	HLDRQ ($\overline{RQ}$/$\overline{AK_0}$)	In (In/Out)	Hold request input (bus hold request input/ acknowledge output 0)
32	$\overline{RD}$	Out	Read strobe output
33	S/$\overline{LG}$	In	Small-scale/large-scale system input
34	LBS_0 (HIGH)	Out	Latched bus status output 0 (always high in large-scale systems)
35 - 38	A_{19}/PS_3 - A_{16}/PS_0	Out	Address bus, high bits or processor status output
39	A_{15}	Out	Address bus, bit 15
40	V_{DD}		Power supply

Notes: * IC should be connected to ground.

Where pins have different functions in small- and large-scale systems, the large-scale system pin symbol and function are in parentheses.

Unused input pins should be tied to ground or V_{DD} to minimize power dissipation and prevent the flow of potentially harmful currents.

Block Diagram



Pin Functions

Some pins of the μPD70108 have different functions according to whether the microprocessor is used in a small- or large-scale system. Other pins function the same way in either type of system.

A_{15} - A_8 [Address Bus]
For small- and large-scale systems.

The CPU uses these pins to output the middle 8 bits of the 20-bit address data. They are three-state outputs and become high impedance during hold acknowledge.

AD_7 - AD_0 [Address/Data Bus]
For small- and large-scale systems.

The CPU uses these pins as the time-multiplexed address and data bus. When high, an AD bit is a one; when low, an AD bit is a zero. This bus contains the lower 8 bits of the 20-bit address during T1 of the bus cycle and is used as an 8-bit data bus during T2, T3, and T4 of the bus cycle.

Sixteen-bit data I/O is performed in two steps. The low byte is sent first, followed by the high byte. The address/data bus is a three-state bus and can be at a high or low level during standby mode. The bus will be high impedance during hold and interrupt acknowledge.

NMI [Nonmaskable Interrupt]
For small- and large-scale systems.

This pin is used to input nonmaskable interrupt requests. NMI cannot be masked by software. This input is positive edge triggered and must be held high for five clocks to guarantee recognition. Actual interrupt processing begins, however, after completion of the instruction in progress.

The contents of interrupt vector 2 determine the starting address for the interrupt-servicing routine. Note that a hold request will be accepted even during NMI acknowledge.

This interrupt will cause the μPD70108 to exit the standby mode.

INT [Maskable Interrupt]
For small- and large-scale systems.

This pin is an interrupt request that can be masked by software.

INT is active high level and is sensed during the last clock of the instruction. The interrupt will be accepted if the interrupt enable flag IE is set. The CPU outputs the INTAK signal to inform external devices that the interrupt request has been granted. INT must be asserted until the interrupt acknowledge is returned.

If NMI and INT interrupts occur at the same time, NMI has higher priority than INT and INT cannot be accepted. A hold request will be accepted during INT acknowledge.

This interrupt causes the μPD70108 to exit the standby mode.

CLK [Clock]
For small- and large-scale systems.

This pin is used for external clock input.

RESET [Reset]
For small- and large-scale systems.

This pin is used for the CPU reset signal. It is an active high level. Input of this signal has priority over all other operations. After the reset signal input returns to a low level, the CPU begins execution of the program starting at address FFFF0H.

In addition to causing normal CPU start, RESET input will cause the μPD70108 to exit the standby mode.

READY [Ready]
For small- and large-scale systems.

When the memory or I/O device being accessed cannot complete data read or write within the CPU basic access time, it can generate a CPU wait state (Tw) by setting this signal to inactive (low level) and requesting a read/write cycle delay.

If the READY signal is active (high level) during either the T3 or Tw state, the CPU will not generate a wait state.

$\overline{\text{POLL}}$ [Poll]
For small- and large-scale systems.

The CPU checks this input upon execution of the $\overline{\text{POLL}}$ instruction. If the input is low, then execution continues. If the input is high, the CPU will check the $\overline{\text{POLL}}$ input every five clock cycles until the input becomes low again.

The $\overline{\text{POLL}}$ and READY functions are used to synchronize CPU program execution with the operation of external devices.

$\overline{\text{RD}}$ [Read Strobe]
For small- and large-scale systems.

The CPU outputs this strobe signal during data read from an I/O device or memory. The IO/$\overline{\text{M}}$ signal is used to select between I/O and memory.

The three-state output is held high during standby mode and enters the high-impedance state during hold acknowledge.

S/$\overline{\text{LG}}$ [Small/Large]
For small- and large-scale systems.

This signal determines the operation mode of the CPU. This signal is fixed at either a high or low level. When

this signal is a high level, the CPU will operate in small-scale system mode, and when low, in the large-scale system mode. A small-scale system will have at most one bus master such as a DMA controller device on the bus. A large-scale system can have more than one bus master accessing the bus as well as the CPU.

Pins 24 to 31 and pin 34 function differently depending on the operating mode of the CPU. Separate nomenclature is adopted for these signals in the two operating modes.

	Function	
Pin No.	S/$\overline{LG}$-high	S/$\overline{LG}$-low
24	$\overline{INTAK}$	QS_1
25	ASTB	QS_0
26	$\overline{BUFEN}$	BS_0
27	BUF$\overline{R}$/W	BS_1
28	IO/$\overline{M}$	BS_2
29	$\overline{WR}$	$\overline{BUSLOCK}$
30	HLDAK	$\overline{RQ}/\overline{AK}_1$
31	HLDRQ	$\overline{RQ}/\overline{AK}_0$
34	LBS_0	Always high

$\overline{INTAK}$ [Interrupt Acknowledge]

For small-scale systems.

The CPU generates the $\overline{INTAK}$ signal low when it accepts an INT signal.

The interrupting device synchronizes with this signal and outputs the interrupt vector to the CPU via the data bus (AD_7 - AD_0).

ASTB [Address Strobe]

For small-scale systems.

The CPU outputs this strobe signal to latch address information at an external latch.

ASTB is held at a low level during standby mode and hold acknowledge.

$\overline{BUFEN}$ [Buffer Enable]

For small-scale systems.

This is used as the output enable signal for an external bidirectional buffer. The CPU generates this signal during data transfer operations with external memory or I/O devices or during input of an interrupt vector.

This three-state output is held high during standby mode and enters the high-impedance state during hold acknowledge.

BUF$\overline{R}$/W [Buffer Read/Write]

For small-scale systems.

The output of this signal determines the direction of data transfer with an external bidirectional buffer. A high output causes transmission from the CPU to the external device; a low signal causes data transfer from the external device to the CPU.

BUF$\overline{R}$/W is a three-state output and becomes high impedance during hold acknowledge.

IO/$\overline{M}$ [IO/Memory]

For small-scale systems.

The CPU generates this signal to specify either I/O access or memory access. A high-level output specifies I/O and a low-level signal specifies memory.

IO/M's output is three state and becomes high impedance during hold acknowledge.

$\overline{WR}$ [Write Strobe]

For small-scale systems.

The CPU generates this strobe signal during data write to an I/O device or memory. Selection of either I/O or memory is performed by the IO/M signal.

This three-state output is held high during standby mode and enters the high-impedance state during hold acknowledge.

HLDAK [Hold Acknowledge]

For small-scale systems.

The HLDAK signal is used to indicate that the CPU accepts the hold request signal (HLDRQ). When this signal is a high level, the address bus, address/data bus, and the control lines become high impedance.

HLDRQ [Hold Request]

For small-scale systems.

This input signal is used by external devices to request the CPU to release the address bus, address/data bus, and the control bus.

LBS_0 [Latched Bus Status 0]

For small-scale systems.

The CPU uses this signal along with the IO/$\overline{M}$ and BUF$\overline{R}$/W signals to inform an external device what the current bus cycle is.

IO/$\overline{M}$	BUFR/$\overline{W}$	LBS_0	Bus Cycle
0	0	0	Program fetch
0	0	1	Memory read
0	1	0	Memory write
0	1	1	Passive state
1	0	0	Interrupt acknowledge
1	0	1	I/O read
1	1	0	I/O write
1	1	1	Halt

A_{19}/PS_3 - A_{16}/PS_0 [Address Bus/Processor Status]

For small- and large-scale systems.

These pins are time multiplexed to operate as an address bus and as processor status signals.

When used as the address bus, these pins are the high 4 bits of the 20-bit memory address. During I/O access, all 4 bits output data 0.

The processor status signals are provided for both memory and I/O use. PS_3 is always 0 in the native mode and 1 in 8080 emulation mode. The interrupt enable flag (IE) is pin on pin PS_2. Pins PS_1 and PS_0 indicate which memory segment is being accessed.

A_{17}/PS_1	A_{16}/PS_0	Segment
0	0	Data segment 1
0	1	Stack segment
1	0	Program segment
1	1	Data segment 0

The output of these pins is three state and becomes high impedance during hold acknowledge.

QS_1, QS_0 [Queue Status]

For large-scale systems.

The CPU uses these signals to allow external devices, such as the floating-point arithmetic processor chip, (μPD72091) to monitor the status of the internal CPU instruction queue.

QS_1	QS_0	Instruction Queue Status
0	0	NOP (queue does not change)
0	1	First byte of instruction
1	0	Flush queue
1	1	Subsequent bytes of instruction

The instruction queue status indicated by these signals is the status when the execution unit (EXU) accesses the instruction queue. The data output from these pins is therefore valid only for one clock cycle immediately following queue access. These status signals are provided so that the floating-point processor chip can monitor the CPU's program execution status and synchronize its operation with the CPU when control is passed to it by the FPO (Floating Point Operation) instructions.

BS_2 - BS_0 [Bus Status]

For large-scale systems.

The CPU uses these status signals to allow an external bus controller to monitor what the current bus cycle is.

The external bus controller decodes these signals and generates the control signals required to perform access of the memory or I/O device.

BS_2	BS_1	BS_0	Bus Cycle
0	0	0	Interrupt acknowledge
0	0	1	I/O read
0	1	0	I/O write
0	1	1	Halt
1	0	0	Program fetch
1	0	1	Memory read
1	1	0	Memory write
1	1	1	Passive state

The output of these signals is three state and becomes high impedance during hold acknowledge.

$\overline{BUSLOCK}$ [Bus Lock]

For large-scale systems.

The CPU uses this signal to secure the bus while executing the instruction immediately following the BUSLOCK prefix instruction, or during an interrupt acknowledge cycle. It is a status signal to the other bus masters in a multiprocessor system, inhibiting them from using the system bus during this time.

The output of this signal is three state and becomes high impedance during hold acknowledge. $\overline{BUSLOCK}$ is high during standby mode except if the HALT instruction has a BUSLOCK prefix.

$\overline{RQ}/\overline{AK}_1$, $\overline{RQ}/\overline{AK}_0$ [Hold Request/Acknowledge]

For large-scale systems.

These pins function as bus hold request inputs ($\overline{RQ}$) and as bus hold acknowledge outputs ($\overline{AK}$). $\overline{RQ}/AK_0$ has a higher priority than $\overline{RQ}/\overline{AK}_1$.

These pins have three-state outputs with on-chip pull-up resistors which keep the pin at a high level when the output is high impedance.

V_{DD} [Power Supply]

For small- and large-scale systems.

This pin is used for the +5 V power supply.

GND [Ground]

For small- and large-scale systems.

This pin is used for ground.

IC [Internally Connected]

This pin is used for tests performed at the factory by NEC. The μPD70108 is used with this pin at ground potential.

Absolute Maximum Ratings

$T_A = +25°C$

Power supply voltage, V_{DD}	−0.5 V to +7.0 V
Power dissipation, PD_{MAX}	0.5 W
Input voltage, V_I	−0.5 V to V_{DD} + 0.3 V
CLK input voltage, V_K	−0.5 V to V_{DD} + 1.0 V
Output voltage, V_O	−0.5 V to V_{DD} + 0.3 V
Operating temperature, T_{OPT}	−40°C to +85°C
Storage temperature, T_{STG}	−65°C to +150°C

Comment: Exposing the device to stresses above those listed in Absolute Maximum Ratings could cause permanent damage. The device is not meant to be operated under conditions outside the limits described in the operational sections of this specification. Exposure to absolute maximum rating conditions for extended periods may affect device reliability.

Capacitance

$T_A = +25°C$, $V_{DD} = 0$ V

Parameter	Symbol	Limits Min	Limits Max	Unit	Test Conditions
Input capacitance	C_I		15	pF	fc = 1 MHz Unmeasured pins returned to 0 V
I/O capacitance	C_{IO}		15	pF	

DC Characteristics

μPD70108-5, $T_A = -40°C$ to +85°C, $V_{DD} = +5$ V ± 10%
μPD70108-8, μPD70108-10, $T_A = -10°C$ to +70°C, $V_{DD} = +5$ V ± 5%

Parameter	Symbol		Limits Min	Limits Typ	Limits Max	Unit	Test Conditions
Input voltage high	V_{IH}		2.2		V_{DD} + 0.3	V	
Input voltage low	V_{IL}		−0.5		0.8	V	
CLK input voltage high	V_{KH}		3.9		V_{DD} + 1.0	V	
CLK input voltage low	V_{KL}		−0.5		0.6	V	
Output voltage high	V_{OH}		0.7 x V_{DD}			V	$I_{OH} = -400$ μA
Output voltage low	V_{OL}				0.4	V	$I_{OL} = 2.5$ mA
Input leakage current high	I_{LIH}				10	μA	$V_I = V_{DD}$
Input leakage current low	I_{LIL}				−10	μA	$V_I = 0$ V
Output leakage current high	I_{LOH}				10	μA	$V_O = V_{DD}$
Output leakage current low	I_{LOL}				−10	μA	$V_O = 0$ V
Supply current	I_{DD}	70108-5 5 MHz	30	60		mA	Normal operation
			5	10		mA	Standby mode
		70108-8 8 MHz	45	80		mA	Normal operation
			6	12		mA	Standby mode
		70108-10 10 MHz	60	100		mA	Normal operation
			7	14		mA	Standby mode

AC Characteristics

μPD70108-5, $T_A = -40°C$ to $+85°C$, $V_{DD} = +5\ V \pm 10\%$
μPD70108-8, μPD70108-10, $T_A = -10°C$ to $+70°C$, $V_{DD} = +5\ V \pm 5\%$

		μPD70108-5		μPD70108-8		μPD70108-10			
Parameter	**Symbol**	**Min**	**Max**	**Min**	**Max**	**Min**	**Max**	**Unit**	**Conditions**
Small/Large Scale									
Clock cycle	t_{CYK}	200	500	125	500	100	500	ns	
Clock pulse width high	t_{KKH}	69		44		41		ns	$V_{KH} = 3.0\ V$
Clock pulse width low	t_{KKL}	90		60		49		ns	$V_{KL} = 1.5\ V$
Clock rise time	t_{KR}		10		8		5	ns	1.5 V to 3.0 V
Clock fall time	t_{KF}		10		7		5	ns	3.0 V to 1.5 V
READY inactive setup to CLK↓	t_{SRYLK}	−8		−8		−10		ns	
READY inactive hold after CLK↑	t_{HKRYH}	30		20		20		ns	
READY active setup to CLK↑	t_{SRYHK}	$t_{KKL} - 8$		$t_{KKL} - 8$		$t_{KKL} - 10$		ns	
READY active hold after CLK↑	t_{HKRYL}	30		20		20		ns	
Data setup time to CLK ↓	t_{SDK}	30		20		10		ns	
Data hold time after CLK ↓	t_{HKD}	10		10		10		ns	
NMI, INT, $\overline{POLL}$ setup time to CLK ↑	t_{SIK}	30		15		15		ns	
Input rise time (except CLK)	t_{IR}		20		20		20	ns	0.8 V to 2.2 V
Input fall time (except CLK)	t_{IF}		12		12		12	ns	2.2 V to 0.8 V
Output rise time	t_{OR}		20		20		20	ns	0.8 V to 2.2 V
Output fall time	t_{OF}		12		12		12	ns	2.2 V to 0.8 V
Small Scale									
Address delay time from CLK	t_{DKA}	10	90	10	60	10	48	ns	
Address hold time from CLK	t_{HKA}	10		10		10		ns	
PS delay time from CLK ↓	t_{DKP}	10	90	10	60	10	50	ns	
PS float delay time from CLK ↑	t_{FKP}	10	80	10	60	10	50	ns	
Address setup time to ASTB ↓	t_{SAST}	$t_{KKL} - 60$		$t_{KKL} - 30$		$t_{KKL} - 30$		ns	
Address float delay time from CLK ↓	t_{FKA}	t_{HKA}	80	t_{HKA}	60	t_{HKA}	50	ns	$C_L = 100\ pF$
ASTB ↑ delay time from CLK ↓	t_{DKSTH}		80		50		40	ns	
ASTB ↓ delay time from CLK ↑	t_{DKSTL}		85		55		45	ns	
ASTB width high	t_{STST}	$t_{KKL} - 20$		$t_{KKL} - 10$		$t_{KKL} - 10$		ns	
Address hold time from ASTB ↓	t_{HSTA}	$t_{KKH} - 10$		$t_{KKH} - 10$		$t_{KKH} - 10$		ns	

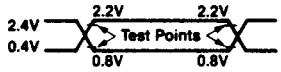
AC Characteristics (cont)

μPD70108-5, $T_A = -40°C$ to $+85°C$, $V_{DD} = +5\ V \pm 10\%$
μPD70108-8, μPD70108-10, $T_A = -10°C$ to $+70°C$, $V_{DD} = +5\ V \pm 5\%$

Parameter	Symbol	μPD70108-5 Min	μPD70108-5 Max	μPD70108-8 Min	μPD70108-8 Max	μPD70108-10 Min	μPD70108-10 Max	Unit	Conditions
Small Scale (cont)									
Control delay time from CLK	t_{DKCT}	10	110	10	65	10	55	ns	
Address float to $\overline{RD}\downarrow$	t_{AFRL}	0		0		0		ns	
$\overline{RD}\downarrow$ delay time from CLK $\downarrow$	t_{DKRL}	10	165	10	80	10	70	ns	
$\overline{RD}\uparrow$ delay time from CLK $\downarrow$	t_{DKRH}	10	150	10	80	10	60	ns	
Address delay time from $\overline{RD}\uparrow$	t_{DRHA}	$t_{CYK} - 45$		$t_{CYK} - 40$		$t_{CYK} - 35$		ns	
$\overline{RD}$ width low	t_{RR}	$2t_{CYK}-75$		$2t_{CYK}-50$		$2t_{CYK}-40$		ns	$C_L = 100$ pF
Data output delay time from CLK $\downarrow$	t_{DKD}	10	90	10	60	10	50	ns	
Data float delay time from CLK $\downarrow$	t_{FKD}	10	80	10	60	10	50	ns	
$\overline{WR}$ width low	t_{WW}	$2t_{CYK}-60$		$2t_{CYK}-40$		$2t_{CYK}-35$		ns	
HLDRQ setup time to CLK $\uparrow$	t_{SHQK}	35		20		20		ns	
HLDAK delay time from CLK $\downarrow$	t_{DKHA}	10	160	10	100	10	60	ns	
Large Scale									
Address delay time from CLK	t_{DKA}	10	90	10	60	10	48	ns	
Address hold time from CLK	t_{HKA}	10		10		10		ns	
PS delay time from CLK $\downarrow$	t_{DKP}	10	90	10	60	10	50	ns	
PS float delay time from CLK $\uparrow$	t_{FKP}	10	80	10	60	10	50	ns	
Address float delay time from CLK $\downarrow$	t_{FKA}	t_{HKA}	80	t_{HKA}	60	t_{HKA}	50	ns	
Address delay time from $\overline{RD}\uparrow$	t_{DRHA}	$t_{CYK} - 45$		$t_{CYK} - 40$		$t_{CYK} - 35$		ns	
ASTB delay time from BS $\downarrow$	t_{DBST}		15		15		15	ns	
BS $\downarrow$ delay time from CLK $\uparrow$	t_{DKBL}	10	110	10	60	10	50	ns	
BS $\uparrow$ delay time from CLK $\downarrow$	t_{DKBH}	10	130	10	65	10	50	ns	
$\overline{RD}\downarrow$ delay time from address float	t_{DAFRL}	0		0		0		ns	$C_L = 100$ pF
$\overline{RD}\downarrow$ delay time from CLK $\downarrow$	t_{DKRL}	10	165	10	80	10	70	ns	
$\overline{RD}\uparrow$ delay time from CLK $\downarrow$	t_{DKRH}	10	150	10	80	10	60	ns	
$\overline{RD}$ width low	t_{RR}	$2t_{CYK}-75$		$2t_{CYK}-50$		$2t_{CYK}-40$		ns	
Date output delay time from CLK $\downarrow$	t_{DKD}	10	90	10	60	10	50	ns	
Data float delay time from CLK $\uparrow$	t_{FKD}	10	80	10	60	10	50	ns	
$\overline{AK}$ delay time from CLK $\downarrow$	t_{DKAK}		70		50		40	ns	
$\overline{RQ}$ setup time to CLK $\uparrow$	t_{SRQK}	20		10		9		ns	
$\overline{RQ}$ hold time after CLK $\uparrow$	t_{HKRQ}	40		30		20		ns	

Timing Waveforms

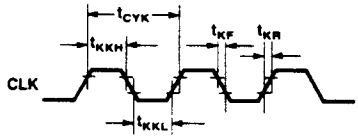
AC Test Input Waveform [Except CLK]



AC Output Test Points

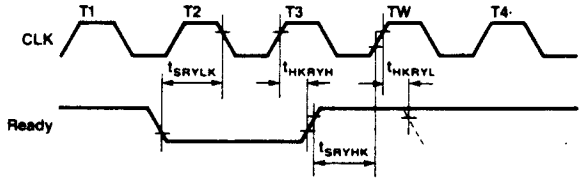
49-000238A

Clock Timing

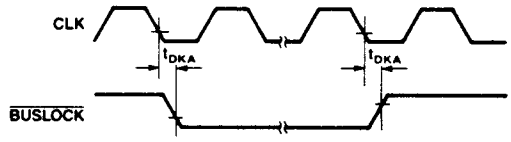


49-000239A

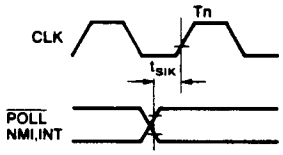
Wait [Ready] Timing



$\overline{\text{BUSLOCK}}$ Output Timing



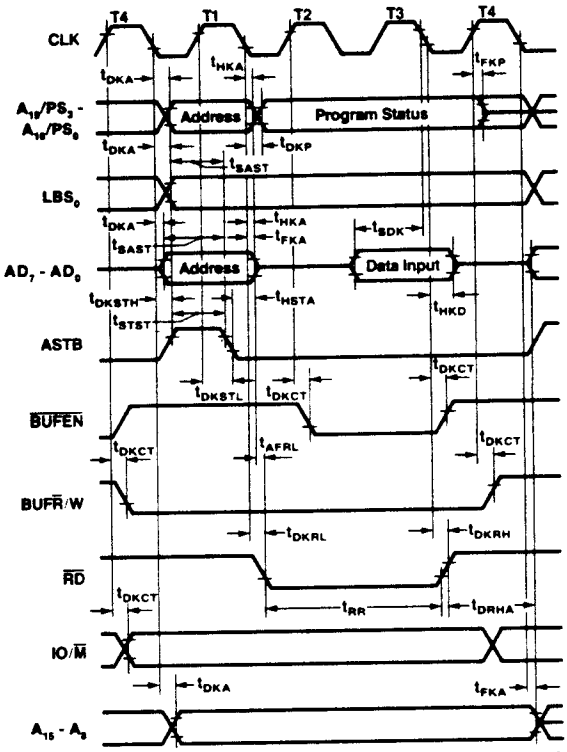
$\overline{\text{POLL}}$, NMI, INT Input Timing



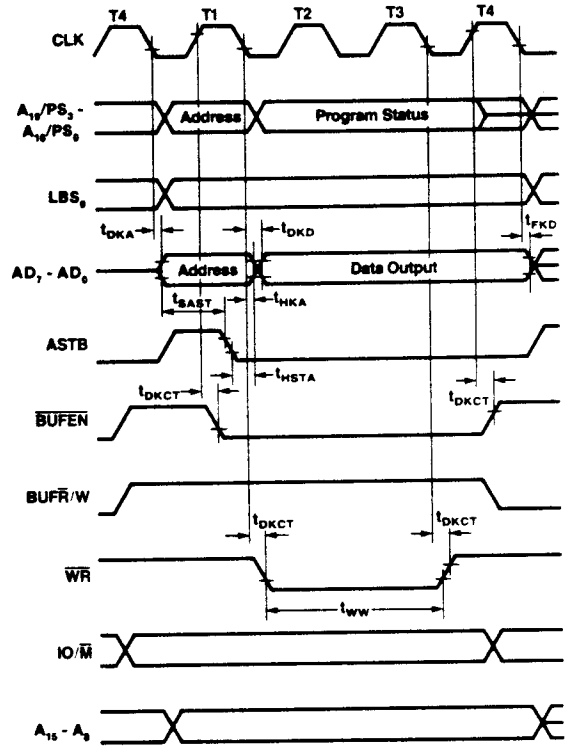
49-000240A

Timing Waveforms (cont)

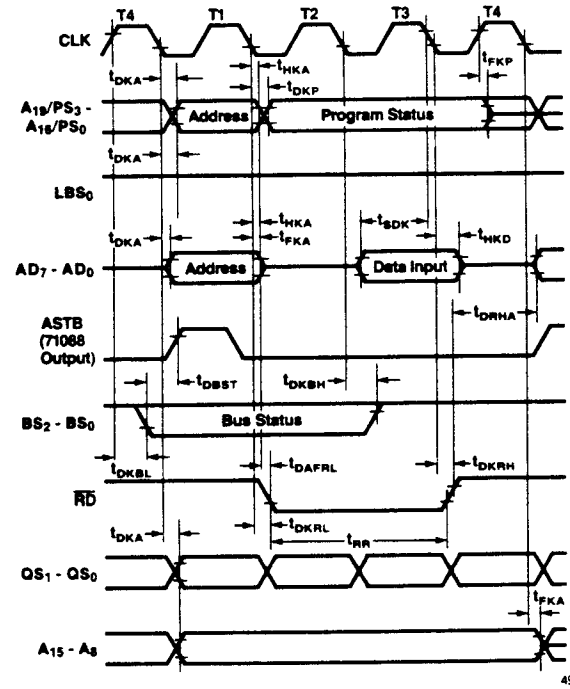
Read Timing [Small Scale]



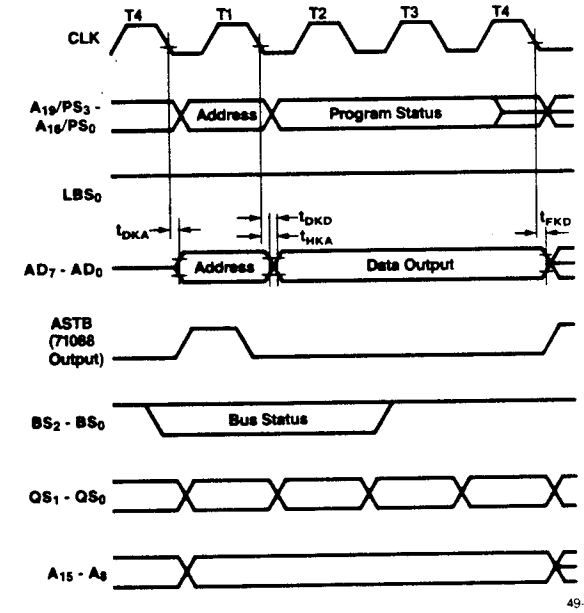
Write Timing [Small Scale]



Read Timing [Large Scale]

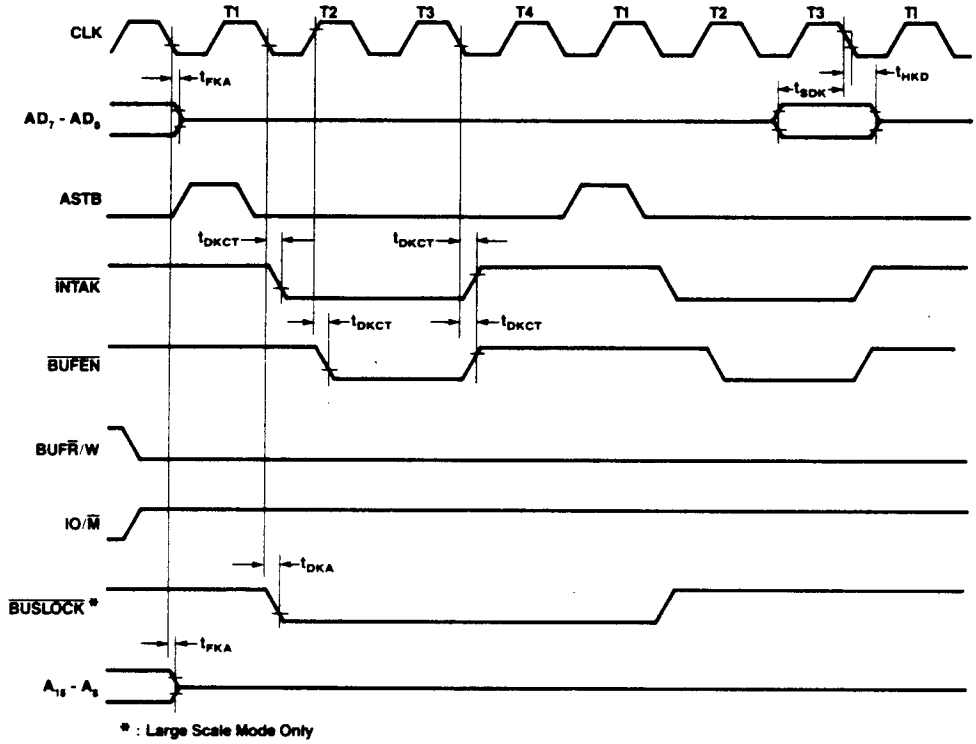


Write Timing [Large Scale]



Timing Waveforms (cont)

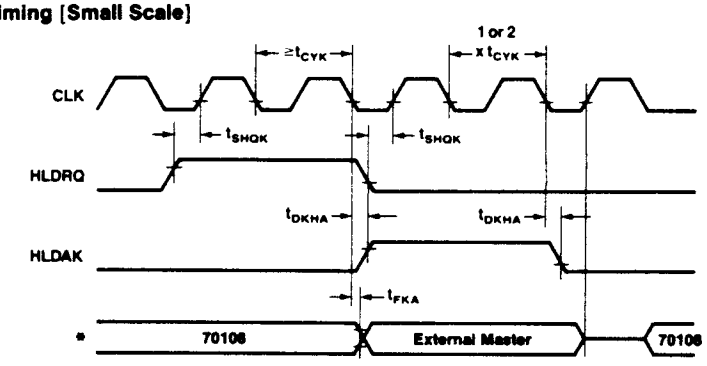
Interrupt Acknowledge Timing



* : Large Scale Mode Only

49:0002458

Hold Request/Acknowledge Timing [Small Scale]

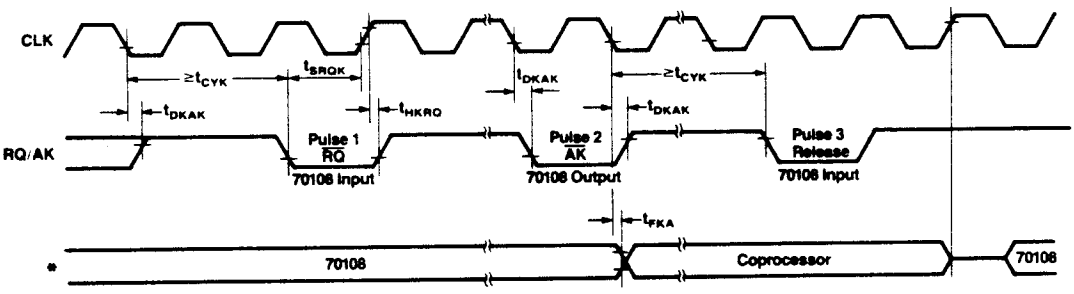


* : A_{19}/PS_3 - A_{16}/PS_0, A_{15} - A_8, AD_7 - AD_0, $\overline{RD}$, LBS_0, IO/$\overline{M}$, BUFR/$\overline{W}$, $\overline{WR}$ $\overline{BUFEN}$

49:0002506

Timing Waveforms (cont)

Bus Request/Acknowledge Timing [Large Scale]



* : A_{19}/PS_3 - A_{16}/PS_0, A_{15} - A_8, AD_7 - AD_0, BS_2 - BS_0, $\overline{RD}$, $\overline{BUSLOCK}$

Register Configuration

Program Counter [PC]

The program counter is a 16-bit binary counter that contains the segment offset address of the next instruction which the EXU is to execute.

The PC increments each time the microprogram fetches an instruction from the instruction queue. A new location value is loaded into the PC each time a branch, call, return, or break instruction is executed. At this time, the contents of the PC are the same as the Prefetch Pointer (PFP).

Prefetch Pointer [PFP]

The prefetch pointer (PFP) is a 16-bit binary counter which contains a segment offset which is used to calculate a program memory address that the bus control unit (BCU) uses to prefetch the next byte for the instruction queue. The contents of PFP are an offset from the PS (Program Segment) register.

The PFP is incremented each time the BCU prefetches an instruction from the program memory. A new location will be loaded into the PFP whenever a branch, call, return, or break instruction is executed. At that time the contents of the PFP will be the same as those of the PC (Program Counter).

Segment Registers [PS, SS, DS_0, and DS_1]

The memory addresses accessed by the μPD70108 are divided into 64K-byte logical segments. The starting (base) address of each segment is specified by a 16-bit segment register, and the offset from this starting address is specified by the contents of another register or by the effective address.

These are the four types of segment registers used.

Segment Register	Default Offset
PS (Program Segment)	PFP
SS (Stack Segment)	SP, effective address
DS_0 (Data Segment 0)	IX, effective address
DS_1 (Data Segment 1)	IY

General-Purpose Registers [AW, BW, CW, and DW]

There are four 16-bit general-purpose registers. Each one can be used as one 16-bit register or as two 8-bit registers by dividing them into their high and low bytes (AH, AL, BH, BL, CH, CL, DH, DL).

Each register is also used as a default register for processing specific instructions. The default assignments are:

AW: Word multiplication/division, word I/O, data conversion

AL: Byte multiplication/division, byte I/O, BCD rotation, data conversion, translation

AH: Byte multiplication/division

BW: Translation

CW: Loop control branch, repeat prefix

CL: Shift instructions, rototation instructions, BCD operations

DW: Word multiplication/division, indirect addressing I/O

Pointers [SP, BP] and Index Registers [IX, IY]

These registers serve as base pointers or index registers when accessing the memory using based addressing, indexed addressing, or based indexed addressing.

These registers can also be used for data transfer and arithmetic and logical operations in the same manner as the general-purpose registers. They cannot be used as 8-bit registers.

Also, each of these registers acts as a default register for specific operations. The default assignments are:

SP: Stack operations

IX: Block transfer (source), BCD string operations

IY: Block transfer (destination), BCD string operations

Program Status Word [PSW]

The program status word consists of the following six status and four control flags.

Status Flags

- V (Overflow)
- S (Sign)
- Z (Zero)
- AC (Auxiliary Carry)
- P (Parity)
- CY (Carry)

Control Flags

- MD (Mode)
- DIR (Direction)
- IE (Interrupt Enable)
- BRK (Break)

When the PSW is pushed on the stack, the word images of the various flags are as shown here.

PSW

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MD	1	1	1	V	DIR	IE	BRK	S	Z	0	AC	0	P	1	CY

The status flags are set and reset depending upon the result of each type of instruction executed.

Instructions are provided to set, reset, and complement the CY flag directly.

Other instructions set and reset the control flags and control the operation of the CPU.

High-Speed Execution of Instructions

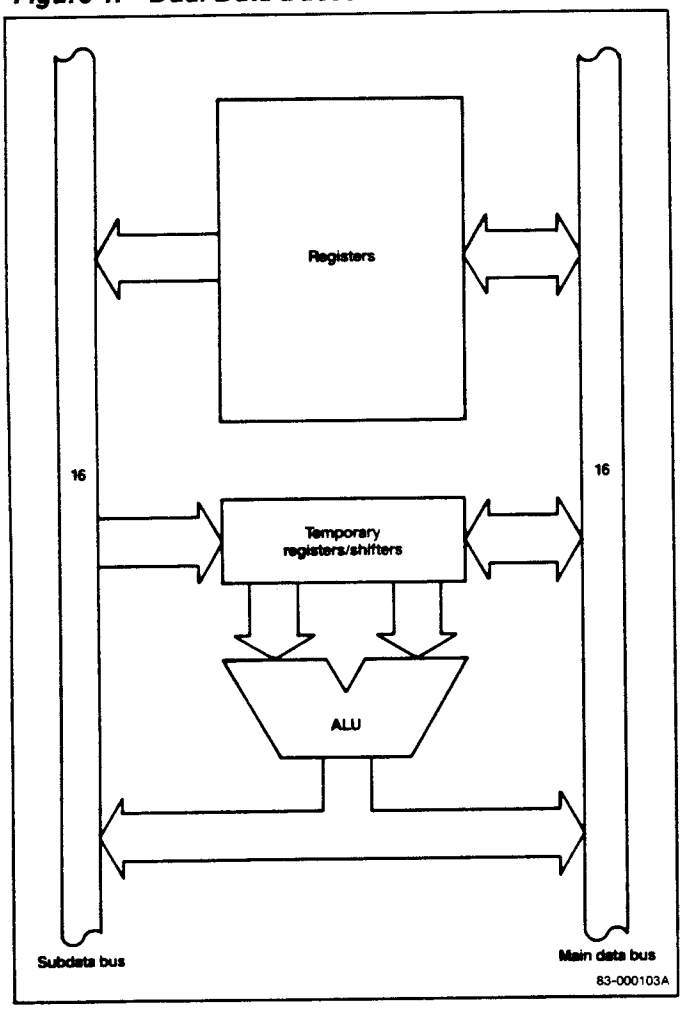
This section highlights the major architectural features that enhance the performance of the μPD70108.

- Dual data bus in EXU
- Effective address generator
- 16/32-bit temporary registers/shifters (TA, TB)
- 16-bit loop counter
- PC and PFP

Dual Data Bus Method

To reduce the number of processing steps for instruction execution, the dual data bus method has been adopted for the μPD70108 (figure 1). The two data buses (the main data bus and the subdata bus) are both 16 bits wide. For addition/subtraction and logical and comparison operations, processing time has been speeded up some 30% over single-bus systems.

Figure 1. Dual Data Buses



Example

ADD AW, BW ; AW ← AW + BW

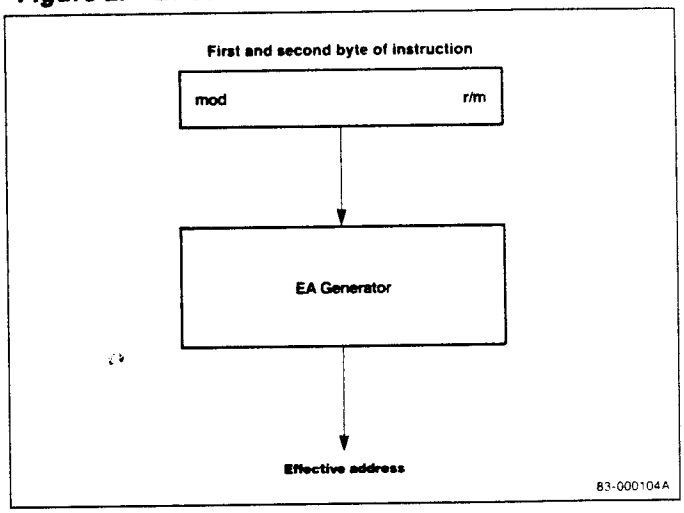
	Single Bus	Dual Bus
Step 1	TA ← AW	TA ← AW, TB ← BW
Step 2	TB ← BW	AW ← TA + TB
Step 3	AW ← TA + TB	

Effective Address Generator

This circuit (figure 2) performs high-speed processing to calculate effective addresses for accessing memory.

Calculating an effective address by the microprogramming method normally requires 5 to 12 clock cycles. This circuit requires only two clock cycles for addresses to be generated for any addressing mode. Thus, processing is several times faster.

Figure 2. Effective Address Generator



16/32-Bit Temporary Registers/Shifters [TA, TB]

These 16-bit temporary registers/shifters (TA, TB) are provided for multiplication/division and shift/rotation instructions.

These circuits have decreased the execution time of multiplication/division instructions. In fact, these instructions can be executed about four times faster than with the microprogramming method.

TA + TB: 32-bit temporary register/shifter for multiplication and division instructions.

TB: 16-bit temporary register/shifter for shift/rotation instructions.

Loop Counter [LC]

This counter is used to count the number of loops for a primitive block transfer instruction controlled by a repeat prefix instruction and the number of shifts that will be performed for a multiple bit shift/rotation instruction.

The processing performed for a multiple bit rotation of a register is shown below. The average speed is approximately doubled over the microprogram method.

Example

RORC AW, CL ; CL = 5

Microprogram method	LC method
8 + (4 x 5) = 28 clocks	7 + 5 = 12 clocks

Program Counter and Prefetch Pointer [PC and PFP]

The μPD70108 microprocessor has a program counter, (PC) which addresses the program memory location of the instruction to be executed next, and a prefetch pointer(PFP), which addresses the program memory location to be accessed next. Both functions are provided in hardware. A time saving of several clocks is realized for branch, call, return, and break instruction execution, compared with microprocessors that have only one instruction pointer.

Enhanced Instructions

In addition to the μPD8088/86 instructions, the μPD70108 has the following enhanced instructions.

Instruction	Function
PUSH imm	Pushes immediate data onto stack
PUSH R	Pushes 8 general registers onto stack
POP R	Pops 8 general registers from stack
MUL imm	Executes 16-bit multiply of register or memory contents by immediate data
SHL imm8 SHR imm8 SHRA imm8 ROL imm8 ROR imm8 ROLC imm8 RORC imm8	Shifts/rotates register or memory by immediate value
CHKIND	Checks array index against designated boundaries
INM	Moves a string from an I/O port to memory
OUTM	Moves a string from memory to an I/O port
PREPARE	Allocates an area for a stack frame and copies previous frame pointers
DISPOSE	Frees the current stack frame on a procedure exit

Enhanced Stack Operation Instructions

PUSH imm

This instruction allows immediate data to be pushed onto the stack.

PUSH R/POP R

These instructions allow the contents of the eight general registers to be pushed onto or popped from the stack with a single instruction.

Enhanced Multiplication Instructions

MUL reg16, imm16/MUL mem16, imm16

These instructions allow the contents of a register or memory location to be 16-bit multiplied by immediate data.

Enhanced Shift and Rotate Instructions

SHL reg, imm8/SHR reg, imm8/SHRA reg, imm8

These instructions allow the contents of a register to be shifted by the number of bits defined by the immediate data.

ROL reg, imm8/ROR reg, imm8/ROLC reg, imm8/ RORC reg, imm8

These instructions allow the contents of a register to be rotated by the number of bits defined by the immediate data.

Check Array Boundary Instruction

CHKIND reg16, mem32

This instruction is used to verify that index values pointing to the elements of an array data structure are within the defined range. The lower limit of the array should be in memory location mem32, the upper limit in mem32 + 2. If the index value in reg16 is not between these limits when CHKIND is executed, a BRK 5 will occur. This causes a jump to the location in interrupt vector 5.

Block I/O Instructions

OUTM DW, src-block/INM dst-block, DW

These instructions are used to output or input a string to or from memory, when preceded by a repeat prefix.

Stack Frame Instructions

PREPARE imm16, imm8

This instruction is used to generate the stack frames required by block-structured languages, such as PASCAL and Ada. The stack frame consists of two areas. One area has a pointer that points to another frame which has variables that the current frame can access. The other is a local variable area for the current procedure.

DISPOSE

This instruction releases the last stack frame generated by the PREPARE instruction. It returns the stack and base pointers to the values they had before the PREPARE instruction was used to call a procedure.

Unique Instructions

In addition to the μPD8088/86 instructions and the enhanced instructions, the μPD70108 has the following unique instructions.

Instruction	Function
INS	Insert bit field
EXT	Extract bit field
ADD4S	Adds packed decimal strings
SUB4S	Subtracts one packed decimal string from another
CMP4S	Compares two packed decimal strings
ROL4	Rotates one BCD digit left through AL lower 4 bits
ROR4	Rotates one BCD digit right through AL lower 4 bits
TEST1	Tests a specified bit and sets/resets Z flag
NOT1	Inverts a specified bit
CLR1	Clears a specified bit
SET1	Sets a specified bit
REPC	Repeats next instruction until CY flag is cleared
REPNC	Repeats next instruction until CY flag is set
FPO2	Additional floating point processor call

Variable Length Bit Field Operation Instructions

This category has two instructions: INS (Insert Bit Field) and EXT (Extract Bit Field). These instructions are highly effective for computer graphics and high-level languages. They can, for example, be used for data structures such as packed arrays and record type data used in PASCAL.

INS reg8, reg8/INS reg8, imm4

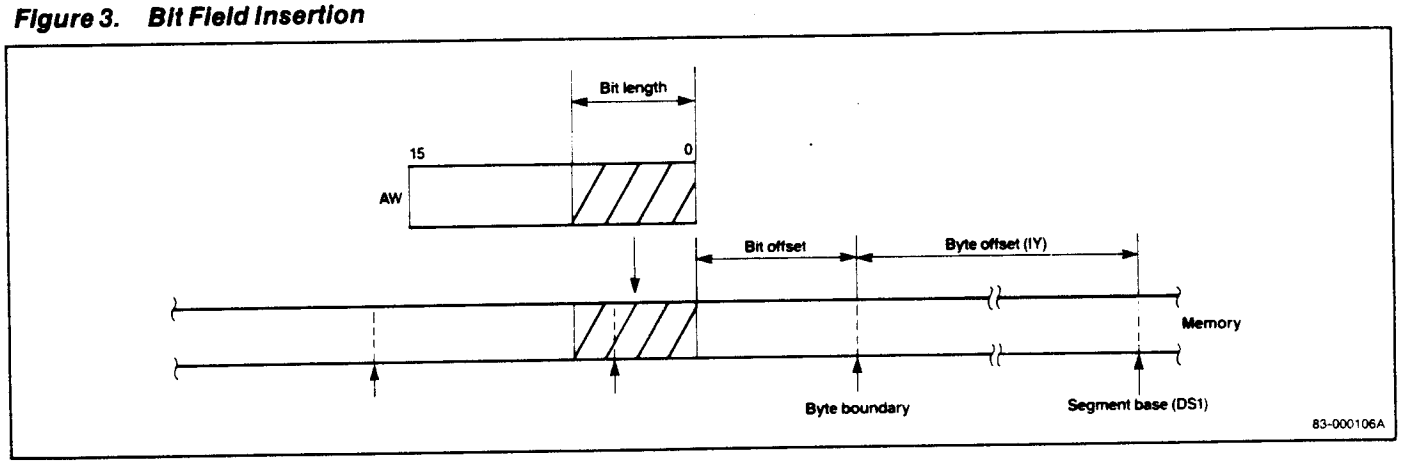
This instruction (figure 3) transfers low bits from the 16-bit AW register (the number of bits is specified by the second operand) to the memory location specified by the segment base (DS_1 register) plus the byte offset (IY register). The starting bit position within this byte is specified as an offset by the lower 4-bits of the first operand.

After each complete data transfer, the IY register and the register specified by the first operand are automatically updated to point to the next bit field.

Either immediate data or a register may specify the number of bits transferred (second operand). Because the maximum transferable bit length is 16-bits, only the lower 4-bits of the specified register (00H to 0FH) will be valid.

Bit field data may overlap the byte boundary of memory.

Figure 3. Bit Field Insertion



EXT reg8, reg8/EXT reg8, imm4

This instruction (figure 4) loads to the AW register the bit field data whose bit length is specified by the second operand of the instruction from the memory location that is specified by the DS0 segment register (segment base), the IX index register (byte offset), and the lower 4-bits of the first operand (bit offset).

After the transfer is complete, the IX register and the lower 4-bits of the first operand are automatically updated to point to the next bit field.

Either immediate data or a register may be specified for the second operand. Because the maximum transferrable bit length is 16 bits, however, only the lower 4-bits of the specified register (0H to 0FH) will be valid.

Bit field data may overlap the byte boundary of memory.

Packed BCD Operation Instructions

The instructions described here process packed BCD data either as strings (ADD4S, SUB4S, CMP4S) or byte-format operands (ROR4, ROL4). Packed BCD strings may be from 1 to 254 digits in length.

When the number of digits is even, the zero and carry flags will be set according to the result of the operation. When the number of digits is odd, the zero and carry flags may not be set correctly in this case, (CL = odd), the zero flag will not be set unless the upper 4 bits of the highest byte are all zero. The carry flag will not be set unless there is a carry out of the upper 4 bits of the highest byte. When CL is odd, the contents of the upper 4 bits of the highest byte of the result are undefined.

ADD4S

This instruction adds the packed BCD string addressed by the IX index register to the packed BCD string addressed by the IY index register, and stores the result in the string addressed by the IY register. The length of the string (number of BCD digits) is specified by the CL register, and the result of the operation will affect the overflow flag (V), the carry flag (CY), and zero flag (Z).

BCD string (IY, CL) ← BCD string (IY, CL) + BCD string (IX, CL)

SUB4S

This instruction subtracts the packed BCD string addressed by the IX index register from the packed BCD string addressed by the IY register, and stores the result in the string addressed by the IY register. The length of the string (number of BCD digits) is specified by the CL register, and the result of the operation will affect the overflow flag (V), the carry flag (CY), and zero flag (Z).

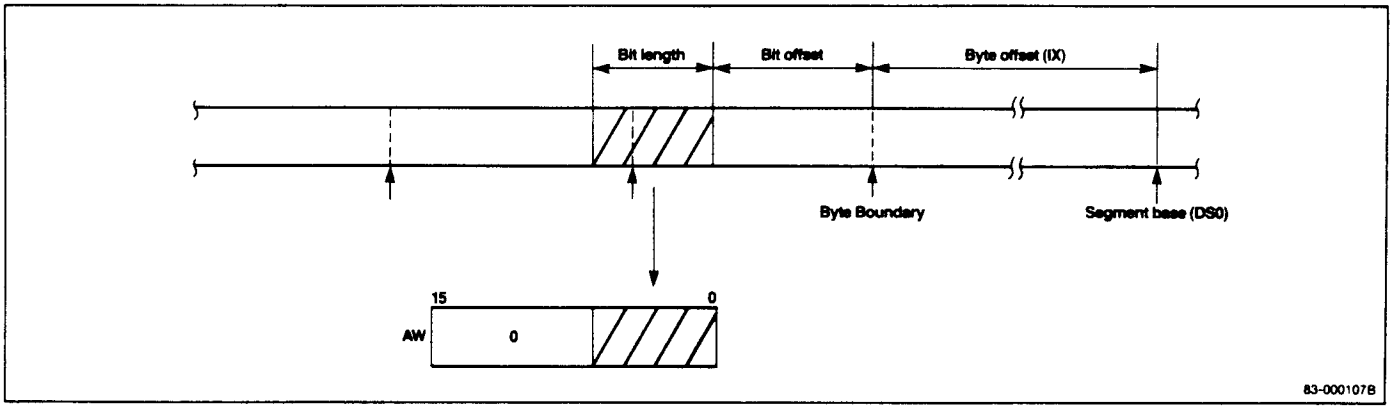
BCD string (IY, CL) ← BCD string (IY, CL) − BCD String (IX, CL)

CMP4S

This instruction performs the same operation as SUB4S except that the result is not stored and only the overflow (V), carry flags (CY) and zero flag (Z) are affected.

BCD string (IY, CL) − BCD string (IX, CL)

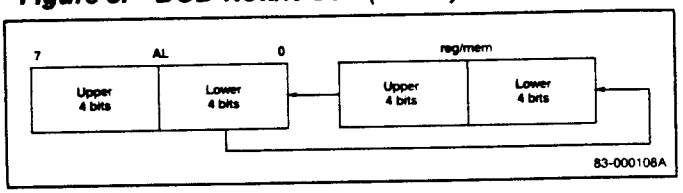
Figure 4. Bit Field Extraction



ROL4

This instruction (figure 5) treats the byte data of the register or memory directly specified by the instruction byte as BCD data and uses the lower 4-bits of the AL register (AL_L) to rotate that data one BCD digit to the left.

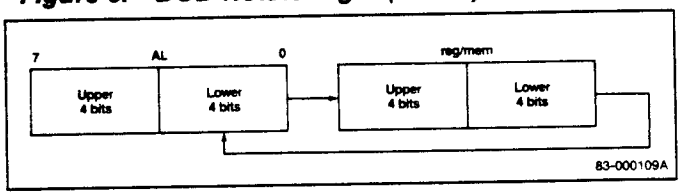
Figure 5. BCD Rotate Left (ROL4)



ROR4

This instruction (figure 6) treats the byte data of the register or memory directly specified by the instruction byte as BCD data and uses the lower 4-bits of the AL register (AL_L) to rotate that data one BCD digit to the right.

Figure 6. BCD Rotate Right (ROR4)



Bit Manipulation Instructions

TEST1

This instruction tests a specific bit in a register or memory location. If the bit is 1, the Z flag is reset to 0. If the bit is 0, the Z flag is set to 1.

NOT1

This instruction inverts a specific bit in a register or memory location.

CLR1

This instruction clears a specific bit in a register or memory location.

SET1

This instruction sets a specific bit in a register or memory location.

Repeat Prefix Instructions

REPC

This instruction causes the μPD70108 to repeat the following primitive block transfer instruction until the CY flag becomes cleared or the CW register becomes zero.

REPNC

This instruction causes the μPD70108 to repeat the following primitive block transfer instruction until the CY flag becomes set or the CW register is decremented to zero.

Floating Point Instruction

FPO2

This instruction is in addition to the μPD8088/86 floating point instruction, FPO1. These instructions are covered in a later section.

Mode Operation Instructions

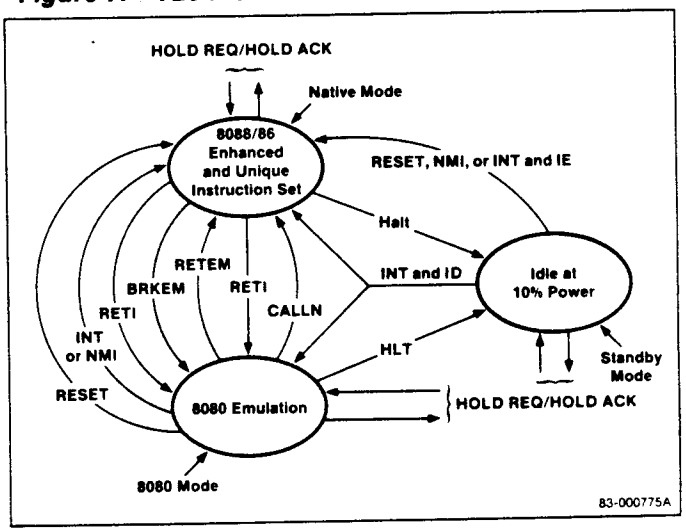
The μPD70108 has two operating modes (figure 7). One is the native mode which executes μPD8088/86, enhanced and unique instructions. The other is the 8080 emulation mode in which the instruction set of the μPD8080AF is emulated. A mode flag (MD) is provided to select between these two modes. Native mode is selected when MD is 1 and emulation mode when MD is 0. MD is set and reset, directly and indirectly, by executing the mode manipulation instructions.

Two instructions are provided to switch operation from the native mode to the emulation mode and back: BRKEM (Break for Emulation), and RETEM (Return from Emulation).

Two instructions are used to switch from the emulation mode to the native mode and back: CALLN (Call Native Routine), and RETI (Return from Interrupt).

The system will return from the 8080 emulation mode to the native mode when the RESET signal is present, or when an external interrupt (NMI or INT) is present.

Figure 7. V20 Modes



BRKEM imm8

This is the basic instruction used to start the 8080 emulation mode. This instruction operates exactly the same as the BRK instruction, except that BRKEM resets the mode flag (MD) to 0. PSW, PS, and PC are saved to the stack. MD is then reset and the interrupt vector specified by the operand imm8 of this command is loaded into PS and PC.

The instruction codes of the interrupt processing routine jumped to are then fetched. Then the CPU executes these codes as μPD8080AF instructions.

In 8080 emulation mode, registers and flags of the μPD8080AF are performed by the following registers and flags of the μPD70108.

	μPD8080AF	μPD70108
Registers:	A	AL
	B	CH
	C	CL
	D	DH
	E	DL
	H	BH
	L	BL
	SP	BP
	PC	PC
Flags:	C	CY
	Z	Z
	S	S
	P	P
	AC	AC

In the native mode, SP is used for the stack pointer. In the 8080 emulation mode this function is performed by BP.

This use of independent stack pointers allows independent stack areas to be secured for each mode and keeps the stack of one of the modes from being destroyed by an erroneous stack operation in the other mode.

The SP, IX, IY and AH registers and the four segment registers (PS, SS, DS_0, and DS_1) used in the native mode are not affected by operations in 8080 emulation mode.

In the 8080 emulation mode, the segment register for instructions is determined by the PS register (set automatically by the interrupt vector) and the segment register for data is the DS_0 register (set by the programmer immediately before the 8080 emulation mode is entered).

It is prohibited to nest BRKEM instructions.

RETEM [no operand]

When RETEM is executed in 8080 emulation mode (interpreted by the CPU as a μPD8080AF instruction), the CPU restores PS, PC, and PSW (as it would when returning from an interrupt processing routine), and returns to the native mode. At the same time, the contents of the mode flag (MD) which was saved to the stack by the BRKEM instruction, is restored to MD = 1. The CPU is set to the native mode.

CALLN imm8

This instruction makes it possible to call the native mode subroutines from the 8080 emulation mode. To return from subroutine to the emulation mode, the RETI instruction is used.

The processing performed when this instruction is executed in the 8080 emulation mode (it is interpreted by the CPU as μPD8080AF instruction), is similar to that performed when a BRK instruction is executed in the native mode. The imm8 operand specifies an interrupt vector type. The contents of PS, PC, and PSW are pushed on the stack and an MD flag value of 0 is saved. The mode flag is set to 1 and the interrupt vector specified by the operand is loaded into PS and PC.

RETI [no operand]

This is a general-purpose instruction used to return from interrupt routines entered by the BRK instruction or by an external interrupt in the native mode. When this instruction is executed at the end of a subroutine entered by the execution of the CALLN instruction, the operation that restores PS, PC, and PSW is exactly the same as the native mode execution. When PSW is restored, however, the 8080 emulation mode value of the mode flag (MD) is restored, the CPU is set in emulation mode, and all subsequent instructions are interpreted and executed as μPD8080AF instructions.

RETI is also used to return from an interrupt procedure initiated by an NMI or INT interrupt in the emulation mode.

Floating Point Operation Chip Instructions

FPO1 fp-op, mem/FPO2 fp-op, mem

These instructions are used for the external floating point processor. The floating point operation is passed to the floating point processor when the CPU fetches one of these instructions. From this point the CPU performs only the necessary auxiliary processing (effective address calculation, generation of physical addresses, and start-up of the memory read cycle).

The floating point processor always monitors the instructions fetched by the CPU. When it interprets one as an instruction to itself, it performs the appropriate processing. At this time, the floating point processor chip uses either the address alone or both the address and read data of the memory read cycle executed by the CPU. This difference in the data used depends on which of these instructions is executed.

Note: During the memory read cycle initiated by the CPU for FPO1 or FPO2 execution, the CPU does not accept any read data on the data bus from memory. Although the CPU generates the memory address, the data is used by the floating point processor.

Interrupt Operation

The interrupts used in the μPD70108 can be divided into two types: interrupts generated by external interrupt requests and interrupts generated by software processing. These are the classifications.

External Interrupts

(a) NMI input (nonmaskable)
(b) INT input (maskable)

Software Processing

As the result of instruction execution

— When a divide error occurs during execution of the DIV or DIVU instruction
— When a memory-boundary-over error is detected by the CHKIND instruction

Conditional break instruction

— When V = 1 during execution of the BRKV instruction

Unconditional break instructions

— 1-byte break instruction: BRK3
— 2-byte break instruction: BRK imm8

Flag processing

— When stack operations are used to set the BRK flag

8080 Emulation mode instructions

— BRKEM imm8
— CALLN imm8

Interrupt Vectors

Starting addresses for interrupt processing routines are either determined automatically by a single location of the interrupt vector table or selected each time interrupt processing is entered.

The interrupt vector table is shown in figure 8. The table uses 1K bytes of memory addresses 000H to 3FFH and can store starting address data for a maximum of 256 vectors (4 bytes per vector).

The corresponding interrupt sources for vectors 0 to 5 are predetermined and vectors 6 to 31 are reserved. These vectors consequently cannot be used for general applications.

The BRKEM instruction and CALLN instruction (in the emulation mode) and the INT input are available for general applications for vectors 32 to 255.

A single interrupt vector is made up of 4 bytes (figure 9). The 2 bytes in the low addresses of memory are loaded into PC as the offset, and the high 2 bytes are loaded into PS as the base address. The bytes are combined in reverse order. The lower-order bytes in the vector become the most significant bytes in the PC and PS, and the higher-order bytes become the least significant bytes.

Figure 8. Interrupt Vector Table

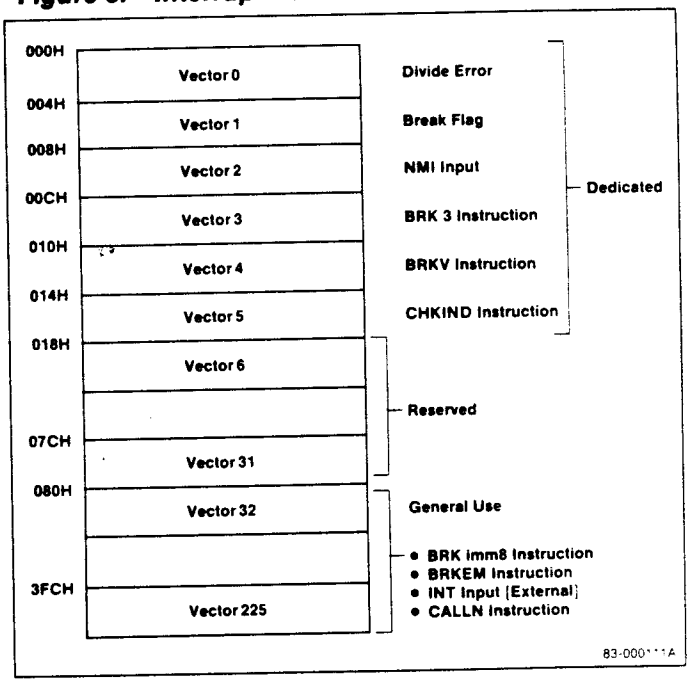
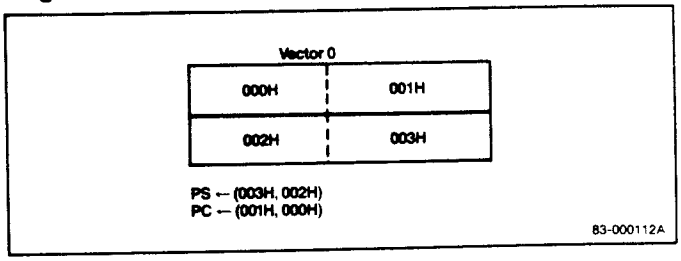


Figure 9. Interrupt Vector 0



Based on this format, the contents of each vector should be initialized at the beginning of the program.

The basic steps to jump to an interrupt processing routine are now shown.

(SP - 1, SP - 2) ← PSW
(SP - 3, SP - 4) ← PS
(SP - 5, SP - 6) ← PC
SP ← SP - 6
IE ← 0, BRK ← 0, MD ← 1
PS ← vector high bytes
PC ← vector low bytes

Standby Function

The μPD70108 has a standby mode to reduce power consumption during program wait states. This mode is set by the HALT instruction in both the native and the emulation mode.

In the standby mode, the internal clock is supplied only to those circuits related to functions required to release this mode and bus hold control functions. As a result, power consumption can be reduced to 1/10 the level of normal operation in either native or emulation mode.

The standby mode is released by inputting a RESET signal or an external interrupt (NMI, INT).

The bus hold function is effective during standby mode. The CPU returns to standby mode when the bus hold request is removed.

During standby mode, all control outputs are disabled and the addres/data bus will be at either high or low levels.

Instruction Set

The following tables briefly describe the μPD70108's instruction set.

- ☐ Operation and Operand Types - defines abbreviations used in the Instruction Set table.
- ☐ Flag Operations - defines the sybols used to describe flag operations.
- ☐ Memory Addressing - shows how mem and mod combinations specify memory addressing modes.
- ☐ Selection of 8- and 16-Bit Registers - shows how reg and W select a register when mod = 111.
- ☐ Selection of Segment Registers - shows how sreg selects a segment register.
- ☐ Instruction Set - shows the instruction mnemonics, their effect, their operation codes the number of bytes in the instruction, the number of clocks required for execution, and the effect on the μPD70108 flags.

Operation and Operand Types

Identifier	Description
reg	8- or 16-bit general-purpose register
reg8	8-bit general-purpose register
reg16	16-bit general-purpose register
dmem	8- or 16-bit direct memory location
mem	8- or 16-bit memory location
mem8	8-bit memory location
mem16	16-bit memory location
mem32	32-bit memory location
imm	Constant (0 to FFFFH)
imm16	Constant (0 to FFFFH)
imm8	Constant (0 to FFH)
imm4	Constant (0 to FH)
imm3	Constant (0 to 7)
acc	AW or AL register
sreg	Segment register
src-table	Name of 256-byte translation table
src-block	Name of block addressed by the IX register
dst-block	Name of block addressed by the IY register
near-proc	Procedure within the current program segment
far-proc	Procedure located in another program segment
near-label	Label in the current program segment
short-label	Label between −128 and +127 bytes from the end of instruction
far-label	Label in another program segment
memptr16	Word containing the offset of the memory location within the current program segment to which control is to be transferred
memptr32	Double word containing the offset and segment base address of the memory location to which control is to be transferred
regptr16	16-bit register containing the offset of the memory location within the program segment to which control is to be transferred
pop-value	Number of bytes of the stack to be discarded (0 to 64K bytes, usually even addresses)
fp-op	Immediate data to identify the instruction code of the external floating point operation

Operation and Operand Types (cont)

Identifier	Description
R	Register set
W	Word/byte field (0 to 1)
reg	Register field (000 to 111)
mem	Memory field (000 to 111)
mod	Mode field (00 to 10)
S:W	When S:W = 01 or 11, data = 16 bits. At all other times, data = 8 bits.
X, XXX, YYY, ZZZ	Data to identify the instruction code of the external floating point arithmetic chip
AW	Accumulator (16 bits)
AH	Accumulator (high byte)
AL	Accumulator (low byte)
BW	BW register (16 bits)
CW	CW register (16 bits)
CL	CW register (low byte)
DW	DW register (16 bits)
SP	Stack pointer (16 bits)
PC	Program counter (16 bits)
PSW	Program status word (16 bits)
IX	Index register (source) (16 bits)
IY	Index register (destination) (16 bits)
PS	Program segment register (16 bits)
SS	Stack segment register (16 bits)
DS_0	Data segment 0 register (16 bits)
DS_1	Data segment 1 register (16 bits)
AC	Auxiliary carry flag
CY	Carry flag
P	Parity flag
S	Sign flag
Z	Zero flag
DIR	Direction flag
IE	Interrupt enable flag
V	Overflow flag
BRK	Break flag
MD	Mode flag
(...)	Values in parentheses are memory contents
disp	Displacement (8 or 16 bits)
ext-disp8	16-bit displacement (sign-extension byte + 8-bit displacement)
temp	Temporary register (8/16/32 bits)

Operation and Operand Types (cont)

Identifier	Description
tmpcy	Temporary carry flag (1 bit)
seg	Immediate segment data (16 bits)
offset	Immediate offset data (16 bits)
←	Transfer direction
+	Addition
−	Subtraction
x	Multiplication
÷	Division
%	Modulo
AND	Logical product
OR	Logical sum
XOR	Exclusive logical sum
XXH	Two-digit hexadecimal value
XXXXH	Four-digit hexadecimal value

Flag Operations

Identifier	Description
(blank)	No change
0	Cleared to 0
1	Set to 1
X	Set or cleared according to the result
U	Undefined
R	Value saved earlier is restored

Memory Addressing

	mod		
mem	00	01	10
000	BW + IX	BW + IX + disp8	BW + IX + disp16
001	BW + IY	BW + IY + disp8	BW + IY + disp16
010	BP + IX	BP + IX + disp8	BP + IX + disp16
011	BP + IY	BP + IY + disp8	BP + IY + disp16
100	IX	IX + disp8	IX + disp16
101	IY	IY + disp8	IY + disp16
110	Direct address	BP + disp8	BP + disp16
111	BW	BW + disp8	BW + disp16

Selection of 8- and 16-Bit Registers (mod 11)

reg	W = 0	W = 1
000	AL	AW
001	CL	CW
010	DL	DW
011	BL	BW
100	AH	SP
101	CH	BP
110	DH	IX
111	BH	IY

Selection of Segment Registers

sreg	
00	DS_1
01	PS
10	SS
11	DS_0

The table on the following pages shows the instruction set.

At "No. of Clocks," for instructions referencing memory operands, the left side of the slash (/) is the number of clocks for byte operands and the right side is for word operands. For conditional control transfer instructions, the left side of the slash (/) is the number of clocks if a control transfer takes place. The right side is the number of clocks when no control transfer or branch occurs. Some instructions show a range of clock times, separated by a hyphen. The execution time of these instructions varies from the minimum value to the maximum, depending on the operands involved.

"No. of Clocks" includes these times:

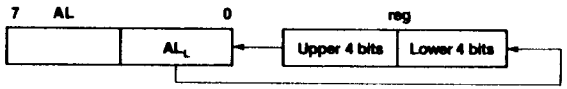
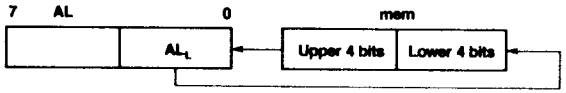
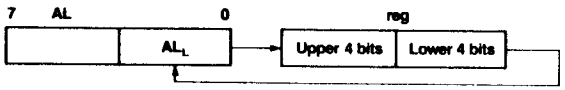
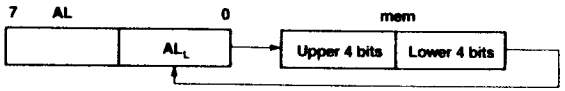
- Decoding
- Effective address generation
- Operand fetch
- Execution

It assumes that the instruction bytes have been prefetched.

Mnemonic	Operand	Operation	Operation Code 7 6 5 4 3 2 1 0 / 7 6 5 4 3 2 1 0	No. of Clocks	No. of Bytes	Flags AC	CY	V	P	S	Z
		Data Transfer Instructions									
MOV	reg, reg	reg ← reg	1 0 0 0 1 0 1 W / 1 1 reg reg	2	2						
	mem, reg	(mem) ← reg	1 0 0 0 1 0 0 W / mod reg mem	9/13	2-4						
	reg, mem	reg ← (mem)	1 0 0 0 1 0 1 W / mod reg mem	11/15	2-4						
	mem, imm	(mem) ← imm	1 1 0 0 0 1 1 W / mod 0 0 0 mem	11/15	3-6						
	reg, imm	reg ← imm	1 0 1 1 W reg	4	2-3						
	acc, dmem	When W = 0 AL ← (dmem) When W = 1 AH ← (dmem + 1), AL ← (dmem)	1 0 1 0 0 0 0 W	10/14	3						
	dmen, acc	When W = 0 (dmem) ← AL When W = 1 (dmem + 1) ← AH, (dmem) ← AL	1 0 1 0 0 0 1 W	9/13	3						
	sreg, reg16	sreg ← reg16 sreg : SS, DS0, DS1	1 0 0 0 1 1 1 0 / 1 1 0 sreg reg	2	2						
	sreg, mem16	sreg ← (mem16) sreg : SS, DS0, DS1	1 0 0 0 1 1 1 0 / mod 0 sreg mem	11/15	2-4						
	reg16, sreg	reg16 ← sreg	1 0 0 0 1 1 0 0 / 1 1 0 sreg reg	2	2						
	mem16, sreg	(mem16) ← sreg	1 0 0 0 1 1 0 0 / mod 0 sreg mem	10/14	2-4						
	DS0, reg16, mem32	reg16 ← (mem32) DS0 ← (mem32 + 2)	1 1 0 0 0 1 0 1 / mod reg mem	18/26	2-4						
	DS1, reg16, mem32	reg16 ← (mem32) DS1 ← (mem32 + 2)	1 1 0 0 0 1 0 0 / mod reg mem	18/26	2-4						
	AH, PSW	AH ← S, Z, x, AC, x, P, x, CY	1 0 0 1 1 1 1 1	2	1	x	x		x	x	x
	PSW, AH	S, Z, x, AC, x, P, x, CY ← AH	1 0 0 1 1 1 1 0	3	1	x	x		x	x	x
LDEA	reg16, mem16	reg16 ← mem16	1 0 0 0 1 1 0 1 / mod reg mem	4	2-4						
TRANS	src-table	AL ← (BW + AL)	1 1 0 1 0 1 1 1	9	1						
XCH	reg, reg	reg ⟷ reg	1 0 0 0 0 1 1 W / 1 1 reg reg	3	2						
	mem, reg or reg, mem	(mem) ⟷ reg	1 0 0 0 0 1 1 W / mod reg mem	16/26	2-4						
	AW, reg16 or reg16, AW	AW ⟷ reg16	1 0 0 1 0 reg	3	1						
		Repeat Prefixes									
REPC		While CW ≠ 0, the next byte of the primitive block transfer instruction is executed and CW is decremented (− 1). If there is a waiting interrupt, it is processed. When CY ≠ 1, exit the loop.	0 1 1 0 0 1 0 1	2	1						
REPNC		While CW ≠ 0, the next byte of the primitive block transfer instruction is executed and CW is decremented (− 1). If there is a waiting interrupt, it is processed. When CY ≠ 0, exit the loop.	0 1 1 0 0 1 0 0	2	1						

Mnemonic	Operand	Operation	Operation Code	No. of Clocks	No. of Bytes	Flags AC	CY	V	P	S	Z
Repeat Prefixes (cont)											
REP REPE REPZ		While CW ≠ 0, the next byte of the primitive block transfer instruction is executed and CW is decremented (− 1). If there is a waiting interrupt, it is processed. If the primitive block transfer instruction is CMPBK or CMPM and Z ≠ 1, exit the loop.	1 1 1 1 0 0 1 1	2	1						
REPNE REPNZ		While CW ≠ 0, the next byte of the primitive block transfer instruction is executed and CW is decremented (− 1). If there is a waiting interrupt, it is processed. If the primitive block transfer instruction is CMPBK or CMPM and Z ≠ 0, exit the loop.	1 1 1 1 0 0 1 0	2	1						
Primitive Block Transfer Instructions											
MOVBK	dst-block, src-block	When W = 0 (IY) ← (IX) DIR = 0: IX ← IX + 1, IY ← IY + 1 DIR = 1: IX ← IX − 1, IY ← IY − 1 When W = 1 (IY + 1, IY) ← (IX + 1, IX) DIR = 0: IX ← IX + 2, IY ← IY + 2 DIR = 1: IX ← IX − 2, IY ← IY − 2	1 0 1 0 0 1 0 W	11 + 8n 11 + 16n	1						
CMPBK	src-block, dst-block	When W = 0 (IX) − (IY) DIR = 0: IX ← IX + 1, IY ← IY + 1 DIR = 1: IX ← IX − 1, IY ← IY − 1 When W = 1 (IX + 1, IX) − (IY + 1, IY) DIR = 0: IX ← IX + 2, IY ← IY + 2 DIR = 1: IX ← IX − 2, IY ← IY − 2	1 0 1 0 0 1 1 W	7 + 14n 7 + 22n	1	x	x	x	x	x	x
CMPM	dst-block	When W = 0 AL − (IY) DIR = 0: IY ← IY + 1; DIR = 1: IY ← IY − 1 When W = 1 AW − (IY + 1, IY) DIR = 0: IY ← IY + 2; DIR = 1: IY ← IY − 2	1 0 1 0 1 1 1 W	7 + 10n 7 + 14n	1	x	x	x	x	x	x
LDM	src-block	When W = 0 AL ← (IX) DIR = 0: IX ← IX + 1; DIR = 1: IX ← IX − 1 When W = 1 AW ← (IX + 1, IX) DIR = 0: IX ← IX + 2; DIR = 1: IX ← IX − 2	1 0 1 0 1 1 0 W	7 + 9n 7 + 13n	1						
STM	dst-block	When W = 0 (IY) ← AL DIR = 0: IY ← IY + 1; DIR = 1: IY ← IY − 1 When W = 1 (IY + 1, IY) ← AW DIR = 0: IY ← IY + 2; DIR = 1: IY ← IY − 2	1 0 1 0 1 0 1 W n: number of transfers	7 + 4n 7 + 8n	1						
Bit Field Transfer Instructions											
INS	reg8, reg8	16-Bit field ← AW	0 0 0 0 1 1 1 1 0 0 1 1 0 0 0 1 1 1 reg reg	35-133	3						
	reg8, imm4	16-Bit field ← AW	0 0 0 0 1 1 1 1 0 0 1 1 1 0 0 1 1 1 0 0 0 reg	35-133	4						

Mnemonic	Operand	Operation	Operation Code 7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	No. of Clocks	No. of Bytes	Flags AC	CY	V	P	S	Z
Bit Field Transfer Instructions (cont)												
EXT	reg8, reg8	AW ← 16-Bit field	0 0 0 0 1 1 1 1	0 0 1 1 reg reg	34-59	3						
	reg8, imm4	AW ← 16-Bit field	0 0 0 0 1 1 1 1	0 0 1 1 0 0 0 reg	34-59	4						
I/O Instructions												
IN	acc, imm8	When W = 0 AL ← (imm8) When W = 1 AH ← (imm8 + 1), AL ← (imm8)	1 1 1 0 0 1 0 W		9/13	2						
	acc, DW	When W = 0 AL ← (DW) When W = 1 AH ← (DW + 1), AL ← (DW)	1 1 1 0 1 1 0 W		8/12	1						
OUT	imm8, acc	When W = 0 (imm8) ← AL When W = 1 (imm8 + 1) ← AH, (imm8) ← AL	1 1 1 0 0 1 1 W		8/12	2						
	DW, acc	When W = 0 (DW) ← AL When W = 1 (DW + 1) ← AH, (DW) ← AL	1 1 1 0 1 1 1 W		8/12	1						
Primitive I/O Instructions												
INM	dst-block, DW	When W = 0 (IY) ← (DW) DIR = 0: IY ← IY + 1; DIR = 1: IY ← IY − 1 When W = 1 (IY + 1, IY) ← (DW + 1, DW) DIR = 0: IY ← IY + 2; DIR = 1: IY ← IY − 2	0 1 1 0 1 1 0 W		9 + 8n 9 + 16n	1						
OUTM	DW, src-block	When W = 0 (DW) ← (IX) DIR = 0: IX ← IX + 1; DIR = 1: IX ← IX − 1 When W = 1 (DW + 1, DW) ← (IX + 1, IX) DIR = 0: IX ← IX + 2; DIR = 1: IX ← IX − 2	0 1 1 0 1 1 1 W	n: number of transfers	9 + 8n 9 + 16n	1						
Addition/Subtraction Instructions												
ADD	reg, reg	reg ← reg + reg	0 0 0 0 0 0 1 W	1 1 reg reg	2	2	x	x	x	x	x	x
	mem, reg	(mem) ← (mem) + reg	0 0 0 0 0 0 0 W	mod reg mem	16/24	2-4	x	x	x	x	x	x
	reg, mem	reg ← reg + (mem)	0 0 0 0 0 0 1 W	mod reg mem	11/15	2-4	x	x	x	x	x	x
	reg, imm	reg ← reg + imm	1 0 0 0 0 0 S W	1 1 0 0 0 reg	4	3-4	x	x	x	x	x	x
	mem, imm	(mem) ← (mem) + imm	1 0 0 0 0 0 S W	mod 0 0 0 mem	18/26	3-6	x	x	x	x	x	x
	acc, imm	When W = 0 AL ← AL + imm When W = 1 AW ← AW + imm	0 0 0 0 0 1 0 W		4	2-3	x	x	x	x	x	x
ADDC	reg, reg	reg ← reg + reg + CY	0 0 0 1 0 0 1 W	1 1 reg reg	2	2	x	x	x	x	x	x
	mem, reg	(mem) ← (mem) + reg + CY	0 0 0 1 0 0 0 W	mod reg mem	16/24	2-4	x	x	x	x	x	x
	reg, mem	reg ← reg + (mem) + CY	0 0 0 1 0 0 1 W	mod reg mem	11/15	2-4	x	x	x	x	x	x
	reg, imm	reg ← reg + imm + CY	1 0 0 0 0 0 S W	1 1 0 1 0 reg	4	3-4	x	x	x	x	x	x
	mem, imm	(mem) ← (mem) + imm + CY	1 0 0 0 0 0 S W	mod 0 1 0 mem	18/26	3-6	x	x	x	x	x	x

Mnemonic	Operand	Operation	Operation Code	No. of Clocks	No. of Bytes	Flags AC	CY	V	P	S	Z
Addition/Subtraction Instructions (cont)											
ADDC	acc, imm	When W = 0 AL ← AL + imm + CY When W = 1 AW ← AW + imm + CY	0 0 0 1 0 1 0 W	4	2-3	x	x	x	x	x	x
SUB	reg, reg	reg ← reg − reg	0 0 1 0 1 0 1 W 1 1 reg reg	2	2	x	x	x	x	x	x
	mem, reg	(mem) ← (mem) − reg	0 0 1 0 1 0 0 W mod reg mem	16/24	2-4	x	x	x	x	x	x
	reg, mem	reg ← reg − (mem)	0 0 1 0 1 0 1 W mod reg mem	11/15	2-4	x	x	x	x	x	x
	reg, imm	reg ← reg − imm	1 0 0 0 0 0 S W 1 1 1 0 1 reg	4	3-4	x	x	x	x	x	x
	mem, imm	(mem) ← (mem) − imm	1 0 0 0 0 0 S W mod 1 0 1 mem	18/26	3-6	x	x	x	x	x	x
	acc, imm	When W = 0 AL ← AL − imm When W = 1 AW ← AW − imm	0 0 1 0 1 1 0 W	4	2-3	x	x	x	x	x	x
SUBC	reg, reg	reg ← reg − reg − CY	0 0 0 1 1 0 1 W 1 1 reg reg	2	2	x	x	x	x	x	x
	mem, reg	(mem) ← (mem) − reg − CY	0 0 0 1 1 0 0 W mod reg mem	16/24	2-4	x	x	x	x	x	x
	reg, mem	reg ← reg − (mem) − CY	0 0 0 1 1 0 1 W mod reg mem	11/15	2-4	x	x	x	x	x	
	reg, imm	reg ← reg − imm − CY	1 0 0 0 0 0 S W 1 1 0 1 1 reg	4	3-4	x	x	x	x	x	x
	mem, imm	(mem) ← (mem) − imm − CY	1 0 0 0 0 0 S W mod 0 1 1 mem	18/26	3-6	x	x	x	x	x	x
	acc, imm	When W = 0 AL ← AL − imm − CY When W = 1 AW ← AW − imm − CY	0 0 0 1 1 1 0 W	4	2-3	x	x	x	x	x	x
BCD Operation Instructions											
ADD4S		dst BCD string ← dst BCD string + src BCD string	0 0 0 0 1 1 1 1 0 0 1 0 0 0 0 0	7 + 19n	2	u	x	u	u	u	x
SUB4S		dst BCD string ← dst BCD string − src BCD string	0 0 0 0 1 1 1 1 0 0 1 0 0 0 1 0	7 + 19n	2	u	x	u	u	u	x
CMP4S		dst BCD string − src BCD string	0 0 0 0 1 1 1 1 0 0 1 0 0 1 1 0 n: number of BCD digits divided by 2	7 + 19n	2	u	x	u	u	u	x
ROL4	reg8		0 0 0 0 1 1 1 1 0 0 1 0 1 0 0 0 1 1 0 0 0 reg	25	3						
	mem8		0 0 0 0 1 1 1 1 0 0 1 0 1 0 0 0 mod 0 0 0 mem	28	3-5						
ROR4	reg8		0 0 0 0 1 1 1 1 0 0 1 0 1 0 1 0 1 1 0 0 0 reg	29	3						
	mem8		0 0 0 0 1 1 1 1 0 0 1 0 1 0 1 0 mod 0 0 0 mem	33	3-5						

Mnemonic	Operand	Operation	Operation Code 7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	No. of Clocks	No. of Bytes	Flags AC	CY	V	P	S	Z
		Increment/Decrement Instructions (cont)										
INC	reg8	reg8 ← reg8 + 1	1 1 1 1 1 1 1 0	1 1 0 0 0 reg	2	2	x		x	x	x	x
	mem	(mem) ← (mem) + 1	1 1 1 1 1 1 1 W	mod 0 0 0 mem	16/24	2-4	x		x	x	x	x
	reg16	reg16 ← reg16 + 1	0 1 0 0 0 reg		2	1	x		x	x	x	x
DEC	reg8	reg8 ← reg8 − 1	1 1 1 1 1 1 1 0	1 1 0 0 1 reg	2	2	x		x	x	x	x
	mem	(mem) ← (mem) − 1	1 1 1 1 1 1 1 W	mod 0 0 1 mem	16/24	2-4	x		x	x	x	x
	reg16	reg16 ← reg16 − 1	0 1 0 0 1 reg		2	1	x		x	x	x	x
		Multiplication Instructions										
MULU	reg8	AW ← AL x reg8 AH = 0: CY ← 0, V ← 0 AH ≠ 0: CY ← 1, V ← 1	1 1 1 1 0 1 1 0	1 1 1 0 0 reg	21-22	2	u	x	x	u	u	u
	mem8	AW ← AL x (mem8) AH = 0: CY ← 0, V ← 0 AH ≠ 0: CY ← 1, V ← 1	1 1 1 1 0 1 1 0	mod 1 0 0 mem	27-28	2-4	u	x	x	u	u	u
	reg16	DW, AW ← AW x reg16 DW = 0: CY ← 0, V ← 0 DW ≠ 0: CY ← 1, V ← 1	1 1 1 1 0 1 1 1	1 1 1 0 0 reg	29-30	2	u	x	x	u	u	u
	mem16	DW, AW ← AW x (mem16) DW = 0: CY ← 0, V ← 0 DW ≠ 0: CY ← 1, V ← 1	1 1 1 1 0 1 1 1	mod 1 0 0 mem	39-40	2-4	u	x	x	u	u	u
MUL	reg8	AW ← AL x reg8 AH = AL sign expansion: CY ← 0, V ← 0 AH ≠ AL sign expansion: CY ← 1, V ← 1	1 1 1 1 0 1 1 0	1 1 1 0 1 reg	33-39	2	u	x	x	u	u	u
	mem8	AW ← AL x (mem8) AH = AL sign expansion: CY ← 0, V ← 0 AH ≠ AL sign expansion: CY ← 1, V ← 1	1 1 1 1 0 1 1 0	mod 1 0 1 mem	39-45	2-4	u	x	x	u	u	u
	reg16	DW, AW ← AW x reg16 DW = AW sign expansion: CY ← 0, V ← 0 DW ≠ AW sign expansion: CY ← 1, V ← 1	1 1 1 1 0 1 1 1	1 1 1 0 1 reg	41-47	2	u	x	x	u	u	u
	mem16	DW, AW ← AW x (mem16) DW = AW sign expansion: CY ← 0, V ← 0 DW ≠ AW sign expansion: CY ← 1, V ← 1	1 1 1 1 0 1 1 1	mod 1 0 1 mem	51-57	2-4	u	x	x	u	u	u
	reg16, (reg16,) imm8	reg16 ← reg16 x imm8 Product ≤ 16 bits: CY ← 0, V ← 0 Product > 16 bits: CY ← 1, V ← 1	0 1 1 0 1 0 1 1	1 1 reg reg	28-34	3	u	x	x	u	u	u
	reg16, mem16, imm8	reg16 ← (mem16) x imm8 Product ≤ 16 bits: CY ← 0, V ← 0 Product > 16 bits: CY ← 1, V ← 1	0 1 1 0 1 0 1 1	mod reg mem	38-44	3-5	u	x	x	u	u	u

Mnemonic	Operand	Operation	Operation Code 7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	No. of Clocks	No. of Bytes	Flags AC	CY	V	P	S	Z
		Multiplication Instructions (cont)										
MUL	reg16, (reg16,) imm16	reg16 ← reg16 x imm16 Product ≤ 16 bits: CY ← 0, V ← 0 Product > 16 bits: CY ← 1, V ← 1	0 1 1 0 1 0 0 1	1 1 reg reg	36-42	4	u	x	x	u	u	u
	reg16, mem16, imm16	reg16 ← (mem16) x imm16 Product ≤ 16 bits: CY ← 0, V ← 0 Product > 16 bits: CY ← 1, V ← 1	0 1 1 0 1 0 0 1	mod reg mem	46-52	4-6	u	x	x	u	u	u
		Unsigned Division Instructions										
DIVU	reg8	temp ← AW When temp ÷ reg8 > FFH (SP − 1, SP − 2) ← PSW, (SP − 3, SP − 4) ← PS (SP − 5, SP − 6) ← PC, SP ← SP − 6 IE ← 0, BRK ← 0, PS ← (3, 2), PC ← (1, 0) All other times AH ← temp % reg8, AL ← temp ÷ reg8	1 1 1 1 0 1 1 0	1 1 1 1 0 reg	19	2	u	u	u	u	u	u
	mem8	temp ← AW When temp ÷ (mem8) > FFH (SP − 1, SP − 2) ← PSW, (SP − 3, SP − 4) ← PS (SP − 5, SP − 6) ← PC, SP ← SP − 6 IE ← 0, BRK ← 0, PS ← (3, 2), PC ← (1, 0) All other times AH ← temp % (mem8), AL ← temp ÷ (mem8)	1 1 1 1 0 1 1 0	mod 1 1 0 mem	25	2-4	u	u	u	u	u	u
	reg16	temp ← AW When temp ÷ reg16 > FFFFH (SP − 1, SP − 2) ← PSW, (SP − 3, SP − 4) ← PS (SP − 5, SP − 6) ← PC, SP ← SP − 6 IE ← 0, BRK ← 0, PS ← (3, 2), PC ← (1, 0) All other times AH ← temp % reg16, AL ← temp ÷ reg16	1 1 1 1 0 1 1 1	1 1 1 1 0 reg	25	2	u	u	u	u	u	u
	mem16	temp ← AW When temp ÷ (mem16) > FFFFH (SP − 1, SP − 2) ← PSW, (SP − 3, SP − 4) ← PS (SP − 5, SP − 6) ← PC, SP ← SP − 6 IE ← 0, BRK ← 0, PS ← (3, 2), PC ← (1, 0) All other times AH ← temp % (mem16), AL ← temp ÷ (mem16)	1 1 1 1 0 1 1 1	mod 1 1 0 mem	35	2-4	u	u	u	u	u	u
		Signed Division Instructions										
DIV	reg8	temp ← AW When temp ÷ reg8 > 0 and temp ÷ reg8 > 7FH or temp ÷ reg8 < 0 and temp ÷ reg8 < 0 - 7FH − 1 (SP − 1, SP − 2) ← PSW, (SP − 3, SP − 4) ← PS (SP − 5, SP − 6) ← PC, SP ← SP − 6 IE ← 0, BRK ← 0, PS ← (3, 2), PC ← (1, 0) All other times AH ← temp % reg8, AL ← temp ÷ reg8	1 1 1 1 0 1 1 0	1 1 1 1 1 reg	29-34	2	u	u	u	u	u	u

Mnemonic	Operand	Operation	Operation Code 7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	No. of Clocks	No. of Bytes	Flags AC	CY	V	P	S	Z
		Signed Division Instructions (cont)										
DIV	mem8	temp ← AW When temp ÷ (mem8) > 0 and (mem8) > 7FH or temp ÷ (mem8) < 0 and temp ÷ (mem8) < 0 - 7FH – 1 (SP – 1, SP – 2) ← PSW, (SP – 3, SP – 4) ← PS (SP – 5, SP – 6) ← PC, SP ← SP – 6 IE ← 0, BRK ← 0, PS ← (3, 2), PC ← (1, 0) All other times AH ← temp % (mem8), AL ← temp ÷ (mem8)	1 1 1 1 0 1 1 0	mod 1 1 1 mem	35-40	2-4	u	u	u	u	u	u
	reg16	temp ← AW When temp ÷ reg16 > 0 and reg16 > 7FFFH or temp ÷ reg16 < 0 and temp ÷ reg16 < 0 - 7FFFH – 1 (SP – 1, SP – 2) ← PSW, (SP – 3, SP – 4) ← PS (SP – 5, SP – 6) ← PC, SP ← SP – 6 IE ← 0, BRK ← 0, PS ← (3, 2), PC ← (1, 0) All other times AH ← temp % reg16, AL ← temp ÷ reg 16	1 1 1 1 0 1 1 1	1 1 1 1 1 reg	38-43	2	u	u	u	u	u	u
	mem16	temp ← AW When temp ÷ (mem16) > 0 and (mem16) > 7FFFH or temp ÷ (mem16) < 0 and temp ÷ (mem16) < 0 - 7FFFH – 1 (SP – 1, SP – 2) ← PSW, (SP – 3, SP – 4) ← PS (SP – 5, SP – 6) ← PC, SP ← SP – 6 IE ← 0, BRK ← 0, PS ← (3, 2), PC ← (1, 0) All other times AH ← temp % (mem16), AL ← temp ÷ (mem16)	1 1 1 1 0 1 1 1	mod 1 1 1 mem	48-53	2-4	u	u	u	u	u	u
		BCD Adjust Instructions										
ADJBA		When (AL AND 0FH) > 9 or AC = 1, AL ← AL + 6, AH ← AH + 1, AC ← 1, CY ← AC, AL ← AL AND 0FH	0 0 1 1 0 1 1 1		3	1	x	x	u	u	u	u
ADJ4A		When (AL AND 0FH) > 9 or AC = 1, AL ← AL + 6, CY ← CY OR AC, AC ← 1, When AL > 9FH, or CY = 1 AL ← AL + 60H, CY ← 1	0 0 1 0 0 1 1 1		3	1	x	x	u	x	x	x
ADJBS		When (AL AND 0FH) > 9 or AC = 1, AL ← AL – 6, AH ← AH – 1, AC ← 1, CY ← AC, AL ← AL AND 0FH	0 0 1 1 1 1 1 1		7	1	x	x	u	u	u	u
ADJ4S		When (AL AND 0FH) > 9 or AC = 1, AL ← AL – 6, CY ← CY OR AC, AC ← 1 When AL > 9FH or CY = 1 AL ← AL – 60H, CY ← 1	0 0 1 0 1 1 1 1		7	1	x	x	u	x	x	x

Mnemonic	Operand	Operation	Operation Code	No. of Clocks	No. of Bytes	Flags AC	CY	V	P	S	Z
Data Conversion Instructions											
CVTBD		AH ← AL ÷ 0AH, AL ← AL % 0AH	1 1 0 1 0 1 0 0 0 0 0 0 1 0 1 0	15	2	u	u	u	x	x	x
CVTDB		AH ← 0, AL ← AH x 0AH + AL	1 1 0 1 0 1 0 1 0 0 0 0 1 0 1 0	7	2	u	u	u	x	x	x
CVTBW		When AL < 80H, AH ← 0, all other times AH ← FFH	1 0 0 1 1 0 0 0	2	1						
CVTWL		When AL < 8000H, DW ← 0, all other times DW ← FFFFH	1 0 0 1 1 0 0 1	4-5	1						
Comparison Instructions											
CMP	reg, reg	reg − reg	0 0 1 1 1 0 1 W 1 1 reg reg	2	2	x	x	x	x	x	x
	mem, reg	(mem) − reg	0 0 1 1 1 0 0 W mod reg mem	11/15	2-4	x	x	x	x	x	x
	reg, mem	reg − (mem)	0 0 1 1 1 0 1 W mod reg mem	11/15	2-4	x	x	x	x	x	x
	reg, imm	reg − imm	1 0 0 0 0 0 S W 1 1 1 1 1 reg	4	3-4	x	x	x	x	x	x
	mem, imm	(mem) − imm	1 0 0 0 0 0 S W mod 1 1 1 mem	13/17	3-6	x	x	x	x	x	x
	acc, imm	When W = 0, AL − imm When W = 1, AW − imm	0 0 1 1 1 1 0 W	4	2-3	x	x	x	x	x	x
Complement Instructions											
NOT	reg	reg ← $\overline{reg}$	1 1 1 1 0 1 1 W 1 1 0 1 0 reg	2	2						
	mem	(mem) ← ($\overline{mem}$)	1 1 1 1 0 1 1 W mod 0 1 0 mem	16/24	2-4						
NEG	reg	reg ← $\overline{reg}$ + 1	1 1 1 1 0 1 1 W 1 1 0 1 1 reg	2	2	x	x	x	x	x	x
	mem	(mem) ← ($\overline{mem}$) + 1	1 1 1 1 0 1 1 W mod 0 1 1 mem	16/24	2-4	x	x	x	x	x	x
Logical Operation Instructions											
TEST	reg, reg	reg AND reg	1 0 0 0 0 1 0 W 1 1 reg reg	2	2	u	0	0	x	x	x
	mem, reg or reg, mem	(mem) AND reg	1 0 0 0 0 1 0 W mod reg mem	10/14	2-4	u	0	0	x	x	x
	reg, imm	reg AND imm	1 1 1 1 0 1 1 W 1 1 0 0 0 reg	4	3-4	u	0	0	x	x	x
	mem, imm	(mem) AND imm	1 1 1 1 0 1 1 W mod 0 0 0 mem	11/15	3-6	u	0	0	x	x	x
	acc, imm	When W = 0, AL AND imm8 When W = 1, AW AND imm8	1 0 1 0 1 0 0 W	4	2-3	u	0	0	x	x	x
AND	reg, reg	reg ← reg AND reg	0 0 1 0 0 0 1 W 1 1 reg reg	2	2	u	0	0	x	x	x
	mem, reg	(mem) ← (mem) AND reg	0 0 1 0 0 0 0 W mod reg mem	16/24	2-4	u	0	0	x	x	x
	reg, mem	reg ← reg AND (mem)	0 0 1 0 0 0 1 W mod reg mem	11/15	2-4	u	0	0	x	x	x
	reg, imm	reg ← reg AND imm	1 0 0 0 0 0 0 W 1 1 1 0 0 reg	4	3-4	u	0	0	x	x	x
	mem, imm	(mem) ← (mem) AND imm	1 0 0 0 0 0 0 W mod 1 0 0 mem	18/26	3-6	u	0	0	x	x	x
	acc, imm	When W = 0, AL ← AL AND imm8 When W = 1, AW ← AW AND imm16	0 0 1 0 0 1 0 W	4	2-3	u	0	0	x	x	x

Mnemonic	Operand	Operation	Operation Code 7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	No. of Clocks	No. of Bytes	Flags AC	CY	V	P	S	Z
Logical Operation Instructions (cont)												
OR	reg, reg	reg ← reg OR reg	0 0 0 0 1 0 1 W	1 1 reg reg	2	2	u	0	0	x	x	x
	mem, reg	(mem) ← (mem) OR reg	0 0 0 0 1 0 0 W	mod reg mem	16/24	2-4	u	0	0	x	x	x
	reg, mem	reg ← reg OR (mem)	0 0 0 0 1 0 1 W	mod reg mem	11/15	2-4	u	0	0	x	x	x
	reg, imm	reg ← reg OR imm	1 0 0 0 0 0 0 W	1 1 0 0 1 reg	4	3-4	u	0	0	x	x	x
	mem, imm	(mem) ← (mem) OR imm	1 0 0 0 0 0 0 W	mod 0 0 1 mem	18/26	3-6	u	0	0	x	x	x
	acc, imm	When W = 0, AL ← AL OR imm8 When W = 1, AW ← AW OR imm16	0 0 0 0 1 1 0 W		4	2-3	u	0	0	x	x	x
XOR	reg, reg	reg ← reg XOR reg	0 0 1 1 0 0 1 W	1 1 reg reg	2	2	u	0	0	x	x	x
	mem, reg	(mem) ← (mem) XOR reg	0 0 1 1 0 0 0 W	mod reg mem	16/24	2-4	u	0	0	x	x	x
	reg, mem	reg ← reg XOR (mem)	0 0 1 1 0 0 1 W	mod reg mem	11/15	2-4	u	0	0	x	x	x
	reg, imm	reg ← reg XOR imm	1 0 0 0 0 0 0 W	1 1 1 1 0 reg	4	3-4	u	0	0	x	x	x
	mem, imm	(mem) ← (mem) XOR imm	1 0 0 0 0 0 0 W	mod 1 1 0 mem	18/26	3-6	u	0	0	x	x	x
	acc, imm	When W = 0, AL ← AL XOR imm8 When W = 1, AW ← AW XOR imm16	0 0 1 1 0 1 0 W		4	2-3	u	0	0	x	x	x
Bit Operation Instructions												
			2nd byte*	3rd byte*								
TEST1	reg8, CL	reg8 bit no. CL = 0: Z ← 1 reg8 bit no. CL = 1: Z ← 0	0 0 0 1 0 0 0 0	1 1 0 0 0 reg	3	3	u	0	0	u	u	x
	mem8, CL	(mem8) bit no. CL = 0: Z ← 1 (mem8) bit no. CL = 1: Z ← 0	0 0 0 1 0 0 0 0	mod 0 0 0 mem	12	3-5	u	0	0	u	u	x
	reg16, CL	reg16 bit no. CL = 0: Z ← 1 reg16 bit no. CL = 1: Z ← 0	0 0 0 1 0 0 0 1	1 1 0 0 0 reg	3	3	u	0	0	u	u	x
	mem16, CL	(mem16) bit no. CL = 0: Z ← 1 (mem16) bit no. CL = 1: Z ← 0	0 0 0 1 0 0 0 1	mod 0 0 0 mem	16	3-5	u	0	0	u	u	x
	reg8, imm3	reg8 bit no. imm3 = 0: Z ← 1 reg8 bit no. imm3 = 1: Z ← 0	0 0 0 1 1 0 0 0	1 1 0 0 0 reg	4	4	u	0	0	u	u	x
	mem8, imm3	(mem8) bit no. imm3 = 0: Z ← 1 (mem8) bit no. imm3 = 1: Z ← 0	0 0 0 1 1 0 0 0	mod 0 0 0 mem	13	4-6	u	0	0	u	u	x
	reg16, imm4	reg16 bit no. imm4 = 0: Z ← 1 reg16 bit no. imm4 = 1: Z ← 0	0 0 0 1 1 0 0 1	1 1 0 0 0 reg	4	4	u	0	0	u	u	x
	mem16, imm4	(mem16) bit no. imm4 = 0: Z ← 1 (mem16) bit no. imm4 = 1: Z ← 0	0 0 0 1 1 0 0 1	mod 0 0 0 mem	17	4-6	u	0	0	u	u	x

2nd byte* 3rd byte*

*Note: First byte = 0FH

Bit Operation Instructions (cont)

Mnemonic	Operand	Operation	Operation Code 7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	No. of Clocks	No. of Bytes	Flags AC	CY	V	P	S	Z
			2nd byte*	3rd byte*								
NOT1	reg8, CL	reg8 bit no. CL ← $\overline{\text{reg8 bit no. CL}}$	0 0 0 1 0 1 1 0	1 1 0 0 0 reg	4	3						
	mem8, CL	(mem8) bit no. CL ← $\overline{\text{(mem8) bit no. CL}}$	0 0 0 1 0 1 1 0	mod 0 0 0 mem	18	3-5						
	reg16, CL	reg16 bit no. CL ← $\overline{\text{reg16 bit no. CL}}$	0 0 0 1 0 1 1 1	1 1 0 0 0 reg	4	3						
	mem16, CL	(mem16) bit no. CL ← $\overline{\text{(mem16) bit no. CL}}$	0 0 0 1 0 1 1 1	mod 0 0 0 mem	26	3-5						
	reg8, imm3	reg8 bit no. imm3 ← $\overline{\text{reg8 bit no. imm3}}$	0 0 0 1 1 1 1 0	1 1 0 0 0 reg	5	4						
	mem8, imm3	(mem8) bit no. imm3 ← $\overline{\text{(mem8) bit no. imm3}}$	0 0 0 1 1 1 1 0	mod 0 0 0 mem	19	4-6						
	reg16, imm4	reg16 bit no. imm4 ← $\overline{\text{(reg16) bit no. imm4}}$	0 0 0 1 1 1 1 1	1 1 0 0 0 reg	5	4						
	mem16, imm4	(mem16) bit no. imm4 ← $\overline{\text{(mem16) bit no. imm4}}$	0 0 0 1 1 1 1 1	mod 0 0 0 mem	27	4-6						
			2nd byte*	3rd byte*								
			*Note: First byte = 0FH									
	CY	CY ← $\overline{\text{CY}}$	1 1 1 1 0 1 0 1		2	1		x				
			2nd byte*	3rd byte*								
CLR1	reg8, CL	reg8 bit no. CL ← 0	0 0 0 1 0 0 1 0	1 1 0 0 0 reg	5	3						
	mem8, CL	(mem8) bit no. CL ← 0	0 0 0 1 0 0 1 0	mod 0 0 0 mem	14	3-5						
	reg16, CL	reg16 bit no. CL ← 0	0 0 0 1 0 0 1 1	1 1 0 0 0 reg	5	3						
	mem16, CL	(mem16) bit no. CL ← 0	0 0 0 1 0 0 1 1	mod 0 0 0 mem	22	3-5						
	reg8, imm3	reg8 bit no. imm3 ← 0	0 0 0 1 1 0 1 0	1 1 0 0 0 reg	6	4						
	mem8, imm3	(mem8) bit no. imm3 ← 0	0 0 0 1 1 0 1 0	mod 0 0 0 mem	15	4-6						
	reg16, imm4	reg16 bit no. imm4 ← 0	0 0 0 1 1 0 1 1	1 1 0 0 0 reg	6	4						
	mem16, imm4	(mem16) bit no. imm4 ← 0	0 0 0 1 1 0 1 1	mod 0 0 0 mem	27	4-6						
			2nd byte*	3rd byte*								
			*Note: First byte = 0FH									
	CY	CY ← 0	1 1 1 1 1 0 0 0		2	1		0				
	DIR	DIR ← 0	1 1 1 1 1 1 0 0		2	1						

Mnemonic	Operand	Operation	Operation Code 7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	No. of Clocks	No. of Bytes	Flags AC	CY	V	P	S	Z
		Bit Operation Instructions (cont)										
SET1	reg8, CL	reg8 bit no. CL ← 1	0 0 0 1 0 1 0 0	1 1 0 0 0 reg	4	3						
	mem8, CL	(mem8) bit no. CL ← 1	0 0 0 1 0 1 0 0	mod 0 0 0 mem	13	3-5						
	reg16, CL	reg16 bit no. CL ← 1	0 0 0 1 0 1 0 1	1 1 0 0 0 reg	4	3						
	mem16, CL	(mem16) bit no. CL ← 1	0 0 0 1 0 1 0 1	mod 0 0 0 mem	21	3-5						
	reg8, imm3	reg8 bit no. imm3 ← 1	0 0 0 1 1 1 0 0	1 1 0 0 0 reg	5	4						
	mem8, imm3	(mem8) bit no. imm3 ← 1	0 0 0 1 1 1 0 0	mod 0 0 0 mem	14	4-6						
	reg16, imm4	reg16 bit no. imm4 ← 1	0 0 0 1 1 1 0 1	1 1 0 0 0 reg	5	4						
	mem16, imm4	(mem16) bit no. imm4 ← 1	0 0 0 1 1 1 0 1 (2nd byte*)	mod 0 0 0 mem (3rd byte*)	22	4-6						
			*Note: First byte = 0FH									
	CY	CY ← 1	1 1 1 1 1 0 0 1		2	1		1				
	DIR	DIR ← 1	1 1 1 1 1 1 0 1		2	1						
		Shift Instructions										
SHL	reg, 1	CY ← MSB of reg, reg ← reg x 2 When MSB of reg ≠ CY, V ← 1 When MSB of reg = CY, V ← 0	1 1 0 1 0 0 0 W	1 1 1 0 0 reg	2	2	u	x	x	x	x	x
	mem, 1	CY ← MSB of (mem), (mem) ← (mem) x 2 When MSB of (mem) ≠ CY, V ← 1 When MSB of (mem) = CY, V ← 0	1 1 0 1 0 0 0 W	mod 1 0 0 mem	16/24	2-4	u	x	x	x	x	x
	reg, CL	temp ← CL, while temp ≠ 0; repeat this operation, CY ← MSB of reg, reg ← reg x 2, temp ← temp − 1	1 1 0 1 0 0 1 W	1 1 1 0 0 reg	7 + n	2	u	x	u	x	x	x
	mem, CL	temp ← CL, while temp ≠ 0, repeat this operation, CY ← MSB of (mem), (mem) ← (mem) x 2, temp ← temp − 1	1 1 0 1 0 0 1 W	mod 1 0 0 mem	19/27 + n	2-4	u	x	u	x	x	x
	reg, imm8	temp ← imm8, while temp ≠ 0, repeat this operation, CY ← MSB of reg, reg ← reg x 2, temp ← temp − 1	1 1 0 0 0 0 0 W	1 1 1 0 0 reg	7 + n	3	u	x	u	x	x	x
	mem, imm8	temp ← imm8, while temp ≠ 0, repeat this operation, CY ← MSB of (mem), (mem) ← (mem) x 2, temp ← temp − 1	1 1 0 0 0 0 0 W	mod 1 0 0 mem n: number of shifts	19/27 + n	3-5	u	x	u	x	x	x
SHR	reg, 1	CY ← LSB of reg, reg ← reg ÷ 2 When MSB of reg ≠ bit following MSB of reg: V ← 1 When MSB of reg = bit following MSB of reg: V ← 0	1 1 0 1 0 0 0 W	1 1 1 0 1 reg	2	2	u	x	x	x	x	x

Mnemonic	Operand	Operation	Operation Code 7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	No. of Clocks	No. of Bytes	Flags AC	CY	V	P	S	Z
		Shift Instructions (cont)										
SHR	mem, 1	CY ← LSB of (mem), (mem) ← (mem) ÷ 2 When MSB of (mem) ≠ bit following MSB of (mem): V ← 1 When MSB of (mem) = bit following MSB of (mem): V ← 0	1 1 0 1 0 0 0 W	mod 1 0 1 mem	16/24	2-4	u	x	x	x	x	x
	reg, CL	temp ← CL, while temp ≠ 0, repeat this operation, CY ← LSB of reg, reg ← reg ÷ 2, temp ← temp − 1	1 1 0 1 0 0 0 W	1 1 1 0 1 reg	7 + n	2	u	x	u	x	x	x
	mem, CL	temp ← CL, while temp ≠ 0, repeat this operation, CY ← LSB of (mem), (mem) ← (mem) ÷ 2, temp ← temp − 1	1 1 0 1 0 0 1 W	mod 1 0 1 mem	19/27+n	2-4	u	x	u	x	x	x
	reg, imm8	temp ← imm8, while temp ≠ 0, repeat this operation, CY ← LSB of reg, reg ← reg ÷ 2, temp ← temp − 1	1 1 0 0 0 0 0 W	1 1 1 0 1 reg	7 + n	3	u	x	u	x	x	x
	mem, imm8	temp ← imm8, while temp ≠ 0, repeat this operation, CY ← LSB of (mem), (mem) ← (mem) ÷ 2, temp ← temp − 1	1 1 0 0 0 0 0 W	mod 1 0 1 mem	19/27+n	3-5	u	x	u	x	x	x
			n: number of shifts									
SHRA	reg, 1	CY ← LSB of reg, reg ← reg ÷ 2, V ← 0 MSB of operand does not change	1 1 0 1 0 0 0 W	1 1 1 1 1 reg	2	2	u	x	0	x	x	x
	mem, 1	CY ← LSB of (mem), (mem) ← (mem) ÷ 2, V ← 0, MSB of operand does not change	1 1 0 1 0 0 0 W	mod 1 1 1 mem	16/24	2-4	u	x	0	x	x	x
	reg, CL	temp ← CL, while temp ≠ 0, repeat this operation, CY ← LSB of reg, reg ← reg ÷ 2, temp ← temp − 1 MSB of operand does not change	1 1 0 1 0 0 1 W	1 1 1 1 1 reg	7 + n	2	u	x	u	x	x	x
	mem, CL	temp ← CL, while temp ≠ 0, repeat this operation, CY ← LSB of (mem), (mem) ← (mem) ÷ 2, temp ← temp − 1 MSB of operand does not change	1 1 0 1 0 0 1 W	mod 1 1 1 mem	19/27+n	2-4	u	x	u	x	x	x
	reg, imm8	temp ← imm8, while temp ≠ 0, repeat this operation, CY ← LSB of reg, reg ← reg ÷ 2, temp ← temp − 1 MSB of operand does not change	1 1 0 0 0 0 0 W	1 1 1 1 1 reg	7 + n	3	u	x	u	x	x	x
	mem, imm8	temp ← imm8, while temp ≠ 0, repeat this operation, CY ← LSB of (mem), (mem) ← (mem) ÷ 2, temp ← temp − 1 MSB of operand does not change	1 1 0 0 0 0 0 W	mod 1 1 1 mem	19/27+n	3-5	u	x	u	x	x	x
			n: number of shifts									

Mnemonic	Operand	Operation	Operation Code 7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	No. of Clocks	No. of Bytes	Flags AC	CY	V	P	S	Z
Rotation Instructions												
ROL	reg, 1	CY ← MSB of reg, reg ← reg x 2 + CY MSB of reg ≠ CY: V ← 1 MSB of reg = CY: V ← 0	1 1 0 1 0 0 0 W	1 1 0 0 0 reg	2	2		x	x			
	mem, 1	CY ← MSB of (mem), (mem) ← (mem) x 2 + CY MSB of (mem) ≠ CY: V ← 1 MSB of (mem) = CY: V ← 0	1 1 0 1 0 0 0 W	mod 0 0 0 mem	16/24	2-4		x	x			
	reg, CL	temp ← CL, while temp ≠ 0, repeat this operation, CY ← MSB of reg, reg ← reg x 2 + CY temp ← temp − 1	1 1 0 1 0 0 1 W	1 1 0 0 0 reg	7 + n	2		x	u			
	mem, CL	temp ← CL, while temp ≠ 0, repeat this operation, CY ← MSB of (mem), (mem) ← (mem) x 2 + CY temp ← temp − 1	1 1 0 1 0 0 1 W	mod 0 0 0 reg	19/27+n	2-4		x	u			
	reg, imm8	temp ← imm8, while temp ≠ 0, repeat this operation, CY ← MSB of reg, reg ← reg x 2 + CY temp ← temp − 1	1 1 0 0 0 0 0 W	1 1 0 0 0 reg	7 + n	3		x	u			
	mem, imm8	temp ← imm8, while temp ≠ 0, repeat this operation, CY ← MSB of (mem), (mem) ← (mem) x 2 + CY temp ← temp − 1	1 1 0 0 0 0 0 W	mod 0 0 0 mem	19/27+n	3-5		x	u			
			n: number of shifts									
ROR	reg, 1	CY ← LSB of reg, reg ← reg ÷ 2 MSB of reg ← CY MSB of reg ≠ bit following MSB of reg: V ← 1 MSB of reg = bit following MSB of reg: V ← 0	1 1 0 1 0 0 0 W	1 1 0 0 1 reg	2	2		x	x			
	mem, 1	CY ← LSB of (mem), (mem) ← (mem) ÷ 2 MSB of (mem) ← CY MSB of (mem) ≠ bit following MSB of (mem): V ← 1 MSB of (mem) = bit following MSB of (mem): V ← 0	1 1 0 1 0 0 0 W	mod 0 0 1 mem	16/24	2-4		x	x			
	reg, CL	temp ← CL, while temp ≠ 0, repeat this operation, CY ← LSB of reg, reg ← reg ÷ 2, MSB of reg ← CY temp ← temp − 1	1 1 0 1 0 0 1 W	1 1 0 0 1 reg	7 + n	2		x	u			
	mem, CL	temp ← CL, while temp ≠ 0, repeat this operation, CY ← LSB of (mem), (mem) ← (mem) ÷ 2, MSB of (mem) ← CY temp ← temp − 1	1 1 0 1 0 0 1 W	mod 0 0 1 mem	19/27+n	2-4		x	u			
			n:number of shifts									

Mnemonic	Operand	Operation	Operation Code 7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	No. of Clocks	No. of Bytes	Flags AC	CY	V	P	S	Z
Rotation Instructions (cont)												
ROR	reg, imm8	temp ← imm8, while temp ≠ 0, repeat this operation, CY ← LSB of reg, reg ← reg ÷ 2, MSB of reg ← CY temp ← temp − 1	1 1 0 0 0 0 0 W	1 1 0 0 1 reg	7 + n	3		x				u
	mem, imm8	temp ← imm8, while temp ≠ 0, repeat this operation, CY ← LSB of (mem), (mem) ← (mem) ÷ 2 temp ← temp − 1	1 1 0 0 0 0 0 W	mod 0 0 1 mem	19/27+n	3-5		x				u
			n: number of shifts									
Rotate Instructions												
ROLC	reg, 1	tmpcy ← CY, CY ← MSB of reg reg ← reg x 2 + tmpcy MSB of reg = CY: V ← 0 MSB of reg ≠ CY: V ← 1	1 1 0 1 0 0 0 W	1 1 0 1 0 reg	2	2		x				x
	mem, 1	tmpcy ← CY, CY ← MSB of (mem) (mem) ← (mem) x 2 + tmpcy MSB of (mem) = CY: V ← 0 MSB of (mem) ≠ CY: V ← 1	1 1 0 1 0 0 0 W	mod 0 1 0 mem	16/24	2-4		x				x
	reg, CL	temp ← CL, while temp ≠ 0, repeat this operation, tmpcy ← CY, CY ← MSB of reg, reg ← reg x 2 + tmpcy temp ← temp − 1	1 1 0 1 0 0 1 W	1 1 0 1 0 reg	7 + n	2		x				u
	mem, CL	temp ← CL, while temp ≠ 0, repeat this operation, tmpcy ← CY, CY ← MSB of (mem), (mem) ← (mem) x 2 + tmpcy temp ← temp − 1	1 1 0 1 0 0 1 W	mod 0 1 0 mem	19/27+n	2-4		x				u
	reg, imm8	temp ← imm8, while temp ≠ 0, repeat this operation, tmpcy ← CY, CY ← MSB of reg, reg ← reg x 2 + tmpcy temp ← temp − 1	1 1 0 0 0 0 0 W	1 1 0 1 0 reg	7 + n	3		x				u
	mem, imm8	temp ← imm8, while temp ≠ 0, repeat this operation, tmpcy ← CY, CY ← MSB of (mem) (mem) ← (mem) x 2 + tmpcy temp ← temp − 1	1 1 0 0 0 0 0 W	mod 0 1 0 mem	19/27+n	3-5		x				u
			n: number of shifts									

Mnemonic	Operand	Operation	Operation Code 7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	No. of Clocks	No. of Bytes	Flags AC	CY	V	P	S	Z
Rotate Instructions (cont)												
RORC	reg, 1	tmpcy ← CY, CY ← LSB of reg reg ← reg ÷ 2, MSB of reg ← tmpcy MSB of reg ≠ bit following MSB of reg: V ← 1 MSB of reg = bit following MSB of reg: V ← 0	1 1 0 1 0 0 0 W	1 1 1 0 1 reg	2	2		x	x			
	mem, 1	tmpcy ← CY, CY ← LSB of (mem) (mem) ← (mem) ÷ 2, MSB of (mem) ← tmpcy MSB of (mem) ≠ bit following MSB of (mem): V ← 1 MSB of (mem) = bit following MSB of (mem): V ← 0	1 1 0 1 0 0 0 W	mod 0 1 1 mem	16/24	2-4		x	x			
	reg, CL	temp ← CL, while temp ≠ 0, repeat this operation, tmpcy ← CY, CY ← LSB of reg, reg ← reg ÷ 2, MSB of reg ← tmpcy, temp ← temp − 1	1 1 0 1 0 0 1 W	1 1 0 1 1 reg	7 + n	2		x	u			
	mem, CL	temp ← CL, while temp ≠ 0, repeat this operation, tmpcy ← CY, CY ← LSB of (mem), (mem) ← (mem) ÷ 2 MSB of (mem) ← tmpcy, temp ← temp − 1	1 1 0 1 0 0 1 W	mod 0 1 1 mem	19/27+n	2-4		x	u			
	reg, imm8	temp ← imm8, while temp ≠ 0 repeat this operation, tmpcy ← CY, CY ← LSB of reg, reg ← reg ÷ 2 MSB of reg ← tmpcy, temp ← temp − 1	1 1 0 0 0 0 0 W	1 1 0 1 1 reg	7 + n	3		x	u			
	mem, imm8	temp ← imm8, while temp ≠ 0, repeat this operation, tmpcy ← CY, CY ← LSB of (mem), (mem) ← (mem) ÷ 2 MSB of (mem) ← tmpcy, temp ← temp − 1	1 1 0 0 0 0 0 W	mod 0 1 1 mem	19/27+n	3-5		x	u			
				n: number of shifts								
Subroutine Control Instructions												
CALL	near-proc	(SP − 1, SP − 2) ← PC, SP ← SP − 2 PC ← PC + disp	1 1 1 0 1 0 0 0		20	3						
	regptr16	(SP − 1, SP − 2) ← PC, SP ← SP − 2 PC ← regptr16	1 1 1 1 1 1 1 1	1 1 0 1 0 reg	18	2						
	memptr16	(SP − 1, SP − 2) ← PC, SP ← SP − 2 PC ← (memptr16)	1 1 1 1 1 1 1 1	mod 0 1 0 mem	31	2-4						
	far-proc	(SP − 1, SP − 2) ← PS, (SP − 3, SP − 4) ← PC SP ← SP − 4, PS ← seg, PC ← offset	1 0 0 1 1 0 1 0		29	5						
	memptr32	(SP − 1, SP − 2) ← PS, (SP − 3, SP − 4) ← PC SP ← SP − 4, PS ← (memptr32 + 2), PC ← (memptr32)	1 1 1 1 1 1 1 1	mod 0 1 1 mem	47	2-4						

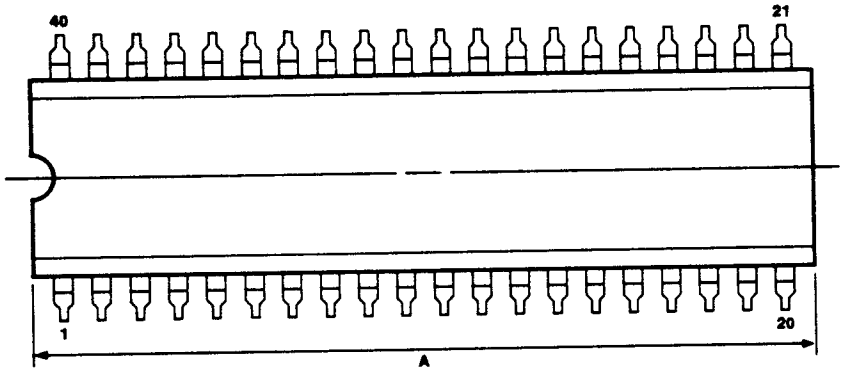
Mnemonic	Operand	Operation	Operation Code 7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	No. of Clocks	No. of Bytes	Flags AC	CY	V	P	S	Z
Subroutine Control Instructions (cont)												
RET		PC ← (SP + 1, SP), SP ← SP + 2	1 1 0 0 0 0 1 1		19	1						
	pop-value	PC ← (SP + 1, SP) SP ← SP + 2, SP ← SP + pop-value	1 1 0 0 0 0 1 0		24	3						
		PC ← (SP + 1, SP), PS ← (SP + 3, SP + 2) SP ← SP + 4	1 1 0 0 1 0 1 1		29	1						
	pop-value	PC ← (SP + 1, SP), PS ← (SP + 3, SP + 2) SP ← SP + 4, SP ← SP + pop-value	1 1 0 0 1 0 1 0		32	3						
Stack Manipulation Instructions												
PUSH	mem16	(SP − 1, SP − 2) ← (mem16), SP ← SP − 2	1 1 1 1 1 1 1 1	mod 1 1 0 mem	26	2-4						
	reg16	(SP − 1, SP − 2) ← reg16, SP ← SP − 2	0 1 0 1 0 reg		12	1						
	sreg	(SP− 1, SP− 2) ← sreg, SP ← SP − 2	0 0 0 sreg 1 1 0		12	1						
	PSW	(SP − 1, SP − 2) ← PSW, SP ← SP − 2	1 0 0 1 1 1 0 0		12	1						
	R	Push registers on the stack	0 1 1 0 0 0 0 0		67	1						
	imm	(SP − 1, SP − 2) ← imm SP ← SP − 2, When S = 1, sign extension	0 1 1 0 1 0 S 0		11/ 12	2-3						
POP	mem16	(mem16) ← (SP + 1, SP), SP ← SP + 2	1 0 0 0 1 1 1 1	mod 0 0 0 mem	25	2-4						
	reg16	reg16 ← (SP + 1, SP), SP ← SP + 2	0 1 0 1 1 reg		12	1						
	sreg	sreg ← (SP + 1, SP) sreg : SS, DS0, DS1 SP ← SP + 2	0 0 0 sreg 1 1 1		12	1						
	PSW	PSW ← (SP + 1, SP), SP ← SP + 2	1 0 0 1 1 1 0 1		12	1	R	R	R	R	R	R
	R	Pop registers from the stack	0 1 1 0 0 0 0 1		75	1						
PREPARE	imm16, imm8	Prepare new stack frame	1 1 0 0 1 0 0 0 *: imm8 = 0: 16 imm8 > 1: 23 + 16 (imm8 − 1)		*	4						
DISPOSE		Dispose of stack frame	1 1 0 0 1 0 0 1		10	1						
Branch Instruction												
BR	near-label	PC ← PC + disp	1 1 1 0 1 0 0 1		13	3						
	short-label	PC ← PC + ext-disp8	1 1 1 0 1 0 1 1		12	2						
	regptr16	PC ← regptr16	1 1 1 1 1 1 1 1	1 1 1 0 0 reg	11	2						
	memptr16	PC ← (memptr16)	1 1 1 1 1 1 1 1	mod 1 0 0 mem	24	2-4						
	far-label	PS ← seg, PC ← offset	1 1 1 0 1 0 1 0		15	5						
	memptr32	PS ← (memptr32 + 2), PC ← (memptr32)	1 1 1 1 1 1 1 1	mod 1 0 1 mem	35	2-4						

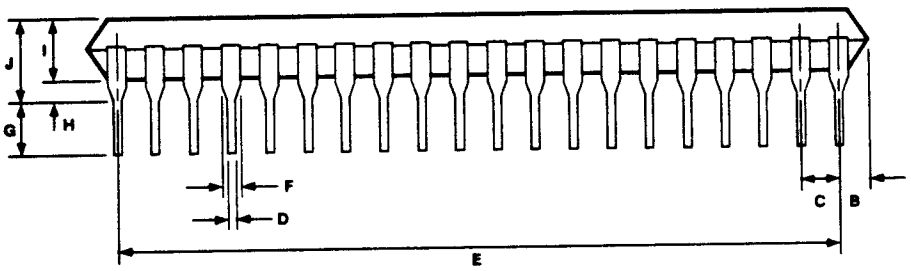
Mnemonic	Operand	Operation	Operation Code 7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0	No. of Clocks	No. of Bytes	Flags AC CY V P S Z
		Conditional Branch Instructions				
BV	short-label	if V = 1, PC ← PC + ext-disp8	0 1 1 1 0 0 0 0	14/4	2	
BNV	short-label	if V = 0, PC ← PC + ext-disp8	0 1 1 1 0 0 0 1	14/4	2	
BC, BL	short-label	if CY = 1, PC ← PC + ext-disp8	0 1 1 1 0 0 1 0	14/4	2	
BNC, BNL	short-label	if CY = 0, PC ← PC + ext-disp8	0 1 1 1 0 0 1 1	14/4	2	
BE, BZ	short-label	if Z = 1, PC ← PC + ext-disp8	0 1 1 1 0 1 0 0	14/4	2	
BNE, BNZ	short-label	if Z = 0, PC ← PC + ext-disp8	0 1 1 1 0 1 0 1	14/4	2	
BNH	short-label	if CY OR Z = 1, PC ← PC + ext-disp8	0 1 1 1 0 1 1 0	14/4	2	
BH	short-label	if CY OR Z = 0, PC ← PC + ext-disp8	0 1 1 1 0 1 1 1	14/4	2	
BN	short-label	if S = 1, PC ← PC + ext-disp8	0 1 1 1 1 0 0 0	14/4	2	
BP	short-label	if S = 0, PC ← PC + ext-disp8	0 1 1 1 1 0 0 1	14/4	2	
BPE	short-label	if P = 1, PC ← PC + ext-disp8	0 1 1 1 1 0 1 0	14/4	2	
BPO	short-label	if P = 0, PC ← PC + ext-disp8	0 1 1 1 1 0 1 1	14/4	2	
BLT	short-label	if S XOR V = 1, PC ← PC + ext-disp8	0 1 1 1 1 1 0 0	14/4	2	
BGE	short-label	if S XOR V = 0, PC ← PC + ext-disp8	0 1 1 1 1 1 0 1	14/4	2	
BLE	short-label	if (S XOR V) OR Z = 1, PC ← PC + ext-disp8	0 1 1 1 1 1 1 0	14/4	2	
BGT	short-label	if (S XOR V) OR Z = 0, PC ← PC + ext-disp8	0 1 1 1 1 1 1 1	14/4	2	
DBNZNE	short-label	CW ← CW − 1 if Z = 0 and CW ≠ 0, PC ← PC + ext-disp8	1 1 1 0 0 0 0 0	14/5	2	
DBNZE	short-label	CW ← CW − 1 if Z = 1 and CW ≠ 0, PC ← PC + ext-disp8	1 1 1 0 0 0 0 1	14/5	2	
DBNZ	short-label	CW ← CW − 1 if CW ≠ 0, PC ← PC + ext-disp8	1 1 1 0 0 0 1 0	13/5	2	
BCWZ	short-label	if CW = 0, PC ← PC + ext-disp8	1 1 1 0 0 0 1 1	13/5	2	
		Interrupt Instructions				
BRK	3	(SP − 1, SP − 2) ← PSW, (SP − 3, SP − 4) ← PS, (SP − 5, SP − 6) ← PC, SP ← SP − 6 IE ← 0, BRK ← 0 PS ← (15, 14), PC ← (13, 12)	1 1 0 0 1 1 0 0	50	1	
	imm8 (≠ 3)	(SP − 1, SP − 2) ← PSW, (SP − 3, SP − 4) ← PS, (SP − 5, SP − 6) ← PC, SP ← SP − 6 IE ← 0, BRK ← 0 PC ← (n x 4, + 1, n x 4) PS ← (n x 4 + 3, n x 4 + 2) n = imm8	1 1 0 0 1 1 0 1	50	2	

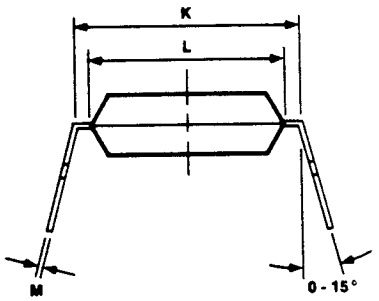
Mnemonic	Operand	Operation	Operation Code 7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	No. of Clocks	No. of Bytes	Flags AC	CY	V	P	S	Z
Interrupt Instructions (cont)												
BRKV		When V = 1 (SP − 1, SP − 2) ← PSW, (SP − 3, SP − 4) ← PS, (SP − 5, SP − 6) ← PC, SP ← SP − 6 IE ← 0, BRK ← 0 PS ← (19, 18), PC ← (17, 16)	1 1 0 0 1 1 1 0		52/3	1						
RETI		PC ← (SP + 1, SP), PS ← (SP + 3, SP + 2), PSW ← (SP + 5, SP + 4), SP ← SP + 6	1 1 0 0 1 1 1 1		39	1	R	R	R	R	R	R
CHKIND	reg16, mem32	When (mem32) > reg16 or (mem32 + 2) < reg16 (SP − 1, SP − 2) ← PSW, (SP − 3, SP − 4) ← PS, (SP − 5, SP − 6) ← PC, SP ← SP − 6 IE ← 0, BRK ← 0, PS ← (23, 22), PC ← (21, 20)	0 1 1 0 0 0 1 0	mod reg mem	73-76/ 26	2-4						
BRKEM	imm8	(SP − 1, SP − 2) ← PSW, (SP − 3, SP − 4) ← PS, (SP − 5, SP − 6) ← PC, SP ← SP − 6 MD ← 0, PC ← (n x 4 + 1, n x 4), MD Bit Write Enable PS ← (n x 4 + 3, n x 4 + 2), n = imm8	0 0 0 0 1 1 1 1	1 1 1 1 1 1 1 1	50	3						
CPU Control Instructions												
HALT		CPU Halt	1 1 1 1 0 1 0 0		2	1						
BUSLOCK		Bus Lock Prefix	1 1 1 1 0 0 0 0		2	1						
FPO1	fp-op	No Operation	1 1 0 1 1 X X X	1 1 Y Y Y Z Z Z	2	2						
	fp-op, mem	data bus ← (mem)	1 1 0 1 1 X X X	mod Y Y Y mem	15	2-4						
FPO2	fp-op	No Operation	0 1 1 0 0 1 1 X	1 1 Y Y Y Z Z Z	2	2						
	fp-op, mem	data bus ← (mem)	0 1 1 0 0 1 1 X	mod Y Y Y mem	15	2-4						
POLL		Poll and wait	1 0 0 1 1 0 1 1	n: number of times POLL pin is sampled	2 + 5n	1						
NOP		No Operation	1 0 0 1 0 0 0 0		3	1						
DI		IE ← 0	1 1 1 1 1 0 1 0		2	1						
EI		IE ← 1	1 1 1 1 1 0 1 1		2	1						
8080 Mode Instructions												
RETEM		PC ← (SP + 1, SP), PS ← (SP + 3, SP + 2), PSW ← (SP + 5, SP + 4), SP ← SP + 6, MD Bit Write Disable	1 1 1 0 1 1 0 1	1 1 1 1 1 1 0 1	39	2	R	R	R	R	R	R
CALLN	imm8	(SP − 1, SP − 2) ← PSW, (SP − 3, SP − 4) ← PS, (SP − 5, SP − 6) ← PC, SP ← SP − 6 MD ← 1, PC ← (n x 4 + 1, n x 4) PS ← (n x 4 + 3, n x 4 + 2), n = imm8	1 1 1 0 1 1 0 1	1 1 1 1 0 1 1 0 1	58	3						

Packaging Information

40-Pin Plastic DIP Package (600 mil)







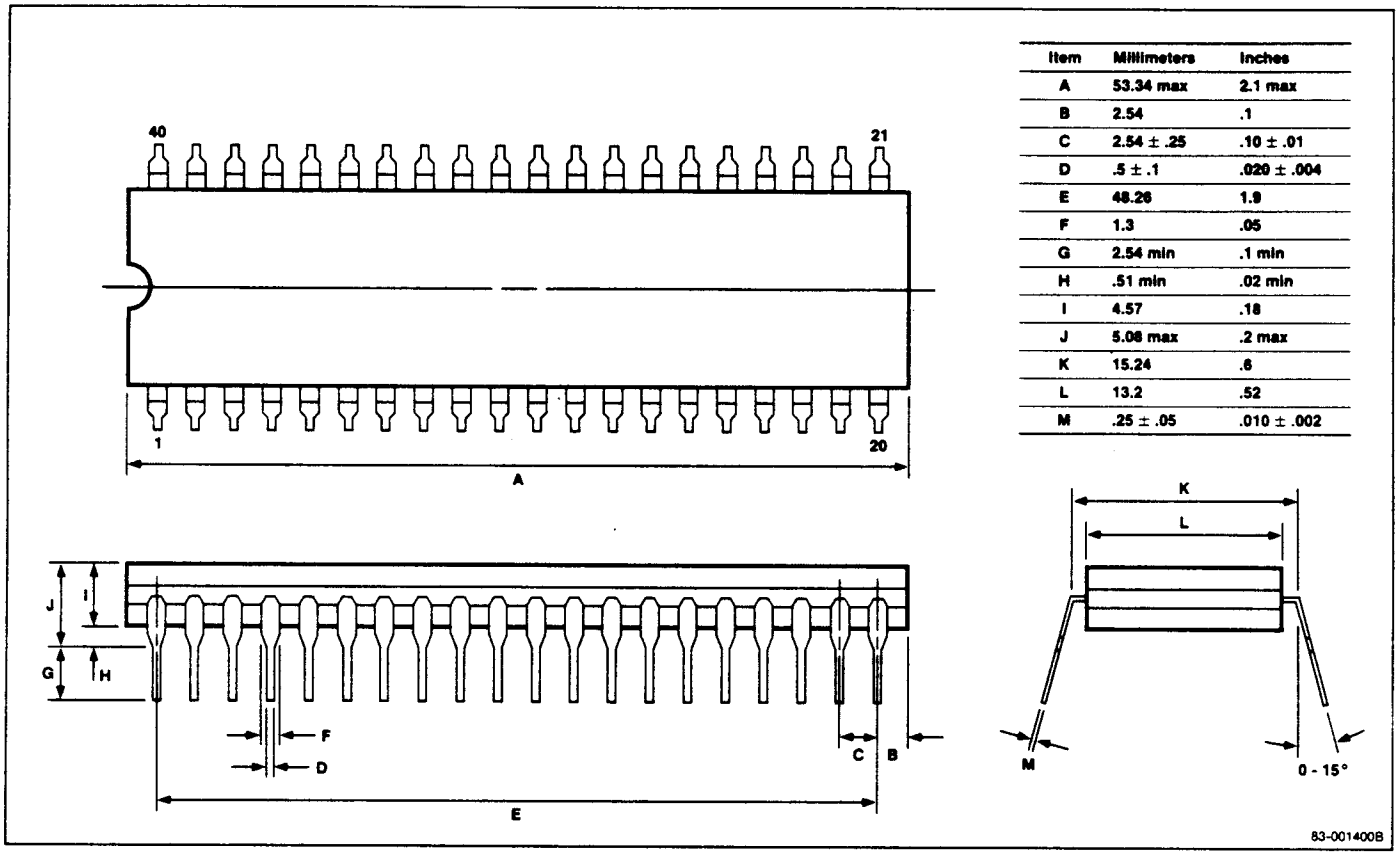
Item	Millimeters	Inches
A	53.34 max	2.1 max
B	2.54 max	.10 max
C	2.54 [T.P.]	.10 [T.P.]
D	.5 ± .10	.02 $^{+.004}_{-.005}$
E	48.26 ± .1	1.9 ± .004
F	1.2 min	.047 min
G	3.6 ± 0.3	.142 ± .012
H	.51 min	.02 min
I	4.31 max	.17 max
J	5.72 max	.226 max
K	15.24 [T.P.]	.60 [T.P.]
L	13.2	.52
M	.25 $^{+.10}_{-.05}$	.01 $^{+.004}_{-.003}$
N	.25	.01

Notes: 1. Each lead centerline is located within 0.25 mm [0.01 inch] of its true position [T.P.] at maximum material condition.
2. Item "K" to center of leads when formed parallel.

83-001399B

Packaging Information (cont)

40-Pin Cerdip Package



Item	Millimeters	Inches
A	53.34 max	2.1 max
B	2.54	.1
C	2.54 ± .25	.10 ± .01
D	.5 ± .1	.020 ± .004
E	48.26	1.9
F	1.3	.05
G	2.54 min	.1 min
H	.51 min	.02 min
I	4.57	.18
J	5.08 max	.2 max
K	15.24	.6
L	13.2	.52
M	.25 ± .05	.010 ± .002

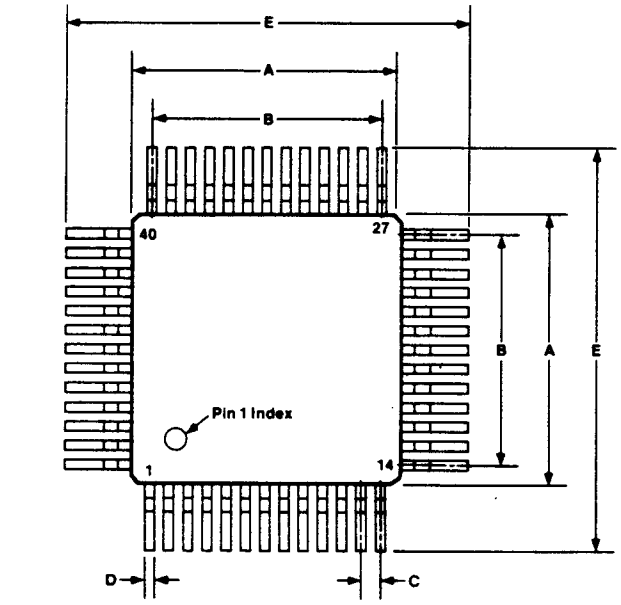
Packaging Information (cont)

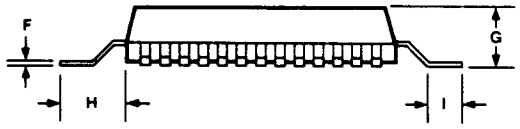
44-Pin Plastic Leadless Chip Carrier (PLCC) Package

(Information available in first quarter of 1986.)

Packaging Information (cont)

52-Pin Plastic Miniflat Package





Item	Millimeters	Inches
A	$14.0^{+.3}_{-.2}$	$.551^{+.012}_{-.008}$
B	12.0 ±.3	.472 ±.012
C	1.00 ±.15	.039 ±.006
D	$.4^{+.2}_{-.1}$	$.016^{+.008}_{-.004}$
E	21.0 ±.4	.827 ±.016
F	$.15^{+.10}_{-.05}$	$.006^{+.004}_{-.002}$
G	2.8 max	.110 max
H	3.3 ±.2	.130 ±.008
I	2.2 ±.2	.087 ±.008

83-001875B

Notes:

REGIONAL SALES AND ENGINEERING SUPPORT OFFICES

NORTHEAST
Twenty Burlington Mall Road, Suite 449
Burlington, MA 01803
TEL 617-272-1774 TWX 710-348-6515

SOUTHEAST
Radice Corporate Center
600 Corporate Drive, Suite 412
Fort Lauderdale, FL 33334
TEL 305-776-0682 TWX 759839

MIDWEST
3025 West Salt Creek Lane, Suite 300
Arlington Heights, IL 60005
TEL 312-577-9090 TWX 910-687-1492

SOUTHCENTRAL
16475 Dallas Parkway, Suite 380
Dallas, TX 75248
TEL 214-931-0641 TWX 910-860-5284

SOUTHWEST
200 East Sandpointe, Building 8 Suite 460
Santa Ana, CA 92707
TEL 714-546-0501 TWX 759845

NORTHWEST
10080 North Wolfe Road, SW3 Suite 360
Cupertino, CA 95014
TEL 408-446-0650 TLX 595497

DISTRICT OFFICES

200 Broadhollow Road, Suite 302
Route 110
Melville, NY 11747
TEL 516-423-2500 TWX 510-224-6090

Beechwood Office Park
385 South Road
Poughkeepsie, NY 12601
TEL 914-452-4747 TWX 510-248-0066

200 Perinton Hills Office Park
Fairport, NY 14450
TEL 716-425-4590 TWX 510-100-8949

5720 Peachtree Parkway, Suite 120
Norcross, GA 30092
TEL 404-447-4409 TWX 910-997-0450

7257 Parkway Drive, Suite 109
Hanover, MD 21076
TEL 301-796-3944 TLX 759847

29200 Southfield Road, Suite 208
Southfield, MI 48076
TEL 313-559-4242 TWX 810-224-4625

Busch Corporate Center
6480 Busch Blvd., Suite 121
Columbus, OH 43229
TEL 614-436-1778 TWX 510-101-1771

8030 Cedar Avenue South, Suite 229
Bloomington, MN 55420
TEL 612-854-4443 TWX 910-997-0726

DISTRICT OFFICES [cont]

Echelon Building 2
9430 Research Boulevard, Suite 330
Austin, TX 78759
TEL 512-346-9280

6150 Canoga Avenue, Suite 112
Woodland Hills, CA 91367
TEL 818-716-1535 TWX 559210

Lincoln Center Building
10300 S.W. Greenburg Road, Suite 540
Portland, OR 97223
TEL 503-245-1600

5445 DTC Parkway, Suite 218
Englewood, CO 80111
TEL 303-694-0041 TWX 510-600-5666

NATICK TECHNOLOGY CENTER

One Natick Executive Park
Natick, MA 01760
TEL 617-655-8833 TWX 710-386-2110

NEC
NEC Electronics Inc.
CORPORATE HEADQUARTERS
401 Ellis Street
P.O. Box 7241
Mountain View, CA 94039
TEL 415-960-6000
TWX 910-379-6985

No part of this document may be copied or reproduced in any form or by any means without the prior written consent of NEC Electronics Inc. The information in this document is subject to change without notice. Devices sold by NEC Electronics Inc. are covered by the warranty and patent indemnification provisions appearing in NEC Electronics Inc. Terms and Conditions of Sale only. NEC Electronics Inc. makes no warranty, express, statutory, implied, or by description, regarding the information set forth herein or regarding the freedom of the described devices from patent infringement. NEC Electronics Inc. makes no warranty of merchantability or fitness for any purpose. NEC Electronics Inc. assumes no responsibility for any errors that may appear in this document. NEC Electronics Inc. makes no commitment to update or to keep current the information contained in this document.

©1985 NEC Electronics Inc./Printed in U.S.A.

NECEL-000010-1185
STOCK NO. 500635

OKI MSM82C51A UART Data Sheet

OKI

JUNE 1984

semiconductor

MSM82C51A
UNIVERSAL SYNCHRONOUS ASYNCHRONOUS RECEIVER TRANSMITTER

GENERAL DESCRIPTION

The MSM82C51A is a USART (Universal Synchronous Asynchronous Receiver Transmitter) for serial data communication for the microcomputer systems.

The MSM82C51A receives parallel data from the CPU and transmits serial data. This device also receives serial data and transmits parallel data to the CPU.

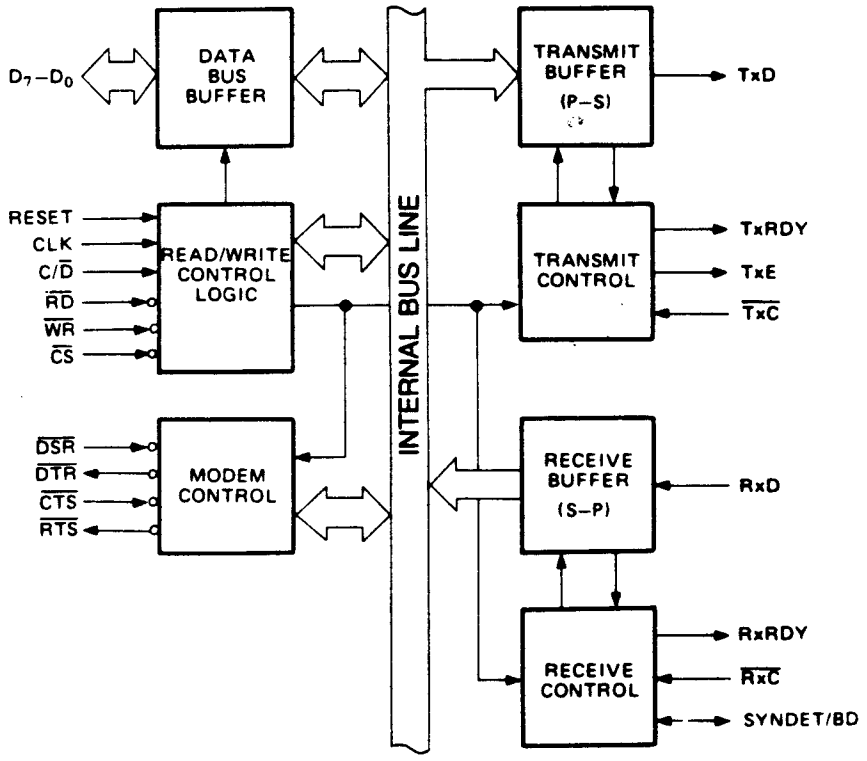
The MSM82C51A is a fully static circuit using silicon gate CMOS technology. It operates on an extremely low power supply at 100 μA (max) of standby current by suspending all the operations.

MSM82C51A is functionally compatible with the 8251A.

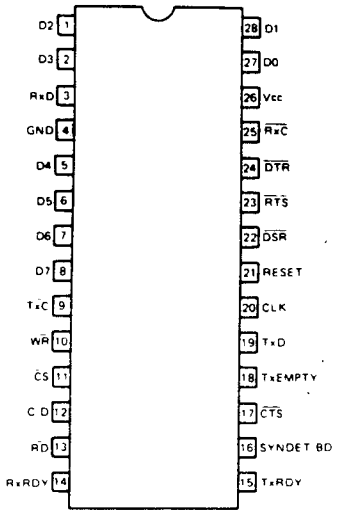
FEATURES

- Wide power supply voltage range from 3 V to 6 V.
- Wide temperature range from −40°C to 85°C.
- Synchronous communication upto 64K baud.
- Asynchronous communication upto 38.4K baud.
- Transmitting/receiving operations under double buffered configuration.
- Error detection (parity, overrun and framing)
- 28-pin DIP (MSM82C51ARS)
- 32-pin flat package (MSM82C51AGSK)

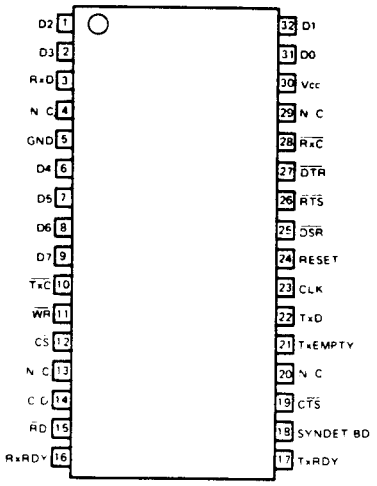
FUNCTIONAL BLOCK DIAGRAM



PIN CONFIGURATION



MSM82C51ARS (Top View)
28 Lead Plastic DIP



MSM82C51AGSK (Top View)
32 Lead Plastic Flat Package

FUNCTION

Outline
MSM82C51A's functional configuration is programmed by the software.

Operation between MSM82C51A and CPU is executed by program control. Table 1 shows the operation between CPU and the device.

Table 1 Operation between MSM82C51A and CPU

$\overline{CS}$	C/$\overline{D}$	$\overline{RD}$	$\overline{WR}$	
1	X	X	X	Data bus 3-state
0	X	1	1	Data bus 3-state
0	1	0	1	Status → CPU
0	1	1	0	Control word ← CPU
0	0	0	1	Data → CPU
0	0	1	0	Data ← CPU

It is necessary to execute a function-setting sequence after resetting on MSM82C51A. Fig. 1 shows the function-setting sequence.

If the function was set, the device is ready to receive a command, thus enabling the transfer of data by setting a necessary command, reading a status and reading/writing data.

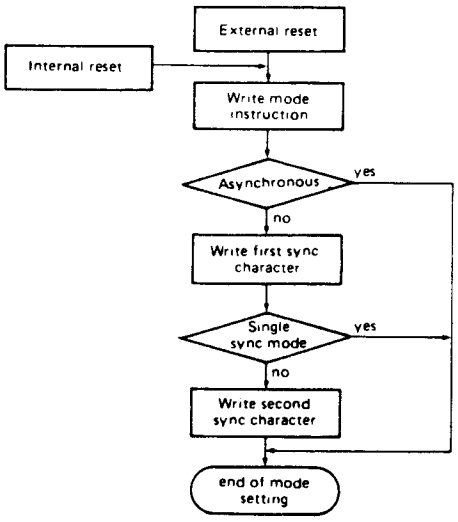


Fig. 1 Function-Setting Sequence (Mode Instruction Sequence)

Control Words
There are two types of control words

1. Mode instruction
2. Command

1. Mode Instruction
Mode instruction is used for setting the function of the MSM82C51A. Mode instruction will be in "wait for write" at either internal reset or external reset. Thus writing a control word after resetting will be recognized as "mode instruction."

Items to be set by mode instruction are as follows:

- Synchronous/asynchronous mode
- Stop bit length (asynchronous mode)
- Character length
- Parity bit
- Baud rate factor (asynchronous mode)
- Internal/external synchronization (synchronous mode)
- No. of synchronous characters (synchronous mode)

The bit configuration of the mode instruction is shown in Figs. 2 and 3. In the case of synchronous mode, it is necessary to write one- or two-sync characters.

If sync characters were written, a function will be set because the writing of sync characters constitutes part of the mode instruction.

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
S_2	S_1	EP	PEN	L_2	L_1	B_2	B_1

Baud rate factor

0	1	0	1
0	0	1	1
*Sync Mode	1x	16x	64x

Character length

0	1	0	1
0	0	1	1
5 bits	6 bits	7 bits	8 bits

Parity check

0	1	0	1
0	0	1	1
Disable	Odd parity	Disable	Even parity

Stop bit length

0	1	0	1
0	0	1	1
Inhibit	1 bit	1.5 bits	2 bits

*Refer to Figure 3

Fig. 2 Bit Configuration of Mode Instruction (Asynchronous)

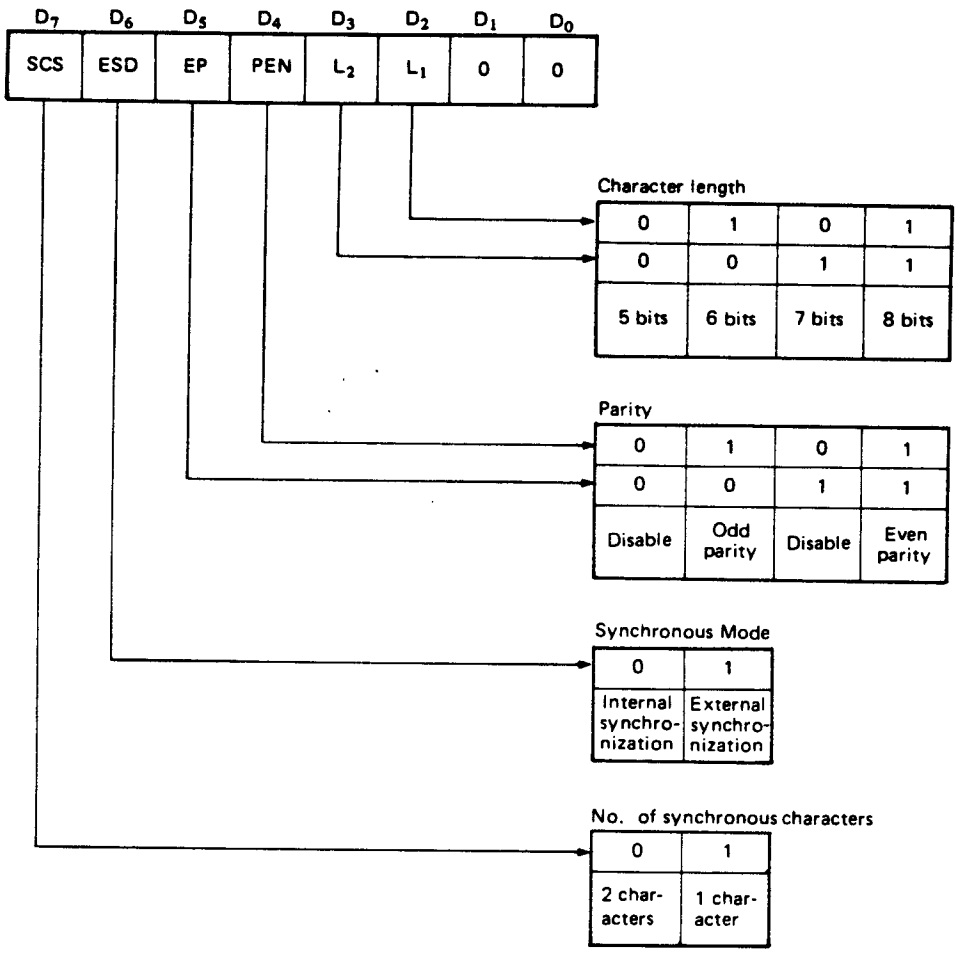


Fig. 3 Bit Configuration of Mode Instruction (Synchronous)

2. Command

The command word is used for setting the operation of MSM82C51A.

It is possible to write a command whenever necessary after writing mode instruction and sync characters.

Items to be set by command are as follows:

- Transmit Enable/Disable
- Receive Enable/Disable
- $\overline{DTR}$, $\overline{RTS}$ Output of data.
- Resetting of error flag.
- Sending of break characters
- Internal reset
- Hunt mode (synchronous mode)

The bit configuration of a command is shown in Fig. 4.

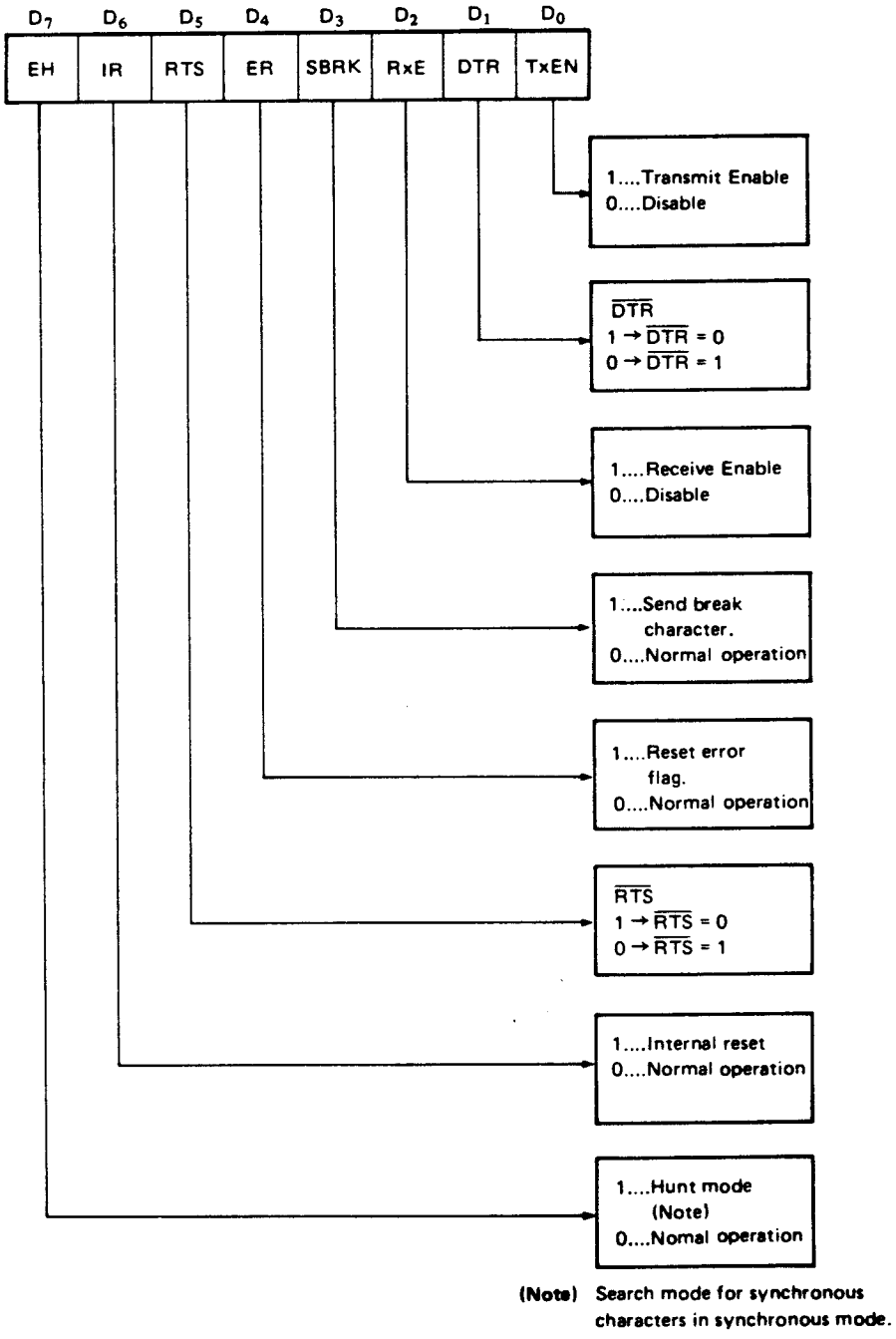


Fig. 4 Bit Configuration of Command

Status Word
It is possible to see the internal status of MSM82C51A by reading a status word.
The bit configuration of status word is shown in Fig. 5.

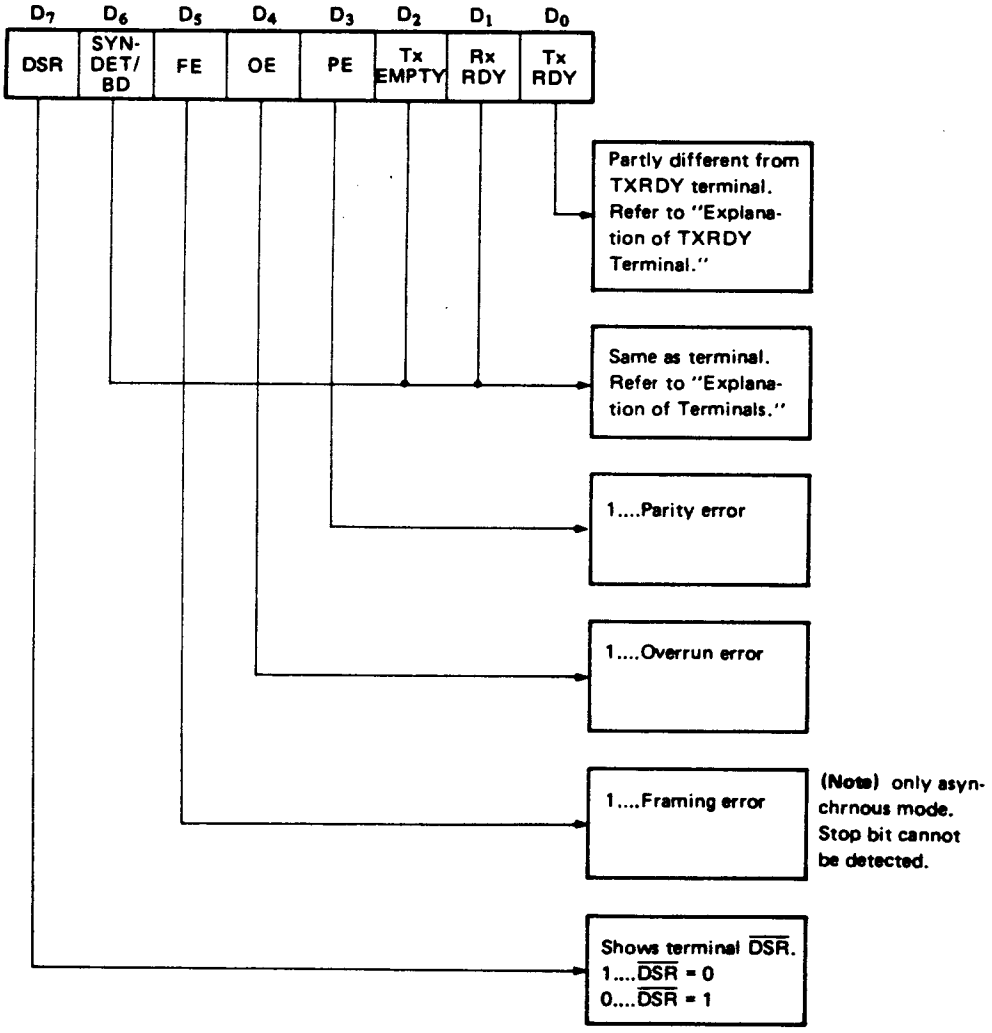


Fig. 5 Bit Configuration of Status Word

Standby Mode

It is possible to put MSM82C51A in "standby mode" for the complete static configuration of CMOS.

It is when the following conditions have been satisfied that MSM82C51A is in "standby mode."

(1) $\overline{CS}$ terminal shall be fixed at Vcc level.

(2) Input pins other than $\overline{CS}$, D0 to D7, $\overline{RD}$, $\overline{WR}$ and C/$\overline{D}$ shall be fixed at Vcc or GND level (including SYNDET in external synchronous mode).

Note: When all outputs current are low, ICCS specification applies.

Explanation of Each Terminal

D0 to D7 (I/O terminal)
This is a bidirectional data bus which receives control words and transmits data from the CPU and sends the status words and received data to the CPU.

RESET (Inpt terminal)
A "High" on this input forces the MSM82C51A into "reset."

The device waits for the "mode instruction."

The min. reset width is six clock inputs.

CLK (Input terminal)
CLK signal is used to generate internal device timing.

CLK signal is independent of $\overline{RXC}$ or $\overline{TXC}$.

However, the frequency of CLK must be greater than 30 times the $\overline{RXC}$ and $\overline{TXC}$ at Synchronous mode and Asynchronous "×1" mode, and must be greater than 5 times at Asynchronous "×16" and "×64" mode.

$\overline{WR}$ (Input terminal)
This is "active low" input terminal which receives a signal for writing transmit data and control words from CPU into the MSM82C51A.

$\overline{RD}$ (Input terminal)
This is "active low" input terminal which receives a signal for reading receive data and status words from the MSM82C51A.

C/$\overline{D}$ (Input terminal)
This is an input terminal which receives a signal for selecting data or command word and status word when MSM82C51A is accessed by CPU.

If C/$\overline{D}$ = low, data will be accessed.

If C/$\overline{D}$ = high, command word or status word will be accessed.

$\overline{CS}$ (Input terminal)
This is "active low" input terminal which selects the MSM82C51A.

Note The device won't be in "standby mode" only setting $\overline{CS}$ = High. Refer to "Explanation of Standby Mode."

TXD (Output terminal)
This is an output for serial transmit data.

The device is in "mark state" (high level) after resetting or when transmit is disabled.

It is also possible to set the device in the "break state" (low level) by a command.

TXRDY (Output terminal)
This is an output which indicates that the MSM82C51A is ready to accept a transmit data character. But the terminal is always at low level if $\overline{CTS}$ = high or the device was set in "TX disable status" by a command.

Note TXRDY of the status word indicates that transmit data character is receivable, regardless of $\overline{CTS}$ or command.

If the CPU writes a data character, TXRDY will be reset by the leading edge of the $\overline{WR}$ signal.

TXEMPTY (Output terminal)
This is an output terminal which indicates that the MSM82C51A transmitted all the characters and has no data characters to send.

In "synchronous mode," the terminal is at high level, if transmit data characters are no longer left (sync characters are automatically transmitted).

If the CPU writes a data character, TXEMPTY will be reset by the leading edge of $\overline{WR}$ signal.

Note As transmitter is disabled by setting CTS "High" or command, data written prior to the transmitter being disabled will be sent out, then TXD and TXEMPTY will be "High." If data is written after the transmitter is disabled, that data is not sent out and TXE will be "High". After re-enabling the transmitter it will be sent. (Refer to Transmitter Control and Flag Timing Chart.)

$\overline{TXC}$ (Input terminal)
This is a clock input signal which determines the transfer speed of transmit data.

In "synchronous mode," the baud rate will be the same as the frequency of $\overline{TXC}$.

In "Asynchronous mode," it is possible to select baud rate factor by the mode instruction.

It can be 1, 1/16, or 1/64 the $\overline{TXC}$.

The falling edge of $\overline{TXC}$ shifts the serial data out of the MSM82C51A.

RXD (Input terminal)
This is a terminal which receives serial data.

RXRDY (Output terminal)
This is a terminal which indicates that MSM82C51A contains a character that is ready to be read.

If CPU reads a data character, RXRDY will be reset by the leading edge of the $\overline{RD}$ signal.

Unless the CPU reads a data character before the next character is received completely, the preceding data will be lost. In such a case, the overrun error flag of the status register will be set.

$\overline{RXC}$ (Input terminal)
This is a clock input signal which determines the transfer speed of the receiver.

In "synchronous mode," the baud rate will be the same as the frequency of $\overline{RXC}$.

In "asynchronous mode," is is possible to select baud rate factor by mode instruction.

It can be 1, 1/16, 1/64 the $\overline{RXC}$.

SYNDET/BD (Input or output terminal)
This is a terminal whose function changes according to the mode.

In "internal synchronous mode," this terminal is at high level, if sync characters are received and synchronized. If status word is read, the terminal will be reset.

In "external synchronous mode," this is an input terminal.

A "High" on this input forces the MSM82C51A to start receiving data characters.

In "asynchronous mode," this is an output terminal which generates a high output upon the detection of a "break" character, if the receiver data contained "low-level" space between stop bits of two continuous characters. The terminal will be reset, if RXD is at high level.

$\overline{DSR}$ (Input terminal)
This is an input port for MODEM interfaces. The input status of the terminal can be read by reading the status register.

$\overline{DTR}$ (Output terminal)
This is an output port for MODEM interfaces. It is possible to set the status of $\overline{DTR}$ by a command.

$\overline{CTS}$ (Input terminal)
This is an input terminal for MODEM interfaces which is used for controlling the transmission. The terminal controls data transmission if the device is set in "TX Enable" status by a command. Data is transmittable if the terminal is at low level.

$\overline{RTS}$ (Output terminal)
This is an output port for MODEM interfaces. It is possible to set the status of $\overline{RTS}$ by a command.

ABSOLUTE MAXIMUM RATINGS

Parameter	Symbol	Limits		Unit	Conditions
		MSM82C51ARS	MSM82C51AGS		
Power supply voltage	V_{CC}	−0.5 ~ +7		V	With respect to GND
Input voltage	V_{IN}	−0.5 ~ V_{CC} + 0.5		V	
Output voltage	V_{OUT}	−0.5 ~ V_{CC} + 0.5		V	
Storage temperature	T_{stg}	−55 ~ 150		°C	—
Power dissipation	P_D	0.9	0.7	W	Ta = 25°C

OPERATING RANGE

Parameter	Symbol	Limits	Unit
Power supply voltage	V_{CC}	3 ~ 6	V
Operating temperature	T_{OP}	−40 ~ 85	°C

RECOMMENDED OPERATING CONDITIONS

Parameter	Symbol	Min.	Typ.	Max.	Unit
Power supply voltage	V_{CC}	4.5	5	5.5	V
Operating temperature	T_{OP}	−40	+25	+85	°C
"L" input voltage	V_{IL}	−0.3		+0.8	V
"H" input voltage	V_{IH}	2.2		V_{CC} + 0.3	V

DC CHARACTERISTICS

(V_{CC} = 4.5 ~ 5.5V Ta = −40°C ~ +85°C)

Parameter	Symbol	Min.	Typ.	Max.	Unit	Measurement Conditions
"L" output voltage	V_{OL}			0.45	V	I_{OL} = 2 mA
"H" output voltage	V_{OH}	3.7			V	I_{OH} = −400 μA
Input leakage current	I_{LI}	−10		10	μA	$0 \le V_{IN} \le V_{CC}$
Output leakage current	I_{LO}	−10		10	μA	$0 \le V_{OUT} \le V_{CC}$
Operating supply current	I_{CCO}			5	mA	Asynchronous X64 during transmitting/receiving
Standby supply current	I_{CCS}			100	μA	All input voltage shall be fixed at V_{CC} or GND level.

AC CHARACTERISTICS

(V_{CC} = 4.5 ~ 5.5V, Ta = −40 ~ 85°C)

CPU Bus Interface Part

Parameter	Symbol	Min.	Max.	Unit	Remarks
Address stable before $\overline{RD}$	t_{AR}	20		NS	Note 2
Address hold time for $\overline{RD}$	t_{RA}	20		NS	Note 2
$\overline{RD}$ pulse width	t_{RR}	250		NS	
Data delay from $\overline{RD}$	t_{RD}		200	NS	
$\overline{RD}$ to data float	t_{DF}	10	100	NS	
Recovery time between $\overline{RD}$	t_{RVR}	6		T_{cy}	Note 5
Address stable before $\overline{WR}$	t_{AW}	20		NS	Note 2
Address hold time for $\overline{WR}$	t_{WA}	20		NS	Note 2
$\overline{WR}$ pulse width	t_{WW}	250		NS	
Data set-up time for $\overline{WR}$	t_{DW}	150		NS	
Data hold time for $\overline{WR}$	t_{WD}	20		NS	
Recovery time between $\overline{WR}$	t_{RVW}	6		T_{cy}	Note 4
RESET pulse width	t_{RESW}	6		T_{cy}	

Serial Interface Part

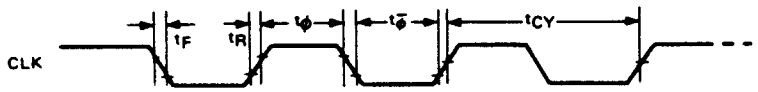
Parameter		Symbol	Min.	Max.	Unit	Remarks
Main clock period		t_{cy}	250		NS	Note 3
Clock low time		t_{ϕ}	90		NS	
Clock high time		t_{ϕ}	120	t_{cy}-90	NS	
Clock rise/fall time		t_R, t_F		20	NS	
TXD delay from falling edge of $\overline{TXC}$		t_{DTX}		1	μS	
Transmitter clock frequency	1X Baud	f_{TX}	DC	64	kHz	Note 3
	16X, Baud	f_{TX}	DC	615	kHz	
	64X, Baud	f_{TX}	DC	615	kHz	
Transmitter clock low time	1X Baud	t_{TPW}	13		T_{cy}	
	16X, 64X Baud	t_{TPW}	2		T_{cy}	
Transmitter clock high time	1X Baud	t_{TPD}	15		T_{cy}	
	16X, 64X Baud	t_{TPD}	3		T_{cy}	
Receiver clock frequency	1X Baud	f_{RX}	DC	64	kHz	Note 3
	16X Baud	f_{RX}	DC	615	kHz	
	64X Baud	f_{RX}	DC	615	kHz	
Receiver clock low time	1X Baud	t_{RPW}	13		T_{cy}	
	16X, 64X Baud	t_{RPW}	2		T_{cy}	
Receiver clock high time	1X Baud	t_{RPD}	15		T_{cy}	
	16X, 64X Baud	t_{RPD}	3		T_{cy}	
Time from the center of last bit to the rise of TXRDY		t_{TXRDY}		8	T_{cy}	
Time from the leading edge of $\overline{WR}$ to the fall of TXRDY		$t_{TXRDY\ CLEAR}$		400	NS	
Time from the center of last bit to the rise of RXRDY		t_{RXRDY}		26	T_{cy}	

Parameter	Symbol	Min.	Max.	Unit	Remarks
Time from the leading edge of $\overline{RD}$ to the fall of RXRDY	$t_{RXRDY\ CLEAR}$		400	NS	
Internal SYNDET delay time from rising edge of $\overline{RXC}$	t_{IS}		26	T_{cy}	
SYNDET setup time for $\overline{RXC}$	t_{ES}	18		T_{cy}	
TXE delay time from the center of last bit	$t_{TXEMPTY}$	20		T_{cy}	
MODEM control signal delay time from rising edge of $\overline{WR}$	t_{WC}	8		T_{cy}	
MODEM control signal setup time for falling edge of $\overline{RD}$	t_{CR}	20		T_{cy}	
RXD setup time for rising edge of $\overline{RXC}$ (1X Baud)	t_{RXDS}	11		T_{cy}	
RXD hold time for falling edge of $\overline{RXC}$ (1X Baud)	t_{RXDH}	17		T_{cy}	

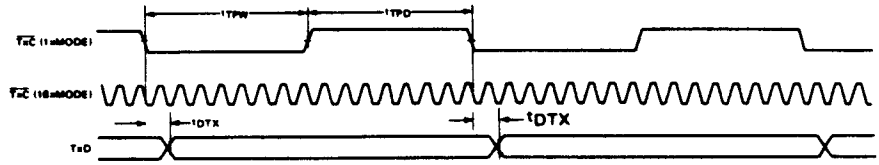
Caution 1) AC characteristics are measured at 150 pF capacity load as an output load based on 0.8V at low level and 2.2 V at high level for output and 1.5 V for input.
2) Addresses are $\overline{CS}$ and C/$\overline{D}$.
3) f_{TX} or $f_{RX} \leq 1/(30\ Tcy)$ 1 × baud
f_{TX} or $f_{RX} \leq (1/5\ Tcy)$ 16 ×, 64 × Baud
4) This recovery time is mode initialization only. Recovery time between command writes for Asynchronous Mode is 8 t_{CY} and for Synchronous Mode is 18 t_{CY}.
Write Data is allowed only when TXRDY = 1.
5) This recovery time is Status read only.
Read Data is allowed only when RXRDY = 1.
6) Status update can have a maximum delay of 28 clock periods from event affecting the status.

TIMING CHART

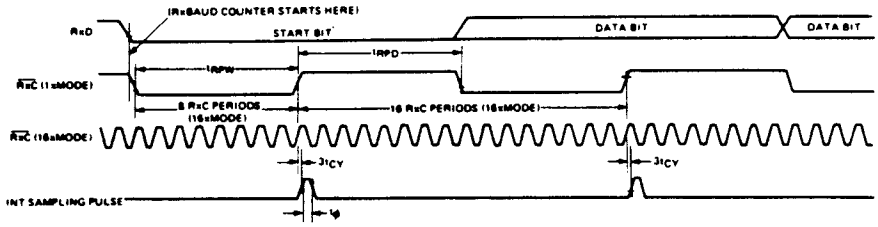
System Clock Input



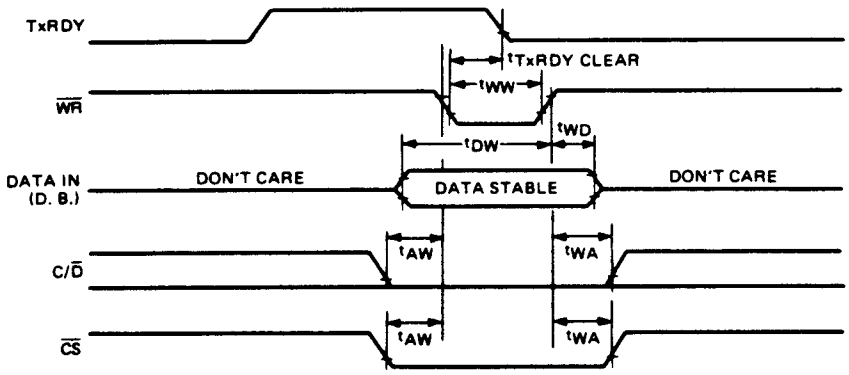
Transmitter Clock and Data



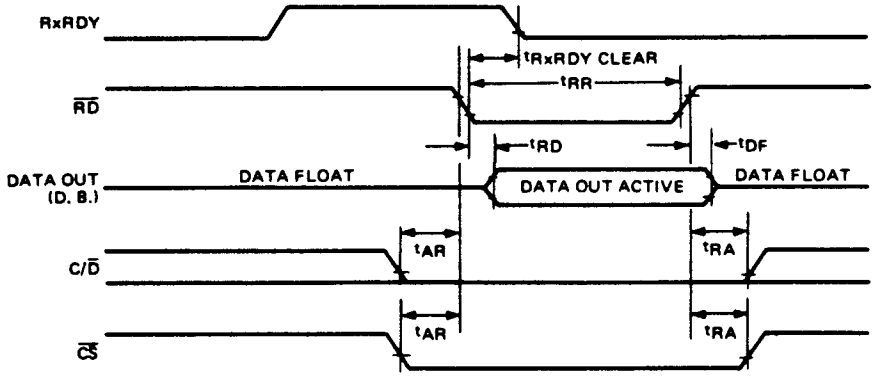
Receiver Clock and Data



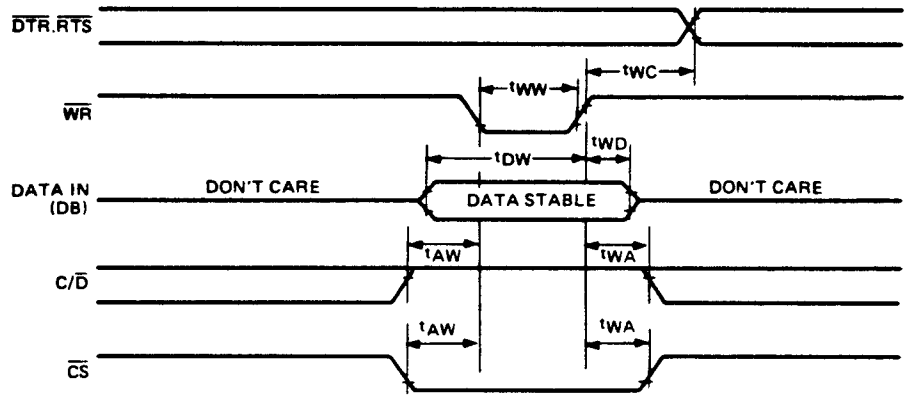
Write Data Cycle (CPU → USART)



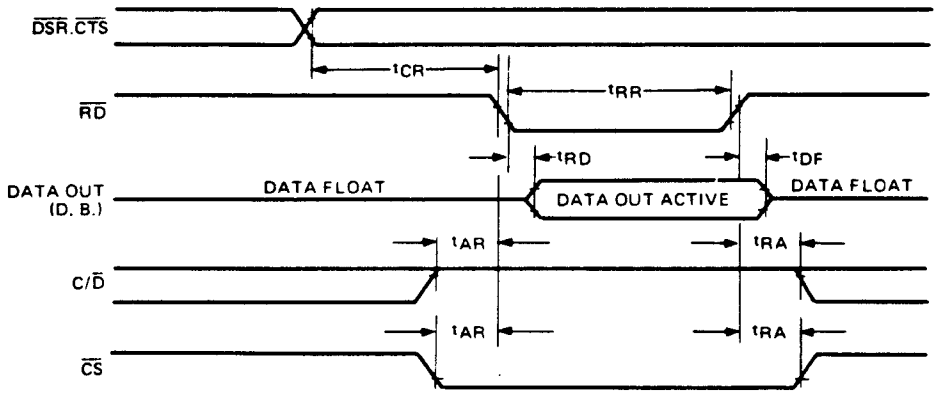
Read Data Cycle (CPU ← USART)



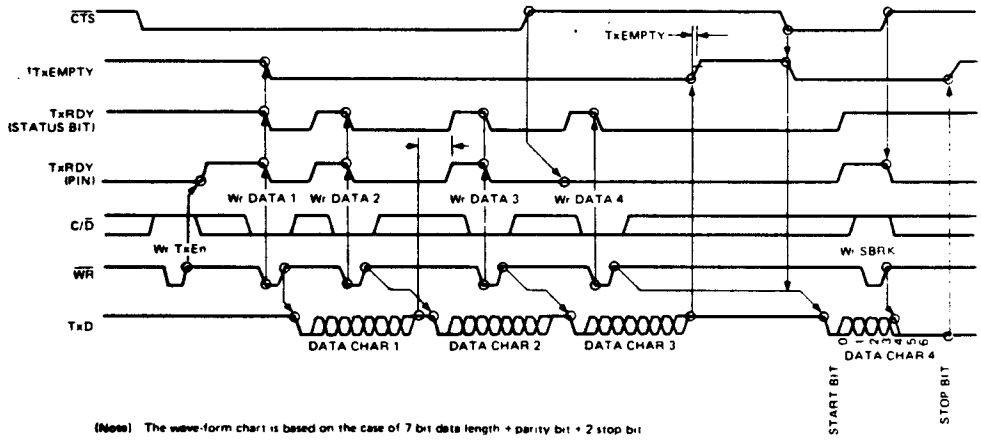
Write Control or Output Port Cycle (CPU → USART)



Read Control or Input Port (CPU ← USART)

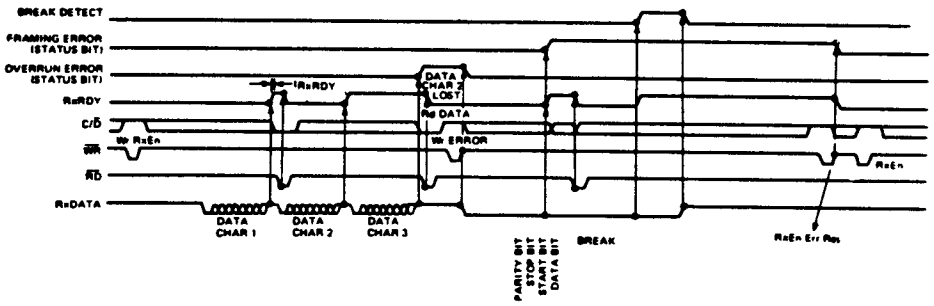


Transmitter Control and Flag Timing (ASYNC Mode)



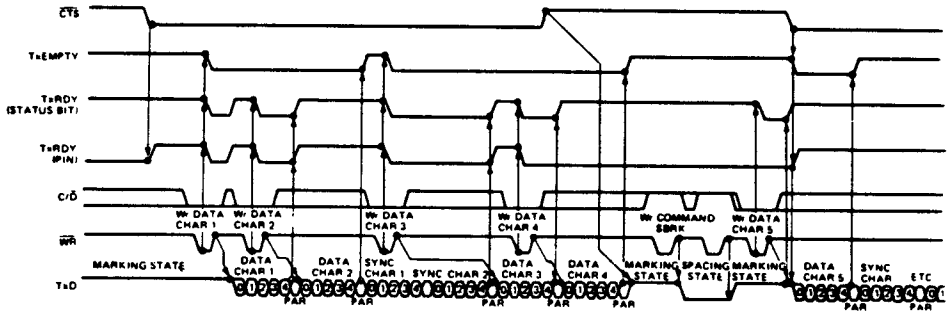
(Note) The wave-form chart is based on the case of 7 bit data length + parity bit + 2 stop bit

Receiver Control and Flag Timing (ASYNC Mode)



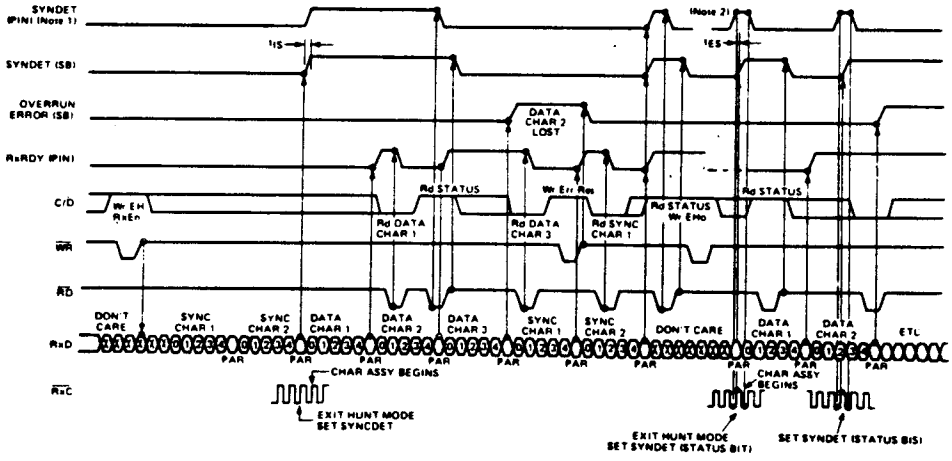
(Note) The wave-form chart is based on the case of 7 data bit length + parity bit + 2 stop bit

Transmitter Control and Flag Timing (SYNC Mode)



(Note) The wave-form chart is based on the case of 5 data bit length + parity bit and 2 synchronous characters

Receiver Control and Flag Timing (SYNC Mode)



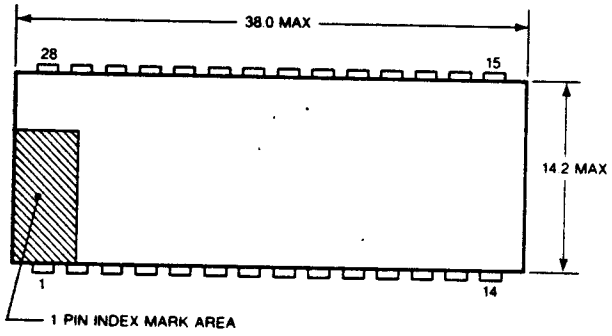
(Note 1) Internal synchronization is based on the case of 5 data bit length + parity bit and 2 synchronous characters

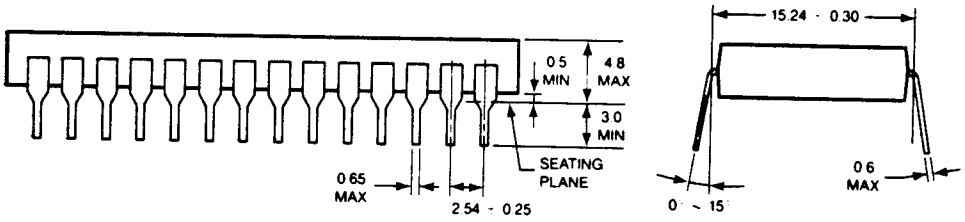
(Note 2) External synchronization is based on the case of 5 data bit length + parity bit

PACKAGE SPECIFICATIONS

MSM82C51ARS
28 LEAD PLASTIC DIP

(UNIT: mm)

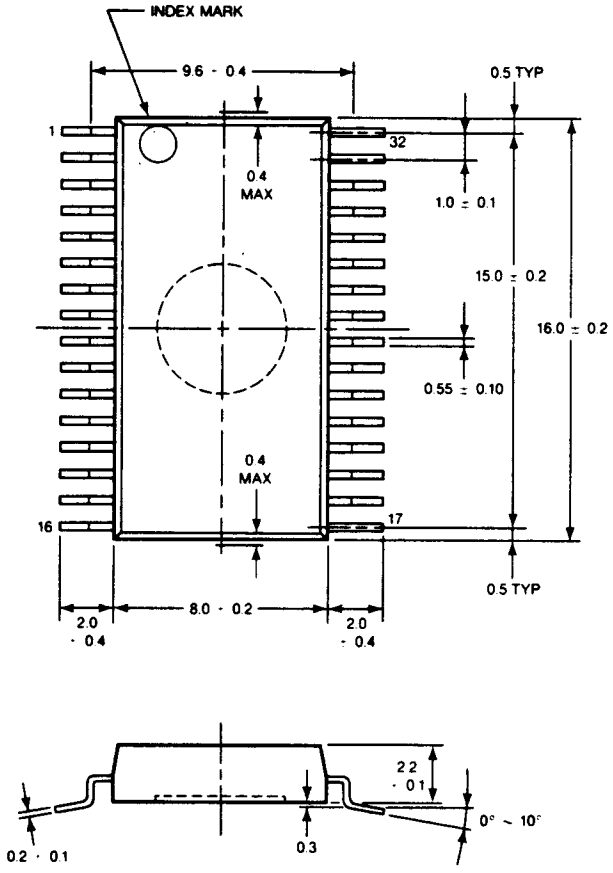




PACKAGE SPECIFICATIONS cont'd

MSM82C51AGSK
32 LEAD PLASTIC FLAT PACKAGE

(UNIT: mm)



OKI SEMICONDUCTOR, INC. 650 N. MARY AVENUE, SUNNYVALE, CA 94086

TELEPHONE: (408) 720-1900 TELEX (25) 910-3380508

OKI Semiconductor reserves the right to make changes in specifications at any time and without notice. The information furnished by OKI Semiconductor in this publication is believed to be accurate and reliable. However, no responsibility is assumed by OKI Semiconductor for its use; nor for any infringements of patents or other rights of third parties resulting from its use. No license is granted under any patents or patent rights of OKI.

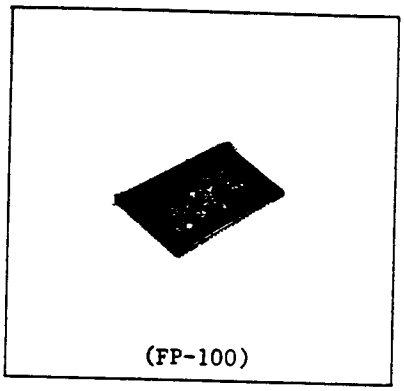
©1984 OKI SEMICONDUCTOR

PRINTED IN U.S.A.

Hitachi HD61102A LCD Column Driver Data Sheet

HD61102 (DOT MATRIX LIQUID CRYSTAL GRAPHIC DISPLAY COMMON DRIVER)

HD61102 is a column (segment) driver for dot matrix liquid crystal graphic display systems. It stores the display data transferred from a 8-bit micro-computer in the internal display RAM and generates dot matrix liquid crystal driving signals.



(FP-100)

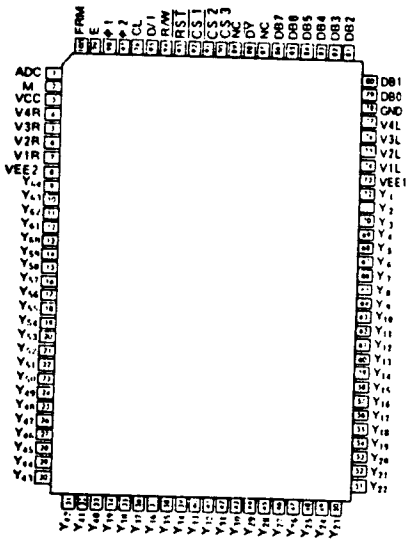
Each bit data of display RAM corresponds to ON/OFF of each dot of liquid crystal display to provide more flexible display.

As it is internally equipped with 64 output drivers for display, it is available for liquid crystal graphic display with many dots.

The HD61102, which is produced in the CMOS process, can accomplish a portable battery drive equipment by combining a CMOS micro-computer, utilizing the liquid crystal display's lower power dissipation.

Moreover it can facilitate dot matrix liquid crystal graphic display system configuration by combining the row (common) driver HD61103A.

■ PIN ARRANGEMENT



(Top view)

■ FEATURES

- Dot matrix liquid crystal graphic display column driver incorporating display RAM.
- RAM data direct display by internal display RAM
  RAM bit data "1" ..... ON
  RAM bit data "0" ..... OFF
- Internal display RAM address counter
  preset, increment
- Display RAM capacity ..... 512 bytes (4096 bits)
- 8-bit parallel interface
- Internal liquid crystal display driver circuit 64
- Display duty
  Combination of frame control signal and data latch synchronization signal make it possible to select out of static through an optional duty.
- Wide range of instruction function
  Display Data Read/Write, Display ON/OFF,
  Set address, Set Display Start line,
  Read Status
- Lower power dissipation ——— during display 2mW max
- Power supply Vcc ——— +5V ± 10%
  VEE ——— 0V ∿ -10V
- Liquid crystal display driving level——15.5V max
- CMOS process
- 100 - pin flat plastic package (FP-100)

■ ABSOLUTE MAXIMUM RATINGS

Item	Symbol	Value	Unit	Note
Supply voltage	V_{CC}	-0.3 ∿ +7.0	V	2
	V_{EE}	V_{CC} -16.5 ∿ V_{CC} +0.3	V	3
Terminal voltage (1)	V_{T1}	V_{EE} -0.3 ∿ V_{CC} +0.3	V	4
Terminal voltage (2)	V_{T2}	-0.3 ∿ V_{CC} +0.3	V	2, 5
Operating temperature	Topr	-20 ∿ +75	°C	
Storage temperature	Tstg	-55 ∿ +125	°C	

(Note 1) LSI's may be destroyed for ever, if being used beyond the absolute maximum ratings.
In ordinary operation, it is desirable to use them observing the recommended operation conditions.
Using beyond these conditions may cause malfunction and poor reliability.

(Note 2) All voltage values are referred to GND=0V.

(Note 3) Apply the same supply voltage to V_{EE} 1 and V_{EE}2.

(Note 4) Applies to V1L, V2L, V3L, V4L, V1R, V2R, V3R and V4R.
Maintain

$$Vcc \geq V1L=V1R \geq V3L=V3R \geq V4L=V4R \geq V2L=V2R \geq VEE$$

(Note 5) Applies to M, FRM, CL, $\overline{RST}$, ADC, ϕ1, ϕ2, $\overline{CS1}$, $\overline{CS2}$, CS3, E, R/W, D/I, ADC and DB0∿7.

■ ELECTRICAL CHARACTERISTICS
(GND=0V, VCC=4.5 ∿ 5.5V, VEE=0∿-10V, Ta=-20∿+75°C)

Item	Symbol	Test condition	Limit			Unit
			min	typ	max	
Input "High" voltage	V_{IHC}		0.7×Vcc	-	Vcc	V
	V_{IHT}		2.0	-	Vcc	V
Input "Low" voltage	V_{ILC}		0	-	0.3×Vcc	V
	V_{ILT}		0	-	0.8	V
Output "High" voltage	V_{OH}	IOH=-205μA	2.4	-	-	V
Output "Low" voltage	V_{OL}	IOL=1.6mA	-	-	0.4	V
Input leakage current	I_{IL}	Vin=GND∿Vcc	-1.0	-	+1.0	μA
Three state (OFF) input current	I_{TSL}	Vin=GND∿Vcc	-5.0	-	+5.0	μA
Liquid crystal supply leakage current	I_{LSL}	Vin=VEE∿Vcc	-2.0	-	+2.0	μA
Driver ON resistance	R_{ON}	Vcc-VEE=15V $\pm I_{LOAD}$=0.1mA	-	-	7.5	KΩ
Dissipation current	Icc(1)	During display	-	-	100	μA
	Icc(2)	During access access cycle=1MHz	-	-	500	μA

(Note 1) Applies to M, FRM, CL, $\overline{RST}$, ADC, ADC, φ1 and φ2.

(Note 2) Applies to $\overline{CS1}$, $\overline{CS2}$, CS3, E, R/W, D/I and DB0 ∿ 7.

(Note 3) Applies to DB0 ∿ 7.

(Note 4) Applies to terminals except for DB0 ∿ 7.

(Note 5) Applies to DB0 ∿ 7 at high impedance.

(Note 6) Applies to V1L ∿ V4L and V1R ∿ V4R.

(Note 7) Applies to Y1 ∿ Y64.

(Note 8) Specified when liquid crystal display is in 1/64 duty.
Operation frequency f_{CLK}=250 kHz (φ1 and φ2 frequency)
Frame frequency f_M =70 Hz (FRM frequency)

Specified in the state of
Output terminal ----- not loaded
Input level --------- V_{IH}=Vcc(V)
V_{IL}=GND (V)

Measured at Vcc terminal

● INTERFACE AC CHARACTERISTICS

(1) MPU Interface

(GND=0V, Vcc=4.5 ∿ 5.5V, V_{EE}=0 ∿ -10V, Ta=-20 ∿ +75°C)

Item	Symbol	min	typ	max	Unit	Note
E cycle time	t_{CYC}	1000	-	-	ns	1, 2
E high level width	P_{WEH}	450	-	-	ns	1, 2
E low level width	P_{WEL}	450	-	-	ns	1, 2
E rise time	tr	-	-	25	ns	1, 2
E fall time	tf	-	-	25	ns	1, 2
Address setup time	t_{AS}	140	-	-	ns	1, 2
Address hold time	t_{AH}	10	-	-	ns	1, 2
Data setup time	t_{DSW}	200	-	-	ns	1
Data delay time	t_{DDR}	-	-	320	ns	2, 3
Data hold time (Write)	t_{DHW}	10	-	-	ns	1
Data hold time (Read)	t_{DHR}	20	-	-	ns	2

(Note 1)

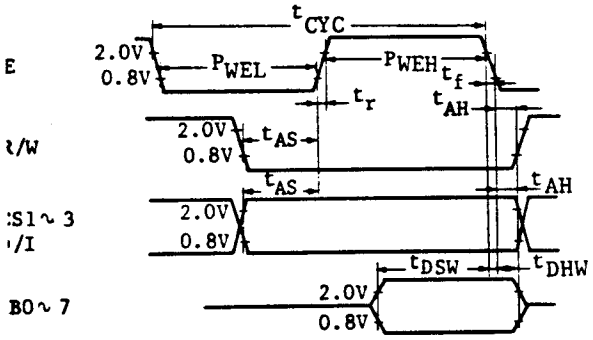


Fig. 1 CPU Write Timing

(Note 2)

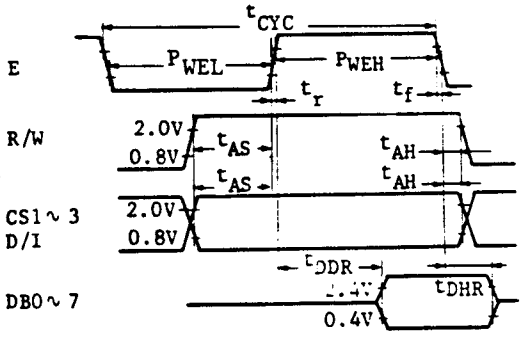
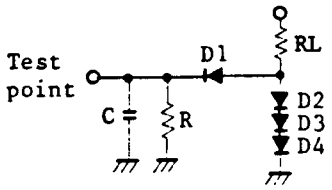


Fig. 2 CPU Read Timing

(Note 3) DB0 ∿ 7 : load circuit



RL=2.4KΩ

R =11KΩ

C =130pF (including jig capacity)

Diodes D1 to D4 are all 1S2074Ⓗ.

(2) Clock Timing

(GND=0V, Vcc=4.5 ∿ 5.5V, V_{EE}=0 ∿ -10V, Ta=20 ∿ +75°C)

Item	Symbol	Test condition	Limit			Unit
			min	typ	max	
φ1, φ2 cycle time	tcyc	Fig. 3	2.5	-	20	μs
φ1 "Low" level width	$t_{WL\phi1}$	Fig. 3	625	-	-	ns
φ2 "Low" level width	$t_{WL\phi2}$	Fig. 3	625	-	-	ns
φ1 "High" level width	$t_{WH\phi1}$	Fig. 3	1875	-	-	ns
φ2 "High" level width	$t_{WH\phi2}$	Fig. 3	1875	-	-	ns
φ1-φ2 phase difference	tD12	Fig. 3	625	-	-	ns
φ2-φ1 phase difference	tD21	Fig. 3	625	-	-	ns
φ1, φ2 rise time	tr	Fig. 3	-	-	150	ns
φ1, φ2 fall time	tf	Fig. 3	-	-	150	ns

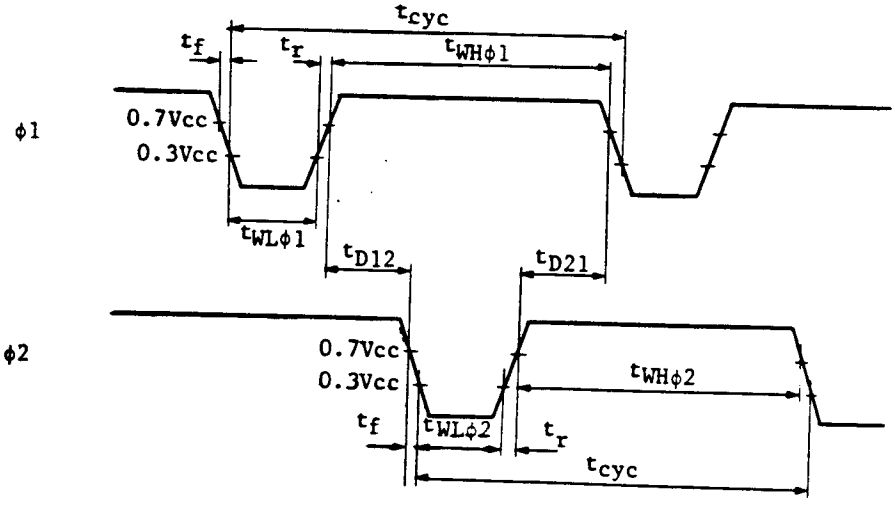


Fig. 3 External Clock Waveform

(3) Display Control Timing

(GND=0V, V_{CC}=4.5 ∿ 5.5V, V_{EE}=0 ∿ -10V, Ta=-20 ∿ +75°C)

Item	Symbol	Test condition	Limit			Unit
			min	typ	max	
FRM delay time	t_{DFRM}	Fig. 4	-2	-	+2	µs
M delay time	t_{DM}	Fig. 4	-2	-	+2	µs
CL "Low" level width	t_{WLCL}	Fig. 4	35	-	-	µs
CL "High" level width	t_{WHCL}	Fig. 4	35	-	-	µs

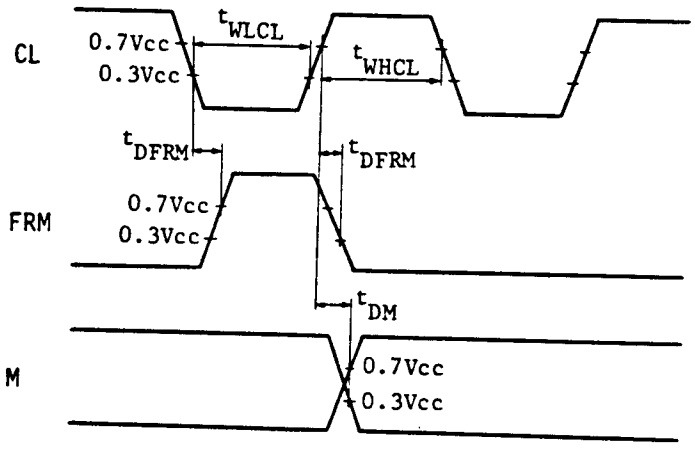
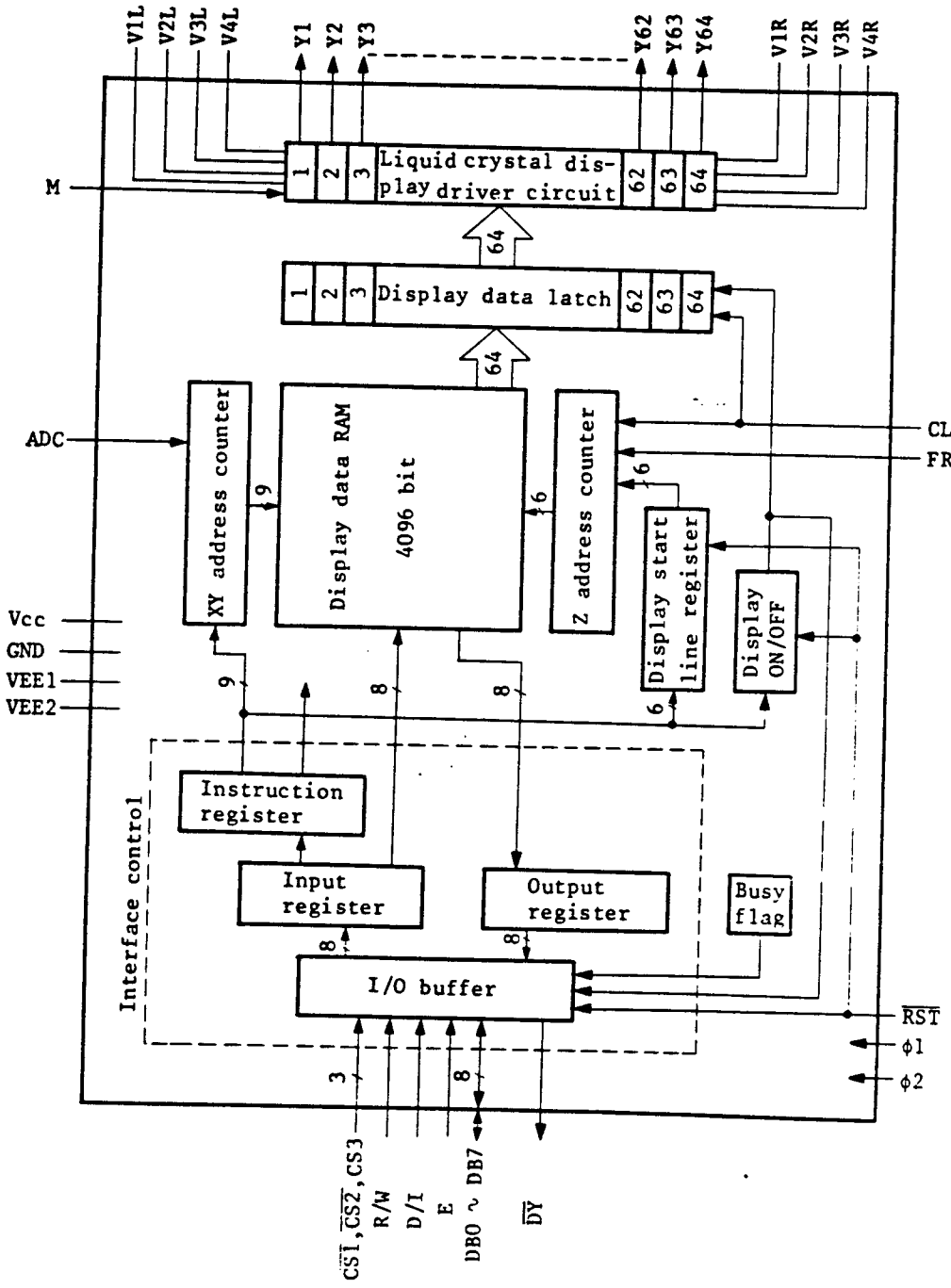


Fig. 4 Display Control Signal Waveform

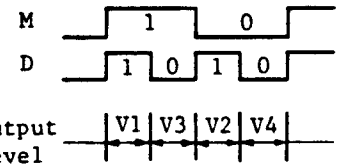
■ BLOCK DIAGRAM



■ TERMINAL FUNCTIONS

Terminal name	Number of terminals	I/O	Connected to	Functions
V_{CC} GND	2		Power supply	Power supply for internal logic. Recommended voltage is GND = 0V V_{CC} = +5V ± 10%
V_{EE} 1 V_{EE} 2	2		Power supply	Power supply for liquid crystal display drive circuit. Recommended power supply voltage is V_{CC} - 15 to GND. Connect the same power supply to V_{EE1} and V_{EE2}. V_{EE1} and V_{EE2} are not connected each other in the LSI.
V1L, V1R V2L, V2R V3L, V3R V4L, V4R	8		Power supply	Power supply for liquid crystal display drive. Apply the voltage specified depending on liquid crystals within the limit of V_{EE} through V_{CC}. V1L(V1R), V2L(V2R)---Selection level V3L(V3R),V4L(V4R)----Non-selection level Power supplies connected with V1L and V1R (V2L & V2R, V3L & V3R, V4L & V4R) should have the same voltages.
$\overline{CS1}$ $\overline{CS2}$ CS3	3	I	MPU	Chip selection. Data can be input or output when the terminals are in the next conditions. Terminal name: $\overline{CS1}$ $\overline{CS2}$ CS3 Condition: 'L' 'L' 'H'
E	1	I	MPU	Enable At write(R/W=L) : Data of DB0 to D[illegible] latched at the [illegible] E. At read(R/W=H) : Data appears at DB0 to DB7 while E is in "High" level.

Terminal name	Number of terminals	I/O	Connected to	Functions
R/W	1	I	MPU	Read/Write R/W=H : Data appears at DB0 to DB7 and can be read by the CPU When E=H, $\overline{CS1}$, $\overline{CS2}$=L and CS3=H. R/W=L : DB0 to DB7 can accept at fall of E when $\overline{CS1}$, $\overline{CS2}$=L and CS3=H.
D/I	1	I	MPU	Data/Instruction D/I=H : Indicates that the data of DB0 to DB7 is display data. D/I=L : Indicates that the data of DB0 to DB7 is display control data.
ADC	1	I	V_{CC}/GND	Adress control signal determine the relation between Y address of display RAM and terminals from which the data is output. ADC=H : Y1-$0, Y64-$63 ADC=L : Y64-$0, Y1-$63
DB0~DB7	8	I/O	MPU	Data bus, three-state I/O common terminal
M	1	I	HD61103A	Switch signal to convert liquid crystal drive waveform into AC.
FRM	1	I	HD61103A	Display synchronous signal (frame signal) This signal presets the 6-bit display line counter and synchronizes a common signal with the frame timing when the FRM signal becomes high.
CL	1	I	HD61103A	Syncronous signal to latch display data. The CL signal indicates to count up the display output adress counter and latch the display data at rising.
ϕ1,ϕ2	1	I	HD61103A	2-phase clock signal for internal operation. The ϕ1 and ϕ2 clocks are used to perform the operations (I/O of display data and execution of instructions) other than display.

Terminal name	Number of terminals	I/O	Connected to	Functions
Y1~Y64	64	O	Liquid criptal display	Liquid crystal display column (segment) drive output. These pins outputs light ON level when "1" is in the display RAM, and light OFF level with "0" in it. Relation among output level, M and display data (D) is as follows. 
$\overline{RST}$	1	I	CPU or external CR	The following registers can be initialized by setting the $\overline{RST}$ signal to "Low" level. (1) ON/OFF register 0 set (display OFF) (2) Display start line register 0 line set (displays from 0 line) After releasing reset, this condition can be changed only by the instruction.
$\overline{DY}$	1	O	Open	Output terminal for test. Usually, don't connect any lines to this terminal.
NC	2		Open	Unused terminals. Don't connect any lines to these terminals.

(Note) "1" corresponds to "High level" in positive logic.

■ FUNCTION OF EACH BLOCK

• Interface Control

(1) I/O buffer

Data is transferred through 8 data buses (DB0 ∿ DB7).

DB7 .... MSB (Most Significant Bit)

DB0 .... LSB (Least Significant Bit)

Data can neither be input nor output unless CS1 to CS3 are in the active mode. Therefore, when CS1 to CS3 are not in active mode it i useless to switch the signals of input terminals except $\overline{RST}$ and ADC, namely, the internal state is maintained and no instruction excute. Besides, pay attention to $\overline{RST}$ and ADC which operate irrespectively b CS1 to CS3.

(2) Register

Both input register and output register are provided to interface MPU of which the speed is different from that of internal operation. The selection of these registers depend on the combination of R/I an D/I signals.

Table 1. Register Selection

D/I	R/W	Operation
1	1	Reads data out of output register as internal operation (display data RAM → output register)
1	0	Writes data into input register as internal operation (input register → display data RAM)
0	1	Busy check. Read of status data.
0	0	Instruction

① Input register

Input register is used to store data temporarily before writing it int display data RAM.

The data from MPU is written into input register, then into display RAM automatically by internal operation.

When CS1 to CS3 are in the active mode and D/I and R/W select the input register as shown in Table 1, data is latched at the fall of E signal.

② Output register

Output register is used to store data temporarily which is read from display data RAM. To read out the data from output register, CS1 to CS3 should be in the active mode and both D/I and R/W should be 1. With READ instruction, data stored in the output register is output while E is "H" level. Then, at the fall of E, the display data at the indicated address is latched into the output register and the address is increased by 1. The contents in the output register is rewritten with READ instruction, while is held with address set instruction, etc.

Therefore, the data of the specified address can not be output with READ instruction soon after the address is set, but can be output at the second read of data. That is to say, one dummy read is necessary. Fig. 5 shows the CPU read timing.

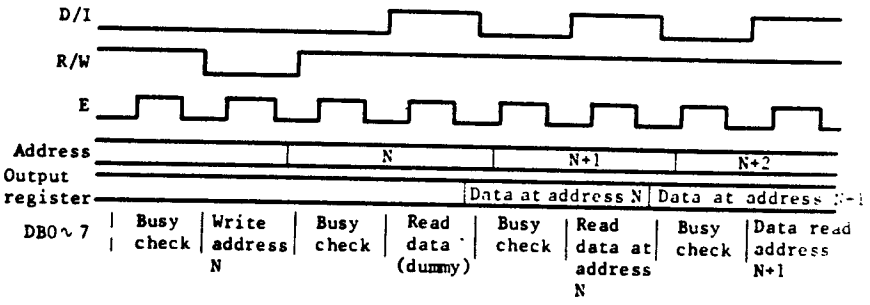
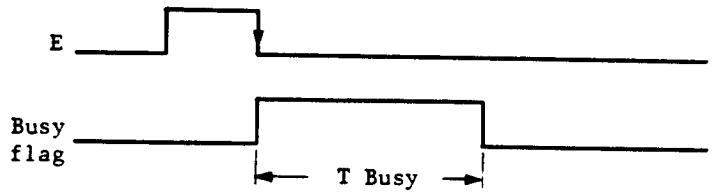


Fig. 5 CPU Read Timing

- Busy Flag

"1" of busy flag indicates that HD61102 is on the move and any instruction except Status Read instruction can not be accepted. The value of the busy flag is read out on DB7 by the Status Read instruction. Make sure that the busy flag is reset ("0") before the issue of instruction.



$1/f_{CLK} \leq T\ Busy \leq 3/f_{CLK}$

f_{CLK} is $\phi 1$, $\phi 2$ frequency

- Display ON/OFF Flip Flop

Display ON/OFF flip flop selects one of two states, ON state and OFF state of segments Y1 to Y64. In ON state, the display data corresponding to that in RAM is output to the segments. On the other hand, the display data at all segments disappear in OFF state independent of the data in RAM.
It is controlled by display ON/OFF instruction. '0' of $\overline{RST}$ signal sets the segments in OFF state. The status of the flip flop is output to DB5 by Status Read instruction. Display ON/OFF instruction does not influence data in RAM. To control display data latch by this flip flop, CL signal (display synchronous signal) should be input correctly.

- Display Start Line Register

The register specifies a line in RAM which corresponds to the top line of LCD panel, when displaying contents in display data RAM on the LCD panel. It is used for scrolling of the screen.
6-bit display start line information is written into this register by displa start line set instruction, with 'H' level of FRM signal instructing to st the display, the information in this register is transferred to Z address counter which controls the display address, and the Z address counter is preset.

- X, Y Address Counter

This is a 9-bit counter which designates addresses of internal display data RAM. X address counter of upper 3 bits and Y address counter of lower 6 bits should be set each address by respective instruction.

(1) X address counter

Ordinary register with no count functions. An address is set in by instructions.

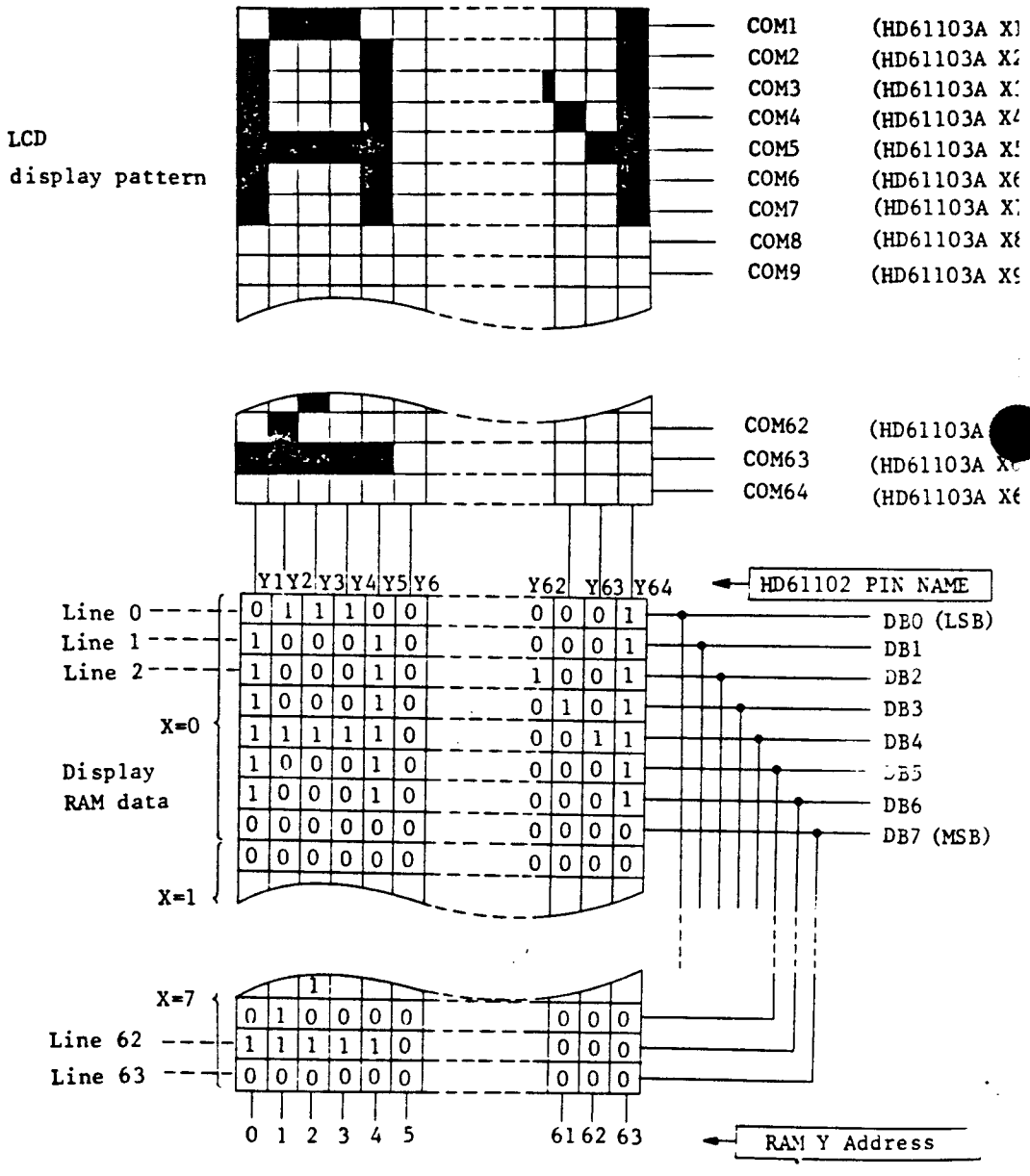
(2) Y address counter

An address is set in by instruction and it is increased by 1 automatically by R/W operations of display data. The Y address counter loops the values of 0 to 63 to count.

- Display Data RAM

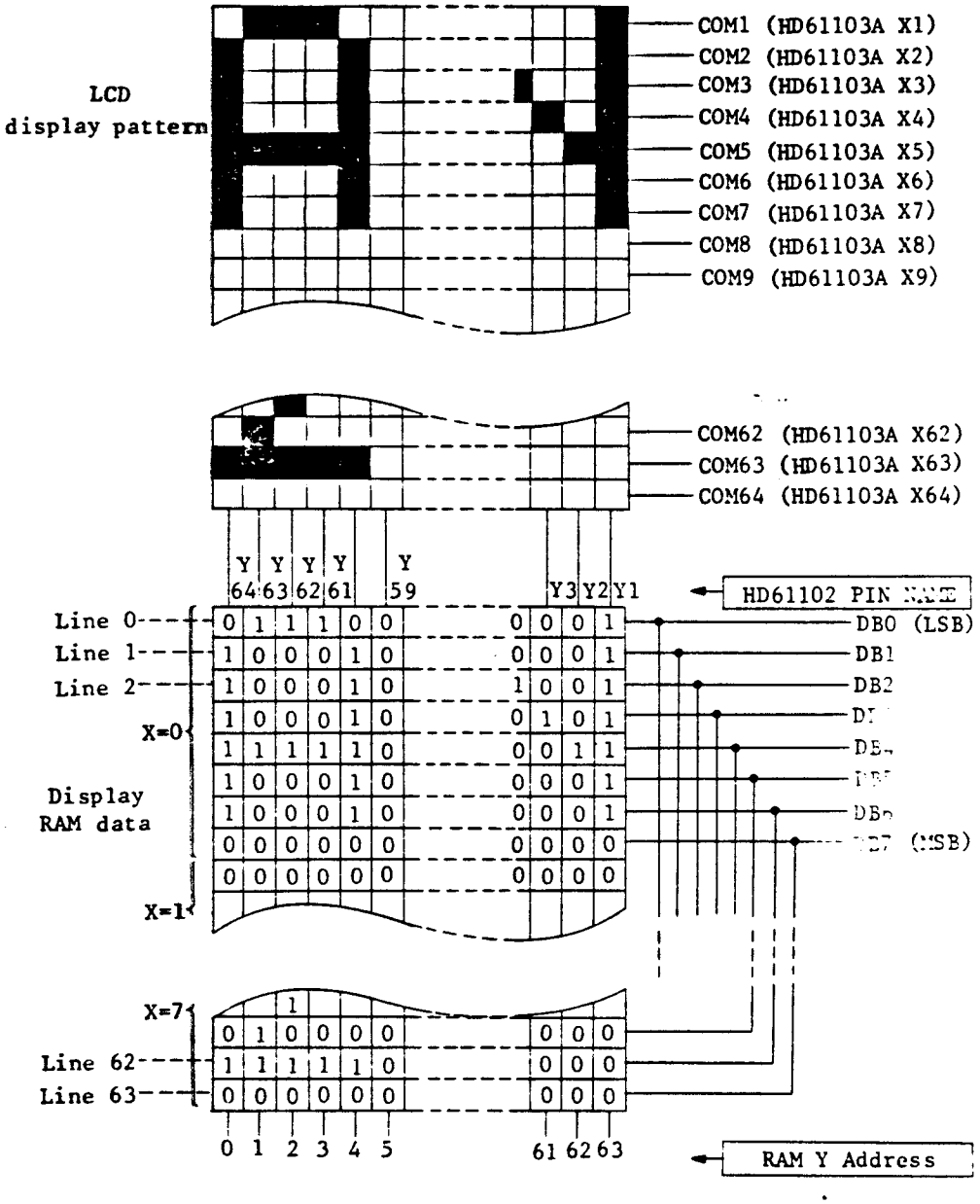
Dot data for display is stored in this RAM. 1-bit data of this RAM corresponds to light ON (data=1) and light OFF (data=0) of 1 dot in the display panel. The correspondence between Y addresses of RAM and segment PINs can be reversed by ADC signal.
As ADC signal controls Y address counter, a reverse of the signal during the operation causes malfunction and destruction of the contents of register and data of RAM. Therefore, never fail to connect ADC pin to V_{CC} or GND when using.

Fig. 6 shows the relations between Y address of RAM and segment pins in the cases of ADC=1 and ADC=0. (display start line=0, 1/64 duty).



(a) ADC="1" (Connected to Vcc)

Fig. 6 Relation between RAM Data and Display



(b) ADC="0" (Connected to GND)

Fig. 6 Relation Between RAM Data and Display

- Z Address Counter

The Z address counter generates addresses for outputting the display data synchronized with the common signal. This counter consists of 6-bit and counts up at the fall of CL signal. With "H" level of FRM, the contents of the display start line register is preset at the Z counter.

- Display data Latch

The display data latch stores the display data temporarily which is output from display data RAM to liquid crystal driving circuit.
Data is latched at the rise of CL signal. Display ON/OFF instruction controls the data in this latch and does not influence data in display data RAM.

- Liquid Crystal Display Driver Circuit

The combination of latched display data and M signal causes one of the 4 liquid crystal driver levels, V1, V2, V3 and V4 to be output.

- Reset

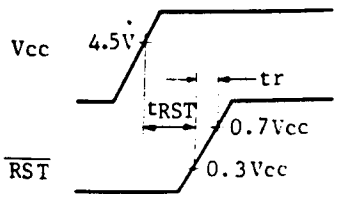
The system can be initialized by setting $\overline{RST}$ terminal at "Low" level when turning power ON.

1) Display-OFF
2) Set display start line register 0 line.

While $\overline{RST}$ is in Low level, any instruction except Status Read cannot be accepted. Therefore, Carry out other instructions after making sure that DB4=0 (clear RESET) and DB7=0 (Ready) by Status Read instruction.
The conditions of Power Supply at initial power up are as follows.

Item	Symbol	Min.	Typ	Max.	Unit
Reset time	t_{RST}	1.0	-	-	μs
Rise time	t_r	-	-	200	ns

Do not fail to set the system again because RESET during operation may destroy the data in all the register except ON/OFF register and in RAM.



■ DISPLAY CONTROL INSTRUCTIONS

• Outline

Table 2 shows the instructions. Read/Write (R/W) signal, Data/Instruction (D/I) signal and Data bus signal (DB0 to DB7) are also called instructions because the internal operation depends on the signals from MPU.
These explanations are detailed from the following page. Generally, there are following three kinds of instructions.

(1) Instruction to give addresses in the internal RAM
(2) Instruction to transfer data from/to the internal RAM
(3) Other instructions

In general use, the instruction (2) are used most frequently. But, since Y address of the internal RAM is increased by 1 automatically after writing (reading) data, the program can be lessened. During the execution of an instruction, the system cannot accept other instructions than Status Read instruction. Send instructions from MPU after making sure if the busy flag is "0", which is the proof an instruction is not being excuted.

Table 2. Instructions

	Instructions	Code										Functions	
		R/W	D/I	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0		
1	Display ON/OFF	0	0	0	0	1	1	1	1	1	1/0	Controls the ON/OFF of display. RAM data and internal status are not affected. 1:ON, 0:OFF.	
2	Display start line	0	0	1	1	display start line (0∿63)						Specifies a RAM line displayed at the top of the screen.	
3	Set page (X address)	0	0	1	0	1	1	1	Page (0∿7)			Sets the page (X address) of RAM at the page (X address) register.	
4	Set Y address	0	0	0	1	Y address (0∿63)						Sets the Y address at the Y address counter	
5	Status Read	1	0	Busy	0	ON/OFF	RESET	0	0	0	0	Reads the status. RESET 1: reset 0:normal ON/OFF 1: display OFF 0:display ON Busy 1: on the internal operation 0: Ready	
6	Write display data	0	1	Write Data								Writes data DB0 (LSB) to DB7 (MSB) on the data bus into display RAM.	Has access to the address of the display RAM specified in advance. After the access, Y address is increased by 1.
7	Read display data	1	1	Read Data								Reads data DB0 (LSB) to DB7 (MSB) from the display RAM to the data bus.	

Note 1) Busy time varies with the frequency (f_{CLK}) of $\phi 1$, and $\phi 2$.

$(1/f_{CLK} \leq T_{BUSY} \leq 3/f_{CLK})$

- **Detailed Explanation**

(1) Display ON/OFF

	R/W	D/I	DB7						DB0
Code	0	0	0	0	1	1	1	1	1	D

← high-order-bit low-order-bit →

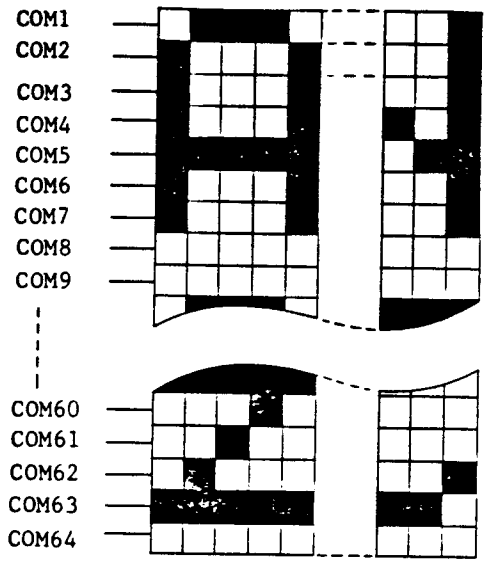
The display data appears when D is 1 and disappears when D is 0. Though the data is not on the screen with D=0, it remains in the display data RAM. Therefore, you can make it appear by changing D=0 into D=1.

(2) Display start line

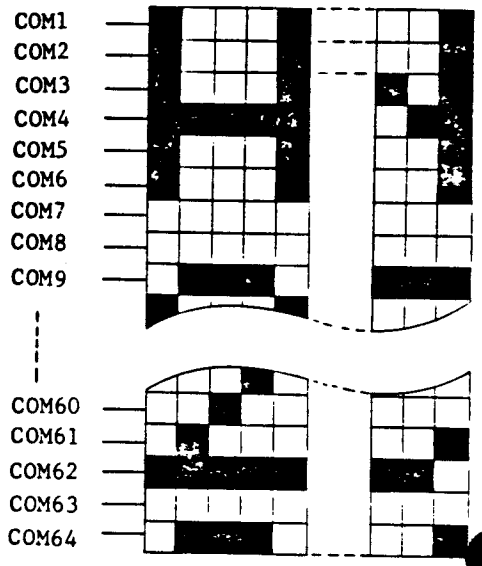
	R/W	D/I	DB7						DB0
Code	0	0	1	1	A	A	A	A	A	A

← high-order-bit low-order-bit →

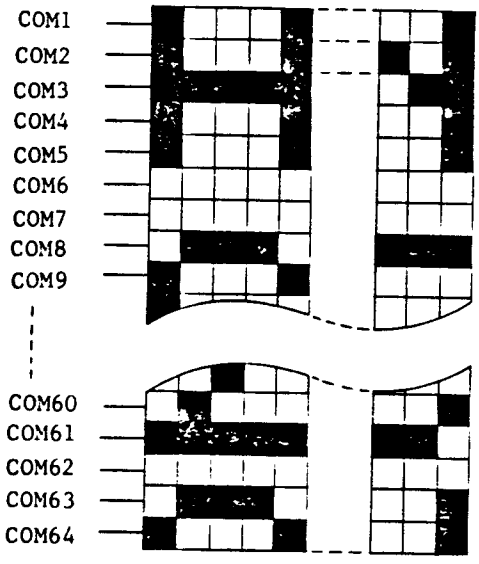
Z address AAAAAA (binary) of the display data RAM is set at the display start line register and displayed at the top of the screen.
Fig. 7 are the examples of display (1/64 duty) when the start line=0 ∿ 3.
When the display duty is 1/64 or more (ex. 1/32, 1/24 etc.), the data of total line number of LCD screen, from the line specified by display start line instruction, is displayed.



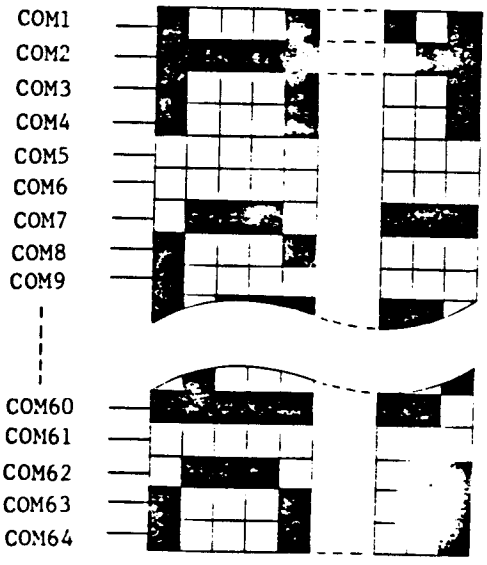
Start line=0



Start line=1



Start line=2



Start line=3

Fig. 7 Relation Between Start Line and Display

(3) Set page (X address)

	R/W	D/I	DB7							DB0
Code	0	0	1	0	1	1	1	A	A	A

← high-order-bit low-order-bit →

X address AAA (binary) of the display data RAM is set at the X address register. After that, writing or reading to or from MPU is executed in this specified page until the next page is set.

(4) Set Y address

	R/W	D/I	DB7							DB0
Code	0	0	0	1	A	A	A	A	A	A

← high-order-bit low-order-bit →

Y address AAAAAA (binary) of the display data RAM is set at the Y address counter. After that, Y address counter is increased by 1 every time the data is written or read to or from MPU.

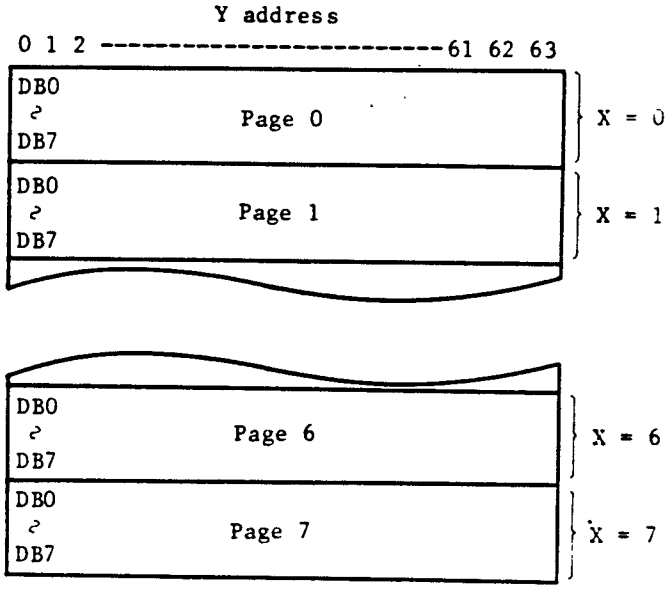
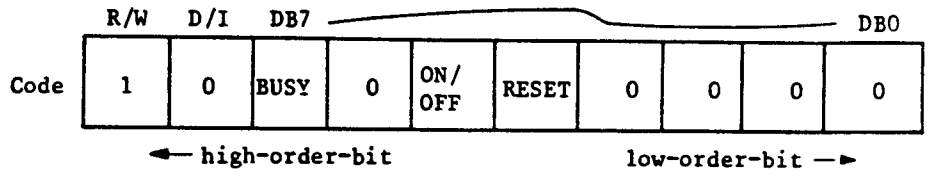


Fig. 8 Address Configuration of Display Data RAM

(5) Status Read

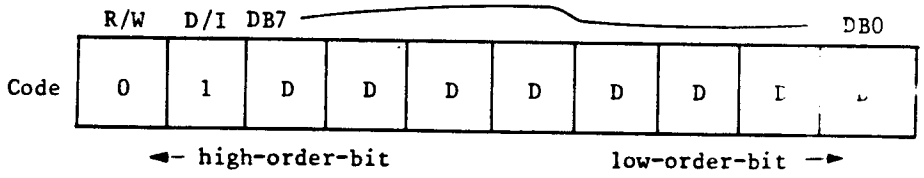


BUSY: When BUSY is 1, the LSI is in internal operation. No instructions are accepted while BUSY is 1, so you should make sure that BUSY is 0 before writing the next instruction.

ON/OFF: This bit shows the liquid crystal display conditions - ON condition or OFF condition.
When ON/OFF is 1, the display is in OFF condition.
When ON/OFF is 0, the display is in ON condition.

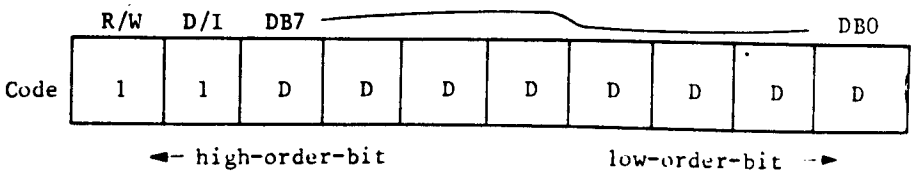
RESET: RESET=1 shows that the system is being initialized. In this condition, any instructions except Status Read instruction cannot be accepted.
RESET=0 shows that initializing has finished and the system is in the usual operation.

(6) Write Display Data



Writes 8-bit data DDDDDDDD (binary) into the display data RAM. Then Y address is increased by 1 automatically.

(7) Read Display Data

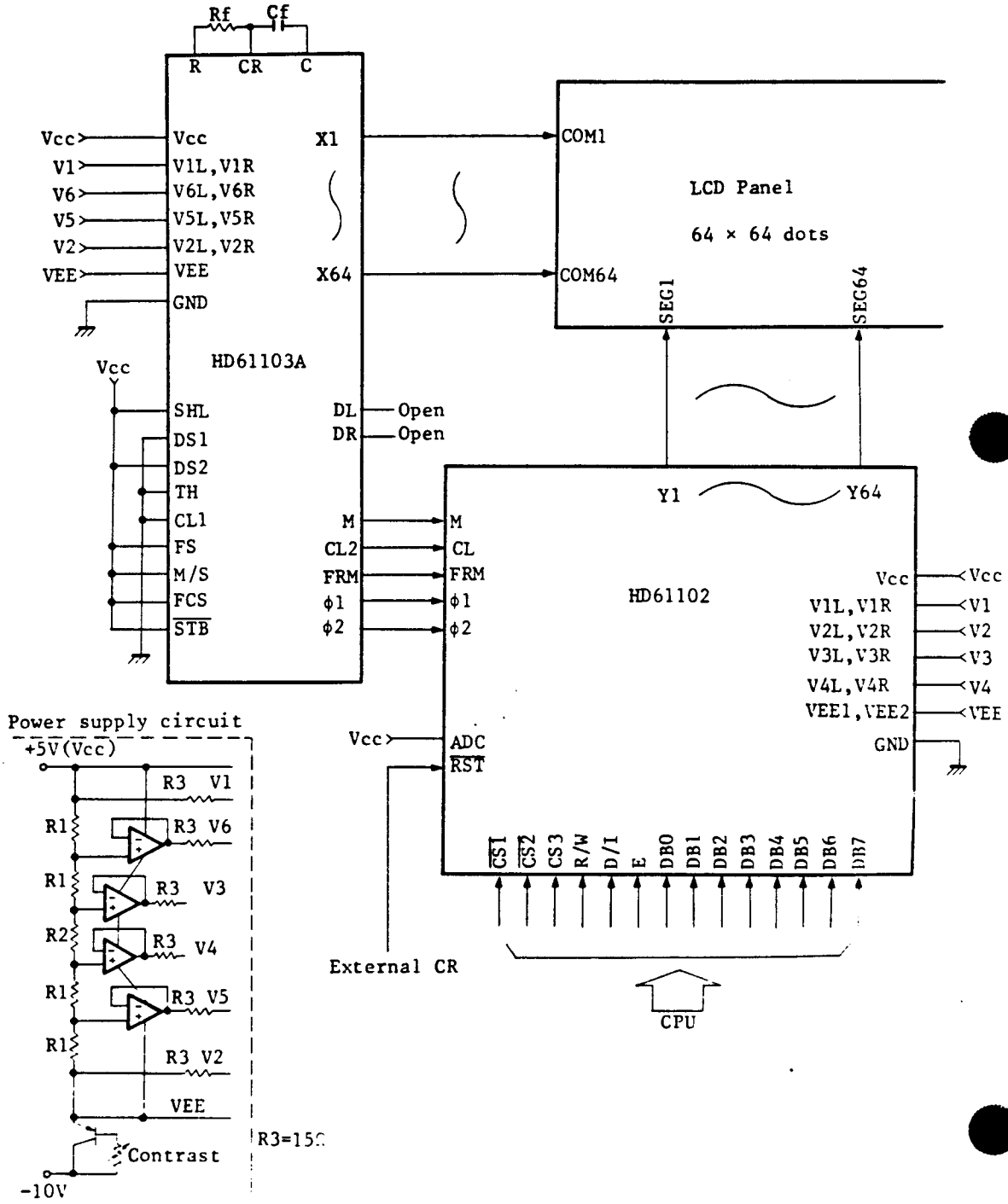


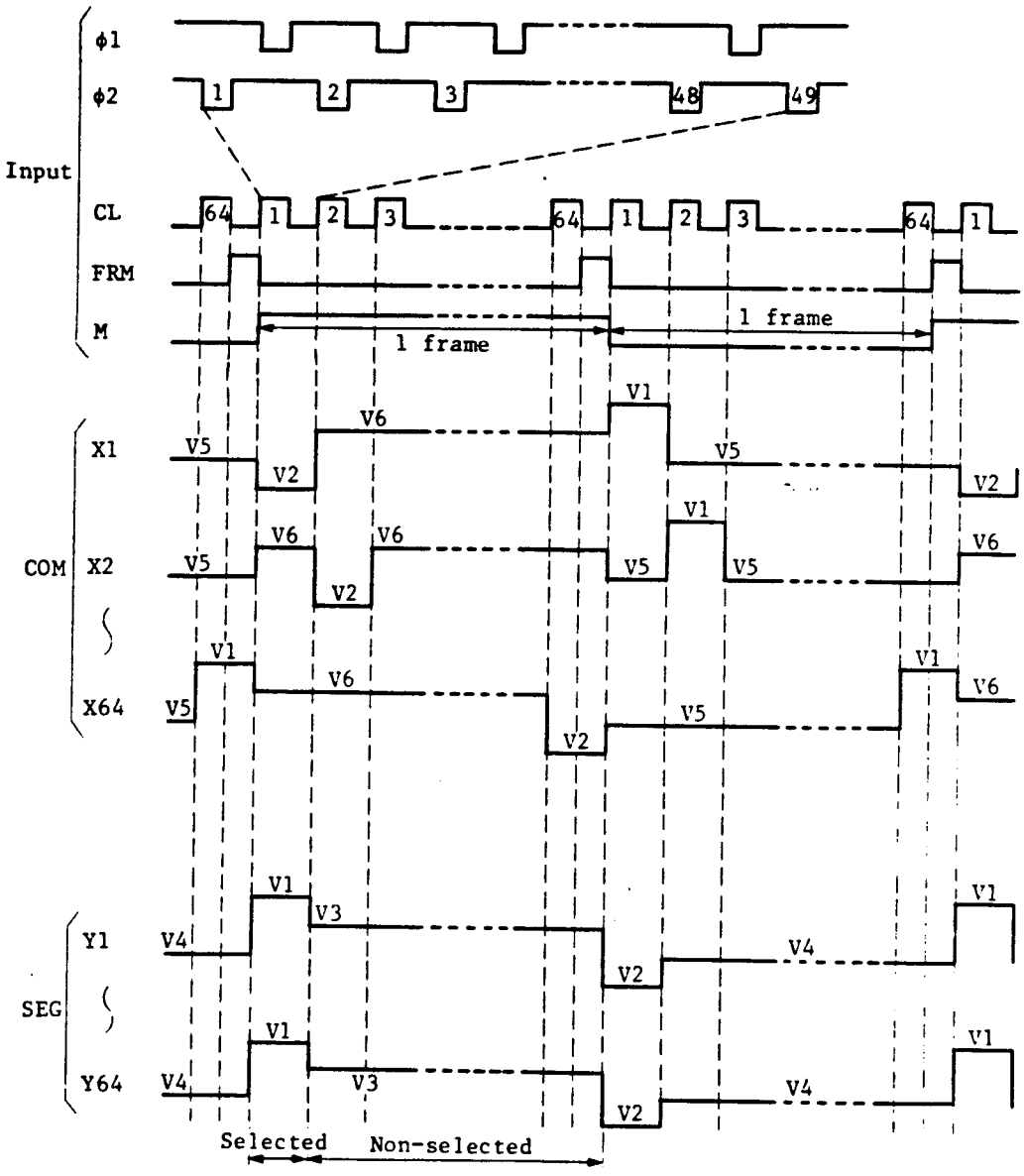
Read out 8-bit data DDDDDDDD (binary) from the display data RAM. Then Y address is increased by 1 automatically.

One dummy read is necessary soon after the address setting. For details, refer to the explanation of output register in "FUNCTION OF EACH BLOCK".

■ THE USAGE OF HD61102

- Interface with HD61103A (1/64 duty)



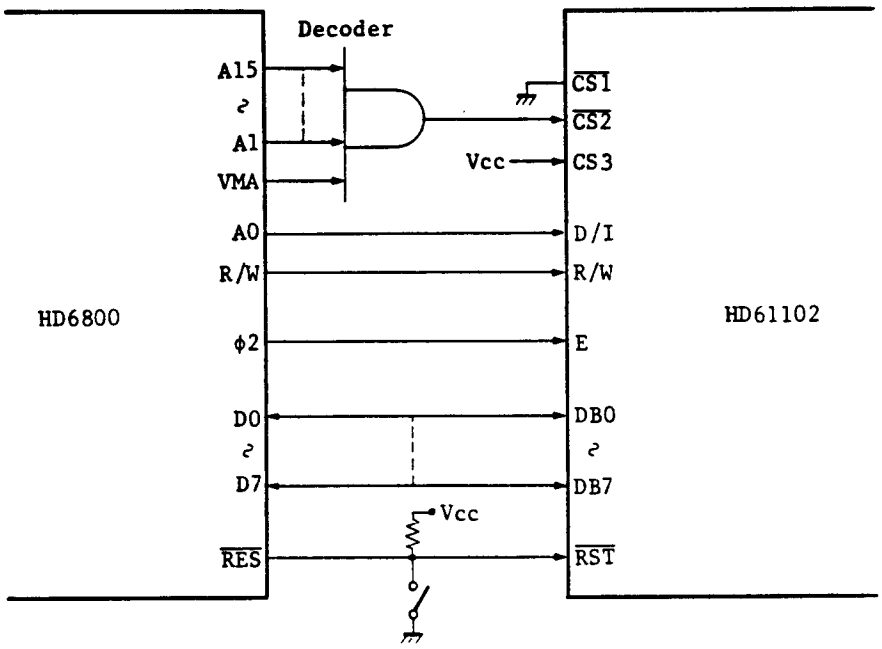


The wave forms of Y1 to Y64 outputs vary with the display data. In this example, the top line of the panel lights up and other dots do not.

Fig. 9 LCD Driver Timing Chart (1/64 duty)

● Interface with CPU

a) Example of connection with HD6800



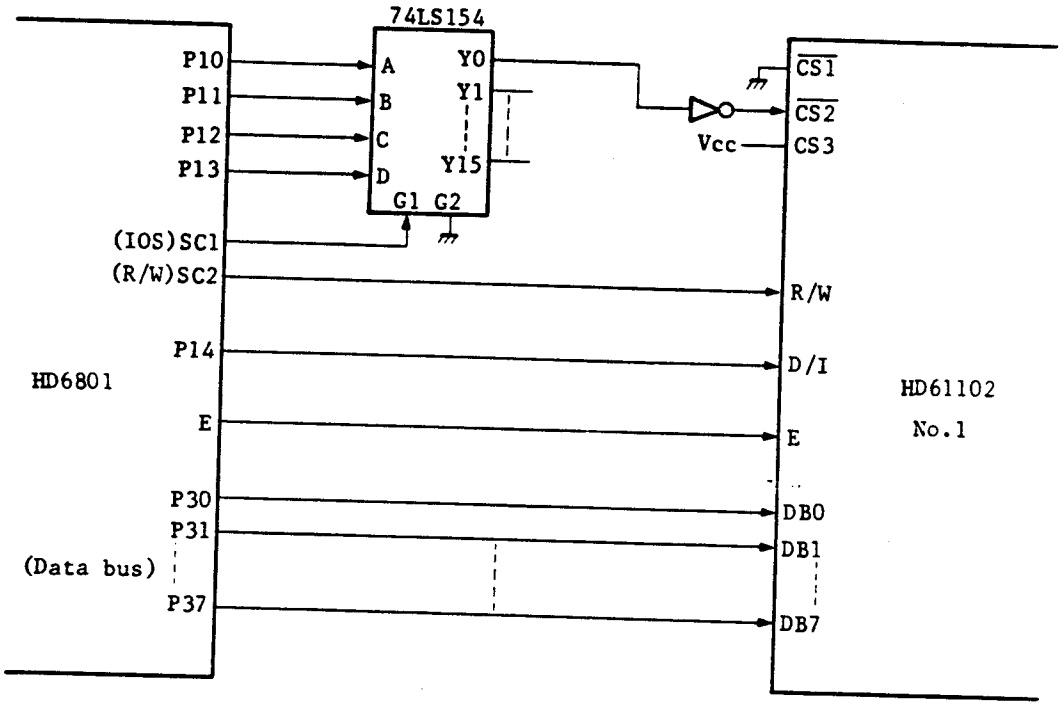
The example of connection with HD6800 series

In this decoder, addresses of HD61102 in the address area of HD6800 are:

Read/Write of the display data	$FFFF
Write of display instruction	$FFFE
Read out of status	$FFFE

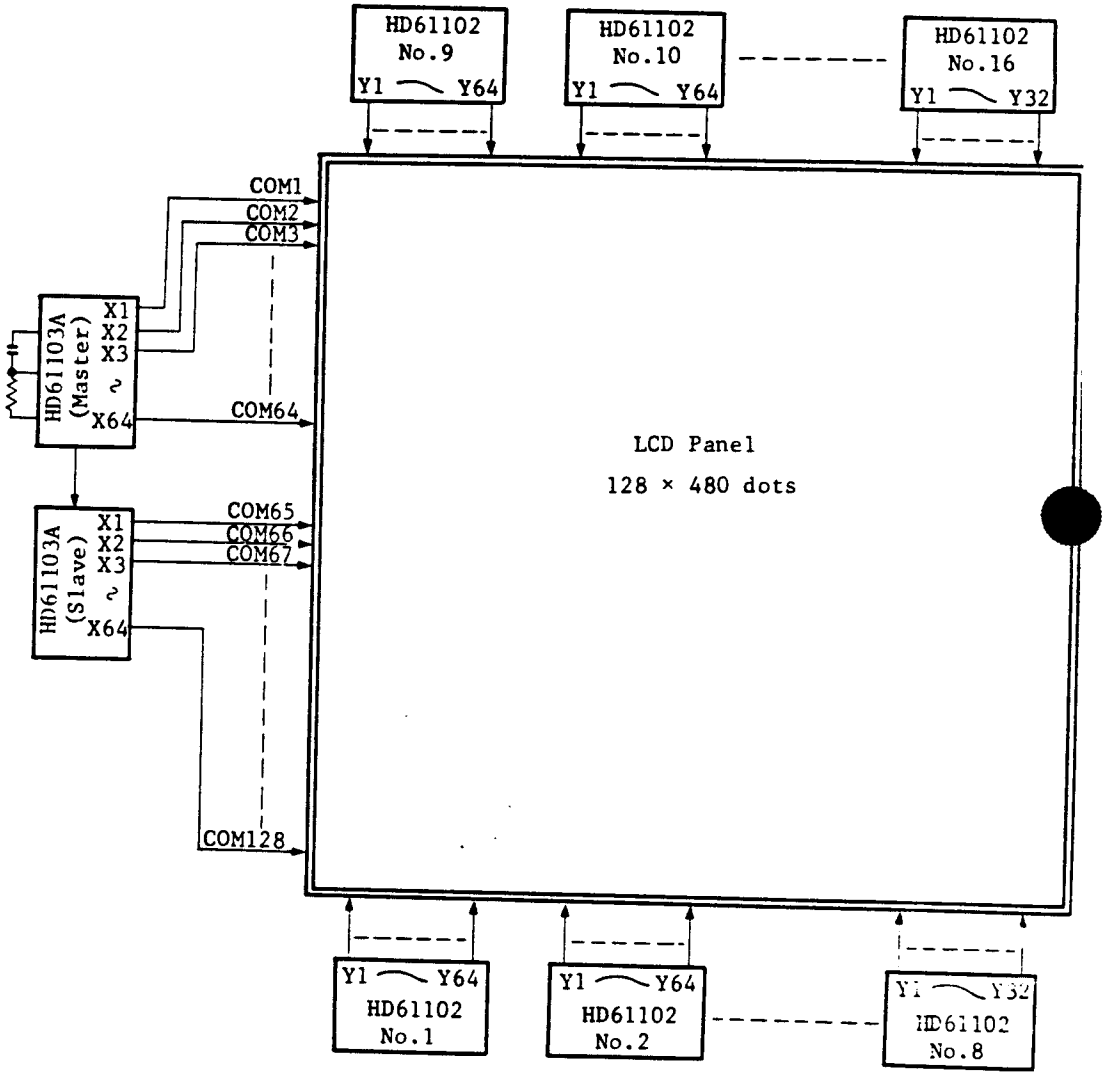
Therefore, you can control HD61102 by reading/writing the data at these addresses.

b) Example of connection with HD6801



- Set HD6801 in Mode 5. P10 to P14 are used as the output port and P30 to P37 as the data bus.
- 74LS154 is 4 to 16 decoder and genelate chip select signal to make specified HD61102 active after decoding 4 bits of P10 to P13.
- Therefore, after making the operation possible by P10 to P13 and specifying D/I signal by P14, read/write from/to the external memory area ($0100 to $01FE) to control HD61102. In this case, IOS signal is output from SC1 and R/W signal from SC2.
- For details of HD6800 and HD6801, refer to the each manual.

- Example of Application



Note) In this example, two HD61103A's output the equivalent waveforms. So, stand-alone operation is possible. In this case, connect COM1 COM65 to X1, COM2 and COM66 to X2, ..., and COM64 and COM128 to X6 However, for the large screen display, you had better drive in 2 row as this example to guarantee the display quality.

Appendixes

A

Resident Debugger

The ROM-resident debugger provides a way to debug application programs written in assembly language. It offers a way to examine or modify data in memory and the CPU registers, set a breakpoint, or single-step an assembly-language program.

The debugger is entered by typing RDB76 [ENTER] at the command mode prompt. The debugger must not use interrupts, so keys are more difficult to enter than in command mode. There is no cursor because cursor blink requires interrupts. Note that the debugger will *not* time out and turn the HP-94 off.

Table A-1. Resident Debugger Commands

Command	Description	Page
D	Display the contents of memory in hexadecimal characters.	A-4
G	Execute code until a breakpoint.	A-6
I *	Input data from an I/O port.	A-7
L †	Enter data in MDS format.	A-8
M	Display or change the contents of memory.	A-9
O *	Output data to an I/O port.	A-10
R	Display or change the contents of CPU registers.	A-11
S	Single-step execution of a program.	A-12
X	Switch the debugger console between HP-94 keyboard and serial port.	A-13

* For the I and O commands, press the [K] or [L] key respectively on the HP-94 keyboard. The command letter is shown in the display.

† The L command can only be entered if the console is set to the serial port.

All the characters recognized by the debugger can be entered without using [SHIFT], including the digit keys [0] through [9]. Some characters are assigned to different keys because the key which has that character printed on it is also a digit key.

Table A-2. Resident Debugger Keyboard Map

Key	Response
SPACE	:
.	,
#	+
K	I
L	O
Q	P
SHIFT	(ignored)

The ROM-resident debugger uses the HP-94 keyboard and display as the console. If the serial port is not used by the code being debugged, the serial port can be used as a console by connecting a terminal to the port. The port configuration is unconditionally set to 9600 baud, 7-bit data with even parity, and one stop bit.

Command Syntax

A parameter enclosed by [ ] is optional and may be omitted.

Underlined characters are characters displayed by the debugger.

In this chapter, the term word means a 16-bit value.

Example:

To input 40:1F, press these keys:

[4] [0] [SPACE] [1] [F]

An address has the following format:

[SSSS:]FFFF

SSSS is the segment expression (default = CS register)
FFFF is the offset expression

Both segment and offset can combine hexadecimal constants or two-character register names in addition or subtraction expressions using the + and - operators.

If more than four hexadecimal digits are entered, only the last four digits are used.

Valid register names are AX, BX, CX, DX, SP, BP, SI, DI, CS, DS, SS, ES, IP, and FL (flags).

Examples:

`41-1:145+AF` is interpreted as `40:1F4`.

`145-34+1` is interpreted as `CS:112`.

`IP+2` (if `IP` is 110h) is interpreted as `CS:112`.

`FFF0145-34+1` is interpreted as `CS:112` (`0145-34+1`).

`ES:BX+1` is interpreted as expected.

D

Display the contents of memory.

Syntax:

D[W]*address1*[, *address2*][ENTER]

Description:

The D command displays in hexadecimal the contents of memory from *address1* to *address2*. If the W option is specified, the display is grouped by words; otherwise the display is grouped by bytes.

Console is the HP-94 keyboard/display:

The contents of memory up to a paragraph boundary (xxxx:xxx0) are displayed. The debugger then waits for a key to be pressed. If [ENTER] is pressed, the contents of the next 16 bytes of memory are displayed. Pressing any other key terminates the D command.

Console is the serial port:

The contents of memory are displayed in hexadecimal. If the W option is not specified, the corresponding ASCII characters are also displayed. Pressing any key terminates the D command. That key is then processed as a debugger command. Since software handshake characters are interpreted as keys, a handshake character such as XOFF from a terminal will terminate the D command.

Example 1: Console is HP-94 keyboard/display; display 0:0 through 0:14 as bytes.

```
*D0:0,14 (D 0 SPACE 0 . 1 4 ENTER)
0000:0000
 5B 0C 00 FC E1 0B
 00 FC 5B 0C 00 FC
 5B 0C 00 FC ENTER
0000:0010
 88 FF 95 7F 07
*
```

Example 2: Console is serial port; display 0:0 through 0:14 as bytes.

```
*D0:0,14
0000:0000  5B 0C 00 FC E1 0B 00 FC 5B 0C 00 FC 5B 0C 00 FC    [.......[...[...
0000:0010  88 FF 95 7F 07                                     .....
*
```

Example 3: Console is HP-94 keyboard/display; display DS:0 through DS:14 as words. Assume DS is 0.

```
*DWDS:0,14 (D W D S SPACE 0 . 1 4 ENTER)
0000:0000
 0C5B FC00 0BE1 FC00
 0C5B FC00 0C5B FC00 ENTER
0000:0010
 FF88 7F95 FF07
*
```

Example 4: Console is serial port; display DS : 0 through DS : 14 as words. Assume DS is 0.

```
*DW DS:0,14
0000:0000  0C5B FC00 0BE1 FC00 0C5B FC00 0C5B FC00
0000:0010  FF88 7F95 FF07
*
```

G

Execute code until a breakpoint.

Syntax:

`G cccc:iiii- dd` [*start address*][, *break address*][ENTER]

Description:

The G command displays in hexadecimal the contents of `CS:IP` and the contents of the byte at that location. If a *break address* is specified, a breakpoint is set by writing an INT 3 (CCh) at that address. If the INT 3 cannot be written, an error occurs. This means it is not possible to set a breakpoint in a program in ROM. If a *start address* is not specified, program execution starts at `CS:IP`. If a *start address* is specified, program execution starts at that address.

When a program reaches the breakpoint, the debugger displays the following message and waits for another command.

```
BR@cccc:iiii
```

`cccc` is the value of the current CS register.
`iiii` is the value of the current `IP` register.

Note that if a program never reaches the breakpoint, the INT 3 remains in the code. The debugger will try to restore the instruction replaced by the INT 3 when the debugger is reentered. Because the program being debugged may have moved, it is recommended that a breakpoint not be set before returning to the operating system from the debugger.

Input data from a port.

Syntax:

`I[W]`*port*`,`

Description:

The `I` command inputs data from port *port* and displays it in hexadecimal. If the W option is specified, one word of data is input from *port* and displayed; otherwise one byte of data is input from *port* and displayed. Data is input from *port* and displayed each time a comma is entered.

Pressing ENTER terminates the `I` command.

L

Enter data in MDS format.

Syntax:

L[*bias*][ENTER]

Description:

The L command inputs data in MDS format and loads the data to main memory. The data is written at the record address contained in the MDS data records added to the value of *bias*. The default *bias* is zero. The segment value can be set with a type 2 MDS record. The default segment is zero.

The L command is available only if the console is the serial port.

The L command discards any data received until the first colon (:) of the MDS file is encountered. If the data which follows the colon is not in MDS format, the L command terminates.

Display or change the contents of memory.

Syntax:

M[W]*address*, dd- [*new data*],

Description:

The **M** command displays in hexadecimal the contents of memory at *address*. If the **W** option is specified, memory is processed in words; otherwise memory is processed in bytes.

When a comma is entered, the contents of the next memory location are displayed.

Pressing ENTER terminates the **M** command.

If *new data* is specified (in hexadecimal), it is written to the memory location currently displayed. A read-after-write check is done to ensure that the data was written correctly. If the data read back does not match the data which was written, such as when trying to write to ROM, the **M** command terminates.

O

Output data to a port.

Syntax:

O[W]*port*,*data*[ENTER]
or
O[W]*port*,*data*,

Description:

The O command outputs *data* to the specified *port*. If the W option is specified, one word of data is output to *port*; otherwise one byte of data is output to *port*.

If the O command is entered with a trailing comma, it writes the data to the port, then prompts for new data with a dash (-).

Pressing [ENTER] terminates the O command.

Display or change the contents of CPU registers.

Syntax:

R [ENTER]
or
R*register*-dddd- [*data*][,][ENTER]

Description:

The R command displays the contents of CPU registers in hexadecimal.

If *register* is specified, the contents of that register are displayed in hexadecimal. If *data* is specified (in hexadecimal), the register is changed to that value. A comma (,) continues on to the next register, if any; [ENTER] terminates the R command.

If *register* is not specified, the contents of all the CPU registers are displayed. The format depends on whether the console is the HP-94 or the serial port:

Console is the HP-94 keyboard/display:

```
*R [ENTER]
 AX=024A BX=0000
 CX=1FA2 DX=000E
 SP=07F2 BP=0250
 SI=0410 DI=0015 [ENTER]

 CS=0128 DS=0128
 SS=1F80 ES=1F80
 IP=0008 FL=F206
*
```

Console is the serial port:

```
*R [ENTER]
 AX=024A BX=0000 CX=1FA2 DX=000E SP=07F2 BP=0250 SI=0410
 DI=0015 CS=0128 DS=0128 SS=1F80 ES=1F80 IP=0008 FL=F206
*
```

S

Single-step execution of a program.

Syntax:

`S cccc:iiii- dd` [*start address*],

Description:

The `S` command displays the current `CS:IP` in hexadecimal and waits for another key. If *start address* is specified, the current `CS:IP` is set to that address.

A single instruction at the current `CS:IP` is executed when a comma (`,`) is entered. The `S` command displays the new `CS:IP` and waits for another key.

Single-step execution terminates when the [ENTER] key is pressed.

NOTE Because the HP-94 has a timer which interrupts every 5 ms, there will almost always be a pending interrupt when single-stepping code. Because all registers are restored before execution, including `FL`, interrupts are enabled unless the `FL` register has been modified to disable interrupts. When using the HP-94 keyboard, there is no key sequence to directly type the letter L. To view the `FL` register using the R command, type `RIP,` ([R][K][Q][.]).

Switch the debugger console between the HP-94 keyboard and serial port.

Syntax:

X [ENTER]

Description:

The X command switches the debugger console between the HP-94 keyboard/display and the serial port. Several commands display information in a format which is easier to read when the console is the serial port.

The X command displays the verification prompt "Ok ? " and waits for a key. If [Y] [ENTER] is entered, the console is switched to the serial port if the console was the HP-94 keyboard/display, or to the HP-94 keyboard/display if the console was the serial port.

When the console is switched to the serial port, the port is set to 9600 baud, even parity, 7-bit data, and one stop bit. The debugger operates without any hardware or software handshaking. Any handshaking characters sent by the terminal will be interpreted as keys, and will have the same effect as pressing keys. This is especially important for the D command.

The console must not be switched to the serial port while an application program which uses the serial port is being debugged.

CAUTION If the console is switched to the serial port which is connected to a terminal that cannot communicate at 9600 baud, even parity, 7-bit data, and one stop bit, or if the console is switched with no terminal attached, the only way to regain control of the HP-94 is to press the reset switch.

B

Errors

Table B-1. Operating System Errors

Hex	Decimal	Meaning
64h	100 *	BASIC interpreter not found
65h	101	Illegal parameter
66h	102	Directory does not exist
67h	103	File not found
68h	104	Too many files
69h	105	Channel not open
6Ah	106	Channel already open
6Bh	107	File already open
6Ch	108	File already exists
6Dh	109	Read-only access
6Eh	110	Access restricted
6Fh	111	No room for file
70h	112	No room to expand file
71h	113	No room for scratch area
72h	114	Scratch area does not exist
73h	115 †	Short record detected
74h	116 †	Terminate character detected
75h	117 †	End-of-data
76h	118	Timeout
77h	119	Power switch pressed
C8h	200	Low battery
C9h	201	Receive buffer overflow
CAh	202	Parity error
CBh	203	Overrun error
CCh	204	Parity and overrun error
CDh	205	Framing error
CEh	206	Framing and parity error
CFh	207	Framing and overrun error
D0h	208	Framing, overrun, and parity error
D1h	209 †	Invalid MDS file received
D2h	210 *	Low backup battery — main memory
D3h	211 *	Low backup battery — 128K memory board or 40K RAM card
D4h	212 *	Checksum error — main memory directory table
D5h	213 *	Checksum error — 40K RAM or ROM/EPROM card directory table
D6h	214 *	Checksum error — reserved scratch space
D7h	215 *	Checksum error — main memory free space
D8h	216 *	Checksum error — main memory file
D9h	217 *	Checksum error — 40K RAM or ROM/EPROM card file
DAh	218	Lost connection while transmitting
DBh	219 †	Illegal use of operating system stack

* Only reported when machine is turned on.

† Never reported by built-in BASIC keywords.

Table B-2. BASIC Interpreter Errors

Message	Meaning
AR	Array subscript error
BM	BASIC interpreter malfunction
BR	Branch destination error
CN	Data conversion error
CO	Conversion overflow
DO	Decimal overflow
DT	Data error
EP	Missing END statement
FN	Illegal DEF FN statement
IL	Illegal argument
IR	Insufficient RAM
IS	Illegal statement
LN	Nonexistent line
MO	Memory overflow
NF	Program not found
RT	RETURN or SYRT error
SY	Syntax error
TY	Data type mismatch
UM	Unmatched number of arguments

C

Keyboard Layout

Table C-1. ASCII Characters and Keycodes for Each Key

Shifted Key (orange)	Shifted Character	Unshifted Key (white)	Unshifted Character	Keycode
A	A (41h)	(unmarked)	user-defined (80h)	01h
B	B (42h)	(unmarked)	user-defined (81h)	06h
C	C (43h)	(unmarked)	user-defined (82h)	0Bh
D	D (44h)	(unmarked)	user-defined (83h)	10h
E	E (45h)	(unmarked)	user-defined (84h)	02h
F	F (46h)	(unmarked)	user-defined (85h)	07h
G	G (47h)	(unmarked)	user-defined (86h)	0Ch
H	H (48h)	7	7 (37h)	11h
I	I (49h)	8	8 (38h)	16h
J	J (4Ah)	9	9 (39h)	1Bh
K	K (4Bh)	(unmarked)	user-defined (87h)	03h
L	L (4Ch)	(unmarked)	user-defined (88h)	08h
M	M (4Dh)	(unmarked)	user-defined (89h)	0Dh
N	N (4Eh)	4	4 (34h)	12h
O	O (4Fh)	5	5 (35h)	17h
P	P (50h)	6	6 (36h)	1Ch
Q	Q (51h)	(unmarked)	user-defined (8Ah)	04h
R	R (52h)	(unmarked)	user-defined (8Bh)	09h
S	S (53h)	(unmarked)	user-defined (8Ch)	0Eh
T	T (54h)	1	1 (31h)	13h
U	U (55h)	2	2 (32h)	18h
V	V (56h)	3	3 (33h)	1Dh
W	W (57h)	(unmarked)	user-defined (8Dh)	05h
X	X (58h)	(unmarked)	user-defined (8Eh)	0Ah
Y	Y (59h)	(unmarked)	user-defined (8Fh)	0Fh
Z	Z (5Ah)	0	0 (30h)	14h
*	* (2Ah)	#	# (23h)	15h
SPACE	(space) (20h)	00	00 (30h 30h)	19h
—	— (2Dh)	—	— (2Dh)	1Ah
.	. (2Eh)	.	. (2Eh)	1Eh
SHIFT	(none)	SHIFT	(none)	1Fh
CLEAR	(CAN) (18h)	CLEAR	(CAN) (18h)	20h
←	(DEL) (7Fh)	←	(DEL) (7Fh)	21h
ENTER	(CR) (0Dh)	ENTER	(CR) (0Dh)	22h

D

Roman-8 Character Set

ASCII Char.	Character Code			
	Hex	Dec	Oct	Binary
NUL	00	0	000	00000000
SOH	01	1	001	00000001
STX	02	2	002	00000010
ETX	03	3	003	00000011
EOT	04	4	004	00000100
ENQ	05	5	005	00000101
ACK	06	6	006	00000110
BEL	07	7	007	00000111
BS	08	8	010	00001000
HT	09	9	011	00001001
LF	0A	10	012	00001010
VT	0B	11	013	00001011
FF	0C	12	014	00001100
CR	0D	13	015	00001101
SO	0E	14	016	00001110
SI	0F	15	017	00001111
DLE	10	16	020	00010000
DC1	11	17	021	00010001
DC2	12	18	022	00010010
DC3	13	19	023	00010011
DC4	14	20	024	00010100
NAK	15	21	025	00010101
SYN	16	22	026	00010110
ETB	17	23	027	00010111
CAN	18	24	030	00011000
EM	19	25	031	00011001
SUB	1A	26	032	00011010
ESC	1B	27	033	00011011
FS	1C	28	034	00011100
GS	1D	29	035	00011101
RS	1E	30	036	00011110
US	1F	31	037	00011111

ASCII Char.	Character Code			
	Hex	Dec	Oct	Binary
space	20	32	040	00100000
!	21	33	041	00100001
"	22	34	042	00100010
#	23	35	043	00100011
$	24	36	044	00100100
%	25	37	045	00100101
&	26	38	046	00100110
'	27	39	047	00100111
(	28	40	050	00101000
)	29	41	051	00101001
*	2A	42	052	00101010
+	2B	43	053	00101011
,	2C	44	054	00101100
-	2D	45	055	00101101
.	2E	46	056	00101110
/	2F	47	057	00101111
0	30	48	060	00110000
1	31	49	061	00110001
2	32	50	062	00110010
3	33	51	063	00110011
4	34	52	064	00110100
5	35	53	065	00110101
6	36	54	066	00110110
7	37	55	067	00110111
8	38	56	070	00111000
9	39	57	071	00111001
:	3A	58	072	00111010
;	3B	59	073	00111011
<	3C	60	074	00111100
=	3D	61	075	00111101
>	3E	62	076	00111110
?	3F	63	077	00111111

ASCII Char.	Character Code			
	Hex	Dec	Oct	Binary
@	40	64	100	01000000
A	41	65	101	01000001
B	42	66	102	01000010
C	43	67	103	01000011
D	44	68	104	01000100
E	45	69	105	01000101
F	46	70	106	01000110
G	47	71	107	01000111
H	48	72	110	01001000
I	49	73	111	01001001
J	4A	74	112	01001010
K	4B	75	113	01001011
L	4C	76	114	01001100
M	4D	77	115	01001101
N	4E	78	116	01001110
O	4F	79	117	01001111
P	50	80	120	01010000
Q	51	81	121	01010001
R	52	82	122	01010010
S	53	83	123	01010011
T	54	84	124	01010100
U	55	85	125	01010101
V	56	86	126	01010110
W	57	87	127	01010111
X	58	88	130	01011000
Y	59	89	131	01011001
Z	5A	90	132	01011010
[	5B	91	133	01011011
\	5C	92	134	01011100
]	5D	93	135	01011101
^	5E	94	136	01011110
_	5F	95	137	01011111

ASCII Char.	Character Code			
	Hex	Dec	Oct	Binary
`	60	96	140	01100000
a	61	97	141	01100001
b	62	98	142	01100010
c	63	99	143	01100011
d	64	100	144	01100100
e	65	101	145	01100101
f	66	102	146	01100110
g	67	103	147	01100111
h	68	104	150	01101000
i	69	105	151	01101001
j	6A	106	152	01101010
k	6B	107	153	01101011
l	6C	108	154	01101100
m	6D	109	155	01101101
n	6E	110	156	01101110
o	6F	111	157	01101111
p	70	112	160	01110000
q	71	113	161	01110001
r	72	114	162	01110010
s	73	115	163	01110011
t	74	116	164	01110100
u	75	117	165	01110101
v	76	118	166	01110110
w	77	119	167	01110111
x	78	120	170	01111000
y	79	121	171	01111001
z	7A	122	172	01111010
{	7B	123	173	01111011
|	7C	124	174	01111100
}	7D	125	175	01111101
~	7E	126	176	01111110
DEL	7F	127	177	01111111

ASCII Char.	Character Code			
	Hex	Dec	Oct	Binary
	80	128	200	10000000
	81	129	201	10000001
	82	130	202	10000010
	83	131	203	10000011
	84	132	204	10000100
	85	133	205	10000101
	86	134	206	10000110
	87	135	207	10000111
	88	136	210	10001000
	89	137	211	10001001
	8A	138	212	10001010
	8B	139	213	10001011
	8C	140	214	10001100
	8D	141	215	10001101
	8E	142	216	10001110
	8F	143	217	10001111
	90	144	220	10010000
	91	145	221	10010001
	92	146	222	10010010
	93	147	223	10010011
	94	148	224	10010100
	95	149	225	10010101
	96	150	226	10010110
	97	151	227	10010111
	98	152	230	10011000
	99	153	231	10011001
	9A	154	232	10011010
	9B	155	233	10011011
	9C	156	234	10011100
	9D	157	235	10011101
	9E	158	236	10011110
	9F	159	237	10011111

ASCII Char.	Character Code			
	Hex	Dec	Oct	Binary
space	A0	160	240	10100000
À	A1	161	241	10100001
Â	A2	162	242	10100010
È	A3	163	243	10100011
Ê	A4	164	244	10100100
Ë	A5	165	245	10100101
Î	A6	166	246	10100110
Ï	A7	167	247	10100111
´	A8	168	250	10101000
`	A9	169	251	10101001
^	AA	170	252	10101010
¨	AB	171	253	10101011
~	AC	172	254	10101100
Ù	AD	173	255	10101101
Û	AE	174	256	10101110
£	AF	175	257	10101111
¯	B0	176	260	10110000
Ý	B1	177	261	10110001
ý	B2	178	262	10110010
°	B3	179	263	10110011
Ç	B4	180	264	10110100
ç	B5	181	265	10110101
Ñ	B6	182	266	10110110
ñ	B7	183	267	10110111
¡	B8	184	270	10111000
¿	B9	185	271	10111001
¤	BA	186	272	10111010
£	BB	187	273	10111011
¥	BC	188	274	10111100
§	BD	189	275	10111101
ƒ	BE	190	276	10111110
¢	BF	191	277	10111111

ASCII Char.	Character Code			
	Hex	Dec	Oct	Binary
â	C0	192	300	11000000
ê	C1	193	301	11000001
ô	C2	194	302	11000010
û	C3	195	303	11000011
á	C4	196	304	11000100
é	C5	197	305	11000101
ó	C6	198	306	11000110
ú	C7	199	307	11000111
à	C8	200	310	11001000
è	C9	201	311	11001001
ò	CA	202	312	11001010
ù	CB	203	313	11001011
ä	CC	204	314	11001100
ë	CD	205	315	11001101
ö	CE	206	316	11001110
ü	CF	207	317	11001111
Å	D0	208	320	11010000
î	D1	209	321	11010001
Ø	D2	210	322	11010010
Æ	D3	211	323	11010011
å	D4	212	324	11010100
í	D5	213	325	11010101
ø	D6	214	326	11010110
æ	D7	215	327	11010111
Ä	D8	216	330	11011000
ì	D9	217	331	11011001
Ö	DA	218	332	11011010
Ü	DB	219	333	11011011
É	DC	220	334	11011100
ï	DD	221	335	11011101
ß	DE	222	336	11011110
Ô	DF	223	337	11011111

ASCII Char.	Character Code			
	Hex	Dec	Oct	Binary
Á	E0	224	340	11100000
Ã	E1	225	341	11100001
ã	E2	226	342	11100010
Ð	E3	227	343	11100011
ð	E4	228	344	11100100
Í	E5	229	345	11100101
Ì	E6	230	346	11100110
Ó	E7	231	347	11100111
Ò	E8	232	350	11101000
Õ	E9	233	351	11101001
õ	EA	234	352	11101010
Š	EB	235	353	11101011
š	EC	236	354	11101100
Ú	ED	237	355	11101101
Ÿ	EE	238	356	11101110
ÿ	EF	239	357	11101111
Þ	F0	240	360	11110000
þ	F1	241	361	11110001
▪	F2	242	362	11110010
µ	F3	243	363	11110011
¶	F4	244	364	11110100
3/4	F5	245	365	11110101
–	F6	246	366	11110110
1/4	F7	247	367	11110111
1/2	F8	248	370	11111000
ª	F9	249	371	11111001
º	FA	250	372	11111010
<<	FB	251	373	11111011
■	FC	252	374	11111100
>>	FD	253	375	11111101
±	FE	254	376	11111110
▩	FF	255	377	11111111

E

Display Control Characters

Table E-1. Display Control Characters

Hex Value	Meaning
01h (SOH)	Turn on cursor.
02h (STX)	Turn off cursor.
06h (ACK)	High tone beep for 0.5 second.
07h (BEL)	Low tone beep for 0.5 second.
08h (BS)	Move cursor left one column. When the cursor reaches the left end of the line, it will back up to the right end of the previous line. When the cursor reaches the top left corner, backspace will have no effect.
0Ah (LF)	Move cursor down one line. If the cursor is on the bottom line, the display contents will scroll up one line.
0Bh (VT)	Clear every character from the cursor position to the end of the current line. The cursor position will be unchanged.
0Ch (FF)	Move cursor to upper left corner and clear the display.
0Dh (CR)	Move cursor to left end of current line.
0Eh (SO)	Change keyboard to numeric mode (underline cursor).
0Fh (SI)	Change keyboard to alpha mode (block cursor).
1Eh (RS)	Turn on display backlight.
1Fh (US)	Turn off display backlight.

F

Memory Map

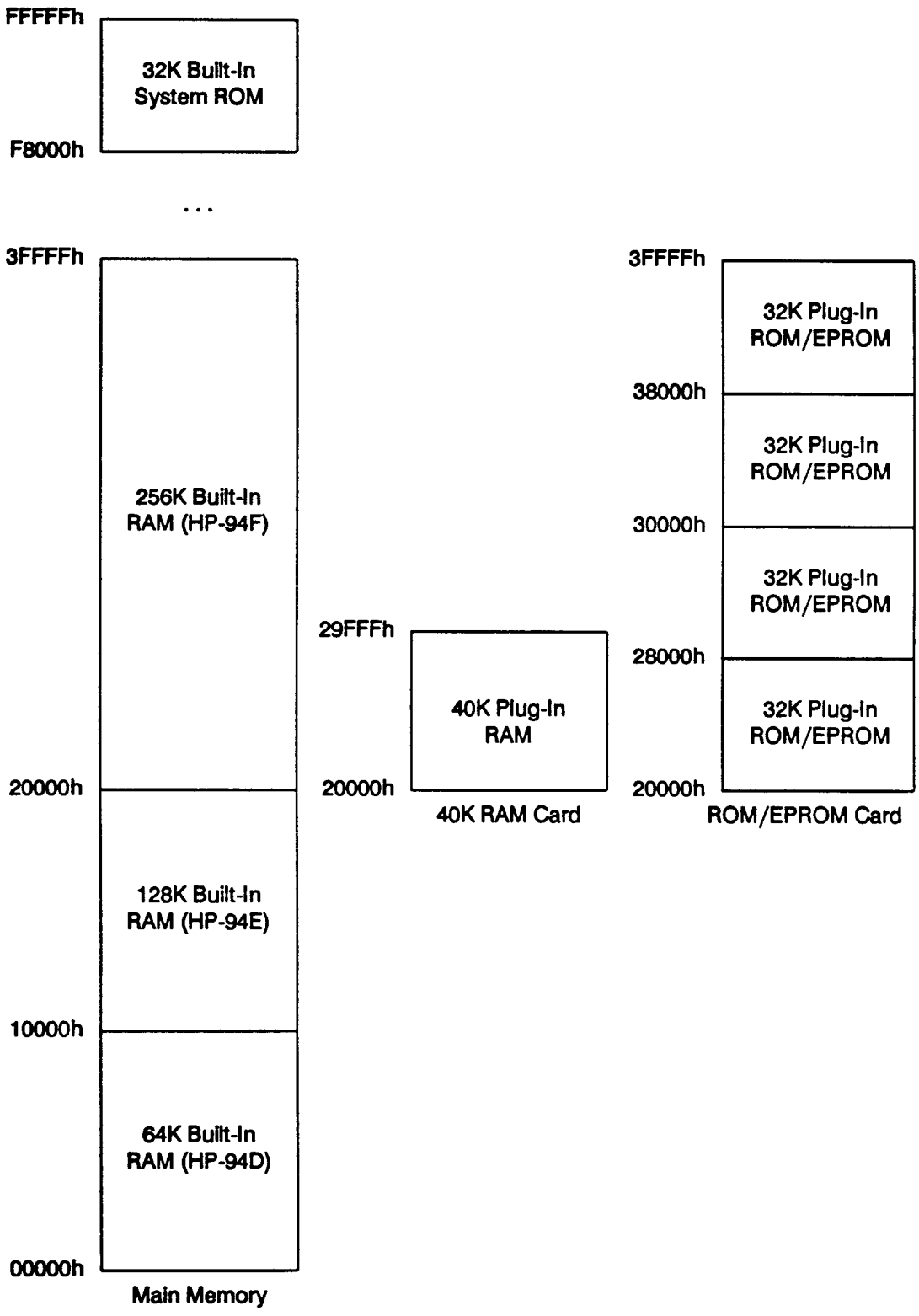


Figure F-1. Memory Map of the HP-94

G

Control and Status Register Addresses

Table G-1. I/O Addresses for Control and Status Registers

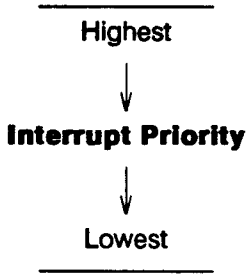
I/O Address	Register Name	Read/ Write
00h	Interrupt Control	W
00h	Interrupt Status	R
01h	Interrupt Clear	W
01h	End of Interrupt	R
02h	System Timer Data	R/W
03h	System Timer Control	W
04h	Bar Code Timer Data (lower 8 bits)	R/W
05h	Bar Code Timer Data (upper 4 bits)	R/W
06h	Bar Code Timer Control	W
07h	Bar Code Timer Value Capture	W
08h	Bar Code Timer Clear	W
0Ah	Baud Rate Clock Value	W
0Bh	Main Control	W
0Bh	Main Status	R
0Ch	Real-Time Clock Control	W
0Ch	Real-Time Clock Status/Data	R
0Eh	Keyboard Control	W
0Eh	Keyboard Status	R
10h	Serial Port Data	R/W
11h	Serial Port Control	W
11h	Serial Port Status	R
12h	Right LCD Driver Control	W
12h	Right LCD Driver Status	R
13h	Right LCD Driver Data	R/W
14h	Left LCD Driver Control	W
14h	Left LCD Driver Status	R
15h	Left LCD Driver Data	R/W
1Bh	Power Control	W

H

Hardware Interrupts

Table H-1. HP-94 Hardware Interrupts

Interrupt Type	Interrupt Name
50h	System Timer
51h	Bar Code Timer
52h	Bar Code Port Transition
53h	Serial Port Data Received
54h	Low Main Battery Voltage
55h	Power Switch
56h	Reserved Interrupt 1
57h	Reserved Interrupt 2



I

Operating System Functions

Table I-1. Operating System Function List

Name	Code	Description
BEEP	07h	Beep a high or low tone for specified duration
BUFFER_STATUS	06h	Get the number of bytes in or flush either the key buffer or the serial port handler receive buffer
CLOSE	10h	Close an I/O channel
CREATE	11h	Create a data file
CURSOR	05h	Read or change the cursor position on the LCD
DELETE	14h	Delete data file
DISPLAY_ERROR	18h	Display numeric error
END_PROGRAM	00h	Terminate the application program
FIND_FILE	16h	Find first occurrence of a file
FIND_NEXT	17h	Find subsequent occurrences of file
GET_CHAR	01h	Get a character from key buffer
GET_LINE	02h	Get a character string from the key buffer
GET_MEM	0Bh	Get a scratch area of memory
MEM_CONFIG	0Dh	Identify memory configuration
OPEN	0Fh	Open an I/O channel
PUT_CHAR	03h	Display a character on the LCD
PUT_LINE	04h	Display a character string on the LCD
READ	12h	Read data from an I/O channel
REL_MEM	0Ch	Release scratch area of memory
ROOM	0Eh	Identify available room in a directory
SEEK	15h	Move data file access pointer
SET_INTR	0Ah	Define power switch or low battery interrupt routines or disable/enable the power switch interrupt
TIMEOUT	09h	Set system or backlight timeout value
TIME_DATE	08h	Set or read the time and date on the real-time clock
WRITE	13h	Write data to an I/O channel

J

BASIC Interpreter Utility Routines

Table J-1. BASIC Interpreter Utility Routine List

Name	Offset	Description
ERROR	34h	Display error and end program
GETARG	3Ch	Convert real or integer into binary
IOERR	38h	Process errors in accordance with SYER
SADD	14h	Add two real numbers
SDIV	20h	Divide two real numbers
SETARG	40h	Convert binary into real or integer
SMUL	1Ch	Multiply two real numbers
SNEG	28h	Change sign of real number
SPOW	24h	Raise one real number to the power of another
SSUB	18h	Subtract two real numbers
TOBIN	30h	Convert integer or real into integer
TOREAL	2Ch	Convert integer or real into real

K

Program Resource Allocation

There are certain resources related to assembly language programs that must be chosen carefully to prevent conflict between different programs. Some of these resources are for any program, while others are for user-defined handlers only. These are as follows:

- Error Numbers
  These are used to report error conditions to calling BASIC or assembly language programs. BASIC programs can report numeric or non-numeric errors, although both internally map to an error number.
- Handler Identifier
  This is returned by the `IDENTIFY` function of the handler `IOCTL` routine.
- Valid Data Flag
  This is used to determine if the data in the parameter scratch area is correct for the handler being used.
- `IOCTL` Function Codes
  These are the function codes for the different functions in the handler `IOCTL` routine.

Refer to the "User-Defined Handlers" chapter in part 1, "Operating System", for details on the last three resources.

Below are tables summarizing usage of these resources by Hewlett-Packard programs. Remember that Hewlett-Packard also reserves `SY` as the first two characters of HP assembly language program and keyword names, and HN as the first two characters of HP handler names.

Table K-1. Error Number Usage

Error Number Range		
Start	**End**	**Reserved For**
00h (0)	00h (0)	No error
01h (1)	13h (19)	BASIC interpreter
64h (100)	77h (119)	Operating system
96h (150)	A9h (169)	*HP-94 Datacomm Utilities Pac*
C8h (200)	DBh (219)	Operating system

Table K-2. Hewlett-Packard Handler Resource Usage

Handler Name	Handler Identifier	Valid Data Flag	IOCTL Function Codes
HNBC	BC	FFh	00h-04h
HNSP	SP	FFh	00h-04h,80h
HNSG	SG	FEh	00h-04h,81h
Reserved	—	80h-FDh	05h-06h

To reserve resources for a particular program, a request should be made in writing to Hewlett-Packard. The request should indicate the resources required and their desired values. Also provide information about the software these resources will be used for (commercial applications for general sale, company-specific internal use only, etc.). This will help us allocate these limited resources as efficiently as possible. Send the request to:

Hewlett-Packard Portable Computer Division
Technical Marketing Software Support Group
1000 N.E. Circle Blvd.
Corvallis, OR 97330

If the desired value is available, it will be reserved for use by the program. If not, it will be necessary to select a different value for that resource.

L

Hewlett-Packard Bar Code Handlers

Hewlett-Packard supplies three bar code handlers with the *HP-94 Software Development System*:

- HNBC, a low-level bar code handler for the bar code port.
- HNSP, a low-level bar code handler for the serial port.
- HNWN, a high-level bar code handler for Hewlett-Packard Smart Wands (HP 39961D, HP 39963D, and HP 39965D).

These are all supplied as EXE files, and will all execute from RAM or ROM. This appendix will discuss details of these handlers important for assembly language programmers, including statistics, behavior of handler routines, errors, and parameter passing.

All three handlers follow the general behavior pattern described in the "User-Defined Handlers" chapter in part 1, "Operating System", so only the specific characteristics that are unique to each handler will be described here. This appendix assumes that the handler descriptions in the *HP-94 BASIC Reference Manual* have been read; that information will not be repeated here.

HNBC Low-Level Handler for Bar Code Port

HNBC is a low-level bar code handler for the bar code port. It is designed to allow "smart" bar code scanning devices to be connected to the bar code port — devices which do on-board decoding of bar code labels into ASCII, and return it as serial data. The HP-94 bar code port does not have a hardware UART to receive serial data, but HNBC performs the functions of a UART in software (assembling the serial bit stream into bytes, and checking for parity and framing errors).

HNBC is designed to work with bar code devices whose electrical characteristics match those of the HP-94 bar code port, and that send data in bursts of no more than 255 characters, with an intercharacter delay (time between characters) of 1-106 ms. HNBC is only supported for Hewlett-Packard Smart Wands (HP 39961D, HP 39963D, and HP 39965D).

HNBC Statistics

Here are pertinent statistics for HNBC.

Table L-1. HNBC Statistics

Item	Value
Version Number	1.00
Handler Identifier	BC
Valid Data Flag	FFh
Length in HP-94	2151 bytes
Scratch Areas Used	1 of 288 bytes *
Handler Information Table Offsets Used	00h, 04h
Valid Channel Numbers	2

* One additional 16-byte parameter scratch area is allocated if it was not allocated before opening the handler.

HNBC Capabilities

HNBC provides the following capabilities:

- Read-Only Operation
  Because the bar code port is read-only, no data can be written to it.
- Good Read Beep
  Automatic beep on successful decoding of a bar code label.
- Key Abort
  Allows a key being pressed to abort waiting for a successful scan.
- Received-Data Buffering
  Received data is placed in a 255-byte buffer. There is no transmit buffer.
- Speeds
  Speeds can be set from 150 to 9600 baud.
- Data Bits
  Seven only.
- Parity
  Zero, one, even, or odd parity.
- Stop Bits
  One only. For a receive-only port, all stop bits received after the first one are treated as intercharacter delay.
- Terminate Character Control
  When defined, a received terminate character will signal the end of the bar code data from the scanning device.

The table below describes how HNBC behaves. It shows the action taken by the handler routines as well as during its interrupt service routine, not including normal handler activities described in the "User-Defined Handlers" chapter. Note that certain actions, such as beeping on a good scan or responding to a received terminate character, will only occur if the appropriate options were enabled when the handler was opened.

Table L-2. Behavior of HNBC

Routine	Activities
CLOSE	Disable interrupt 52h and restore interrupt vector Turn off power to bar code port Release handler scratch area
IOCTL	Implement the reserved IOCTL functions 00h-04h
OPEN	Allocate parameter scratch area if needed Allocate handler scratch area Take over interrupt 52h vector Initialize operating configuration * Supply power to bar code port Return handler scratch area address in CX †
POWERON	Do nothing
READ	Wait for key up before accepting data Report error CDh (205) if no bar code device connected Discard data until none for 106 ms to avoid reading middle of label Return no data and error 75h (117) if read aborted by pressing key Monitor and report low battery, power switch, and timeout errors ‡ Enable interrupt 52h Wait for first byte received End read operation if no subsequent data received for 106 ms Compute parity of received data Report error 74h (116) if terminate character detected Report error 75h (117) if scanned length less than requested length Report errors detected in interrupt service routine Issue high beep if scan successful Return data from receive buffer
RSVD2	Do nothing
RSVD3	Do nothing
TERM	Flush receive buffer
WARM	Perform all OPEN routine activities except those related to scratch areas
WRITE	Return error 6Dh (109).
Interrupt Service	Read bar code status from main control register for all bits in each byte Monitor framing and receive buffer overflow errors Accumulate data into receive buffer

* Baud rate, parity, key abort, good read beep, and terminate character.

† Handlers are not required to return this, but HNBC does.

‡ System timeout only monitored until first byte received. After that, no data received for 106 ms signals the end of the label. Consequently, all characters must be received in a burst in which the intercharacter delay (time between characters) must be less than 106 ms.

CAUTION The HP-94 is unable to receive data through the serial port while the READ routine is executing. READ prevents this from happening by disabling the serial port data received interrupt. If both the serial and bar code ports must be open simultaneously, programs should halt serial port input before calling the HNBC READ routine (perhaps by sending an XOFF).

While the READ routine is executing, the background timer routine (interrupt 1Ch) must not clear the CPU interrupt flag (CLI), write the interrupt control register (00h) to enable any interrupts, or issue software interrupts (such as interrupt 1Ah for the operating system functions). Doing so may cause loss of bar code data, resulting in parity or framing errors.

The background timer accuracy will be degraded if the baud rate of the bar code device is less than 2400 baud or if the device sends its data with an intercharacter delay of less than 1 ms.

The errors reported by HNBC are shown in the following table.

Table L-3. Errors Reported by HNBC

Routine	Errors
CLOSE	6Eh
IOCTL	65h
OPEN	65h,67h,6Eh,71h
POWERON	None
READ	74h,75h,76h,77h,C8h,C9h,CAh,CDh,CEh
RSVD2	None
RSVD3	None
TERM	None
WARM	None
WRITE	6Dh
Interrupt Service *	C9h,CDh

* Detected by interrupt service routine, but reported by READ routine.

NOTE Two errors reported by the READ routine (74h, terminate character detected, and 75h, end of data) do not indicate error conditions, but signal the end of bar code data for the BASIC GET # and INPUT # statements. Assembly language programs using HNBC should handle these two errors differently than other errors from the READ routine.

If READ has not transferred all the data in its receive buffer when any read error occurs, it will flush the buffer. The next time READ is called, it will wait for a new bar code scan.

Parameters at OPEN Time

When HNBC is opened, it looks at offset 04h of the handler information table. If the value is zero, it allocates a one-paragraph parameter scratch area, places the default configuration in it, and places the scratch area address in the table. If the value is non-zero, it uses the value as the segment address of an existing parameter scratch area, and reads the configuration to use from that scratch area. The meanings of the parameters are shown below. In these figures, the offsets are from the start of the parameter scratch area. A copy of these parameters are pointed to by ES:DX in the GET_CONFIG and CHANGE_CONFIG reserved IOCTL functions (01h and 02h).

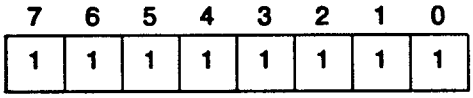


Figure L-1. HNBC Valid Data Flag — Parameter Byte 1 (Offset 00h)

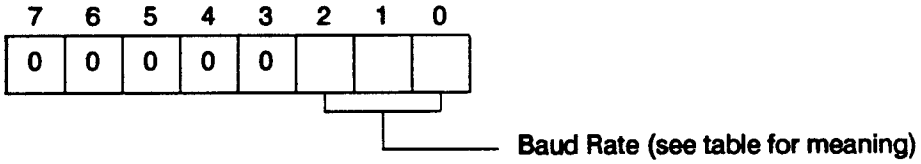
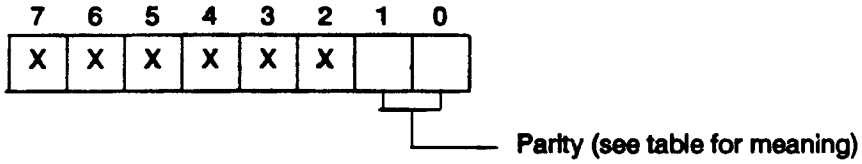


Figure L-2. HNBC Baud Rate — Parameter Byte 2 (Offset 01h)

Table L-4. HNBC Baud Rate Values

Value	Baud Rate
1	9600
2	4800
3	2400
4	1200
5	600
6	300
7	150

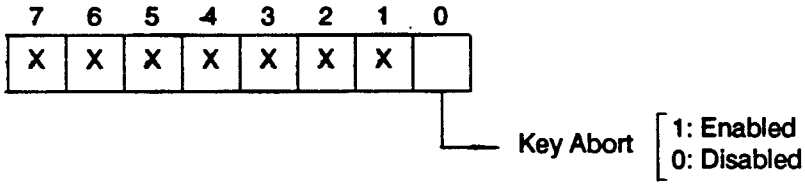


X = don't care

Figure L-3. HNBC Parity — Parameter Byte 3 (Offset 02h)

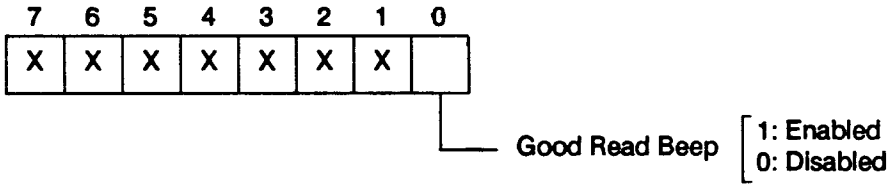
Table L-5. HNBC Parity Values

Value	Parity
0	Zero
1	One
2	Even
3	Odd



X = don't care

Figure L-4. HNBC Key Abort — Parameter Byte 4 (Offset 03h)



X = don't care

Figure L-5. HNBC Good Read Beep — Parameter Byte 5 (Offset 04h)

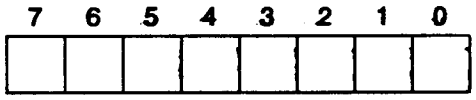


Figure L-6. HNBC Terminate Character * — Parameter Byte 6 (Offset 05h)

The default values for the parameters are FFh (valid data flag), 01h (9600 baud), 00h (zero parity), 01h (key abort enabled), 01h (good read beep enabled), and 00h (no terminate character).

HNSP Low-Level Handler for Serial Port

HNSP is a low-level bar code handler for the serial port. It is designed to allow "smart" bar code scanning devices to be connected to the serial port — devices which do on-board decoding of bar code labels into ASCII, and return it as serial data.

HNSP is designed to work with bar code devices whose electrical characteristics match those of the HP-94 serial port, and that send data in bursts of no more than 255 characters, with an intercharacter delay (time between characters) of 0-106 ms. HNSP is only supported for Hewlett-Packard Smart Wands (HP 39961D, HP 39963D, and HP 39965D).

HNSP Statistics

Here are pertinent statistics for HNSP.

Table L-6. HNSP Statistics

Item	Value
Version Number	1.00
Handler Identifier	SP
Valid Data Flag	FFh
Length in HP-94	2332 bytes
Scratch Areas Used	1 of 288 bytes *
Handler Information Table Offsets Used	02h
Valid Channel Numbers	1

* One additional 16-byte parameter scratch area is allocated if it was not allocated before opening the handler.

HNSP Capabilities

HNSP provides the following capabilities:

- Read/Write Operation
  Bar code data can be read from the port, and commands and data can be written to the bar code device. XON/XOFF handshaking is automatically used to pace transmission only.
- Good Read Beep
  Automatic beep on successful decoding of a bar code label.

* To disable use of the terminate character, set it to zero.

- Key Abort
  Allows a key being pressed to abort waiting for a successful scan.
- Received-Data Buffering
  Received data is placed in a 255-byte buffer. There is no transmit buffer.
- Speeds
  Speeds can be set from 150 to 9600 baud.
- Data Bits
  Seven only.
- Parity
  Zero, one, even, or odd parity.
- Stop Bits
  One for received data like HNBC. Two for transmitted data (actually, one plus intercharacter delay), which allows transmitting to devices that use either one or two stop bits.
- Terminate Character Control
  When defined, a received terminate character will signal the end of the bar code data from the scanning device. A terminate character will be sent after sending every block of data.

The table below describes how HNSP behaves. It shows the action taken by the handler routines as well as during its interrupt service routine, not including normal handler activities described in the "User-Defined Handlers" chapter. Note that certain actions, such as beeping on a good scan or responding to a received terminate character, will only occur if the appropriate options were enabled when the handler was opened.

Table L-7. Behavior of HNSP

Routine	Activities
CLOSE	Complete transmission of current byte Disable interrupt 53h and restore interrupt vector Lower RTS and DTR Wait 60 ms for signals to stabilize Disable 82C51 and turn off power to serial port Release handler scratch area
IOCTL	Implement the reserved IOCTL functions 00h-04h and 80h
OPEN	Allocate parameter scratch area if needed Allocate handler scratch area Take over interrupt 53h vector Enable 82C51 and supply power to serial port Initialize operating configuration * Raise RTS and DTR Return handler scratch area address in CX †
POWERON	Do nothing
READ	Wait for key up before accepting data Discard data until none for 106 ms to avoid reading middle of label Return no data and error 75h (117) if read aborted by pressing key Monitor and report low battery, power switch, and timeout errors ‡ Enable interrupt 53h Wait for first byte received End read operation if no subsequent data received for 106 ms Compute parity of received data Report error 74h (116) if terminate character detected Report error 75h (117) if scanned length less than requested length Report errors detected in interrupt service routine Issue high beep if scan successful Return data from receive buffer
RSVD2	Do nothing
RSVD3	Do nothing
TERM	Flush receive buffer
WARM	Perform all OPEN routine activities except those related to scratch areas
WRITE	Monitor and report low battery, power switch, and timeout errors Monitor CTS indirectly and report error DAh (218) if lost Write data to 82C51 Send terminate character at end of data
Interrupt Service	Monitor parity, framing, overrun, and receive buffer overflow errors Read data from 82C51 and accumulate data into receive buffer Disable/enable transmission when XOFF/XON received

* Baud rate, parity, key abort, good read beep, and terminate character.

† Handlers are not required to return this, but HNSP does.

‡ System timeout only monitored until first byte received. After that, no data received for 106 ms signals the end of the label. Consequently, all characters must be received in a burst in which the intercharacter delay (time between characters) must be less than 106 ms.

CAUTION While the READ routine is executing, the background timer routine (interrupt 1Ch) must not clear the CPU interrupt flag (CLI), write the interrupt control register (00h) to enable any interrupts, or issue software interrupts (such as interrupt 1Ah for the operating system functions). Doing so may cause loss of bar code data, resulting in parity or framing errors.

The errors reported by HNSP are shown in the following table.

Table L-8. Errors Reported by HNSP

Routine	Errors
CLOSE	6Eh
IOCTL	65h
OPEN	65h,67h,6Eh,71h
POWERON	None
READ	74h,75h,76h,77h,C8h,C9h,CAh,CBh,CCh,CDh,CEh,CFh,D0h
RSVD2	None
RSVD3	None
TERM	None
WARM	None
WRITE	76h,77h,C8h,DAh
Interrupt Service *	C9h,CAh,CBh,CCh,CDh,CEh,CFh,D0h
* Detected by interrupt service routine, but reported by READ routine.	

NOTE Two errors reported by the READ routine (74h, terminate character detected, and 75h, end of data) do not indicate error conditions, but signal the end of bar code data for the BASIC GET # and INPUT # statements. Assembly language programs using HNSP should handle these two errors differently than other errors from the READ routine.

If READ has not transferred all the data in its receive buffer when any read error occurs, it will flush the buffer. The next time READ is called, it will wait for a new bar code scan.

Parameters at OPEN Time

When HNSP is opened, it looks at offset 02h of the handler information table. If the value is zero, it allocates a one-paragraph parameter scratch area, places the default configuration in it, and places the scratch area address in the table. If the value is non-zero, it uses the value as the segment address of an existing parameter scratch area, and reads the configuration to use from that scratch area. The

meanings of the parameters are shown below. In these figures, the offsets are from the start of the parameter scratch area. A copy of these parameters are pointed to by ES:DX in the GET_CONFIG and CHANGE_CONFIG reserved IOCTL functions (01h and 02h).

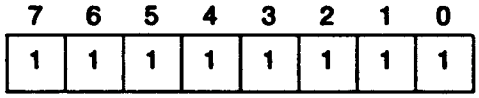


Figure L-7. HNSP Valid Data Flag — Parameter Byte 1 (Offset 00h)

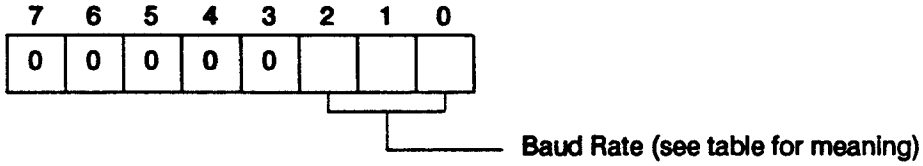
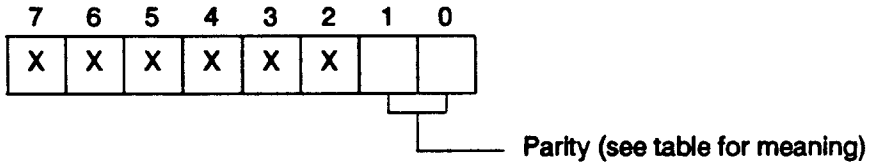


Figure L-8. HNSP Baud Rate — Parameter Byte 2 (Offset 01h)

Table L-9. HNSP Baud Rate Values

Value	Baud Rate
1	9600
2	4800
3	2400
4	1200
5	600
6	300
7	150

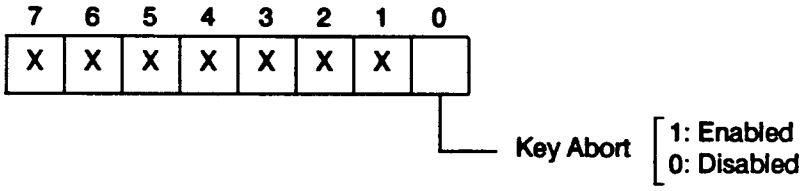


X = don't care

Figure L-9. HNSP Parity — Parameter Byte 3 (Offset 02h)

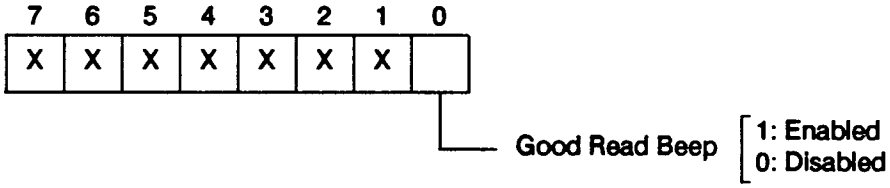
Table L-10. HNSP Parity Values

Value	Parity
0	Zero
1	One
2	Even
3	Odd



X = don't care

Figure L-10. HNSP Key Abort — Parameter Byte 4 (Offset 03h)



X = don't care

Figure L-11. HNSP Good Read Beep — Parameter Byte 5 (Offset 04h)

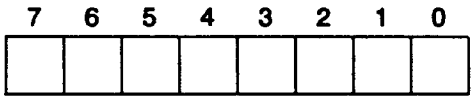


Figure L-12. HNSP Terminate Character * — Parameter Byte 6 (Offset 05h)

The default values for the parameters are FFh (valid data flag), 01h (9600 baud), 00h (zero parity), 01h (key abort enabled), 01h (good read beep enabled), and 00h (no terminate character).

* To disable use of the terminate character, set it to zero.

Write With Read Enabled IOCTL Function

HNSP implements an additional IOCTL function, function 80h, called WR_RD_EN (*write with read enabled*). It is invoked by setting AH to 80h when calling the HNSP IOCTL routine, and it returns 00h in AL (no errors). This is used when requesting status from a bar code device. To request status from a Hewlett-Packard Smart Wand, for example, a program would normally send it an escape sequence via the WRITE routine. The Smart Wand returns its status almost immediately — before the HNSP READ routine is ready to accept it. The READ routine will not read this status successfully because it ignores all data received after it is called until a quiet period of 106 ms has elapsed.

The WR_RD_EN function enables HNSP to receive data that arrives immediately after its WRITE routine completes writing data to the bar code device. The received data is stored in a buffer and returned to the calling program the next time the READ routine is called.

When the WR_RD_EN function is called, it enables a write with read enabled *only for the next write operation*. The next time the WRITE routine is called, it will actually do two separate I/O operations: first a write, then a read.

- The WRITE routine first performs the write operation the same way it would for a normal write. WRITE writes CX bytes of data starting at ES:BX out the serial port (for HP Smart Wands, this would be the status request escape sequence).
- The WRITE routine then performs the read operation the same way it would for a normal read. It waits until it receives the data from the serial port, or until the system timeout period expires. Once a byte has been received, the system timeout is no longer monitored, and WRITE assumes all data has arrived when the serial port does not receive any bytes for 106 ms.
- After all data has arrived, the WRITE routine checks for parity and framing errors. It does *not* beep, even if the beeper is enabled for normal bar code data. WRITE stores the received data in the receive buffer.
- WRITE returns the number of bytes actually written (*not* the number of bytes read) in CX and the error code in AL (00h if no errors). The error code is for both the write *and* read portions of the operation. The calling program will not know whether the error occurred during the write or the read, except for the context of the error message. For example, error DAh (218) can only occur during a write, while error CAh (202) can only occur during a read.

The next time the READ routine is called, it behaves as a normal read with data already available in the receive buffer, even though the data was actually received by the WRITE routine. The number of bytes actually read is returned in CX, and the data is returned in the read buffer specified by the READ caller.

The next time the WRITE routine is called, it behaves as a normal write — no write with read enabled operation will be performed unless WR_RD_EN is called again.

Because WR_RD_EN is treated as two separate I/O operations, the system timeout is restarted twice. It is started for the write operation and then stopped when the write is completed. It is then restarted for the read operation and stopped when the read is complete. If a system timeout occurs during either operation, AL is set to 76h (118). (For the read operation, it is only monitored until the first byte is received. After that, no data received for 106 ms signals the end of the label.)

XON/XOFF Handshaking During WR_RD_EN

XON/XOFF handshaking is normally done only during the WRITE routine. When the write with read enabled operation occurs, however, XON/XOFF handshaking is performed during both the write *and* the read portions of the operation. If status information returned by a bar code device contains an XON or an XOFF as legitimate data, those characters will be used to pace communications. They will not be passed back to the caller as part of the status.

HP Smart Wands do not send XON or XOFF characters as part of their status information.

HNWN High-Level Handler for Bar Code Handlers

HNWN is a high-level bar code handler for either the bar code port or the serial port. Because it is a high-level handler, it only communicates with low-level handlers, and specifically with HNBC and HNSP. It is designed to accommodate the unique features of Hewlett-Packard Smart Wands (HP 39961D, HP 39963D, and HP 39965D), and is only supported for these devices.

Throughout this section, there are references to features, behavior, and escape sequences sent or recognized by the Smart Wand. Refer to the *HP Smart Wand User's Manual* (part number HP 39960-90001) for details. There are also references to *configuration menus*, which provide optical configuration of the Smart Wand. This allows changing the Smart Wand's behavior by scanning bar code labels that are interpreted as commands, not as data. (Since the HP-94 bar code port is read-only, commands to change configuration cannot be sent to the Smart Wand through the bar code port). Refer to the *Smart Wand Configuration Menus* (part number HP 39960-90002) for details.

HNWN Statistics

Here are pertinent statistics for HNWN.

Table L-11. HNWN Statistics

Item	Value
Version Number	1.00
Handler Identifier	WN
Valid Data Flag	FFh
Length in HP-94	2217 bytes
Scratch Areas Used	1 of 272 bytes *
Handler Information Table Offsets Used	02h or 04h
Valid Channel Numbers	1 and 2
Valid Low-Level Handlers	HNBC, HNSP

* One additional 16-byte parameter scratch area is allocated if it was not allocated before opening the handler.

HNWN Capabilities

HNWN provides the following capabilities:

- Ignore or transmit Smart Wand escape sequences
  Causes escape sequences sent by the Smart Wand when it is in configuration mode to be sent to the calling program. Different beeps than for normal bar code data help distinguish received configuration escape sequences.
- Synchronize parity and baud rate of HP-94 port and Smart Wand
  Allows the HP-94 serial port or bar code port to track the Smart Wand's parity and baud rate without closing and reopening the port.

The table below describes how HNWN behaves. It shows the action taken by the handler routines, not including normal handler activities described in the "User-Defined Handlers" chapter. Note that certain actions, such as responding to escape sequences from the HNWN caller or from the Smart Wand, will only occur if the appropriate options were enabled when the handler was opened. Since high-level handlers interact with low-level handlers but not with I/O port hardware, HNWN has no interrupt service routine.

Table L-12. Behavior of HNWN

Routine	Activities
CLOSE	Call low-level handler CLOSE routine Release scratch area
IOCTL	Call low-level handler IOCTL routine
OPEN	Allocate parameter scratch area if needed Allocate handler scratch area Call low-level handler OPEN routine
POWERON	Do nothing
READ	Read data from low-level handler by calling its READ routine Ignore or transmit escape sequences
RSVD2	Call low-level handler RSVD2 routine
RSVD3	Call low-level handler RSVD3 routine
TERM	Call low-level handler TERM routine
WARM	Call low-level handler WARM routine
WRITE	Parse escape sequences being sent to low-level handler Take appropriate action for special escape sequences * Pass data to low-level handler by calling its WRITE routine

* Discussed later in this section.

NOTE HNWN cannot be used by itself — it must be used in conjunction with either HNBC or HNSP. To open HNWN with one of the low-level handlers, use the following as the handler name given to the BASIC OPEN # statement or the OPEN function (0Fh):

"HNWN;HNBC" for the bar code port (channel 2)

"HNWN;HNSP" for the serial port (channel 1)

When the low-level handlers are copied into the HP-94, their file names must be either of the low-level handlers must be either HNBC or HNSP — if the file names are different, HNWN will not be able to open them.

The errors reported by HNWN are shown in the following table. In addition, HNWN will report errors returned to it by either HNBC or HNSP.

Table L-13. Errors Reported by HNWN

Routine	Errors
CLOSE	6Eh
IOCTL	None
OPEN	65h,66h,67h,6Eh,71h
POWERON	None
READ	None
RSVD2	None
RSVD3	None
TERM	None
WARM	None
WRITE	None

Parameters at OPEN Time

When HNWN is opened, it looks in the handler information table. If it is opened to channel 1, it looks at offset 02h of the table. If it is opened to channel 2, it looks at offset 04h of the table. If the value is zero, it allocates a one-paragraph parameter scratch area, places the default configuration in it, and places the scratch area address in the table. If the value is non-zero, it uses the value as the segment address of an existing parameter scratch area, and reads the configuration to use from that scratch area. The meanings of the parameters are shown below. In these figures, the offsets are from the start of the parameter scratch area.

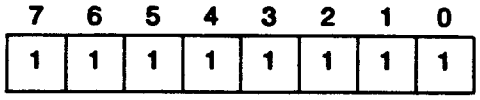


Figure L-13. HNWN Valid Data Flag — Parameter Byte 1 (Offset 08h)

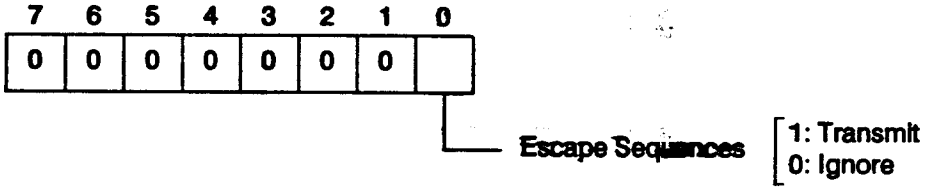


Figure L-14. HNWN Escape Sequences — Parameter Byte 2 (Offset 09h)

The default values for the parameters are FFh (valid data flag) and 00h (ignore escape sequences).

Response to Escape Sequences From Smart Wand

When the Smart Wand scans bar codes in a configuration menu, it sends one of six types of responses:

- Configuration Complete (Esc *)
  This is sent to signify that the Smart Wand has completed the configuration operation specified by the menu.
- Configuration Partially Complete (Esc \ +)
  This is sent to signify the Smart Wand has completed a portion of the configuration operation. This is sent for intermediate steps in configuration operations that require more than one scan.
- Syntax Error (Esc \ -)
  This is sent to signify that the configuration menu was out of context. This may be caused by scanning configuration bar codes in the wrong order, that are the wrong type, or that are numerically out of range.
- Configuration Dump (Esc \ * CR LF Esc \ ...)
  This contains status information about the Smart Wand. If the Smart Wand is in HP-94 default mode, the length of this status is 223 characters.
- Hard Reset Message (`ready XX.X`)
  This message is sent if the configuration bar code that specifies a hard reset is scanned. `XX.X` is the Smart Wand's firmware version number.
- No Read Message (user-defined, default is CR LF)
  This message is sent only if the Smart Wand is enabled to send the no read message and if the Smart Wand reads a bar code label but is unable to decode it.

HNWN provides special responses only for the four escape sequences. It treats the hard reset message and no read message the same as standard bar code data. It is the responsibility of the calling program to provide special handling of these messages.

When escape sequences are received, HNWN will respond in one of two ways:

- Ignore Escape Sequences (default behavior)
  If this mode is selected, HNWN will discard all strings received from the Smart Wand that begin with Esc \. There is no beep, and the string is not passed to the calling program. This mode may be used if it is desirable to prevent configuration messages from accidentally being interpreted by an application as legitimate bar code data.
- Transmit Escape Sequences
  In this mode HNWN will transmit to the calling program all strings received from the Smart Wand that begin with Esc \. When escape sequences are received, HNWN causes the HP-94 to generate different sounding beeps in response to the configuration mode escape sequences. These are generated only if there are no parity or framing errors when the configuration bar code was scanned, and are generated whether or not the good read beep is enabled for normal bar code data.

Table L-14. Beeps From HNWN for Smart Wand Escape Sequences

Smart Wand Escape Sequence	Number of Beeps	Beep Tone
Configuration Complete	4	High
Configuration Partially Complete	2	High
Syntax Error	4	Low

NOTE Because Code 128 bar code labels can contain any of 128 ASCII characters, it is possible (although unlikely) to encounter Code 128 labels that decode to strings beginning with Esc \. If such labels are encountered, HNWN will respond to them as if they were configuration sequences (assuming the transmit escape sequences option is being used). Applications that may encounter this situation should use HNWN with the ignore escape sequences option, or use HNBC or HNSP alone (without HNWN).

Response to Escape Sequences From Calling Program

The Smart Wand will respond to a number of escape sequences sent to it through its serial port (using HNSP). Four of these also invoke special responses from HNWN:

- Serial Port Configuration (Esc - y *n* p)
- Status Request (Esc - y *n* s)
- Hard Reset (Esc - y 1 z and Esc E)
- Save Configuration to Non-Volatile Memory (Esc - y 5 z)

These may be sent to HNWN;HNSP with the BASIC PRINT # and PUT # statements or with the WRITE function (13h). The last three cannot be sent to the Smart Wand using HNBC since the bar code port is read-only (the first one is handled by HNWN;HNBC as a special case). Refer to the "Hardware Specifications" for the pin assignments of a cable that will connect the serial port to the Smart Wand.

Serial Port Configuration Escape Sequence

The format of this escape sequence is as follows:

$$^{E}c\text{-}y\,n\,p$$

where *n* is a sequence of numeric characters (30h through 39h) that specifies a decimal number between 0 and 255. If this decimal number is converted to the equivalent binary number, the bit pattern has the following meaning:

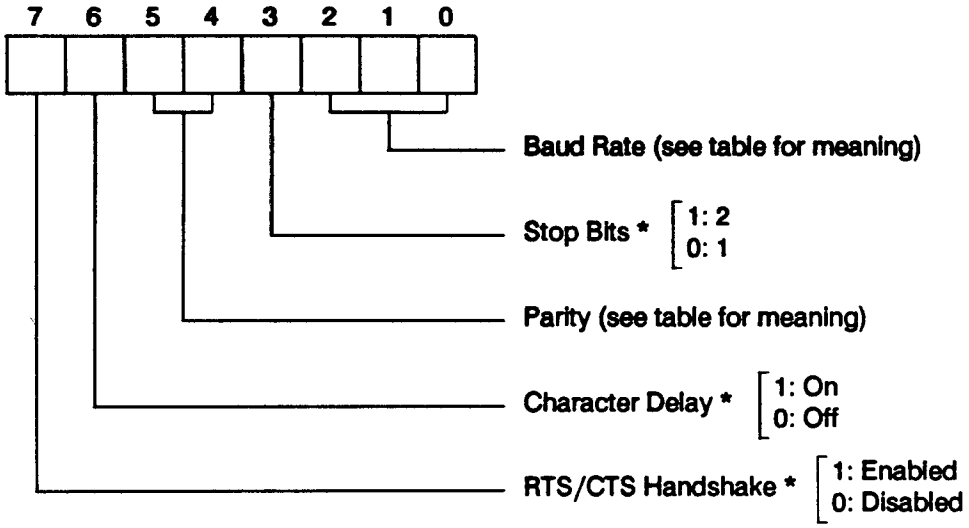


Figure L-15. Serial Port Configuration Escape Sequence

Table L-15. Smart Wand Baud Rate Values

Value	Baud Rate
0	150
1	300
2	600
3	1200
4	2400
5	4800
6	9600

* Ignored by HNWN. Only affects Smart Wand.

Table L-16. Smart Wand Parity Values

Value	Parity
0	Zero
1	One
2	Even
3	Odd

If the Smart Wand receives this escape sequence through its serial port, it changes its serial configuration as specified. For example, Esc - y 62 p would set the Smart Wand serial port to 9600 baud, 2 stop bits, odd parity, character delay off, and RTS/CTS handshake disabled (because 62 decimal corresponds to a bit pattern of 00111110).

The manner in which HNWN responds depends on which channel it is open to. If it is open to the serial port (channel 1) through HNSP, HNWN sends the escape sequence on to the Smart Wand at the current baud rate and parity. It then changes the baud rate and parity of the HP-94 serial port to the values specified by the sequence. This causes the Smart Wand and the HP-94 to track each other's serial configuration. When the port configuration is changed this way, the new configuration is assumed only for as long as the port is open. If the bar code or serial port is closed and then reopened, it will assume the baud rate and parity specified in the parameter scratch area at the time the port is reopened.

If HNWN is open to the bar code port (channel 2) through HNBC, HNWN changes the baud rate and/or the parity of the port. It does not try to write the escape sequence to the bar code port, since the port is read-only. (For this reason, the serial port configuration escape sequence is the only sequence that may be written to HNWN ; HNBC without causing an error.) It is assumed that immediately after this sequence is sent to HNWN ; HNBC, the operator will scan a configuration bar code that causes the Smart Wand to change parity and baud rate to match those of the HP-94. If this is not done, all subsequent scans will result in parity or framing errors.

If it is desirable to change both baud rate and parity for the bar code port, the baud rate should be changed first. The changes should be done as two separate operations, since each change involves sending an escape sequence to HNWN and having the operator scan the appropriate configuration bar code.

When the Smart Wand is powered off, then back on, it may or may not return to its default serial configuration. This depends on whether the Smart Wand's serial port configuration has been saved (discussed in "Save Configuration to Non-Volatile Memory Escape Sequence").

Status Request Escape Sequence

This escape sequence can be used to obtain various types of status from the Smart Wand. The format is as follows:

Esc - y *n* s

where *n* is a decimal number from 1-6. The meanings of the different values of *n* are as follows:

Table L-17. Status Request Escape Sequence Parameter

Value	Type of Status Returned
1	Status message followed by $^{C}_{R}$
2	Status message with selected trailer
3	Message ready/not ready response (for Single Read Mode 2)
4 *	Smart Wand configuration screen message
5	Serial number
6	Configuration dump †

* The Smart Wand responds to $^{E}_{C}$ - y 4 s by sending its configuration screen message, which is not usable by the HP-94. HNWN traps this sequence, does not send it to the Smart Wand, and returns error 65h (101).

† If the Smart Wand is in HP-94 default mode, this is 223 bytes long.

If this escape sequence is written to channel 1, it alters the behavior of HNWN;HNSP. Normally, HNWN;HNSP discards all data received by the serial port unless its READ routine is called. However, when this escape sequence is received, HNWN invokes the WR_RD_EN function of the HNSP IOCTL routine, writes the escape sequence to HNSP, then places the status returned by the Smart Wand in the receive buffer. The status will be returned to the calling program the next time the READ routine is called. The beeper does not sound when the status information is received, even if beeps are enabled for normal bar code data.

If this escape sequence is written to channel 2, HNWN;HNBC returns error 6Dh (109).

NOTE The status messages returned by the Smart Wand are escape sequences, and HNWN must be configured to transmit escape sequences in order for the calling program to receive the status messages.

Wand Hard Reset Escape Sequences

There are two escape sequences that cause the Smart Wand to perform a hard reset. These are:

$^{E}_{C}$ - y 1 z

$^{E}_{C}$ E

When the Smart Wand receives one of these escape sequences, it becomes unable to parse escape sequences for 516 ms (worst case), until the reset operation is complete. When HNWN receives either of these escape sequences, it sends it to the Smart Wand (through HNSP only) and then waits 530 ms before returning to the calling program or sending any more characters in the output string. This gives the Smart Wand enough time to perform the reset operation.

Save Configuration to Non-Volatile Memory Escape Sequence

This escape sequence has the following format:

$$^{E}c\text{-}y5z$$

It causes the Smart Wand to write its current configuration to the Smart Wand's built-in non-volatile memory (EEPROM). This operation requires 2.78 seconds (worst case). When HNWN receives this escape sequence, it sends it to the Smart Wand (through HNSP only) and then waits 2.9 seconds before returning to the calling program or sending any more characters in the output string. The HP-94 power switch is disabled during this period to prevent powering down of the HP-94 and the Smart Wand.

CAUTION The Smart Wand must not be powered down by turning off the HP-94 while the save configuration operation is in progress. Although the power switch is disabled, the reset switch or automatic turn off after very low battery could still turn the 94 and Smart Wand off. If this occurs, the Smart Wand may become inoperable, requiring that it be sent to a Hewlett-Packard service center to be restored to proper operation.

M

Disc-Based Utility Routines

The disc included with the *HP-94 Technical Reference Manual* contains 17 utility routines. These utilities are include files with the extension ASM. They can be included as part of assembly language programs (using the INCLUDE assembler directive), and can be executed from either RAM or ROM. Below is a list of all the utilities.

Table M-1. Utility Routines on Technical Reference Manual Disc

File Name	Description	Page
BLINK.ASM *	Blink the cursor	M-3
EQUATES.ASM	Equates for HP-94 operating system	M-5
FINDOS.ASM	Locate operating system file in system ROM	M-8
INTERNAL.ASM	Call internal entry point of BASIC keyword	M-10
IOABORT.ASM	Check for low battery, power switch, or timeout	M-14
IOWAIT.ASM	Enable I/O wait state	M-18
ISOPEN.ASM	Determine if a channel is open	M-20
LLHLINKG.ASM	Call low-level handler from high-level handler	M-22
NOIOWAIT.ASM	Disable I/O wait state	M-34
READCTRL.ASM	Examine hardware status	M-36
READINTR.ASM	Examine interrupt status	M-38
SCANKYBD.ASM *	Check if key down	M-40
SETCTRL.ASM	Write to saved copy of main control register	M-42
SETINTR.ASM	Write to saved copy of interrupt control register	M-44
VERSION.ASM *	Return version of operating system or program	M-46
XIOCTL.ASM	Execute IOCTL routine in any handler	M-49
XTIMEOUT.ASM *	Execute timeout process when timeout occurs	M-51

* Requires the FINDOS.ASM utility.

These utilities were written to be assembled using the Microsoft assembler MASM. Conditional assembly is used to allow the Hewlett-Packard copyright notice to appear in the source code, but not be printed in the list file (extension LST). The copyright notice allows the utilities to be reproduced for inclusion in an application or for archival purposes *without* prior written consent of Hewlett-Packard.

Conditional assembly is also used for the FINDOS utility. This utility is required by the BLINK, SCANKYBD, VERSION and XTIMEOUT utilities, and is included with each of them (using the INCLUDE assembler directive). The conditional assembly prevents FINDOS from being included more than once in the source file.

Utility Routine Descriptions

Utility routine descriptions consist of the following:

- A brief description of the utility.
- Information on when the utility should be used.
- Program listing.

The program listings start with a comment block that describes the following:

- What the utility does.
- How to call the utility.
- What is returned by the utility.
- Registers altered by the utility.

BLINK.ASM

The BLINK utility in this include file blinks the cursor. Normally, the system timer interrupt service routine causes the cursor to blink every 500 ms. However, in time-critical handlers such as for the bar code port, the system timer may be disabled while waiting for data to be received at the port (to prevent bar code port transition interrupts from being missed). BLINK performs the operating system cursor blink operation when the system timer is disabled.

The BLINK utility should be called approximately every 100 ms while the handler waits for data, thereby allowing the cursor to continue blinking. This helps prevent users from gaining the perception of no machine activity that accompanies an idle cursor. (The 100 ms calling interval is consistent with the frequency with which the system timer interrupt routine calls BLINK.)

BLINK uses FINDOS to find the operating system file and the start of the operating system jump table.

Program listing:

```
                        .sfcond
                        if1
;*************************************************************
;*     "(c) Copyright Hewlett-Packard Company, 1987.  All    *
;*     rights are reserved.  Copying or other reproduction   *
;*     of this program for inclusion in an application or    *
;*     for archival purposes is permitted without the prior  *
;*     written consent of Hewlett-Packard Company."          *
;*************************************************************
                        endif
                        .lfcond

                        include findos.asm

;*************************************************************
;*
;* Name: BLINK
;*
;* Version: 1.3
;*
;* Description:
;*   Call the cursor blink routine in the operating system
;*
;* Call with:
;*   None
;*
;* Returns:
;*   None
;*
;* Registers altered:
;*   None
;*
;* Notes:
;*   The BLINK routine is normally called every 100 ms by the
;*   system timer interrupt service routine, and should be called
;*   here only when the system timer interrupt is disabled.
;*
;*   The BLINK routine decrements a count in the operating system
;*   scratch space each time it is called.  The cursor state is
;*   changed only when the count reaches 0. When the cursor state
```

```
;*   is changed, the count is reset to 5.
;*
;*************************************************************

BLINK                   proc    near
                        push    bx
                        push    si
                        push    ds
                        push    es
                        call    FINDOS
; DS is SYOS segment
; ES is operating system pointer table segment
                        push    ds                  ; Push cursor blink routine segment
                        push    es:[38h]            ; Push cursor blink routine offset
                        mov     si,es:[00h]
                        mov     ds,si               ; DS = operating system data segment
                        mov     si,sp
                        call    dword ptr ss:[si]
                        add     sp,4
                        pop     es
                        pop     ds
                        pop     si
                        pop     bx
                        ret
BLINK                   endp
```

EQUATES.ASM

The EQUATES include file is a set of symbolic names for use in writing HP-94 assembly language programs. These include names for operating system functions, register locations in the register save area for handler routines, and certain operating system values and locations.

Program listing:

```
                        .sfcond
                        if1
;*****************************************************************
;*     "(c) Copyright Hewlett-Packard Company, 1986, 1987.  All *
;*     rights are reserved.  Copying or other reproduction of   *
;*     this program for inclusion in an application or for      *
;*     archival purposes is permitted without the prior written *
;*     consent of Hewlett-Packard Company."                     *
;*****************************************************************
                        endif
                        .lfcond

; Equate values for the HP-94

; Macros to push and pop registers in order expected by handlers
pushregs                macro
                        pushf
                        push        bp
                        push        es
                        push        ds
                        push        di
                        push        si
                        push        dx
                        push        cx
                        push        bx
                        push        ax
                        mov         bp,sp
                        endm

popregs                 macro
                        pop         ax
                        pop         bx
                        pop         cx
                        pop         dx
                        pop         si
                        pop         di
                        pop         ds
                        pop         es
                        pop         bp
                        popf
                        endm

; Symbolic names for registers relative to BP
; (e.g. AXREG[BP] is the saved value of AX)

AXREG                   equ         00h
ALREG                   equ         00h
AHREG                   equ         01h
;
BXREG                   equ         02h
BLREG                   equ         02h
BHREG                   equ         03h
```

```
;
CXREG                   equ     04h
CLREG                   equ     04h
CHREG                   equ     05h
;
DXREG                   equ     06h
DLREG                   equ     06h
DHREG                   equ     07h
;
SIREG                   equ     08h
DIREG                   equ     0Ah
DSREG                   equ     0Ch
ESREG                   equ     0Eh
BPREG                   equ     10h


OS_PTRTBL_SEG           equ     16h             ; Operating system pointer table segment
;
; HP-94 ADDRESSES
;
; OS_PTRTBL_SEG is a segment address of a table of operating system pointers.
; The contents of this table point to where the system addresses
; are located.  All entries in the address table are 2-byte entries.
; The values below are the offset addresses in OS_PTRTBL_SEG.
;
OSSCRATCH_SEG           equ     00h             ; System RAM data segment
;
SYSROM_SEG              equ     08h             ; System ROM segment
;
OPENTBLSIZE             equ     0Ah             ; Number of open table entries
;
RESTART_STATUS          equ     14h             ; Offset of status area in OSRAM_SEG
;
MAXDIR                  equ     18h             ; Offset of maximum directory # in OSRAM_SEG
EVENTTBL                equ     1Ah             ; Offset of system timer event table in OSRAM_SEG
;
OPENTBL                 equ     24h             ; Offset of channel # table in OSRAM_SEG
;
KEY_SCAN                equ     36h             ; Offset (in "SYOS" file) of jump to key scan routine
CURSOR_BLINK            equ     38h             ; Offset (in "SYOS" file) of jump to cursor blink rout
TIMEOUT                 equ     3Ah             ; Offset (in "SYOS" file) of jump to timeout utility
VERSION                 equ     3Ch             ; Offset (in "SYOS" file) of system ROM version number

;
; FUNCTION CALLS
;
; Each function call equate has two forms, one for directly loading
; AH (mov  ah,FUNCTION), the other for loading AX with a word value.
; The form for loading AX has "x100h" appended to the base name.
;
; Call operating system functions as follows:
;
;  Example which loads AH:
;    Output the line pointed to by ES:BX
;
;                       mov     ah,PUT_LINE
;                       int     1Ah
;
END_PROGRAM             equ     00h
GET_CHAR                equ     01h
```

```
GET_LINE                equ     02h
PUT_CHAR                equ     03h
PUT_LINE                equ     04h
CURSOR                  equ     05h
BUFFER_STATUS           equ     06h
BEEP                    equ     07h
TIME_DATE               equ     08h
TIMEOUT                 equ     09h
SET_INTR                equ     0Ah
GET_MEM                 equ     0Bh
REL_MEM                 equ     0Ch
MEM_CONFIG              equ     0Dh
ROOM                    equ     0Eh
OPEN                    equ     0Fh
CLOSE                   equ     10h
CREATE                  equ     11h
READ                    equ     12h
WRITE                   equ     13h
DELETE                  equ     14h
SEEK                    equ     15h
FIND_FILE               equ     16h
FIND_NEXT               equ     17h
DISPLAY_ERROR           equ     18h
;
;  Example which loads AX:
;    Output the letter 'j' to the LCD
;
;                       mov     ax,PUT_CHARx100h + "j"
;                       int     1Ah
;
END_PROGRAMx100h        equ     0000h
GET_CHARx100h           equ     0100h
GET_LINEx100h           equ     0200h
PUT_CHARx100h           equ     0300h
PUT_LINEx100h           equ     0400h
CURSORx100h             equ     0500h
BUFFER_STATUSx100h      equ     0600h
BEEPx100h               equ     0700h
TIME_DATEx100h          equ     0800h
TIMEOUTx100h            equ     0900h
SET_INTRx100h           equ     0A00h
GET_MEMx100h            equ     0B00h
REL_MEMx100h            equ     0C00h
MEM_CONFIGx100h         equ     0D00h
ROOMx100h               equ     0E00h
OPENx100h               equ     0F00h
CLOSEx100h              equ     1000h
CREATEx100h             equ     1100h
READx100h               equ     1200h
WRITEx100h              equ     1300h
DELETEx100h             equ     1400h
SEEKx100h               equ     1500h
FIND_FILEx100h          equ     1600h
FIND_NEXTx100h          equ     1700h
DISPLAY_ERRORx100h      equ     1800h
```

FINDOS.ASM

The FINDOS utility in this include file finds the operating system (file SYOS) in the system ROM. The FIND_FILE and FIND_NEXT functions (16h and 17h) cannot be used because running programs do not have access to directory 5, the system ROM directory. The FINDOS utility searches the system ROM directory table to locate SYOS.

This utility is used by the BLINK, SCANKYBD, and XTIMEOUT utilities, all of which utilities call routines whose locations are defined by a jump table at a known location in the operating system file. It is also used by VERSION to locate the version number at a known location.

Program listing:

```
                        .xlist      ; Suppress findoshere macro listing
findoshere              macro
                        .sfcond
                        if1
;*****************************************************************
;*     "(c) Copyright Hewlett-Packard Company, 1987.   All       *
;*     rights are reserved.  Copying or other reproduction       *
;*     of this program for inclusion in an application or        *
;*     for archival purposes is permitted without the prior      *
;*     written consent of Hewlett-Packard Company."              *
;*****************************************************************
                        endif
                        .lfcond

TABLE_SEG               EQU         16h
SYSROM_SEG              EQU         08h
;*****************************************************************
;*
;* Name: FINDOS
;*
;* Version: 1.3
;*
;* Description:
;*   Find the start of the SYOS file
;*
;* Call with:
;*   None
;*
;* Returns:
;*   DS = start of SYOS file (0 if SYOS not found)
;*   ES = start of operating system pointer table
;*
;* Registers altered:
;*   DS, ES
;*
;* Notes:
;*   If SYOS is not found, DS = 0.
;*
;*****************************************************************

FINDOS                  proc        near
                        push        bx
                        push        cx

                        mov         bx,TABLE_SEG
                        mov         es,bx               ; ES is TABLE_SEG
```

```
                        mov     bx,es:[SYSROM_SEG]
                        mov     ds,bx               ; DS is SYSROM_SEG
                        mov     cx,ds:[06h]         ; Get start of files pointer
                        sub     cx,bx               ; CX is number of paragraphs in directory
                        dec     cx                  ; Account for "*DIR*" entry
FIND1:
                        mov     bx,ds
                        inc     bx
                        mov     ds,bx               ; DS[0] is name
                        cmp     ds:[2],'O'+'S'*100h ; 'OS'
                        jne     FIND2
                        cmp     ds:[0],'S'+'Y'*100h ; 'SY'
                        je      FIND3
FIND2:
                        loop    FIND1
NOFIND:
                        sub     bx,bx
                        mov     ds,bx               ; Set DS = 0 (not found)
                        jmp     short FIND4
;-
FIND3:
                        mov     ds,ds:[7]           ; DS is SYOS segment
FIND4:
                        pop     cx
                        pop     bx
                        ret
FINDOS                  endp
                        endm
                        .list
                        ifndef  FINDOS
                        findoshere
                        endif
                        if      FINDOS eq $
                        findoshere
                        endif
```

INTERNAL.ASM

The INTERNAL utility in this include file calls the internal entry point of a type A file. The internal entry point is the address at offset 02h in the file — the second pair of bytes in the program header. It is used mainly for type A files that are new BASIC keywords, allowing access to the functionality of the keyword without using the interaction between the keyword and the BASIC interpreter. Refer to the "Program Execution" chapter in part 1, "Operating System", for details.

The INTERNAL utility calls the internal entry point with a FAR CALL, so the called program should end with a FAR RET.

Program listing:

```
                        .sfcond
                        if1
;*****************************************************************
;*     "(c) Copyright Hewlett-Packard Company, 1987.  All        *
;*     rights are reserved.  Copying or other reproduction       *
;*     of this program for inclusion in an application or        *
;*     for archival purposes is permitted without the prior      *
;*     written consent of Hewlett-Packard Company."              *
;*****************************************************************
                        endif
                        .lfcond

;*****************************************************************
;*
;* Name: INTERNAL
;*
;* Version: 1.3
;*
;* Description:
;*   Call the internal entry point of a type "A" file
;*
;* Call with:
;*   SS:SP+2 = segment address of file name
;*   SS:SP = offset address of file name
;*
;* Returns:
;*   AL = Error code:
;*        00h       No error
;*        65h (101) Illegal parameter
;*        66h (102) Invalid directory number
;*        67h (103) File not found
;*
;* Registers altered:
;*   AL (if the internal entry point is called, the return
;*   value in AL is the value returned by the internal entry
;*   point)
;*
;* Notes:
;*   INTERNAL verifies that the file is type "A", and that
;*   the internal entry point offset is within the file.
;*
;*   All registers passed to INTERNAL are preserved for the
;*   call to the internal entry point.
;*
;*   The address of the file name is passed on the stack so that
;*   all registers may be passed to the internal entry point routine
```

```
;*
;*************************************************************

FIND_FILE               equ     16h
BUFFER_SIZE             equ     0Eh

AXREG                   equ     00h
ALREG                   equ     00h
AHREG                   equ     01h
;
BXREG                   equ     02h
BLREG                   equ     02h
BHREG                   equ     03h
;
CXREG                   equ     04h
CLREG                   equ     04h
CHREG                   equ     05h
;
DXREG                   equ     06h
DLREG                   equ     06h
DHREG                   equ     07h
;
SIREG                   equ     08h
DIREG                   equ     0Ah
DSREG                   equ     0Ch
ESREG                   equ     0Eh
BPREG                   equ     10h
;
FLAGREG                 equ     12h
;
REGSAVE_SIZE            equ     14h
FILE_SEGMENT            equ     REGSAVE_SIZE+BUFFER_SIZE+4
FILE_OFFSET             equ     REGSAVE_SIZE+BUFFER_SIZE+2

pushregs                macro
                        pushf
                        push    bp
                        push    es
                        push    ds
                        push    di
                        push    si
                        push    dx
                        push    cx
                        push    bx
                        push    ax
                        mov     bp,sp
                        endm

popregs                 macro
                        pop     ax
                        pop     bx
                        pop     cx
                        pop     dx
                        pop     si
                        pop     di
                        pop     ds
                        pop     es
                        pop     bp
                        popf
                        endm
```

```
INTERNAL                proc    near
                        sub     sp,BUFFER_SIZE      ; Reserve file information buffer
;
                        pushregs
                        lea     dx,[bp+REGSAVE_SIZE]; offset of file information buffer
;
                        push    ss
                        pop     ds                  ; DS = SS
                        mov     es,FILE_SEGMENT[bp]
                        mov     bx,FILE_OFFSET[bp]
; DS:DX = address of file information buffer
                        mov     ah,FIND_FILE
                        int     1Ah                 ; O.S. function call
; Check for errors...
                        or      al,al
                        jnz     ENTRY_ERROR
; CX:DX = directory table entry of the file
                        push    ss
                        pop     ds
                        mov     si,sp
                        lea     si,[si+REGSAVE_SIZE]
; DS:SI = directory table entry of the file
                        mov     al,ds:[si+07h]
                        cmp     al,"A"
                        jne     ENTRY_NOT_A
                        mov     cx,ds:[si+0Ch]      ; high byte of end-of-data address
                        mov     dx,ds:[si+0Ah]      ; low word of end-of-data address
                        mov     ds,ds:[si+08h]      ; segment address of file
                        mov     ax,ds:[02h]         ; Internal entry point offset
                        cmp     ax,06h              ; Check for valid offset (must be >= 6)
                        jb      ENTRY_BAD
                        or      cx,cx
                        jnz     ENTRY_CALL
                        cmp     ax,dx               ; low word of end-of-data address
                        jae     ENTRY_BAD
; Address is OK
; DS is segment of file
ENTRY_CALL:
                        mov     FILE_SEGMENT[bp],ds ; segment of internal entry point
                        mov     FILE_OFFSET[bp],ax  ; offset of internal entry point
                        popregs                     ; restore all registers
                        add     sp,BUFFER_SIZE      ; discard buffer (no longer needed)
                        push    cs
                        push    sp                  ; leave room for offset of INTERNAL1
                        pushf
                        sub     sp,4                ; leave room for segment and offset addresses
                        push    bp
                        mov     bp,sp
                        push    ax
;
; Stack relative to BP (* means not yet filled in)
;
; 10h                   FILE_SEGMENT
; 0Eh                   FILE_OFFSET
; 0Ch                   Caller's return address (offset)
; 0Ah                   CS (my segment)
; 08h                   * offset of INTERNAL1
; 06h                   Flag register
; 04h                   * Segment of internal entry point
; 02h                   * Offset of internal entry point
```

```
; 00h                   my BP
;-02h                  my AX
;
                        mov     ax,offset INTERNAL1
                        mov     [bp+08h],ax
                        mov     ax,[bp+10h]         ; file segment address
                        mov     [bp+4],ax
                        mov     ax,[bp+0Eh]         ; file offset address
                        mov     [bp+2],ax
                        pop     ax
                        pop     bp
                        iret                        ; Internal entry point ends with a FAR RET
INTERNAL1:
                        ret     4                   ; NEAR RET and add 4 to SP
;-
ENTRY_NOT_A:
ENTRY_BAD:
                        mov     al,65h              ; Illegal parameter
ENTRY_ERROR:
                        mov     bp,sp
                        mov     ALREG[bp],al
                        popregs
                        add     sp,BUFFER_SIZE
                        ret     4                   ; NEAR RET and add 4 to SP

INTERNAL                endp
```

IOABORT.ASM

The IOABORT utility in this include file allows a handler to check for system errors that should cause I/O to be aborted: low battery, power switch pressed, and system timeout. IOABORT will report errors C8h (200), 77h (119), and 76h (118) respectively for these conditions, but only if the operating system I/O wait state has been enabled using IOWAIT. To use this during the READ or WRITE routine, the handler would do the following:

- Enable the operating system I/O wait state by calling IOWAIT.
- Call IOABORT periodically while waiting to receive or transmit data, and check if it returns an error code that indicates I/O should be aborted. It can be called as often as is convenient, such as in the main READ or WRITE routine wait loop. It should be called at least every second, since that is the system timeout resolution (although low battery or power switch may not occur exactly on a 1 second time boundary).
- If the timeout error is reported by IOABORT, the user-defined timeout interrupt routine defined by SET_INTR (0Ah) will *not* have been executed. The handler should call XTIMEOUT which will call the user-defined timeout interrupt routine if one was defined, or turn the machine off. If the low battery or power switch errors are reported by IOABORT, user-defined low battery or power switch interrupt routines defined by SET_INTR (0Ah) will *already* have been executed.
- Abort I/O by halting the process of receiving or sending data.
- Disable the operating system I/O wait state by calling NOIOWAIT.
- End the READ or WRITE routine, and return the error code from IOABORT to the caller.

IOABORT must be used in conjunction with IOWAIT, which sets the operating system I/O wait state. The tables below show how the I/O wait state affects how each of these error conditions are reported by IOABORT.

Table M-2. Low Battery Interrupt Routine Behavior During I/O

I/O Wait State	User-Defined Behavior	Default Behavior
Waiting *	IOABORT reports error C8h (200). User-defined low battery interrupt routine executed when low battery condition occurs. †	Program halted, Error 200 displayed, and machine waits for power switch to be pressed to turn off.
Not waiting	IOABORT does not report an error. User-defined low battery interrupt routine executed when low battery condition occurs.	Program halted, Error 200 displayed, and machine waits for power switch to be pressed to turn off.

* Only if IOWAIT was called.

† Routine has already been executed by the time IOABORT reports the error.

Table M-3. Power Switch Interrupt Routine Behavior During I/O

I/O Wait State	User-Defined Behavior	Default Behavior
Waiting *	IOABORT reports error 77h (119). User-defined power switch interrupt routine executed when power switch pressed. † Error not reported and interrupt routine not called if power switch disabled.	Machine turns off. No default action taken if power switch disabled.
Not waiting	IOABORT does not report an error. User-defined power switch interrupt routine executed when power switch pressed. Interrupt routine not called if power switch disabled.	Machine turns off.

* Only if IOWAIT was called.
† Routine has already been executed by the time IOABORT reports the error.

Table M-4. Timeout Interrupt Routine Behavior During I/O

I/O Wait State	User-Defined Behavior	Default Behavior
Waiting *	IOABORT reports error 76h (118). Handler must call XTIMEOUT, which will execute user-defined timeout interrupt routine. Error not reported if timeout disabled.	IOABORT reports error 76h (118). Handler must call XTIMEOUT, which will turn machine off. Error not reported if timeout disabled.
Not waiting	IOABORT does not report an error. User-defined timeout interrupt routine not executed.	IOABORT does not report an error. No default action taken.

* Only if IOWAIT was called.

Program listing:

```
                        .sfcond
                        if1
;*****************************************************************
;*     "(c) Copyright Hewlett-Packard Company, 1986.  All        *
;*     rights are reserved.  Copying or other reproduction       *
;*     of this program for inclusion in an application or        *
;*     for archival purposes is permitted without the prior      *
;*     written consent of Hewlett-Packard Company."              *
;*****************************************************************
                        endif
                        .lfcond

;*****************************************************************
```

...IOABORT.ASM

```
;*
;* Name: IOABORT
;*
;* Version: 1.3
;*
;* Description:
;*   Check for any error conditions while waiting for data
;*   (designed for use by a handler while doing I/O)
;*
;* Call with:
;*   None
;*
;* Returns:
;*   AL = Error code:
;*        00h        No error
;*        76h (118)  Timeout
;*        77h (119)  Power switch pressed
;*        C8h (200)  Low battery
;*
;* Registers altered:
;*   AL
;*
;* Notes:
;*   Timeout is checked first, followed by low battery,
;*   and finally power switch pressed; if there are multiple
;*   error conditions, only the first one found will be
;*   reported.  Subsequent calls to IOABORT will report
;*   any errors not previously reported.
;*
;*   IOABORT assumes that IOWAIT and NOIOWAIT are used
;*   by the handler to set up timeout processing.
;*
;*************************************************************

IOABORT                 proc        near

                        push        bx
                        push        cx
                        push        ds
                        mov         bx,16h
                        mov         ds,bx
                        mov         bx,ds:[14h]
                        add         bx,4                ; BX points to current status
                        mov         ds,ds:[00h]
                        mov         al,ds:[bx]
                        mov         cx,0FE76h
                        test        al,01H              ;timeout?
                        jne         IOABORT1            ;(yes)
                        mov         cx,0F7C8h
                        test        al,08h              ;low battery?
                        jne         IOABORT1            ;(yes)
                        mov         cx,0EF77h
                        test        al,10H              ;power SW off?
                        jne         IOABORT1            ;(yes)
                        mov         cx,0FF00h
IOABORT1:
                        and         byte ptr ds:[bx],ch ;clear flag with CH
                        mov         al,cl
                        pop         ds
                        pop         cx
```

```
                        pop     bx
                        ret
IOABORT                 endp
```

IOWAIT.ASM

The IOWAIT utility in this include file enables the operating system I/O wait state. This is the state in which low battery, power switch, and timeout errors can be reported by the IOABORT utility while handler READ or WRITE routines are waiting for I/O.

The low battery error will occur when the operating voltage drops to 4.6 ± 0.05 volts or below. The power switch error will occur when the power switch is pressed. It will not occur if the power switch has been disabled using the SET_INTR function (0Ah). The timeout error will occur when the current system timeout value expires. The system timeout value is set by the TIMEOUT function (09h), and has a default time of 120 seconds. The timeout error will not occur if the timeout has been disabled by setting it to zero.

Whenever IOWAIT is called, it resets the timeout to the system timeout value. This allows a handler to restart the timeout period after each byte is sent or received by calling IOWAIT again.

When a handler READ or WRITE routine ends, it must call NOIOWAIT to indicate that I/O is not waiting.

The I/O wait state set by IOWAIT determines how IOABORT reports these error conditions. Refer to the IOABORT utility for details.

Program listing:

```
                        .sfcond
                        if1
;*************************************************************
;*     "(c) Copyright Hewlett-Packard Company, 1986.  All    *
;*     rights are reserved.  Copying or other reproduction   *
;*     of this program for inclusion in an application or    *
;*     for archival purposes is permitted without the prior  *
;*     written consent of Hewlett-Packard Company."          *
;*************************************************************
                        endif
                        .lfcond

;*************************************************************
;*
;* Name: IOWAIT
;*
;* Version: 1.4
;*
;* Description:
;*   Enable I/O wait state
;*
;* Call with:
;*   None
;*
;* Returns:
;*   None
;*
;* Registers altered:
;*   None
;*
;* Notes:
;*   Enables timeout if timeout interval is not zero.
```

```
;*
;*************************************************************

IOWAIT          proc    near
                push    ax
                push    bx
                push    si
                push    di
                push    bp
                push    ds
                mov     ax,16h
                mov     ds,ax
                mov     bx,ds:[14h]
                add     bx,4                    ; BX points to current status
                mov     si,ds:[1Ah]             ; System timer event control table
                mov     di,ds:[18h]
                mov     bp,di
                add     di,2                    ; DI points to timeout interval
                add     bp,0bh                  ; BP points to timeout counter
                mov     ds,ds:[00h]             ; DS = operating system data segment
                or      byte ptr ds:[bx],20h; I/O waiting flag set
                mov     ax,word ptr ds:[di] ; Read timeout interval
                or      ax,ax
                jz      IOWAIT9                 ; No timeout if zero
                mov     ds:[bp],ax              ; Timeout counter set to timeout interval
                mov     byte ptr ds:13[si],200; Timeout enable (1 second)
IOWAIT9:
                pop     ds
                pop     bp
                pop     di
                pop     si
                pop     bx
                pop     ax
                ret
IOWAIT          endp
```

ISOPEN.ASM

The ISOPEN utility in this include file checks if a channel is open. The primary use of ISOPEN is for configuration programs to determine if a handler is already open.

When a handler is closed, configuration programs create a parameter scratch area, write configuration parameters into it, and put the scratch area address in the appropriate entry in the handler information table. If the scratch area already exists, the handler information table entry will already point to the scratch area, and the configuration program will write its parameters into the existing scratch area.

When a handler is open, however, the entry in the handler information table is the address of the handler scratch area, not of the parameter scratch area. If the configuration program is run after the handler is open, it could misinterpret the handler information table entry, and modify the handler scratch area by mistake. ISOPEN allows configuration programs to check if the handler is open regardless of the meaning of the entry in the handler information table. This allows configuration programs to take different action depending on whether or not the handler is open. Refer to the "User-Defined Handlers" chapter in part 1, "Operating System", for information about using the handler information table.

Program listing:

```
                        .sfcond
                        if1
;*************************************************************
;*     "(c) Copyright Hewlett-Packard Company, 1986.  All    *
;*     rights are reserved.  Copying or other reproduction   *
;*     of this program for inclusion in an application or    *
;*     for archival purposes is permitted without the prior  *
;*     written consent of Hewlett-Packard Company."          *
;*************************************************************
                        endif
                        .lfcond

;*************************************************************
;*
;* Name: ISOPEN
;*
;* Version: 1.3
;*
;* Description:
;*   Check if a channel is open
;*
;* Call with:
;*   AL = Channel number
;*
;* Returns:
;*   AL = Error code:
;*        00h       No error (channel open)
;*        65h (101) Illegal parameter
;*        69h (105) Channel not open
;*
;* Registers altered:
;*   AL
;*
;* Notes:
;*   None
;*
```

```
;*************************************************************

OBIT                    EQU     10h

ISOPEN                  proc    near
                        push    ds
                        push    si
                        push    ax
                        mov     si,16h
                        mov     ds,si
                        mov     si,ds:[0Ah]         ; Size of open table
                        sub     ah,ah               ; Set AH=0
                        cmp     ax,si               ; Check for valid channel #
                        jae     ISOPEN_ERR
                        mov     si,ds:[24h]         ; Open channel table
                        mov     ds,ds:[00h]         ; DS = operating system data segment
                        mov     ah,0Ch              ; 0Ch (12) bytes per open table entry
                        mul     ah                  ; Result to AX
                        add     si,ax               ; DS:SI points to channel entry
                        pop     ax                  ; Restore AH
                        mov     al,69h              ; Preload "Channel not open"
                        test    byte ptr ds:[si],OBIT; Is channel open?
                        jz      ISOPEN1             ; Channel not open.
                        xor     al,al               ; Channel is open.  Return 00h.
ISOPEN1:
                        pop     si
                        pop     ds
                        ret
;-
ISOPEN_ERR:
                        pop     ax                  ; Restore AH
                        mov     al,65h              ; Illegal parameter
                        jmp     ISOPEN1

ISOPEN                  endp
```

LLHLINKG.ASM

The handler linkage routines are in LLHLINKG.ASM. This set of utilities is a single include file that can be used as part of a high-level handler to call a low-level handler. Below is a list of all the linkage routines.

Table M-5. Handler Linkage Routine List

Name	Description
LLH_CLOSE	Call CLOSE routine of low-level handler
LLH_IOCTL	Call IOCTL routine of low-level handler
LLH_OPEN	Call OPEN routine of low-level handler
LLH_READ	Call READ routine of low-level handler
LLH_RSVD2	Call RSVD2 routine of low-level handler
LLH_RSVD3	Call RSVD3 routine of low-level handler
LLH_TERM	Call TERM routine of low-level handler
LLH_WARM	Call WARM routine of low-level handler
LLH_WRITE	Call WRITE routine of low-level handler

All the information about the linkage routines and how to use them is in the "User-Defined Handlers" chapter in part 1, "Operating System".

Program listing:

```
                        .sfcond
                        if1
;*************************************************************
;*     "(c) Copyright Hewlett-Packard Company, 1986.  All    *
;*     rights are reserved.  Copying or other reproduction   *
;*     of this program for inclusion in an application or    *
;*     for archival purposes is permitted without the prior  *
;*     written consent of Hewlett-Packard Company."          *
;*************************************************************
                        endif
                        .lfcond

; List of functions:
;
; LLH_OPEN  - calls OPEN routine of specified handler
;
; LLH_CLOSE  - calls CLOSE routine of specified handler
;
; LLH_READ  - calls READ routine of specified handler
;
; LLH_WRITE  - calls WRITE routine of specified handler
;
; LLH_WARM  - calls WARM routine of specified handler
;
; LLH_TERM  - calls TERM routine of specified handler
;
; LLH_IOCTL  - calls IOCTL routine of specified handler
;
; LLH_RSVD2  - calls RSVD2 routine of specified handler
;
; LLH_RSVD3  - calls RSVD3 routine of specified handler
```

```
;
                        page
AXREG                   equ     00h
ALREG                   equ     00h
AHREG                   equ     01h
;
BXREG                   equ     02h
BLREG                   equ     02h
BHREG                   equ     03h
;
CXREG                   equ     04h
CLREG                   equ     04h
CHREG                   equ     05h
;
DXREG                   equ     06h
DLREG                   equ     06h
DHREG                   equ     07h
;
SIREG                   equ     08h
DIREG                   equ     0Ah
DSREG                   equ     0Ch
ESREG                   equ     0Eh
BPREG                   equ     10h

OBIT                    equ     10h

save_regs               macro
;
; Save registers on stack in the order used by handler call
;
                        push    ds
                        push    bp
                        pushf
                        push    cs
                        mov     bp,offset RET_TO_HLH
                        push    bp

                        push    bp
                        push    es
                        push    ds
                        push    di
                        push    si
                        push    dx
                        push    cx
                        push    bx
                        push    ax
                        mov     bp,sp
                        mov     ss:BPREG[bp],bp

                        endm


restore_regs            macro
;
; Pop registers off of stack and return to high-level handler
;
                        pop     bx                      ; Get BH into AH
                        mov     ah,bh
                        pop     bx
                        pop     cx
```

```
                        pop     dx
                        pop     si
                        pop     di
                        pop     ds
                        pop     es
                        pop     bp
                        iret
                        endm
                        page
;*************************************************************
;*
;* Name: LLH_OPEN
;*
;* Version: 1.5
;*
;* Description:
;*   Call the OPEN routine of the low-level handler
;*
;* Call with:
;*   AL = Channel number to open
;*   ES:BX = Address of handler name string to open
;*
;* Returns:
;*   AL = Error code:
;*        00h       No errors
;*        65h (101) Illegal parameter
;*        66h (102) Directory does not exist
;*        67h (103) File not found
;*        6Ah (106) Channel already open
;*        6Eh (110) Access restricted
;*        (any others returned by handler)
;*
;* Registers altered:
;*   AL
;*
;* Notes:
;*   None.
;*
;*************************************************************

LLH_OPEN                proc    near

                        save_regs                       ; Save regs on stack
;
; Save DI and SI for FIND_HNDLR
;
                        mov     di,es
                        mov     si,bx
;
; Get segment addr of work area out of channel table
;
                        cmp     al,5                    ; Valid device channel
                        jl      LLH_OPEN_1              ; Yes, continue
                        mov     al,65h                  ; Invalid parameter
                        jmp     short LLH_OPEN_5        ; Exit with return code in AL

LLH_OPEN_1:
                        mov     bx,16h                  ; Table of addresses
                        mov     ds,bx
                        mov     bx,ds:24h               ; Get offset of open table
```

```
                        mov     ds,ds:00h
                        mov     ah,0Ch              ; Length of each entry in open table
                        mul     ah                  ; Multiply by channel number
                        add     bx,ax
                        test    byte ptr ds:[bx],08IT; Is channel open?
                        jnz     LLH_OPEN_2          ; Yes, continue
                        mov     al,69h              ; Channel not open
                        jmp     short LLH_OPEN_5    ; Exit with return code in AL
LLH_OPEN_2:
                        mov     cx,ds:5[bx]         ; Get stored value in open table
                        push    ds                  ; Address and value of
                        push    bx                  ; DS entry in open table
                        push    cx
;
; Set up my return address for open entry of handler
;
                        push    cs
                        mov     dx,offset LLH_OPEN_4 ; Return address this program
                        push    dx                  ; Return address in DX
;
; Set up address of handler entry on stack for IRET to branch to
;
                        call    FIND_HNDLR          ; Get segment address of handler
                        and     al,al               ; Any errors
                        je      LLH_OPEN_3          ; No, continue
                        add     sp,4                ; Yes, clean up stack
                        jmp     short LLH_OPEN_4    ; Return with error code in AL
;-
LLH_OPEN_3:
                        mov     ds:08h[bx],cx       ; Save handler CS
                        pushf                       ; Put flags for IRET
                        push    cx                  ; Segment address of handler
                        mov     bx,6                ; Offset of OPEN entry
                        push    bx
;
; Restore registers to the values they had when function was called
;
                        cli
                        add     sp,10h              ; Point to register values
                        pop     ax
                        pop     bx
                        pop     cx
                        pop     dx
                        pop     si
                        pop     di
                        pop     ds
                        pop     es
                        pop     bp
;
; Branch to handler entry
;
                        sub     sp,9*2+10h          ; Point to IRET addr
                        iret
;-
;
; Return to calling program with return code in AL
; NOTE: it is the responsibility of the handler entry that was
; just executed to set AL to appropriate return code
;
                                                    ; Address in DX
```

```
LLH_OPEN_4:
                        pop         cx                  ; Get the saved copy of segment address
                                                        ; of high-level handler's work area
                        pop         bx
                        pop         ds
                        xchg        ds:05h[bx],cx       ; Restore to open table and get
                                                        ; saved copy of segment address of
                                                        ; low-level handler's work area
                        mov         ds:0Ah[bx],cx       ; Save low-level handler's work area seg
LLH_OPEN_5:
                        restore_regs                    ; Return to high-level handler
;-
LLH_OPEN                endp
                        page
;*************************************************************
;*
;* Name: LLH_CLOSE
;*
;* Version: 1.5
;*
;* Description:
;*   Call the CLOSE routine of the low-level handler
;*
;* Call with:
;*   AL = Channel number to close
;*
;* Returns:
;*   AL = Error code
;*
;* Registers altered:
;*   AL
;*
;* Notes:
;*   None.
;*
;*************************************************************

LLH_CLOSE               proc        near

                        save_regs
                        mov         bx,09h              ; Offset of CLOSE entry
                        jmp         CALL_HNDLR          ; Go to handler
LLH_CLOSE               endp
                        page
;*************************************************************
;*
;* Name: LLH_READ
;*
;* Version: 1.5
;*
;* Description:
;*   Call the READ routine of the low-level handler
;*
;* Call with:
;*   AL = Channel number from which to read
;*   CX = Number of bytes to read
;*   ES:BX = Address of read buffer
;*
;* Returns:
```

```
;*   AL = Error code
;*   CX = Number of bytes actually read
;*
;* Registers altered:
;*   AL,CX
;*
;* Notes:
;*   None
;*
;*****************************************************************

LLH_READ                proc        near

                        save_regs
                        mov         bx,0Ch              ; Offset of READ entry
                        jmp         CALL_HNDLR          ; Go to handler
LLH_READ                endp
                        page
;*****************************************************************
;*
;* Name: LLH_WRITE
;*
;* Version: 1.5
;*
;* Description:
;*   Call the WRITE routine of the low-level handler
;*
;* Call with:
;*   AL = Channel number to write
;*   CX = Number of bytes to write
;*   ES:BX = Address of write buffer
;*
;* Returns:
;*   AL = Error code
;*   CX = Number of bytes actually written
;*
;* Registers altered:
;*   AL,CX
;*
;* Notes:
;*   None
;*
;*****************************************************************

LLH_WRITE               proc        near

                        save_regs
                        mov         bx,0Fh              ; Offset of WRITE entry
                        jmp         CALL_HNDLR          ; Go to handler
LLH_WRITE               endp
                        page

;*****************************************************************
;*
;* Name: LLH_WARM
;*
;* Version: 1.5
;*
;* Description:
;*   Call the WARM routine of the low-level handler
```

```
;*
;* Call with:
;*   AL = Channel number
;*
;* Returns:
;*   AL = Error code
;*
;* Registers altered:
;*   AL
;*
;* Notes:
;*   None
;*
;*************************************************************

LLH_WARM                proc        near

                        save_regs
                        mov         bx,12h              ; Offset of WARM entry
                        jmp         CALL_HNDLR          ; Go to handler
LLH_WARM                endp

                        page
;*************************************************************
;*
;* Name: LLH_TERM
;*
;* Version: 1.5
;*
;* Description:
;*   Call the TERM routine of the low-level handler
;*
;* Call with:
;*   AL = Channel number
;*   AH = Cause of termination
;*
;* Returns:
;*   AL = Error code
;*
;* Registers altered:
;*   AL
;*
;* Notes:
;*   Note that entry conditions are different for LLH_TERM
;*   than those which are seen by the low-level handler (AH, AL).
;*   LLH_TERM moves AH (cause of termination) into AL before
;*   calling the low-level handler.
;*
;*************************************************************

LLH_TERM                proc        near

                        save_regs
                        pop         bx                  ; Get original AX into BX
                        xchg        bh,bl               ; Exchange AH and AL
                        push        bx                  ; Put back on the stack
                        mov         bx,15h              ; Offset of TERM entry
                        jmp         CALL_HNDLR          ; Go to handler
LLH_TERM                endp
                        page
```

```
;*************************************************************
;*
;* Name: LLH_IOCTL
;*
;* Version: 1.5
;*
;* Description:
;*   Call the IOCTL routine of the low-level handler
;*
;* Call with:
;*   AL = Channel number
;*   AH = IOCTL function code
;*   (others as defined by handler)
;*
;* Returns:
;*   AL = Error code
;*   (others as defined by handler)
;*
;* Registers altered:
;*   AL
;*   (others as defined by handler)
;*
;* Notes:
;*   None
;*
;*************************************************************

LLH_IOCTL               proc        near

                        save_regs
                        mov         bx,1Bh              ; Offset of IOCTL entry
                        jmp         CALL_HNDLR          ; Go to handler
LLH_IOCTL               endp
                        page
;*************************************************************
;*
;* Name: LLH_RSVD2
;*
;* Version: 1.5
;*
;* Description:
;*   Call the RSVD2 routine of the low-level handler
;*
;* Call with:
;*   AL = Channel number
;*   (others as defined by handler)
;*
;* Returns:
;*   AL = Error code
;*   (others as defined by handler)
;*
;* Registers altered:
;*   AL
;*   (others as defined by handler)
;*
;* Notes:
;*   None
;*
;*************************************************************
```

```
LLH_RSVD2               proc        near

                        save_regs
                        mov         bx,1Eh              ; Offset of RSVD2 entry
                        jmp         CALL_HNDLR          ; Go to handler
LLH_RSVD2               endp
                        page
;*************************************************************
;*
;* Name: LLH_RSVD3
;*
;* Version: 1.5
;*
;* Description:
;*   Call the RSVD3 routine of the low-level handler
;*
;* Call with:
;*   AL = Channel number
;*   (others as defined by handler)
;*
;* Returns:
;*   AL = Error code
;*   (others as defined by handler)
;*
;* Registers altered:
;*   AL
;*   (others as defined by handler)
;*
;* Notes:
;*   None
;*
;*************************************************************

LLH_RSVD3               proc        near

                        save_regs
                        mov         bx,21h              ; Offset of RSVD3 entry
                        jmp         CALL_HNDLR          ; Go to handler
LLH_RSVD3               endp

                        page
;*************************************************************
;*
;* Name: CALL_HNDLR
;*
;* Version: 1.5
;*
;* Description:
;*   Call the selected routine of the low-level handler
;*
;* Call with:
;*   AL = Channel number
;*   BX = Offset of handler entry to call
;*
;* Returns:
;*   AL = Error code
;*
;* Registers altered:
;*   AL
;*
```

```
;* Notes:
;*   None
;*
;*************************************************************

CALL_HNDLR              proc        near

;
; Set up my return address for handler entry to return to
;
                        push        cs
                        mov         dx,offset CALL_HNDLR_EXIT; Return address this program
                        push        dx                  ; Return address in DX
;
; Set up address of handler entry on stack for IRET to branch to
;
                        pushf                           ; Put flags for IRET
                        mov         si,16h
                        mov         ds,si
                        mov         si,ds:24h           ; DS:SI point to open table
                        mov         ds,ds:00h
                        mov         ah,0Ch
                        mul         ah                  ; Multiply channel number by length of each entry
                        add         si,ax               ; DS:SI points to table entry for this channel
                        mov         cx,ds:0Ah[si]       ; Get low-level handler's work area segment
                        mov         ss:DSREG[bp],cx     ; Send to low-level handler
                        mov         cx,ds:08h[si]       ; Get low-level handler's CS
CALL_HNDLR_1:
                        push        cx                  ; Segment address of handler
                        push        bx                  ; Offset of handler entry
;
; Restore registers to the values they had when function was called
;
; Stack now:
; 0Ah                   (saved registers)
; 08h                   My CS
; 06h                   My IP (points to CALL_HNDLR_EXIT)
; 04h                   flags
; 02h                   low-level handler's segment address
; 00h                   low-level handler's offset address
;
                        cli
                        add         sp,0Ah              ; Point to register values
                        pop         ax
                        pop         bx
                        pop         cx
                        pop         dx
                        pop         si
                        pop         di
                        pop         ds
                        pop         es
                        pop         bp
;
; Branch to handler entry
;
                        sub         sp,9*2+0Ah          ; Point to IRET addr
                        iret
;-
;
; Return to calling program with return code in AL
```

```
; NOTE: it is the responsibility of the handler entry that was
; just executed to set AL to appropriate return code
;
CALL_HNDLR_EXIT:
                        restore_regs                    ; Return to high-level handler

CALL_HNDLR              endp
                        page
;*************************************************************
;*
;* Name: FIND_HNDLR
;*
;* Version: 1.5
;*
;* Description:
;*   Find a handler program whose name is at DI:SI
;*
;* Call with:
;*   DI:SI = Pointer to handler name
;*
;* Returns:
;*   AL = Error code
;*        00h       No errors
;*        65h (101) Illegal parameter
;*        66h (102) Directory does not exist
;*        67h (103) File not found
;*        6Eh (110) Access restricted
;*   CX = Segment address of handler
;*
;* Registers altered:
;*   AX,CX
;*
;* Notes:
;*   None
;*
;*************************************************************

FIND_HNDLR              proc        near

                        push        bx
                        push        dx
                        push        di
                        push        bp
                        push        ds
                        push        es
                        mov         bp,sp               ; Save SP
                        sub         sp,0Eh              ; Get room for file information buffer
                        mov         bx,si               ; Set ES:BX to file name address
                        mov         es,di
                        mov         di,sp               ; Get offset address of file information buffer
                        push        ss
                        pop         ds                  ; Get segment address of file information buffer
                        mov         ah,16h              ; FIND_FILE function
FIND_HNDLR1:
                        mov         dx,di               ; Offset address of file information buffer
                        int         1Ah
;
; If there are any errors, exit with appropriate error code
;
                        or          al,al               ; Any errors?
```

```
                        jnz         FIND_HNDLR_EXIT     ; Yes, exit with error code in AL

;
; If this is not a handler program (type "H"), generate "Access restricted"
; Get another file with same name and correct type
;
                        mov         al,6Eh              ; "Access restricted" error code
                        cmp         byte ptr ss:7[di],"H"; Is it a handler?
                        jne         FIND_HNDLR_EXIT     ; No, exit with error
                        xor         al,al               ; Set "No error" code
                        mov         cx,ss:08h[di]       ; Get segment address of handler

FIND_HNDLR_EXIT:
                        mov         sp,bp
                        pop         es
                        pop         ds
                        pop         bp
                        pop         di
                        pop         dx
                        pop         bx
                        ret
FIND_HNDLR              endp



RET_TO_HLH              proc        near
;
; Restore BP and DS, then return to calling program
;
                        pop         bp
                        pop         ds
                        ret
RET_TO_HLH              endp
```

NOIOWAIT.ASM

The NOIOWAIT utility in this include file disables the operating system I/O wait state. This is the state in which low battery, power switch, and timeout errors can be reported by the IOABORT utility while handler READ or WRITE routines are waiting for I/O.

The I/O wait state is enabled with IOWAIT. NOIOWAIT should be called when the handler READ or WRITE routines end, whether the routines are ending because I/O is being aborted, or because of a normal end. Refer to IOWAIT and IOABORT for further information.

Program listing:

```
                        .sfcond
                        if1
;*****************************************************************
;*     "(c) Copyright Hewlett-Packard Company, 1986.  All        *
;*     rights are reserved.  Copying or other reproduction       *
;*     of this program for inclusion in an application or        *
;*     for archival purposes is permitted without the prior      *
;*     written consent of Hewlett-Packard Company."              *
;*****************************************************************
                        endif
                        .lfcond

;*****************************************************************
;*
;* Name: NOIOWAIT
;*
;* Version: 1.3
;*
;* Description:
;*   Disable I/O wait state
;*
;* Call with:
;*   None
;*
;* Returns:
;*   None
;*
;* Registers altered:
;*   None
;*
;* Notes:
;*   None.
;*
;*****************************************************************


NOIOWAIT                proc    near
                        push    bx
                        push    si
                        push    ds
                        mov     bx,16h
                        mov     ds,bx
                        mov     bx,ds:[14h]
                        add     bx,4                ; BX points to current status
                        mov     si,ds:[1Ah]         ; System timer event control table
                        mov     ds,ds:[00h]         ; DS = operating system data segment
                        and     byte ptr ds:[bx],0DFh; I/O waiting flag clear
```

```
                        mov     byte ptr ds:13[si],-1; Timeout disable
                        pop     ds
                        pop     si
                        pop     bx
                        ret
NOIOWAIT                endp
```

READCTRL.ASM

The READCTRL utility in this include file reads the saved copy of the main control register (I/O address 0Bh). When hardware status is changed by an assembly language program, the program must use the following procedure to ensure that hardware devices unaffected by the change remain in their current state:

- Read the saved copy of the main control register using READCTRL.
- Change the bits needed to cause the hardware status to change.
- Write the updated value back to its saved location and output the updated value to the main control register using SETCTRL.

Refer to the "Hardware Control and Status Registers" chapter in part 1, "Operating System", for further information.

Program listing:

```
                        .sfcond
                        if1
;*************************************************************
;*     "(c) Copyright Hewlett-Packard Company, 1986.  All    *
;*     rights are reserved.  Copying or other reproduction   *
;*     of this program for inclusion in an application or    *
;*     for archival purposes is permitted without the prior  *
;*     written consent of Hewlett-Packard Company."          *
;*************************************************************
                        endif
                        .lfcond

;*************************************************************
;*
;* Name: READCTRL
;*
;* Version: 1.3
;*
;* Description:
;*   Read the saved copy of the main control register
;*   (I/O address 0Bh)
;*
;* Call with:
;*   None
;*
;* Returns:
;*   AH = Main control register value
;*
;* Registers altered:
;*   AH
;*
;* Notes:
;*   The control register bits have the following meanings:
;*
;*  ...............................................
;*  .   .   .   .   .   .   .   .   .   .   .   .
;*  . 7 . 6 . 5 . 4 . 3 . 2 . 1 . 0 .
;*  .   .   .   .   .   .   .   .   .   .   .   .
;*  ...............................................
;*                    .   .   .   .   .
;*                    .   .   .   .   .
```

```
;*                      .    .    .    .    .
;*                      .    .    .    .......
;*                      .    .    .        .
;*                      .    .    .        .
;*                      .    .    .        ... beeper control
;*                      .    .    .            11 = low tone
;*                      .    .    .            01 = high tone
;*                      .    .    .            00 = off
;*                      .    .    .
;*                      .    .    .... serial port power control
;*                      .    .         1 = enable
;*                      .    .         0 = disable
;*                      .    .
;*                      .    .....bar code port power control
;*                      .         1 = enable
;*                      .         0 = disable
;*                      .
;*                      .....bar code port transition control
;*                           1 = enable
;*                           0 = disable
;*
;************************************************************

READCTRL        proc    near

                push    bx
                push    ds
                mov     bx,16h              ; Get operating system pointer table
                mov     ds,bx
                mov     bx,ds:[14h]         ; Get offset of status area
                mov     ds,ds:[00h]
                inc     bx
                inc     bx                  ; Point to saved copy of main control register
                mov     ah,ds:[bx]          ; Read it
                pop     ds
                pop     bx
                ret

READCTRL        endp
```

READINTR.ASM

The READINTR utility in this include file reads the saved copy of the interrupt control register (I/O address 00h). When interrupt status is changed by an assembly language program, the program must use the following procedure to ensure that interrupts unaffected by the change remain in their current state:

- Read the saved copy of the interrupt control register using READINTR.
- Change the bits needed to cause the interrupt status to change.
- Write the updated value back to its saved location and output the updated value to the interrupt control register using SETINTR.

Refer to the "Hardware Control and Status Registers" and "Interrupt Controller" chapters in part 1, "Operating System", for further information.

Program listing:

```
                        .sfcond
                        if1
;*****************************************************************
;*     "(c) Copyright Hewlett-Packard Company, 1986.  All        *
;*     rights are reserved.  Copying or other reproduction       *
;*     of this program for inclusion in an application or        *
;*     for archival purposes is permitted without the prior      *
;*     written consent of Hewlett-Packard Company."              *
;*****************************************************************
                        endif
                        .lfcond

;*****************************************************************
;*
;* Name: READINTR
;*
;* Version: 1.3
;*
;* Description:
;*   Read the saved copy of the interrupt control register
;*   (I/O address 00h)
;*
;* Call with:
;*   None
;*
;* Returns:
;*   AH = interrupt control register value
;*
;* Registers altered:
;*   AH
;*
;* Notes:
;*   The interrupt control register bits have the following meanings:
;*
;*     (bit set to 1 -> corresponding interrupt enabled)
;*     (bit set to 0 -> corresponding interrupt disabled)
;*
;*   ...............................................
;*   .   .   .   .   .   .   .   .   .   .   .   .
;*   . 7 . 6 . 5 . 4 . 3 . 2 . 1 . 0 .
;*   .   .   .   .   .   .   .   .   .   .   .   .
```

```
;*   ..............................................
;*      .     .     .     .     .     .     .     .
;*      .     .     .     .     .     .     .     .
;*      .     .     .     .     .     .     .     .
;*      .     .     .     .     .     .     .     ... system timer interrupt
;*      .     .     .     .     .     .     .         (type 50h)
;*      .     .     .     .     .     .     .
;*      .     .     .     .     .     .     ... bar code timer interrupt
;*      .     .     .     .     .     .         (type 51h)
;*      .     .     .     .     .     .
;*      .     .     .     .     .     ... bar code port transition interrupt
;*      .     .     .     .     .         (type 52h)
;*      .     .     .     .     .
;*      .     .     .     .     ... serial port data received interrupt
;*      .     .     .     .         (type 53h)
;*      .     .     .     .
;*      .     .     .     ... low main battery voltage interrupt
;*      .     .     .         (type 54h)
;*      .     .     .
;*      .     .     ... power switch interrupt
;*      .     .         (type 55h)
;*      .     .
;*      .     ... reserved interrupt 1
;*      .         (type 56h)
;*      .
;*      ... reserved interrupt 2
;*          (type 57h)
;*
;*************************************************************

READINTR                proc        near

                        push        bx
                        push        ds
                        mov         bx,16h              ; Get operating system pointer table
                        mov         ds,bx
                        mov         bx,ds:[14h]         ; Get offset of status area
                        mov         ds,ds:[00h]
                        add         bx,3                ; Point to saved copy of interrupt control register
                        mov         ah,ds:[bx]          ; Read it
                        pop         ds
                        pop         bx
                        ret

READINTR                endp
```

SCANKYBD.ASM

The SCANKYBD utility in this include file scans the keyboard and returns the keycode of the first key found down. Normally, the system timer interrupt service routine causes the keyboard to be scanned every 5 ms. However, in time-critical handlers such as for the bar code port, the system timer may be disabled while waiting for data to be received at the port (to prevent bar code port transition interrupts from being missed). SCANKYBD performs the operating system keyboard scan operation when the system timer is disabled.

The SCANKYBD utility can be called while the handler waits for data, thereby allowing the handler to respond to keyboard input. This helps prevent users from gaining the perception of no machine activity that accompanies no keyboard response.

The keyboard columns are scanned from right to left, and the rows from top to bottom. The first key found down in that scanning sequence will be reported as a keycode. Other keys to the left or below the first key found will be ignored.

The keycodes corresponding to each key position and the corresponding ASCII characters are described in the "Keyboard" chapter in part 1, "Operating System".

SCANKYBD uses FINDOS to find the operating system file and the start of the operating system jump table.

Program listing:

```
                        .sfcond
                        if1
;*****************************************************************
;*     "(c) Copyright Hewlett-Packard Company, 1987.  All        *
;*     rights are reserved.  Copying or other reproduction       *
;*     of this program for inclusion in an application or        *
;*     for archival purposes is permitted without the prior      *
;*     written consent of Hewlett-Packard Company."              *
;*****************************************************************
                        endif
                        .lfcond

                        include findos.asm

;*****************************************************************
;*
;* Name: SCANKYBD
;*
;* Version: 1.3
;*
;* Description:
;*   Call the operating system routine to scan the keyboard
;*
;* Call with:
;*   None
;*
;* Returns:
;*   AL = 0 if no key down
;*   AL = HP-94 keycode (see chapter 8, "Keyboard", for keycode values)
;*
;* Registers altered:
```

```
;*   AX
;*
;* Notes:
;*   The key is NOT debounced by this routine - the routine
;*   simply reports what key is down at the present time.
;*
;*****************************************************************

SCANKYBD                proc        near
                        push        bx
                        push        si
                        push        ds
                        push        es
                        call        FINDOS
; DS is SYOS segment
; ES is operating system pointer table segment
                        push        ds
                        push        es:[36h]            ; Get offset of keyscan routine
                        mov         si,es:[00h]
                        mov         ds,si               ; DS = operating system data segment
                        mov         si,sp
                        call        dword ptr ss:[si]
                        add         sp,4
                        pop         es
                        pop         ds
                        pop         si
                        pop         bx
                        ret
SCANKYBD                endp
```

SETCTRL.ASM

The SETCTRL utility in this include file writes a value to the location of the saved copy of the main control register (I/O address 0Bh), then writes the value to the main control register as well. When hardware status is changed by an assembly language program, the program must use the following procedure to ensure that hardware devices unaffected by the change remain in their current state:

- Read the saved copy of the main control register using READCTRL.
- Change the bits needed to cause the hardware status to change.
- Write the updated value back to its saved location and output the updated value to the main control register using SETCTRL.

Refer to the "Hardware Control and Status Registers" chapter in part 1, "Operating System", for further information.

Program listing:

```
                        .sfcond
                        if1
;*************************************************************
;*     "(c) Copyright Hewlett-Packard Company, 1986.  All    *
;*     rights are reserved.  Copying or other reproduction   *
;*     of this program for inclusion in an application or    *
;*     for archival purposes is permitted without the prior  *
;*     written consent of Hewlett-Packard Company."          *
;*************************************************************
                        endif
                        .lfcond

;*************************************************************
;*
;* Name: SETCTRL
;*
;* Version: 1.3
;*
;* Description:
;*   Set the mail control register (I/O address 0Bh) and its
;*   saved copy to the value specified in AH
;*
;* Call with:
;*   AH = main control register value
;*
;* Returns:
;*   The main control register and its saved copy are set
;*   to the value in AH
;*
;* Registers altered:
;*   None
;*
;* Notes:
;*   The control register bits have the following meanings:
;*
;*  ...............................................
;*  .    .    .    .    .    .    .    .    .
;*  . 7  . 6  . 5  . 4  . 3  . 2  . 1  . 0  .
;*  .    .    .    .    .    .    .    .    .
;*  ...............................................
;*                      .    .    .    .    .
```

```
;*                      .    .    .    .    .
;*                      .    .    .    .    .
;*                      .    .    .    .......
;*                      .    .    .        .
;*                      .    .    .        .
;*                      .    .    .        ... beeper control
;*                      .    .    .            11 = low tone
;*                      .    .    .            01 = high tone
;*                      .    .    .            00 = off
;*                      .    .    .
;*                      .    .    .... serial port power control
;*                      .    .         1 = enable
;*                      .    .         0 = disable
;*                      .    .
;*                      .    .....bar code port power control
;*                      .         1 = enable
;*                      .         0 = disable
;*                      .
;*                      .....bar code port transition control
;*                           1 = enable
;*                           0 = disable
;*
;*************************************************************

SETCTRL                 proc    near

                        push    ax
                        push    bx
                        push    ds
                        pushf                       ; Save interrupt flag (CLI below)
                        mov     bx,16h              ; Get operating system pointer table
                        mov     ds,bx
                        mov     bx,ds:[14h]         ; Get offset of status area
                        mov     ds,ds:[00h]
                        inc     bx
                        inc     bx                  ; Point to saved copy of main control register
                        mov     al,ah               ; Get AH into AL
                        cli
                        mov     ds:[bx],al          ; Send value to saved copy location
                        out     08h,al              ; Send to main control register
                        popf                        ; Restore interrupt flag
                        pop     ds
                        pop     bx
                        pop     ax
                        ret

SETCTRL                 endp
```

SETINTR.ASM

The SETINTR utility in this include file writes a value to the location of the saved copy of the interrupt control register (I/O address 00h), then writes the value to the interrupt control register as well. When interrupt status is changed by an assembly language program, the program must use the following procedure to ensure that interrupts unaffected by the change remain in their current state:

- Read the saved copy of the interrupt control register using READINTR.
- Change the bits needed to cause the interrupt status to change.
- Write the updated value back to its saved location and output the updated value to the interrupt control register using SETINTR.

Refer to the "Hardware Control and Status Registers" and "Interrupt Controller" chapters in part 1, "Operating System", for further information.

Program listing:

```
                        .sfcond
                        if1
;*************************************************************
;*     "(c) Copyright Hewlett-Packard Company, 1986.  All    *
;*     rights are reserved.  Copying or other reproduction   *
;*     of this program for inclusion in an application or    *
;*     for archival purposes is permitted without the prior  *
;*     written consent of Hewlett-Packard Company."          *
;*************************************************************
                        endif
                        .lfcond

;*************************************************************
;*
;* Name: SETINTR
;*
;* Version: 1.3
;*
;* Description:
;*   Set the interrupt control register (I/O address 00h)
;*   and its saved copy to the value in AH
;*
;* Call with:
;*   AH = interrupt control register value
;*
;* Returns:
;*   The interrupt control register and its saved copy are
;*   set to the value in AH
;*
;* Registers altered:
;*   None
;*
;* Notes:
;*   The interrupt control register bits have the following meanings:
;*
;*     (bit set to 1 -> corresponding interrupt enabled)
;*     (bit set to 0 -> corresponding interrupt disabled)
;*
;*   ...........................................
;*   .   .   .   .   .   .   .   .   .   .   .
;*   . 7 . 6 . 5 . 4 . 3 . 2 . 1 . 0 .
```

```
;*   .    .    .    .    .    .    .    .    .    .
;*   ..........................................
;*        .    .    .    .    .    .    .    .
;*        .    .    .    .    .    .    .    .
;*        .    .    .    .    .    .    .    .
;*        .    .    .    .    .    .    .    ... system timer interrupt
;*        .    .    .    .    .    .    .        (type 50h)
;*        .    .    .    .    .    .    .
;*        .    .    .    .    .    .    ... bar code timer interrupt
;*        .    .    .    .    .    .        (type 51h)
;*        .    .    .    .    .    .
;*        .    .    .    .    .    ... bar code port transition interrupt
;*        .    .    .    .    .        (type 52h)
;*        .    .    .    .    .
;*        .    .    .    .    ... serial port data received interrupt
;*        .    .    .    .        (type 53h)
;*        .    .    .    .
;*        .    .    .    ... low main battery voltage interrupt
;*        .    .    .        (type 54h)
;*        .    .    .
;*        .    .    ... power switch interrupt
;*        .    .        (type 55h)
;*        .    .
;*        .    ... reserved interrupt 1
;*        .        (type 56h)
;*        .
;*        ... reserved interrupt 2
;*            (type 57h)
;*
;*****************************************************************

SETINTR         proc near

                push    ax
                push    bx
                push    ds
                pushf                           ; Save interrupt flag (CLI below)
                mov     bx,16h                  ; Get operating system pointer table
                mov     ds,bx
                mov     bx,ds:[14h]             ; Get offset of status area
                mov     ds,ds:[00h]
                add     bx,3                    ; Point to saved copy of interrupt control register
                mov     al,ah                   ; Get AH into AL
                cli
                mov     ds:[bx],al              ; Write value to saved copy location
                out     00h,al                  ; Write to interrupt control register
                popf                            ; Restore interrupt flag
                pop     ds
                pop     bx
                pop     ax
                ret

SETINTR         endp
```

VERSION.ASM

The VERSION utility in this include file returns the version number from the specified program file (type A, B, or H). The version number is the two-byte value at offset 04h in the file — the third pair of bytes in the program header. If the specified program is a handler (type H), VERSION also returns the handler identifier, which is the two-byte identifier at offset 02h in the file — the second pair of bytes in the handler header. VERSION can return the version of the system ROM instead of the version of a program. The system ROM version is part of the copyright message that appears when the machine enters command mode.

Program listing:

```
                        .sfcond
                        if1
;*****************************************************************
;*     "(c) Copyright Hewlett-Packard Company, 1987.  All        *
;*     rights are reserved.  Copying or other reproduction       *
;*     of this program for inclusion in an application or        *
;*     for archival purposes is permitted without the prior      *
;*     written consent of Hewlett-Packard Company."              *
;*****************************************************************
                        endif
                        .lfcond

AXREG                   equ         00h
ALREG                   equ         00h
AHREG                   equ         01h
;
BXREG                   equ         02h
BLREG                   equ         02h
BHREG                   equ         03h
;
CXREG                   equ         04h
CLREG                   equ         04h
CHREG                   equ         05h
;
DXREG                   equ         06h
DLREG                   equ         06h
DHREG                   equ         07h
;
SIREG                   equ         08h
DIREG                   equ         0Ah
DSREG                   equ         0Ch
ESREG                   equ         0Eh
BPREG                   equ         10h

FIND_FILE               equ         16h
BUFFER_SIZE             equ         0Eh

pushregs                macro
                        pushf
                        push        bp
                        push        es
                        push        ds
                        push        di
                        push        si
                        push        dx
                        push        cx
                        push        bx
```

```
                        push    ax
                        mov     bp,sp
                        endm

popregs                 macro
                        pop     ax
                        pop     bx
                        pop     cx
                        pop     dx
                        pop     si
                        pop     di
                        pop     ds
                        pop     es
                        pop     bp
                        popf
                        endm

                        include findos.asm

;*************************************************************
;*
;* Name: VERSION
;*
;* Version: 1.5
;*
;* Description:
;*   Return the version number of the specified file
;*
;* Call with:
;*   ES:BX points to a file name (see Notes, below)
;*
;* Returns:
;*   AL = Error code:
;*        00h       No error
;*        65h (101) Illegal parameter
;*        66h (102) Invalid directory number
;*        67h (103) File not found
;*
;*   If AL=00h:
;*     DX = version (DH = integer part, DL = fractional part)
;*     For type "H" files only:
;*       CX = handler identifier (bytes 2 and 3 of the program header)
;*
;* Registers altered:
;*   CX,DX
;*
;* Notes:
;*   If ES and BX are both zero, the version returned is that
;*   of the system ROM (the version shown in the copyright
;*   message which is displayed when the HP-94 enters command mode).
;*
;*************************************************************

VERSION                 proc    near
                        pushregs
                        mov     ax,es
                        add     ax,bx           ; Check for zero
                        ja      VERSION_FILE
; This is for the system ROM
                        call    FINDOS
```

```
                        mov     ax,ds
                        or      ax,ax
                        mov     al,67h          ; File not found
                        jz      VERSION_RET
; DS is start of SYOS file, ES is start of operating system pointer table
                        mov     si,es:[3Ch]     ; Pointer to version
                        sub     dx,dx           ; Clear out counter
                        mov     bx,010Ah        ; Decade value and "." flag
VERSION_OS1:
                        lodsb
                        sub     al,'9'+1
                        add     al,'9'+1-'0'
                        jnc     VERSION_OS2     ; Not in 0...9
                        xchg    al,dh
                        mul     bl
                        add     dh,al           ; New sum in DH
                        jmp     VERSION_OS1
;-
VERSION_OS2:
                        cmp     al,'.'-'0'
                        jne     VERSION_OS3     ; "." (continue with fraction)
                        sub     bh,01h          ; decrement "." flag
                        jc      VERSION_EXIT    ; already had "." (exit now)
                        xchg    dh,dl
                        jmp     VERSION_OS1
;-
VERSION_OS3:
                        or      bh,bh
                        jnz     VERSION_EXIT
                        xchg    dh,dl
                        jmp     short VERSION_EXIT  ; Done
;-
VERSION_FILE:
                        sub     sp,BUFFER_SIZE  ; allocate file information buffer
                        push    ss
                        pop     ds
                        mov     dx,sp
                        mov     ah,FIND_FILE
                        int     1Ah             ; O.S. function call
                        or      al,al
                        jnz     VERSION_RET
                        mov     si,sp
                        mov     ds,ss:[si+08h]  ; start segment address of the file
                        mov     al,ss:[si+07h]  ; get file type
                        mov     dx,ds:[04h]     ; Fetch version from file
                        cmp     al,'H'
                        jne     VERSION_EXIT
                        mov     cx,ds:[02h]     ; Fetch handler identifier
                        mov     CXREG[bp],cx
VERSION_EXIT:
                        mov     DXREG[bp],dx
                        sub     al,al           ; result code = 00h
VERSION_RET:
                        mov     ALREG[bp],al    ; return result code to user
                        mov     sp,bp           ; restore stack pointer
                        popregs
                        ret
VERSION                 endp
```

XIOCTL.ASM

The XIOCTL utility in this include file allows an assembly language program to call the IOCTL routine in an open low-level handler. IOCTL routines are usually called by a high-level handler to cause its low-level handler to take some action, such as change operating configuration or flush its receive buffer. The XIOCTL utility allows any application to direct the behavior of a low-level handler in the same manner. The behavior of the IOCTL routine is described in the "User-Defined Handlers" chapter in part 1, "Operating System", and the "Hewlett-Packard Bar Code Handlers" appendix.

Program listing:

```
                        .sfcond
                        if1
;*****************************************************************
;*     "(c) Copyright Hewlett-Packard Company, 1986.  All        *
;*     rights are reserved.  Copying or other reproduction       *
;*     of this program for inclusion in an application or        *
;*     for archival purposes is permitted without the prior      *
;*     written consent of Hewlett-Packard Company."              *
;*****************************************************************
                        endif
                        .lfcond

;*************************************************************
;*
;* Name: XIOCTL
;*
;* Version: 1.5
;*
;* Description:
;*   Call the IOCTL routine in a handler
;*
;* Call with:
;*   AL = Channel number
;*   AH = IOCTL function code
;*   Other registers as defined by the handler's IOCTL routine
;*
;* Returns:
;*   AL = Error code:
;*        00h       No error
;*        69h (105) Channel not open
;*   Other registers as defined by the handler's IOCTL routine
;*
;* Registers altered:
;*   AL (handler's AL if channel is open)
;*   Other registers as defined by the handler's IOCTL routine
;*
;* Notes:
;*   See the Technical Reference Manual, Part 1, chapter 3 "User-
;*   Defined Handlers" for more information about IOCTL.
;*
;*   When the IOCTL routine of the handler is called, XIOCTL also
;*   sets up these registers:
;*     DS = Segment address of handler scratch area
;*     BP = Stack offset address of register save area
;*
;*************************************************************
OBIT                    equ         10h
HBIT                    equ         20h
```

...XIOCTL.ASM

```
XIOCTL          proc    near
                pushf
                push    bp
                push    es
                push    ds
                push    di
                push    si
                push    dx
                push    cx
                push    bx
                push    ax
                mov     bp,sp
                mov     bx,16h
                mov     ds,bx
                mov     bx,ds:[0Ah]         ; Size of open table
                cmp     al,bl               ; Check for valid channel #
                jae     XIOCTL_ERROR
                mov     bx,ds:[24h]         ; Open channel table
                mov     ds,ds:[00h]         ; Operating system segment
                mov     ah,0Ch              ; 0Ch (12) bytes per open table entry
                mul     ah                  ; Result to AX
                add     bx,ax               ; DS:BX points to channel entry
                mov     ax,0[bp]            ; Restore AX
                mov     al,byte ptr ds:[bx] ; Read channel status
                and     al,OBIT+HBIT
                cmp     al,OBIT+HBIT        ; Is channel open for a device handler?
                mov     al,69h              ; Preload "Channel not open"
                jne     XIOCTL_END
                xor     al,al
                push    cs
                mov     cx,offset XIOCTL_END
                push    cx                  ; Stack has return address for XIOCTL_END
                push    ds:[bx+3]
                mov     cx,ds:[bx+1]
                add     cx,15h
                push    cx                  ; Stack has execute address for IOCTL
                mov     ds,ds:[bx+5]        ; Load handler data segment
                mov     cx,4[bp]
                mov     bx,2[bp]            ; Restore these registers
                db      0CBh                ; FAR RET (go to IOCTL handler)
;-
XIOCTL_ERROR:
                mov     al,65h              ; Illegal parameter
XIOCTL_END:
                pop     bx                  ; (really AX contents)
                mov     ah,bh               ; Leave AL unchanged from handler IOCTL
                pop     bx
                pop     cx
                pop     dx
                pop     si
                pop     di
                pop     ds
                pop     es
                pop     bp
                popf
                ret
XIOCTL          endp
```

XTIMEOUT.ASM

The XTIMEOUT utility in this include file executes a user-defined timeout interrupt routine if one was defined, or turns the machine off if none was defined. It is used by a handler READ or WRITE routine that is waiting for I/O when the IOABORT utility indicates that the timeout occurred. Refer to IOWAIT and IOABORT for further information.

XTIMEOUT uses FINDOS to find the operating system file and the start of the operating system jump table.

Program listing:

```
                        .sfcond
                        if1
;*****************************************************************
;*     "(c) Copyright Hewlett-Packard Company, 1987.  All        *
;*     rights are reserved.  Copying or other reproduction       *
;*     of this program for inclusion in an application or        *
;*     for archival purposes is permitted without the prior      *
;*     written consent of Hewlett-Packard Company."              *
;*****************************************************************
                        endif
                        .lfcond

                        include findos.asm

;***************************************************************
;*
;* Name: XTIMEOUT
;*
;* Version: 1.3
;*
;* Description:
;*   Execute a user-defined timeout routine, if any.  If no
;*   user-defined timeout routine is present, turn the HP-94
;*   off (will cold start when next turned on)
;*
;* Call with:
;*   None
;*
;* Returns:
;*   None
;*
;* Registers altered:
;*   None
;*
;* Notes:
;*   If there is no user-specified timeout routine, XTIMEOUT
;*   does not return to the caller.  The HP-94 is turned off,
;*   and will cold start when it is turned on again.
;*
;*   If there is a user-specified timeout routine, it will be
;*   executed before XTIMEOUT returns.
;*
;***************************************************************


XTIMEOUT                proc        near
                        push        ax
```

```
                        push    bx
                        push    si
                        push    ds
                        push    es
                        call    FINDOS
; DS is SYOS segment
; ES is operating system pointer table segment
                        push    ds
                        push    es:[3Ah]            ; Get offset of timeout routine
                        mov     si,es:[00h]
                        mov     ds,si               ; Set up operating system data segment
                        mov     si,sp
                        call    dword ptr ss:[si]
                        add     sp,4
                        pop     es
                        pop     ds
                        pop     si
                        pop     bx
                        pop     ax
                        ret
XTIMEOUT                endp
```