AT&T

999-802-200IS

System Programmer's Guide

AT&T Personal Computer 6300

NOTICE

The information in this document is subject to change without notice. AT&T assumes no responsibility for any errors that may appear in this document.

MS-DOS is a registered trademark of Microsoft Corp.

Contents

1

System Programming Concepts

Purpose of this Manual 1-2
Notation 1-3
Programming Steps 1-4

2

MS-LINK

Overview 2-2
MS-LINK File Usage 2-3
Segments, Groups, and Classes 2-7
Invoking MS-LINK 2-9
Sample MS-LINK Session 2-21
MS-LINK Error Messages 2-24

3

DEBUG

Overview 3-2
How to Invoke DEBUG 3-3
Debugging Commands 3-6
Command Parameters 3-7
DEBUG Error Messages 3-44

4

8086 Addressing Scheme

Overview 4-2
The 20-Bit Address 4-3
Aligned and Non-Aligned Words 4-5
Registers and Flags 4-6
Code, Data, and Stack Segments 4-11
Addressing Modes 4-12

5 Memory Maps Control Blocks Diskette Allocation

Overview 5-2
The Address Space 5-3
Low Memory Map 5-4
ROM BIOS Data Area 5-5
File Control Blocks 5-6
ASCIIZ Strings 5-11
Handles 5-12
Diskette Layout 5-13
Diskette Directory 5-14
File Allocation Table 5-18
Diskette Formats 5-22

6 Program File Structure and Loading

Overview 6-2
Pros and Cons for Selecting a Program Format 6-3
EXE2BIN 6-5
File Header Format 6-10
Relocation Process for .EXE Files 6-13
Program Segment Prefix 6-15
Program Loading Process 6-18

7 System Calls

Quick Reference: Functions and Interrupts 7-2
Overview 7-5
Programming Considerations 7-6
Interrupts 7-7
Functions 7-8
System Calls Description 7-10

8 ROM BIOS Service Routines

Overview 8-2
Conventions 8-3
Interrupt Vector List 8-4
Video Control 8-5
Diskette Services 8-17
Communications Services 8-19
Keyboard Handling 8-22
Printer Routines 8-29
Miscellaneous ROM-BIOS Services 8-30
Bypassing the BIOS 8-32
CONFIG.SYS 8-33

9 MS-DOS Device Drivers

Overview 9-2
MS-DOS Device Drivers 9-8
Asynchronous Communications Element 9-27
DMA Controller 9-36
Floppy Diskette Interface and Controller 9-45
Hard Disk Controller 9-66
Keyboard Interface 9-94
Parallel Printer Interface 9-100
Programmable Interrupt Controller 9-105
Programmable Interval Timer 9-116
Real Time Clock and Calendar 9-123
Serial Communications Controller 9-128
Speaker 9-147
Video Controller 9-150

Supplement: The Display Enhancement Board

1 System Programming Concepts

- Purpose of this Manual
- Notation
- Programming Steps

Purpose of this Manual

This guide provides you with in-depth information on the AT&T Personal Computer program development tools. The guide focuses on what you need to know to make use of the existing AT&T Personal Computer 6300 hardware and hardware interfaces.

The final chapter on programming devices assumes that you have a working knowledge of the principles of designing device drivers and need the technical details on how to program the AT&T Personal Computer.

Notation

The following syntax is used throughout this manual in descriptions of command and statement syntax:

[ ] Square brackets indicate that the enclosed entry is optional.

{ } Braces indicate a choice between two or more entries. At least one of the entries enclosed in braces must be chosen.

... Ellipses indicate that an entry may be repeated as many times as needed.

This guide contains examples of prompts and messages displayed on the screen. These system-displayed items are indented from the main body of the text so that you can easily distinguish them. For example, MS-LINK prompts:

OBJECT MODULES[.OBJ]

Descriptions or examples that show a required response are indented and presented in boldface type:

LINK OBJ1+OBJ2+OBJ3,MAP

Programming Steps

This section shows where to go for information on the task you are performing.

Running High-Level Language

If you are running a program via the BASIC interpreter, the section on "System Calls" is applicable, since you can call these functions via a BASIC program.

If you are running a compiled program, read the section on MS-LINK as well as the section on System Calls.

Writing Assembler Programs

The first eight chapters are aimed at programmers writing assembler programs. If you have not used the 8088 or 8086 assembly language, the section "8086 Addressing Scheme" gives you a good start. The sections on the linker and debugger are fundamental to writing and debugging assembler programs. Also read the sections on "System Calls" and "ROM BIOS Service Calls."

Writing Utilities

If you are writing a supplementary utility program, read the sections on assembly programs, the sections on "Memory Maps, Control Blocks, and Diskette Allocation," and "Program File Structure and Loading."

Programming Devices Directly

Every section applies to writing device drivers, especially the chapter on "MS-DOS Device Drivers."

2 MS-LINK

- Overview
- MS-LINK File Usage
- Segments, Groups, and Classes
- Invoking MS-LINK
- Sample MS-LINK Session
- MS-LINK Error Messages

Overview

MS-LINK is an executable program on your DOS Supplemental Programs diskette. MS-LINK combines object modules that are the output of the MACRO-86 assembler or a compatible compiler. It produces a relocatable run file (load module) and a list file of external references and error messages.

To run MS-LINK, you provide object, run, list, and library file parameters. You may optionally enter switches that modify the operation of MS-LINK.

"Invoking the Linker" describes the three ways to run MS-LINK: interactive entry, command line entry, and automatic response file entry. Interactive entry is used most frequently, so its section contains information common to all three methods.

If you are linking a high-level language program, the compiler determines the arrangement of your object modules in memory. If you are using assembler, however, you have more control over your program's organization. The section "Segments, Groups, and Classes" shows you how to specify the order of your object modules at run time.

MS-LINK File Usage

The link process involves the use of several files.

MS-LINK:

- Works with one or more input files
- Produces two output files
- Creates a temporary disk file if necessary
- Searches up to eight library files

The format for MS-LINK file specifications is the same as that of any disk file:

Syntax [d:][path]filename[.ext]

d: the drive designation. Permissible drive designations for MS-LINK are A: through O:.

path a path of directory names.

filename any legal filename of one to eight characters.

ext a one- to three-character extension to the filename.

If no filename extensions are given in the input (object) file specifications, MS-LINK recognizes the following extensions by default:

.OBJ Object
.LIB Library

MS-LINK appends the following default extensions to the output (Run and List) files:

.EXE Run (may not be overridden)
.MAP List (may be overridden)

VM.TMP File

MS-LINK uses available memory for the link session. If an output file exceeds available memory, MS-LINK creates a temporary file, names it VM.TMP, and puts it on the disk in the default drive. If MS-LINK creates VM.TMP, it will display the message:

```
VM.TMP has been created.
Do not change diskette in drive, <d:>
```

Once this message is displayed, do not remove the diskette from the default drive until the link session ends. If the diskette is removed, the operation of MS-LINK is unpredictable and MS-LINK usually displays the error message:

```
Unexpected end of file on VM.TMP
```

MS-LINK writes the contents of VM.TMP to the file named following the Run File: prompt. VM.TMP is a working file only and is deleted at the end of the linking session.

Do not use VM.TMP as a filename for any file. If MS-LINK requires the VM.TMP file, MS-LINK deletes the VM.TMP already on disk and creates a new VM.TMP. Thus, the contents of the previous VM.TMP file are lost.

Changing diskettes

You may want to change diskettes during the link operation. If MS-LINK cannot find an object file on the specified diskette, it prompts you to change diskettes rather than aborting the session. If you enter the /PAUSE switch, MS-LINK pauses and prompts you to change diskettes before it creates the run file. You may change diskettes when prompted except in the following cases:

- the diskette you want to change has a VM.TMP file on it.
- you have requested a list file on the diskette you want to change.

Segments, Groups, and Classes

Below terms are explained to help you understand how MS-LINK works. Generally, if you are linking object modules from a high-level language compiler, you do not need to know these terms. If you are linking assembly language modules, read this section carefully.

Segment

The segment is one of the most basic units of program memory organization. A segment is a contiguous area of memory up to 64K bytes long, and may be located anywhere in RAM. The contents of a segment are addressed by a segment:offset address pair, where "segment" is the segment's base or lowest address (see "The 20-Bit Address" in chapter 4).

Each segment has a class name in addition to its segment name. All segments with the same class name are loaded into memory contiguously by the linker from the first segment of that class to the last.

Class

A class is a collection of related segments. By naming the segments of your assembly language program to classes, you control the order in which they are loaded into memory (for high level languages, the compiler does this for you).

MS-LINK loads segments into memory on a class-by-class basis. Starting with the first class encountered in the first object file, all of the segments of each class are loaded. Within each class, the linker loads the segments in the order in which it finds them in the object files. Therefore, you can control the order in which classes are loaded by the order in which segments from different classes appear in the object files.

To ensure that classes are loaded in the order you desire, you can create a dummy module to feed to the linker as the first object file. This module declares empty-segment classes in the order you want the classes loaded. For example, one such file might look like this:

```
A  SEGMENT 'CODE'
A  ENDS
B  SEGMENT 'CONST'
B  ENDS
C  SEGMENT 'DATA'
C  ENDS
D  SEGMENT STACK 'STACK'
D  ENDS
```

If this method is used, be sure to declare all the classes used in your program in the dummy module; otherwise, you lose absolute control over the ordering of classes. Also, this method should only be used when linking assembly language programs. Do not create a dummy module if linking object files for a compiler, or unpredictable results may occur. Classes may be any length.

Group

Just as classes allow you to combine segments in a way that is logical, groups combine segments in 64K byte chunks to make them easily addressable. The segments in a group need not be contiguous, but when loaded they must fit within 64K bytes. This way each segment in the group can be fully addressed by an offset to one segment address, which is the start address of the lowest segment in the group. Segments are named to groups by the assembler or compiler or, as is possible in assembly language programs, by the programmer. Note that a segment can be large enough to be an entire group by itself.

Invoking MS-LINK

Ways to Invoke MS-LINK

MS-LINK is invoked in one of three ways. The first method, interactive entry, requires you to respond to individual prompts.

For the second method, command line entry, type all commands on the same line used to start MS-LINK.

To use the third method, automatic response file entry, create a response file that contains all the necessary commands and tell MS-LINK where that file is when you run MS-LINK.

Interactive Entry	LINK
Command Line Entry	LINK filenames[/switches]
Automatic Response File Entry	LINK @filespec

Interactive Entry

To invoke MS-LINK interactively, type:

LINK

MS-LINK loads into memory, then displays four prompts, one at a time. At the end of each line, after typing your response to the prompt, you may type one or more switches preceded by a forward slash.

The command prompts are summarized below. Defaults appear in square brackets ([]) after the prompt. **Object Modules** is the only prompt that requires a response.

MS-LINK Prompts

Prompt	Responses
Object Modules[.OBJ]:	[d:][path]filename[.ext] [+[[d:][path]filename[.ext]]...]
Run File[filename.EXT]:	[d:][path][filename[.ext]]
List File[NUL.MAP]:	[d:][path][filename[.ext]]
Libraries[.LIB]:	[d:][path][filename[.ext]] [+[d:][[path]filename[.ext]]...]

Notes:

- If you enter a filename without specifying the drive, the default drive is assumed. If you enter a filename without specifying the path, the default path is assumed.The libraries prompt is an exception — if the linker looks for the libraries on the default drive and doesn't find them, it looks on the drive specified by the compiler.

- To select default responses to all remaining prompts, use a single semicolon (;) followed immediately by <return> at any time after the second prompt (Run File:).

  Once you enter the semicolon, you can no longer respond to any of the prompts for that link session. Use the <RETURN> key to skip prompts.

- Use <CONTROL-C> to abort the link session at any time.

Object Modules to be Included

Object Modules [.OBJ]:

List .OBJ files to be linked. They must be separated by blank spaces or plus signs (+). If the plus sign is the last character typed, this prompt will reappear so that you can enter more object modules.

MS-LINK assumes that object modules have the extension .OBJ unless you explicitly specify some other extension. Object filenames may not begin with the @ symbol (@ is used for specifying an automatic response file).

The order in which you key in the the object files is significant. See section on segments, groups, and classes for more information.

Load Module

Run File [Obj-file.EXE]:

Give filename for executable object code. The default is: <first-object-filename>.EXE. (You cannot change the output extension.)You can specify just the drive designation or just a path for this prompt.

Listing

List File [NUL.MAP]:

Give filename for listing (also known as a linker map). The listing is not created if you select the default. You can request a listing by entering a drive designator, path, or filename[.ext]. If you do not specify an extension, the default .MAP is used.

You can have the listing printed by specifying a print device instead of a filename or have the listing displayed on the screen by specifying CON. If you display the linker map, you can also print it by pressing Ctrl-PrtSc.

Libraries to be Searched

```
Libraries [.LIB]:
```

List filenames to be searched separated by blank spaces or plus signs (+). If a plus sign is the last character typed, the prompt will reappear.

MS-LINK searches library files in the order listed to resolve external references. When it finds the module that defines the external symbol, MS-LINK processes that module as another object module.

There is no default library search for MACRO assembler object modules. For compiled modules, if you select the default for this prompt, MS-LINK looks for the compiler package's library on the default drive. If not found there, MS-LINK looks on the drive specified by the compiler.

If MS-LINK cannot find a library file, it displays:

```
Cannot find library <library-name>
Type new drive letter:
```

Press the letter for the drive designation (for example, B).

If two libraries have the same filename, only the first in the list is searched.

MS-LINK Switches

The seven MS-LINK switches control various MS-LINK functions. Type switches at the end of a prompt response regardless of which method you use to start MS-LINK. Switches may be grouped at the end of any response, or may be scattered at the end of several. Even if you type more than one switch at the end of one response, each switch must be preceded by a forward slash (/).

All switches may be abbreviated. The only restriction is that an abbreviation must be sequential from the first letter through the last typed; no gaps or transpositions are allowed. For example:

Legal	Illegal
/D	/DSL
/DS	/DAL
/DSA	/DLC
/DSALLOCA	/DSALLOCT

/DSALLOCATE

/DSALLOCATE tells MS-LINK to load all data at the high end of the Data Segment. Otherwise, MS-LINK loads all data at the low end of the Data Segment. At runtime, the DS pointer is set to the lowest possible address to allow the entire DS segment to be used. Use of /DSALLOCATE in combination with the default load low (that is, the /HIGH switch is not used) permits the user application to dynamically allocate any available memory below the area specifically allocated within DGroup yet to remain addressable by the same DS pointer. This dynamic allocation is needed for Pascal and FORTRAN programs.

Your application program may dynamically allocate up to 64K bytes (or the actual amount of memory available) less the amount allocated within DGroup.

/HIGH

/HIGH causes MS-LINK to place the Run file as high as possible in memory. Otherwise, MS-LINK places the Run file as low as possible.

Note:
Do not use /HIGH with Pascal or FORTRAN programs.

/LINENUMBERS

/LINENUMBERS tells MS-LINK to include in the List file the line numbers and addresses of the source statements in the input modules. Otherwise, line numbers are not included in the List file.

Not all compilers produce object modules that contain line number information. In these cases, of course, MS-LINK cannot include line numbers.

/MAP

/MAP directs MS-LINK to list all public (global) symbols defined in the input modules. If /MAP is not given, MS-LINK will list only errors (including undefined globals).

The symbols are listed alphabetically. For each symbol, MS-LINK lists its value and its segment:offset location in the Run file. The symbols are listed at the end of the List file.

/PAUSE

/PAUSE causes MS-LINK to pause in the link session when the switch is encountered. Normally, MS-LINK performs the linking session from beginning to end without stopping. This switch enables you to swap the diskettes before MS-LINK outputs the Run (.EXE) file.

When MS-LINK encounters /PAUSE, it displays the message:

```
About to generate .EXE file
Change disks <hit any key>
```

MS-LINK resumes processing when you press any key.

Note
Do not remove the disk which will receive the List file, or the disk used for the VM.TMP file, if one has been created.

/STACK: <number>

Stack number represents any positive numeric value (in hexadecimal radix) up to 65536 bytes. If a value from 1 to 511 is typed, MS-LINK will use 512. If /STACK is not used for a link session, MS-LINK calculates the necessary stack size automatically.

All compilers and assemblers should provide information in the object modules that allow the linker to compute the required stack size.

At least one object (input) module must contain a stack allocation statement. If not, MS-LINK will display the following error message:

```
WARNING: NO STACK STATEMENT
```

/NO

/NO is short for NODEFAULTLIBRARY-SEARCH. This switch applies only to higher level language modules. This switch tells MS-LINK not to search the default libraries in the object modules. For example, if you are linking object modules in Pascal, specifying /NO tells MS-LINK not to automatically search the library named PASCAL.LIB to resolve external references.

Command Line Entry

Purpose

You may invoke MS-LINK by typing all commands on one line. The entries following LINK are responses to the command prompts. The entry fields for the different prompts must be separated by commas. Use the following syntax:

Syntax

```
LINK <obj-list>,<runfile>,<listfile>,
<lib-list>[/switch...]
```

obj-list

a list of object modules, separated by plus signs or spaces.

runfile

name of the file to receive the executable output.

listfile

name of the file to receive the listing.

lib-list

list of library modules to be searched, separated by spaces or plus signs.

/switch

refers to optional switches which may be placed following any of the response entries (just before any of the commas or after the <lib-list>, as shown).

To select the default for a field, simply type a second comma with no spaces between the two commas.

Example:

**LINK FUN+TEXT+TABLE+CARE,
FUNLIST, COBLIB.LIB**

This command causes MS-LINK to load. Then the object modules FUN.OBJ, TEXT.OBJ, TABLE.OBJ and CARE.OBJ are loaded. MS-LINK links the object modules and writes the output to FUN.EXE (by default), creates a List file named FUNLIST.MAP, and searches the library file COBLIB.LIB.

Automatic Response File Entry

It is often convenient to save responses to the linker for re-use at a later time. This is especially useful when a long list of object modules needs to be specified. The use of an automatic response file allows you to do this.

Before using this option, you must create the response file. Each line of text corresponds to one MS-LINK prompt. The responses must be typed in the same order as they are when entered interactively. To continue a line, type a plus sign (+) at the end of the line.

You can enter the name of more than one automatic response file on the command line and combine response file names with additional parameters. The combined series of resulting parameters must be a valid sequence of MS-LINK prompts.

Use switches and special characters (+ and ;) in the response file the same way they are used when entered interactively.

To invoke the linker using a response file, type

```
LINK @ <filespec>
```

Filespec is the name of a response file.

When the session begins, MS-LINK displays each prompt with the corresponding response from the response file. If the response file does not contain answers for all the prompts, MS-LINK displays the prompt which does not have a response and waits for a response. When you type a legal response, MS-LINK continues the link session.

Example:

```
FUN TEXT TABLE CARE
/PAUSE /MAP
FUNLIST
COBLIB.LIB
```

This response file tells MS-LINK to load the four object modules named FUN.OBJ, TEXT.OBJ, TABLE.OBJ, and CARE.OBJ. MS-LINK pauses before producing a public symbol map to permit you to swap disks. When you press any key, the output files will be named FUN.EXE and FUNLIST.MAP. MS-LINK will search the library file COBLIB.LIB.

Sample MS-LINK Session

This sample shows you the type of information displayed during an MS-LINK session.

In response to the MS-DOS prompt, type:

LINK

The system displays the following messages and prompts:

```
    Microsoft Object Linker V2.01 (Large)
(C) Copyright 1982,1983 by Microsoft Inc.

Object Modules [.OBJ]: IO SYSINIT
Run File [IO.EXE]:
List File [NUL.MAP]: PRN /MAP /LINE
Libraries [.LIB]: ;
```

Notes:

- By specifying /MAP, you get both an alphabetic listing and a chronological listing of public symbols.
- By responding PRN to the List File: prompt, you can redirect your output to the printer.
- By specifying the /LINE switch, MS-LINK gives you a listing of all line numbers for all modules. (Note that /LINE can generate a large volume of output.)

Once MS-LINK locates all libraries, the linker map displays a list of segments in the order of their appearance within the load module. The list might look like this:

```
Start    Stop     Length  Name
00000H   009ECH   09EDH   CODE
009F0H   01166H   0777H   SYSINITSEG
```

The information in the Start and Stop columns shows the 20-bit hex address of each segment relative to location zero. Location zero is the beginning of the load module.

The addresses displayed are not the absolute addresses where these segments are loaded. See the following section on the MS-LINK DEBUG program for information on how to determine the absolute address of a segment.

Because the /MAP switch was used, MS-LINK displays the public symbols by name and value. For example:

```
ADDRESS      PUBLICS BY NAME
009F:0012    BUFFERS
009F:0005    CURRENT DOS LOCATION
009F:0011    DEFAULT DRIVE
009F:000B    DEVICE LIST
009F:0013    FILES
009F:0009    FINAL DOS LOCATION
009F:000F    MEMORY SIZE
009F:0000    SYSINIT

ADDRESS      PUBLICS BY VALUE
009F:0000    SYSINIT
009F:0005    CURRENT DOS LOCATION
009F:0009    FINAL DOS LOCATION
009F:000B    DEVICE LIST
009F:000F    MEMORY SIZE
009F:0011    DEFAULT DRIVE
009F:0012    BUFFERS
009F:0013    FILES
```

The final line in the listing file describes the program's entry point:

Program entry point at 0009F:0000

MS-LINK Error Messages

All errors, except for the two warning messages, cause the link session to abort. After the cause has been found and corrected, MS-LINK must be rerun. The following error messages are displayed by MS-LINK:

Attempt to access data outside of segment bounds, possibly bad object module

There is probably a bad object file.

Bad numeric parameter

Numeric value is not in digits.

Cannot open temporary file

MS-LINK is unable to create the file VM.TMP because the disk directory is full. Insert a new disk. Do not remove the disk that will receive the List. MAP file.

Error: dup record too complex

DUP record in assembly language module is too complex. Simplify DUP record in assembly language program.

Error: fixup offset exceeds field width

An assembly language instruction references an address with a short or near instruction instead of a long or far instruction. Edit assembly language source and reassemble.

Input file read error

There is probably a bad object file.

Invalid object module

An object module(s) is incorrectly formed or incomplete (as when assembly is stopped in the middle).

Symbol defined more than once

MS-LINK found two or more modules that define a single symbol name.

Program size or number of segments exceeds capacity of linker

The total size may not exceed 384K bytes and the number of segments may not exceed 255.

Requested stack size exceeds 64K

Specify a size less than or equal to 64K bytes with the /STACK switch.

Segment size exceeds 64K

64K bytes is the addressing system limit.

Symbol table capacity exceeded

Very many and/or very long names were typed exceeding the limit of approximately 50K bytes.

Too many external symbols in one module

The limit is 256 external symbols per module.

Too many groups

The limit is ten groups.

Too many libraries specified

The limit is 8 libraries.

Too many public symbols

The limit is 1024 public symbols.

Too many segments or classes

The limit is 256 (segments and classes together must total 256 or less).

Unresolved externals: <list>

The external symbols listed have no defining module among the modules or library files specified.

VM read error

This is a disk error; it is not caused by MS-LINK.

Warning: no stack segment

None of the object modules specified contains a statement allocating stack space.

Warning: segment of absolute or unknown type

There is a bad object module or an attempt has been made to link modules that MS-LINK cannot handle (e.g., an absolute object module).

Write error in TMP file

No more disk space remains to expand the VM.TMP file.

Write error on run file

Usually, this means there is not enough disk space for the Run file.

3 DEBUG

- **Overview**
- **How to Invoke DEBUG**
- **Debugging Commands**
- **Command Parameters**

  A Assemble
  C Compare
  D Display
  E Enter
  F Fill
  G Go
  H Hexarithmetic
  I Input
  L Load
  M Move
  N Name
  O Output
  Q Quit
  R Register
  S Search
  T Trace
  U Unassemble
  W Write

- **DEBUG Error Messages**

Overview

The DEBUG utility is an executable object program that resides on your MS-DOS diskette. DEBUG performs the following functions:

- Allows you to single step through a program, instruction by instruction, for testing purposes.
- Changes register and file contents during the DEBUG session so that you can test a code change without reassembling your program.
- Makes permanent changes to diskette files so you can use DEBUG to recover files that may otherwise be lost.
- Supports a disassemble command so you can translate machine code instructions into their assembly language equivalents for testing purposes.

How to Invoke DEBUG

The DEBUG program is invoked as follows:

DEBUG [filespec [,arglist]]

filespec the name of the program file to be debugged.

arglist An optional list of file name parameters and switches. These will be passed to the program specified by the filespec parameter. When the program is loaded into memory, it is loaded as if it had been invoked with the command

filespec arglist

That is, filespec indicates the file to be debugged, and arglist is the rest of the command line that is used when the file is invoked and loaded in memory via COMMAND.COM.

If you enter DEBUG without parameters, since no file name has been specified, current memory, disk blocks, or disk files can be manipulated.

Comments

On entering the DEBUG environment DEBUG responds with the hyphen (-) prompt and underline (_) cursor. You now may enter any DEBUG command.

If you include the filespec in the command line, the specified file is loaded into memory starting at location 100 (hexadecimal). However, if you specify a file with a .EXE extension, the program is relocated to the address specified in the header of the file. See the chapter on "Program Structure and Loading" for information on the format of the file header.

If the file has the HEX extension, the file is loaded beginning at the address specified in the HEX file. HEX files are in INTEL hex format and are converted to memory image format by DEBUG.

All DEBUG commands may be aborted at any time by pressing <CTRL-C>. Pressing <CTRL-S> suspends the display, so that you can read it before the output scrolls away. After suspending the display, press any key (except <CTRL-S> or <CTRL-C>) to continue scrolling.

Examples

DEBUG <**CR**>.

The DEBUG session begins, but without loading a file.

DEBUG b:myprog <**CR**>.

The DEBUG environment is entered and the file named "myprog" is loaded into memory from drive B.

When you invoke DEBUG, it sets up a program segment prefix at offset 0 in the program work area. You can overwrite this area if you enter DEBUG without parameters. Moreover, if you are debugging a file with a COM or EXE extension, do not tamper with the program header below location 5CH, or DEBUG will terminate.

Do not restart a program after a "Program terminated normally" message is displayed. You must reload the program with the N and L commands for it to run properly.

Debugging Commands

This section describes the DEBUG commands in alphabetical order for ease of reference.

- Commands can be entered in either upper or lower case.

- Command keywords and command parameters can be separated from each other by spaces or commas for readability but need not be, except where two hexadecimal numbers are entered as parameters, in which case they must be separated by a comma or space. For brevity, the syntax of this chapter will always indicate a comma where separation is obligatory, but note that a space can alternatively be used.

- Commands only become effective after entering <CR>.

- If you make a syntax error when entering a command, the message "Error" will be displayed. You must re-enter the command using the correct syntax.

Command Parameters

The following DEBUG command parameters require definition.

address a hex value in one of the following formats:

- a segment register designation and a hex offset separated from each other by a colon. For example:

  DS:0300

- a hexadecimal segment and offset separated from each other by a colon. For example:

  9D0:0100

- a hexadecimal offset value. The DEBUG command will use a default segment value from either the DS or CS registers, depending on the command. For example:

  200

byte a one or two character hexadecimal value.

drive 0, 1, or 2 depending on whether you wish to select drive A, drive B or drive C, respectively.

range a range of addresses. The range can be specified as

```
address L value
```

where address specifies the start of the range and value specifies the length of the range. For example:

```
DS:300L30
```

indicates a range of 48 locations starting at address 300 in the segment indicated by the DS register.

The specified range cannot be greater than 10000 (hexadecimal). To specify this value enter 0000 (or 0) as the value parameter.

A range can also be specified as:

```
address,address
```

where the two addresses indicate the limits of the range. A space may be used instead of a comma.

value a 1 to 4 character hexadecimal value.

A (ASSEMBLE)

Assembles 8086 mnemonics directly into memory.

Syntax

A [address]

Address is the start address into which the subsequently entered line of mnemonics is to be assembled. If this parameter is omitted, offset 100 from the segment in the CS register is assumed, if you did not enter an Assemble command previously. If you did enter Assemble previously, the code assembles into the address following the last instruction loaded by the previous Assemble command.

Comments

- After you enter the Assemble command, DEBUG displays the specified address followed by the cursor. You may then enter a line of 8086 assembler mnemonics. On terminating the line with <CR>, the line will be assembled into memory starting at the specified location. The address of the byte subsequent to the assembled code will be displayed on the next line along with the cursor to enable you to enter the next line of code. If, instead of a line of 8086 mnemonics, you simply enter <CR>, the Assemble command terminates and the DEBUG prompt reappears.
- All numeric values are hexadecimal and must be entered as 1 to 4 characters without a trailing H. Prefix mnemonics must be specified in front of the opcode to which they refer. You may also enter them on a separate line.
- The segment override mnemonics are CS:, DS:, ES: and SS:. The mnemonic for the far return is RETF. String manipulation mnemonics must explicitly state the string size. For example, use MOVSB to move byte strings.

- The Assemble command will automatically assemble short, near, or far jumps and calls, depending on byte displacement with respect to the destination address. These may be overridden with the NEAR or FAR prefix. For example:

```
0100:0500 JMP 502           ;a two-byte
                            ;short jump
0100:0502 JMP NEAR 505      ;a three-byte
                            ;near jump
0100:505 JMP FAR 50A        ;a five-byte far
                            ;jump
```

  The NEAR prefix may be abbreviated to NE, but the FAR prefix cannot be abbreviated.

- DEBUG cannot tell whether some operands refer to a word memory location or to a byte memory location. In this case the data type must be explicitly stated with the prefix "WORD PTR" or "BYTE PTR". Acceptable abbreviations are "WO" and "BY". For example:

```
NEG    BYTE PTR [128]
DEC    WO [SI]
```

- DEBUG cannot distinguish whether an operand refers to a memory location or to an immediate operand. Enclose operands that refer to memory locations in square brackets. For example:

```
MOV AX,21       ;Load AX with 21H
MOV AX,[21]     ;Load AX with the contents of
                ;location 21H
```

- Two pseudo-instructions are available with the Assemble command. The DB opcode will assemble byte values directly into memory. The DW opcode assembles word values into memory. For example:

```
DB 1,2,3,4,"THIS IS AN EXAMPLE"
DB 'THIS IS A QUOTE:"'
DB "THIS IS A QUOTE'"
DW 1000,2000,3000,"BACH"
```

- The Assemble command supports all forms of register indirect addressing. For example:

```
ADD BX,34[BP+2].[SI-1]
POP [BP+DI]
PUSH [SI]
```

All opcode synonyms are supported. For example:

```
LOOPZ  100
LOOPE  100
JA     200
JNBE   200
```

Example

1 Enter **A200** <**CR**>.

2 DEBUG displays **09AC:0200_**.

3 Enter **MOV AX,[21]** <**CR**>.

4 The 8086 mnemonics are assembled starting at location 200. The byte location subsequent to the assembled code is then displayed:

```
09AC:0203_
```

5 Enter <**CR**>.

6 The Assemble command terminates and the DEBUG prompt reappears.

C (COMPARE)

Compares the contents of two areas of memory.

Syntax

C range,address

range — the range of addresses defining the first area to be compared. If no segment is specified, then the segment specified in the DS register is assumed.

address — the start of the area to be compared with the area specified by the range parameter.

Comments

- The Compare command compares the area of memory specified by the range parameter with an area of the same size starting at the location specified by the address parameter.
- If the contents of the two areas are identical, nothing is displayed. If there are differences, then the differences are displayed in the form

  <address1><contents1><contents2><address2>

  <address1> indicates the address in the first area and <contents1> its contents. <address2> indicates the corresponding address in the second area and <contents2> its contents.

Example

1 Enter **C100,1FF,300** <CR> or

C100L100, 300 <CR>.

2 The area of memory from 100 to 1FF is compared with the area of memory from 300 to 3FF.

D (DISPLAY)

Displays an area of memory.

Syntax

D [range] or
D [address]

range

the range of addresses whose contents are to be displayed. If you enter only offsets, then the segment specified in the DS register is assumed.

address

the address from which the display is to start. The contents of this address and the subsequent 127 locations are displayed. If only an offset is entered, then the segment specified in the DS register is assumed.

Comments

- If D is specified without parameters, then the 128 bytes following the last address to be displayed are displayed. If no location has yet been accessed, the display will start from location DS:100.
- If D and the range parameter are specified, the contents of that range of addresses are displayed. If this takes more than 24 screen lines, the display is scrolled until the contents of the final address in the range are displayed on line 24.
- The display is displayed in two portions:

  A hexadecimal display, where each byte is represented by its hexadecimal value, and an ASCII display, where the equivalent ASCII character for the byte is displayed. If there is no corresponding printable ASCII character, a period (.) is displayed.

- Each line of the display begins with an address followed by the hexadecimal contents of the 16 bytes starting from the addressed location. The eighth and ninth bytes are separated by a hyphen (-). The right-hand columns display the equivalent ASCII values. Each line of the display, except possibly the first, begins on a 16 byte boundary.

Example

1 Enter D 100,110 <CR>.

2 Lines 100H to 110H (inclusive) are displayed.

3 Enter D <CR>.

4 The 128 bytes starting from location 111H are displayed.

5 Enter D200 <CR>.

6 The 128 bytes starting from location 200H are displayed.

E (ENTER)

Replaces the contents of memory locations at the byte address(es) specified.

Syntax **E address[,bytevalue[,bytevalue...]]**

address the address of the location whose value is to be replaced; or the address of the first of a succession of locations whose contents are to be replaced. If only an offset is specified, then the segment indicated by the DS register is assumed.

bytevalue the value that is to replace the contents of the specified address. The first bytevalue parameter will replace the contents of the location specified by the address parameter. A second bytevalue will replace the contents of the location following that specified by the address parameter, and so on.

Comments

- If the command is entered without the byte value list, then DEBUG displays the specified address and its contents. The Enter command then waits for you to perform one of the following:

1 Replace the displayed bytevalue by entering another value. Enter the new value after the current value. If you enter an illegal value, or if you type more than two digits, the illegal or extra character is not echoed.

2 Advance to the next byte by pressing <SPACE>. To change the value of this byte simply enter the value as described above. If you advance beyond an eight-byte boundary, DEBUG starts a new display line with the address displayed at the start of the line. To advance to the next byte without changing the current byte, press <SPACE> again.

3 To return to the previous byte enter hyphen (-). DEBUG then starts a new display line with the address of the byte you have returned to and its contents. You can then change the contents of this location as described above. To move back one byte further without changing this value, enter hyphen again, and another new display line will be generated.

4 Terminate the Enter command by pressing <CR>. This key may be pressed in any byte position.

- If you specify byte values in the command line, then the first of these byte values will replace the contents of the location specified by the address parameter. Subsequent entries in the list of byte values will replace subsequent bytes in memory.

Example

1 Enter **E100<CR>**.

2 DEBUG displays something like **058D:0100 CD._**

3 Enter **26**.

4 the value of location 100 is changed to 26 and DEBUG displays:

```
058D:0100 CD.26_
```

5 Enter <**SPACE**>.

6 The next byte (location 101) is displayed

```
058D:0100 CD.26 20._
```

7 Enter <**SPACE**>.

8 The next byte (location 102) is displayed

```
058D:100 CD.26 20. 00._
```

9 Enter <->.

10 The previous byte (location 101) is displayed on the next line

```
058D:0100 CD.26 20. 00.
058D:0101 20._
```

11 Enter **30** <**CR**>.

12 The contents of location 101 are changed to 30 and the Enter command is terminated.

```
058D:0100 CD.26. 20. 00.
058D:0101 20.30
-_
```

13 Enter **E 200,26,0A,19,23** <**CR**>.

14 The contents of byte locations 200, 201, 202 and 203 are changed to 26, 0A, 19 and 23, respectively.

F (FILL)

Fills an area of memory with specified byte values.

Syntax **F range,bytevalue[,bytevalue...]**

range the range of addresses whose contents are to be overwritten with the specified bytevalues. If only the offset is specified, then the segment indicated by the DS register is assumed.

bytevalue a two digit hexadecimal value that is to overwrite the contents of the specified address(es).

Comments

- If the specified range contains more bytes than the list of byte values, then the list of byte values is repeated until the specified range is filled.
- If the list of byte values is longer than the specified range, the extra byte values are ignored.

Example

1. Enter **F04BA:100L100,42,45,48,37,20** <**CR**>.
2. DEBUG fills memory locations **04BA:100** to **04BA:1FF** with the byte values specified. The five values are repeated until all 256 locations are filled.

G (GO)

Executes the program currently in memory, optionally halting at specified breakpoint(s) and displaying information about the system and program environment.

Syntax

G [=address][,address...]

=address

the address in memory at which program execution is to start. "=" must be entered to distinguish a start address from a breakpoint address.

address

the breakpoint address. You can specify up to ten breakpoints, in any order.

Comments

If you enter G without parameters, the program currently in memory is executed starting from the address specified by the CS and IP registers.

If you specify the =address parameter, the contents of the CS and IP registers are changed to those specified by the =address parameter and the program in memory is executed, starting from the address you specified.

If you specify one or more breakpoint addresses, program execution stops at the first such address encountered and displays the contents of the registers, the state of the flags and the next instruction to be executed (see the Register command for a description of the display).

- If only an offset is entered for an address, the GO command assumes the segment in the CS register.

- If you enter more than ten breakpoints, DEBUG will display

  BP Error

- Before executing the program, the GO command replaces the contents of the breakpoint locations with an interrupt instruction (hexadecimal CC). Therefore, each breakpoint address that you specify must point to the first byte of an 8086 instruction, or unpredictable results occur.

  When program execution halts at a breakpoint DEBUG restores the original values of all the specified breakpoint locations. However, if the program terminates normally (that is, not at a specified breakpoint), the original values are not restored.

  Note: Once a program has reached completion (DEBUG has displayed "Program terminated normally") you must reload the program before you can re-execute it.

  The stack segment must have six bytes available at the stack pointer for this command, otherwise unpredictable results occur. This is because the GO command jumps into the user program with the IRET instruction. The flag, CS, and IP registers have to be pushed onto the stack in preparation for the IRET, taking up six bytes.

Example

1. Enter **G=200,1AF,141** <CR>.

2. The program currently in memory is executed starting from location 200. Assuming location 141 is encountered before 1AF, then the program halts at location 141 and the register and flag values are displayed along with the next instruction to be executed. If neither breakpoint location is encountered, then the program terminates normally.

3. Enter **G** <CR>.

4. If, in step two, the program had halted at location 141, then program execution continues from that address.

H (HEXARITHMETIC)

Calculates and displays the sum and the difference of two hexadecimal values.

Syntax **H value_a,value_b**

value_a The first of two hexadecimal values.

value_b The hexadecimal value that is to be added to or subtracted from value_a.

Comments The hexadecimal values may be up to four digits long.

The Hex command displays two four-digit values:

— the first is the result of adding value_b to value_a

— the second is the result of subtracting value_b from value_a

Example

1 Enter **H19F,10A** <CR>.

2 DEBUG displays

```
02A9 0095
```

3 Enter **HFFFF,2** <CR>.

4 DEBUG displays

```
0001 FFFD
```

I (INPUT)

Inputs and displays (in hexadecimal) one byte from the specified port.

Syntax I value

value the address of the port that the byte is to be input from.

Comments The port address can be up to 16 bits.

Example

1 Enter **I2F8**.

2 the byte at the addressed port is input and displayed.

L (LOAD)

Loads a file or absolute disk blocks into memory.

Syntax **L [address[,drive,block,count]]**

address the address in memory at which the file or range of blocks is to be loaded. If only an offset is entered, then the segment indicated by the CS register is assumed.

drive the drive from which disk blocks are to be loaded. For drive A you must enter 0, for drive B you must enter 1, etc.

block the first of a range of blocks to be loaded from the disk specified by the drive parameter.

count the number of blocks to be loaded.

Comments

- If all parameters are specified, then DEBUG loads blocks of information from disk into memory.
- If you enter L without parameters, or with just the address parameter, the file whose file control block is correctly formatted at location CS:5C is loaded into memory. The file control block at CS:5C is set either to the filespec specified when the DEBUG command was invoked, or to the filespec specified by the most recent "Name" command.
- The default location for programs to load is at CS:100. If you specify L and the address parameter, the file is loaded at the specified

address unless it is a .EXE or .HEX file. In any case DEBUG sets the BX:CX registers to the number of bytes loaded.

- If the file has an EXE extension, then it is relocated to the load address specified in the header of the .EXE file. That is, the address parameter to the Load command is ignored. The header itself is stripped off the .EXE file before the file is loaded into memory. Thus the size of the .EXE file on disk will differ from its size in memory.
- If the file is a .HEX file, entering the Load command with no parameters causes the file to be loaded starting at the address specified within the .HEX file. If the address parameter, however, is specified, then loading starts at the address which is the sum of the address specified and the address in the .HEX file.

Examples

The following examples assume the system to be initially in MS-DOS.

1 Enter **debug <CR>**
Nb:file.com <CR>
L <CR>.

Debug is entered and the subsequent Name command sets the file control block at CS:5C to identify file "file.com" on the diskette inserted in drive B. The Load command then loads this file into memory starting at CS:100 (the default address).

2 Enter **debug b:file.com <CR>**
L300 <CR>.

file.com is loaded into memory at location CS:100 by the DEBUG command. It is then relocated to CS:300 by the Load command.

M (MOVE)

Moves the contents of a specified range of memory addresses to the locations starting at a specified address.

Syntax **M range,address**

range The area of memory whose contents are to be moved. If you only enter an offset, the segment indicated in the DS register is assumed.

address The start of the destination area. If you only enter an offset, then the segment indicated by the DS register is assumed.

Comments If the source and destination areas overlap, the move is performed without loss of data.

The contents of the source area are not changed by the move, unless the destination area overlaps it.

If you specify an address as the end of the range, you must only enter the offset. The segment specified, or defaulted to, in the start address of the range is assumed.

Example

1 Enter **MCS:100,110,CS:500 <CR>** or
MCS:100L11,CS:500 <CR>.

2 The 17 bytes starting at location CS:100 are copied to the 17 bytes starting at location CS:500.

N (NAME)

Provides file names for the Load and Write commands or file name parameters for the program to be debugged.

Syntax **N filespec[,filespec...]**

filespec the file specifier of a file to be loaded into memory, written to diskette, or used as a file name parameter to the file currently in memory.

Comments The Name command can be used to provide:

the name of the disk file to be loaded into memory by a subsequent Load command

the name to be assigned to the file currently in memory when the file is subsequently written to disk

file name parameters to the file in memory to be debugged.

The first case enables you to specify the file you wish to debug after entering the DEBUG environment. That is, you can enter DEBUG without specifying parameters, then use the Name command to name the disk file you wish to debug, then load the file into memory using the Load command. This has the same effect as entering the file name as the first parameter to the DEBUG command upon invocation. In either case the file control block for the file to be debugged is set up at location CS:5C and the file is loaded.

In the second case, the file is already in memory and the Name command sets up the file control block for the specified file name at location CS:5C. When a Write command is subsequently entered the file in memory is written to disk with the file name whose file control block is set up at location CS:5C.

In the third case, the Name command provides file name parameters for the program currently in memory. Whatever file control block was set at CS:5C is replaced by that of the first such parameter. If a second file parameter is specified, its file control block is set up at location CS:6C. Only two file control blocks are set up, although additional file name parameters may be included if required. All the specified — including any delimiters and switches that may have been typed — are placed in a save area at CS:81, with CS:80 containing a character count. Parameters specified in this way are analogous to file names specified in the argument list to the DEBUG command.

Examples 1 Enter **DEBUG <CR>**
Nb:file.com <CR>
L <CR>.

The system enters the DEBUG environment and FILE.COM resident on drive B has its file control block set up at location CS:5C. The Load command subsequently loads this file into memory.

This sequence has the same effect as entering

"DEBUG b:file.com"

2 Enter **Nb:newfile.com** <CR>
W <CR>

The file control block is set up at location CS:5C for the file specifier "b:newfile.com". The subsequent Write command writes the file currently in memory to drive B and names the file "newfile.com".

3 Enter **DEBUG b:file1.com** <CR>
Nfile2.dat,file3.dat <CR>
G <CR>

The DEBUG command loads the file named "file1. com" from drive B to be debugged. The Name command sets up two file control blocks at locations CS:5C and CS:6C for the file specifiers b:file2.dat and b:file3.dat, respectively. These files then become parameters to file1.COM when the subsequent GO command executes file1-.COM. Therefore, the file is executed as if the following command line had been typed:

b:file1 file2.dat file3.dat

O (OUTPUT)

Sends a specified byte to an output port.

Syntax **O value,byte**

value the address of the output port. It must be specified in hexadecimal and can be up to 16 bits.

byte a two-digit hexadecimal value to be sent to the specified port.

Example

1 O1E, 27 <CR>

2 the byte value 27H is output to the port 1EH.

Q (QUIT)

Terminates the DEBUG program.

Syntax Q

Comments The Quit command terminates the debugger without saving the file you are working on. Control is returned to MS-DOS command mode.

R (REGISTER)

Displays the contents of the registers and flag settings, or displays the contents of a specified register with the option to change that value, or displays the flag settings with the option of reversing any number of those settings.

Syntax

R [register-name pipe: **F**]

register-name

any valid register name whose contents are to be examined and optionally changed. This may be one of:

```
AX  DX  SI  ES  IP
BX  SP  DI  SS  PC
CX  BP  DS  CS
```

Note: IP and PC both refer to the Instruction Pointer.

F

the flag settings are to be displayed and optionally changed.

Comments

If you enter R without parameters, then the contents of all registers are displayed along with the flag settings and the next instruction to be executed. For Example:

```
AX=058D BX=0000 CS=0000 DX=0000
SP=FFF0 BP=0000 SI=0000 DI=0000 DS=058D
ES=058D CS=058D IP=013B
NV UP EI PL NZ NA PO NC
058D:013B 83D8      MOV DS,AX
```

If you enter R with a register name, then DEBUG displays the contents of that register. The command then waits for you to do one of the following:

- press <CR> to terminate the Register command without changing the value of the displayed register.

- change the value of the register by entering the four-digit hexadecimal value, then terminate the Register command by entering <CR>.

The valid flag values are shown in the following table:

Flag Name	Set	Clear
Overflow	OV (yes)	NV (no)
Direction	DN (decrement)	UP (increment)
Interrupt	EI (enabled)	DI (disabled)
Sign	NG (negative)	PL (plus)
Zero	ZR (yes)	NZ (no)
Auxiliary Carry	AC (yes)	NA (no)
Parity	PE (even)	PO (odd)
Carry	CY (yes)	NC (no)

If you enter RF, then the current flag settings are displayed. You can then either

- press <CR> to terminate the Register command without changing the flag values, or

- change the setting of one or more flags by entering the alternate value of the appropriate flags. The new values may be entered in any order, with or without delimiters.

Example

1 Enter **R** <CR>.

2 DEBUG displays the contents of all registers, flag settings and the next instruction to be executed.

3 Enter **RIP** <CR>.

4 DEBUG displays the contents of the Instruction Pointer. For example:

```
IP 0139

:—
```

5 Enter **0138** <CR>.

6 the contents of the Instruction Pointer are changed to 0138.

7 Enter **RF** <CR>.

8 DEBUG displays the flag settings. For example:

```
NV UP EI PL NZ NA PO NC -
```

9 Enter **PE ZR DI NG** <CR>.

10 The Parity flag is set to even (PE), the Zero flag is set (ZR), the Interrupt flag is cleared (DI), and the Sign flag is set (NG).

11 Enter **RF** <CR>.

12 DEBUG displays the new state of the flags

```
NV UP DI NG ZR NA PE NC-
```

S (SEARCH)

Searches a specified range for a list of bytes.

Syntax **S range,list**

range the range of addresses within which the search is to be made. If you only enter the offset, the segment indicated by the DS register is assumed.

list the list of one or more bytes to be searched for. Bytes in the list must be separated by a space or a comma.

Comments For each occurrence of the list of bytes within the specified range, DEBUG returns the address of the first byte. If no address is returned, no match was found.

Example

1 Enter **S100L100,20<CR>** or **S100,1FF,20<CR>**.

2 DEBUG displays the address of every occurrence of byte value 20 in the address range 100 to 1FF, inclusive, for example:

```
058D: 010C
058D: 0110
058D: 0115
058D: 0118
058D: 0120
058D: 0128
058D: 01CE
```

T (TRACE)

Executes one or more instructions and displays the register contents, flag settings and the next instruction to be executed.

Syntax **T [=address][,value]**

= address DEBUG is to commence execution at this address.

value the number of instructions to be executed.

Comments If the =address parameter is not specified, execution begins at CS:IP.

If the value parameter is not specified, only one instruction is executed.

The display generated is of the same format as that of the Register command (without parameters).

Example

1 Enter **T = 200,5** <CR>.

2 Five instructions, starting with the one at location CS:200, are executed, and the register and flag values following each instruction are displayed along with the next instruction to be executed.

3 Enter **T** <CR>.

4 The instruction pointed to by CS:IP is executed and the register and flag contents are displayed along with the next instruction to be executed.

U (UNASSEMBLE)

Disassembles strings of bytes in memory and displays them as assembler-like statements along with their corresponding addresses.

Syntax

```
U [range]
  or
U [address]
```

range the range of addresses whose byte values are to be disassembled. If you do not specify the segment, then the segment indicated by the CS register is assumed.

address the start of a 32 byte area of memory to be disassembled. If you only enter an offset, then the segment indicated by the CS register is assumed.

Comments

- If neither the range nor address parameter is specified, then 32 bytes are disassembled starting at location CS:IP. If the Unassemble command is given more than once, each subsequent invocation starts at the address following the last disassembled location.
- The number of bytes disassembled may be slightly more than the number you specified. This is because instructions are not always the same length and the final address in a range will not always contain the last byte of an instruction.
- The first address of a range, or the address parameter, must always refer to the first byte of an 8086 instruction, otherwise results are unpredictable.

Example

1 Enter **U058D:204L8** <CR>.

2 Eight bytes starting at location 058D:204 are disassembled and the result displayed:

```
058D:0204 8D16DFOD LEA  DX,[ODDF]
058D:0208 42       INC  DX
058D:0209 03DO     ADD  DX,AX
058D:020B 8916E50B MOV  [OBE5],DX
```

W (WRITE)

Writes the file being debugged to disk.

Syntax W [address[,drive,block,count]

address the start address of the code in memory that is to be written to disk. If you enter only an offset, then the segment indicated in the CS register is assumed.

drive the drive containing the specified blocks to which code in memory is to be written. For drive A you must enter 0, for drive B you must enter 1, etc.

block the block number on disk that is the first of a contiguous range of blocks to be overwritten with code from memory.

count the number of disk blocks to be overwritten with code from memory.

Comments

- If you enter the WRITE command without parameters, then the file is written to disk starting from memory address CS:100. If you specify the address parameter, then the file in memory, starting from the specified address, is written to disk.
- In either case, before executing the WRITE command, BX:CX must be set to the number of bytes to be written if the count parameter is not included. This value was set up correctly when the file was loaded (either by the Load command or the DEBUG command itself). However, if, since loading the file, you have executed a GO or

TRACE command, then the value of BX:CX will have been changed. Be sure this value is set up correctly.

- When the WRITE command writes a file to disk, it obtains the drive specifier and file name via the file control block set up at CS:5C. If no drive specifier is set up, then the default is assumed. This file control block is set up either by the DEBUG command (for the file you specify as a parameter to DEBUG) or by a subsequent NAME command. If it does not indicate the file specifier you require, you must set up this file control block using the NAME command. Refer to "Memory Maps, Control Blocks, and Diskette Allocation" for further details.

- When the file is written to disk it overwrites the version currently on disk unless the specified file name does not exist, in which case a new file is created.

- If all parameters are specified, then the code in memory is written to the drive specified by the parameter. The data to be written starts at the memory location specified by the address parameter, and is written to the blocks on the disk specified by the block and count parameters. Be extremely careful to correctly specify the blocks, since information stored there previously will be destroyed by this operation.

Examples **1** Enter **W <CR>**.

The file in memory, starting from location CS:100, is written to disk with the file specifier defined by the file control block set up at location CS:5C. The number of bytes written is given by BX:CX.

2 Enter **W200** <**CR**>.

The file in memory, starting from location CS:200, is written to disk with the file specifier defined by the file control block set up at location CS:5C. The number of bytes written is given by BX:CX.

3 Enter **W200,1,1F,20** <**CR**>.

Blocks 1F through 3F on drive B are overwritten with the data starting at memory location CS:200.

DEBUG ERROR MESSAGES

BF Bad Flag
You attempted to alter a flag, but entered some characters that are not acceptable pairs of flag values. See R (Register) command for the list of acceptable flag entries.

BP Too many Breakpoints
You specified more than ten breakpoints as parameters to the GO command. Reenter the command with ten or fewer breakpoints.

BR Bad Register
You entered the R command with an invalid register name.

DF Double Flag
You entered two values for one flag.

4 8086 Addressing Scheme

- Overview
- The 20-Bit Address
- Aligned and Non-Aligned Words
- Registers and Flags
- Code, Data, and Stack Segments
- Addressing Modes

Overview

The 8086 microprocessor has an extremely flexible addressing scheme. The 8086 uses a 16-bit word, but can address a megabyte of memory. The 8086 supports seven different addressing modes.

To take advantage of the flexibility of the 8086, so that you can write assembly language code and navigate through programs while debugging, study the addressing scheme by carefully reading this chapter.

The 20-Bit Address

The AT&T Personal Computer 6300 utilizes the full address space that is available due to the design of the 8086 microprocessor. The addresses are 20 bits long, so the address space is two to the twentieth power, 1024K, or one megabyte.

The 8086 has a 16-bit word. To convert 16-bit words to a 20-bit address, the 8086 uses "segmented addressing." A 20-bit address is created by using values from two separate registers. Two 16-bit numbers are used.

The binary representation of the first number is considered to have four binary zeroes tacked on to its end. This effectively multiplies the number by 16. This value is known as the segment portion of the address. The segment portion can point to any 16-byte segment of memory in the megabyte address space. However, with four zeroes as its least significant bits, it cannot "zero in" on individual bytes. The segment register's function is just to point to a 16-byte boundary (also known as a paragraph boundary).

Once a segment is located, the other register comes into play. This "offset register" points to the relative part of the address. The 16 bits that comprise the offset register point to an individual byte which is relative to the start of the segment.

The 8086 locates a particular address by:

1. Shifting the segment register to the left by four bits

2. Adding the contents of the offset register

The 20-bit address is conventionally expressed in special notation:

Syntax

<segment register>:<offset register>

Example

009F:0012

Segment address	009F0
+ Relative address	0012
= Actual Address	00A02

Aligned and Non-Aligned Words

The instructions for the 8086 are made up of from one to six bytes. Instructions can start at either an even or odd address. The 8086 is capable of accessing two bytes of data in memory in a single memory cycle. When the CPU accesses a word (16 bits) located at an even address, it is accessing an "aligned" word. The word is aligned because both bytes are located at the same word address and can be accessed in a single memory cycle.

When the CPU accesses a word starting at an odd address, it is accessing a "non-aligned" word. Since the two bytes comprising the word do not occupy the same word address, two memory cycles are required to read the entire word.

The importance of aligned or non-aligned words is determined by the importance of execution speed in your application. It is good programming practice to store data starting at an even address. If your program accesses or manipulates many word quantities, this will help speed program execution. If you are writing a device driver and instruction cycle times affect the execution of your program, the impact of aligned and non-aligned words should be taken into consideration.

Registers and Flags

General Registers

There are two main groups of general registers used by the 8086: the data group and the pointer and index group. Each register is 16 bits wide.

- The data registers are AX, BX, CX, and DX. Each can be used as a single 16-bit register or as two 8-bit registers. When they are used as two 8-bit registers, they are divided into an upper(H) and lower(L) half and called AH, AL, BH, BL, CH, CL, DH, and DL.

- The pointer and index registers are 16-bit registers. They are named according to their functions: SP (stack pointer), BP (base pointer), SI (source index), and DI (destination index).

Segment Registers

There are four segment registers in the 8086. Each register is 16 bits and their names reflect their use:

- CS — Code Segment
  Always defines the current code segment.
- DS — Data Segment
  Usually defines the current data segment.
- SS — Stack Segment
  Always defines the current stack segment.
- ES — Extra Segment
  Can define an auxiliary data segment.

These registers are used in combination with other registers to form the 20-bit address. Each segment begins on a paragraph (16 byte) boundary. There are four "current" segments at any one time. The contents of each segment register is called the "segment base value". The sections on "Code, Data, and Stack Segments" and "Addressing Modes" give details on how these registers are utilized.

Instruction Pointer

The instruction pointer (IP) is used in conjunction with the Code Segment register to point to the address of the next executable instruction. The IP is also a 16-bit register.

Flags

The 8086 has nine 1-bit status or condition flags that are used to indicate the condition of the result of an arithmetic or logical operation that has just occurred. Some of the assembly language instructions use these flags to conditionally change the execution path of a program.

Flag Definitions

AF Auxiliary Carry Flag
This flag is set (i.e., equal to 1) under two conditions:

During addition there is a carry of the low nybble to the high nybble. (nybble = 4 bits)

During subtraction there is a borrow from the low nybble to the high.

CF Carry Flag
This flag is set when there has been a carry or a borrow to the high-order bit of the (8- or 16-bit) result of an operation.

OF Overflow Flag
When this flag is set, an arithmetic overflow has occurred and a significant digit has been lost.

SF Sign Flag
This flag is set when the high-order bit of the result of an operation is a logical 1. Since negative binary numbers are represented using two's complement notation, SF reflects the sign of the result: 0 indicates a positive number and 1 indicates a negative number.

PF Parity Flag
If this flag is set, the result of the operation has an even number of ones in it. Use this flag to check for data transmission errors.

ZF Zero Flag
This flag is set when the result of an operation is zero.

Flag Definitions (Cont'd)

TF Trap Flag
When set, the trap flag puts the system into single-step mode for the purposes of debugging. An internal interrupt is generated after each instruction so that you can inspect your program one instruction at a time.

IF Interrupt-enable Flag
If this flag is set, external (maskable) interrupts are recognized by the 8086.

DF Direction Flag
This flag is set and cleared by the STD (Set Direction Flag) and CLD (Clear Direction Flag) instructions. If it has the value 1, SI and DI are decremented during string move operations. If it has the value of 0, SI and DI are incremented during string move operations. This flag is used for the following instructions: MOVS, MOVSB, MOVSW, CMPS, CMPSB, and CMPSW.

The flag register looks like this:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	—	—	—	—	OF	DF	IF	TF	SF	ZF	—	AF	—	PF	—	CF
	FLAGSh								FLAGS1							

CPU Registers

AX	AH	AL	accumulator
BX	BH	BL	base
CX	CH	CL	count
DX	DH	DL	data
	SP		stack pointer
	BP		base pointer
	SI		source index
	DI		destination index
	IP		instruction pointer
	FLGSh	FLGS1	flags
	CS		code segment
	DS		data seg
	ES		extra seg
	SS		stack seg

Code, Data, and Stack Segments

When you invoke a program, MS-DOS loads all of its segments into memory on paragraph boundaries. The segment registers are set to point to these locations. The data, code, and stack segments aren't necessarily far apart in memory; they may, in fact, overlap. Each segment may be up to 64 KB in length.

Code Segment

Programs are limited to 64K of code, unless they change the value in the CS register. If a program changes the CS register, it may address up to 1024K of code.

The CS register is modified by the FAR CALL and FAR RETurn instructions. Use these instructions to execute code that is located outside the bounds of the current segment.

Data Segment

Most programs use a maximum of 64 KB of memory for data. This includes Pascal and compiled BASIC. Assembly language programs, however, can use additional memory for data by employing the Extra Segment.

Extra Segment

The extra segment may be used in any manner you wish but is often used for transferring large blocks of data quickly in memory or as a storage area for a second stack.

Stack Segment

Stacks are used for temporarily storing register contents and other important values under these conditions:

- Interrupts
- Inter-segment calls
- One program calls another

Addressing Modes

General Comments

The flexible architecture of the 8086 supports many different memory-addressing modes. These can be broken down into six main types of addressing: immediate, register, direct, register indirect, and two kinds of calculated addressing. The following section discusses these modes and concerns the nature of the operand.

Immediate Addressing

In the immediate addressing mode, the operand appears in the instruction. For example,

```
MOV AX,333
```

moves the constant value 333 into the AX register.

Register Addressing

The register addressing mode uses the contents of one of the registers as the operand for the instruction. The instruction can specify that either 8 bits or 16 bits are to be manipulated. For example:

```
MOV AX,BX   ;moves 16 bits from BX to AX

MOV AL,BL   ;moves 8 bits from BL to AL
```

Direct Addressing

The direct addressing mode specifies a location in memory whose contents are used as the operand for the instruction. Example:

```
MOV CX,COUNT
```

This instruction uses the value found in the memory location designated by the symbol COUNT. Unless otherwise specified, COUNT is expected to be somewhere in the Data Segment. To specify that the operand is located in a segment other than the data segment, use the "segment override prefix:"

```
MOV CX,ES:COUNT
```

This syntax specifies that COUNT is located in the Extra Segment.

Register Indirect Addressing

With the register indirect addressing mode, the 16-bit offset address is contained in a base or index register. That is, the offset address resides in the BX, BP, SI, or DI register. Example:

```
MOV AX,[SI]
```

The 16-bit offset contained in the SI register is combined with the data segment register to compute the 20-bit address of the operand to move into register AX. Which segment register is used to compute the address depends on which instruction you are using (i.e., data segment or segment override for MOV, code segment for JMP or CALL, etc.).

Calculated Addressing Modes

The calculated addressing modes are like a combination of register indirect mode and direct addressing mode. There are two calculated addressing modes: single index and double index. In single index addressing, a 16-bit offset from the BX, BP, SI or DI register is added to an offset location in memory specified in the instruction. The combined value of these two items provides the offset into memory from which the operand is fetched. If the BP register is used the offset is from within the stack segment; otherwise the offset is from within the data segment. As always, use of a segment override prefix can change this. Examples:

```
MOV   AX, COUNT[DI]
MOV   AX, RECORD[BP]
```

In double index calculated addressing mode, values from two 16-bit registers are added to an optionally specified location in memory to produce the final offset. Either the BX or BP register is used for one of the register values, and the SI or DI register is used for the other one. If only two registers are given with no memory location, the memory location defaults to 0000 (start of segment). Once again, the default calculation is from the stack segment if the BP register is used; if BX is used the default is from the data segment. Examples:

```
MOV   AX, COUNT[BX+SI]
MOV   AX, RECORD[BP] [DI]
MOV   AX, [BX] [DI]
```

5 Memory Maps Control Blocks Diskette Allocation

- Overview
- The Address Space
- Low Memory Map
- ROM BIOS Data Area
- File Control Blocks
- ASCIIZ Strings
- Handles
- Diskette Layout
- Diskette Directory
- File Allocation Table
- Diskette Formats

Overview

The purpose of this chapter is to enable you to locate items in memory or on diskette for the purposes of programming and debugging.

The first portion of the chapter contains detailed memory maps of the RAM and ROM memory areas. The sections on control blocks deal with program file formats and I/O data structures. The last part of this chapter describes how data is organized on the diskette.

The Address Space

Hex	Decimal	Contents
00000	0K	Interrupt vectors (see detail in low memory map)
04000	16K	DOS software
08000	32K	Language, applications programs and data Note: There is at least 98,000 hex or 608K of address space reserved for user programs and data. To take advantage of the full amount, you must have purchased and installed the physical memory.
		• • •
A0000	640K	Reserved for extended graphics
B0000	704K	Monochrome display buffer — Not used
B8000	736K	Color/graphics display buffer(s)
C8000	800K	Fixed disk adapter's ROM (Optional)
F0000	960K	Reserved for ROM expansion
FC000	1008K	ROM BIOS

Low Memory Map

Hexadecimal addresses are in segment:offset format.

Hex	Decimal	Contents
0:0000	0	Interrupt vectors 0 - 7 8259 interrupt controller vectors (8-F)
0:0040	64	BIOS interrupt vectors 10-1F
0:0080	128	MS-DOS interrupt vectors 20-3F
0:0100	256	Assignable interrupt vectors (40-FF) Note: These vectors may be assigned to non-Intel hardware and software products.
0:0400	1024	ROM BIOS data area (also called BIOS communications area) See map on next page.
0:0500	1280	DOS data area (also called DOS communications area)
0:0600	1536	

ROM BIOS Data Area

Hex addresses are in segment:offset format.

Hex	Decimal	Contents
0:0400	1024	Hardware environment parameters (printer and RS232C device addresses, memory size, etc.)
0:0417	1047	Keyboard buffer and status bytes
0:043E	1086	Floppy and hard disk status bytes
0:0449	1097	Video display area (current mode, color pallette, cursor position, active page numbers, etc.)
0:0467	1127	Data area for option ROM and 8253 timer chip
0:0471	1137	Fixed disk, I/O timeouts, and more keyboard status information
0:0488	1160	RESERVED
0:0500	1280	Inter-applications communications area

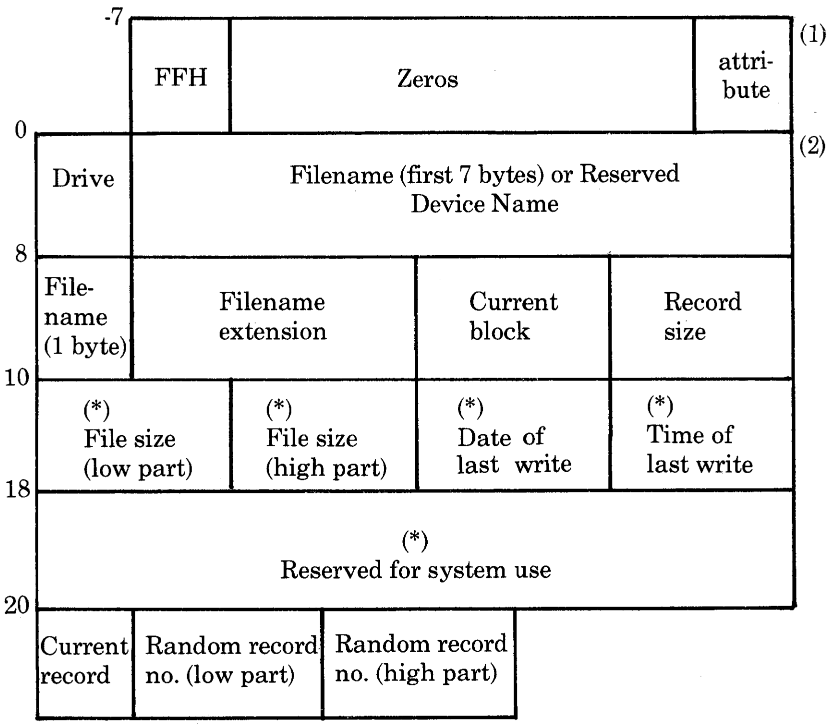
File Control Blocks

FCB Format

The standard File Control Block (FCB) contains 37 bytes of file control information. The extended File Control Block is used to create or search for files in the disk directory that have special attributes. If the extended FCB is used, it adds a 7-byte prefix to the standard FCB.

Any of the DOS functions which employ FCBs may use either an FCB or an extended FCB. (See chapter 7 for a description of each DOS function call.)

If you are using an extended FCB, set the appropriate register to the first byte of the prefix, not to the first byte of the standard FCB, before executing the function call.



(1) FCB extension
(2) Standard FCB

(*) Areas with an asterisk are filled by DOS and must not be modified. Other areas must be filled by the using program.

Offsets are in decimal.

Byte	Function
0	Drive number Before open: 0 — default drive 1 — drive A 2 — drive B etc. After open: 1 — drive A 2 — drive B etc. A 0 is replaced by the actual drive number when the file is opened.
1-8	Filename, left-justified, padded with trailing blanks. If a reserved device name is placed here (e.g., LPT1,) do not include the optional colon.
9-OB	Filename extension, left-justified, with trailing blanks (may be all blanks).
OC-OD	Current block number relative to the beginning of the file (starting at zero). A block is defined as a group of 128 records. This field is set to 0 when the file is opened. This field and the Current Record field (offset 20H) make up the record pointer that is used for sequential reads and writes.
OE-OF	Logical record size in bytes. Automatically set to 80H during open. If this is not correct, set it to the correct value.
10-13	File size in bytes. The first word of the field is the low-order part of the size.

Byte	Function
14-15	Date the file was created or last updated. The bits correspond to the date as follows: < 15 >< 14 > 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 y y y y y y y m m m m d d d d d yy = 0 - 119 (1980 to 2099) mm = 1 - 12 dd = 1 - 31
16-17	Time the file was created or last updated. The bits correspond to the time as follows: < 17 >< 16 > 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 h h h h h m m m m m m s s s s s hh = 0 - 23 mm = 0 - 59 ss = 0 - 30 (number of 2-second increments)
18-1F	Reserved for use by DOS.
20	Current relative record number (0-127) within the current block. Set this field before doing sequential read/write operations; it is not initialized by the Open File function call.
21-24	Relative record number; relative to the beginning of the file, starting with zero.

Byte	Function
	This field is not initialized by the Open File function call and must be set before doing a random read or write. If the record size is less than 64 bytes, both words of the field are used; otherwise, only the first three bytes are used. Note: If you use the FCB at offset 5CH of the Program Segment Prefix, the last byte of the Relative Record field is the first byte of the unformatted parameter area that starts at offset 80H. This is the default Disk Transfer Address.

Extended Control Block

Byte	Function
FCB-7	Contains hex FF to indicate this is extended FCB.
FCB-6 to FCB-2	Reserved
FCB-1	Attribute byte. 02 = Hidden file 04 = System file

ASCIIZ Strings

MS-DOS version 2.11 provides a set of new function calls for file I/O that are easier to use than the "traditional" calls that were used in past versions. These new calls do not utilize file control blocks. To open a diskette file, you simply provide information to identify the file in the form of an ASCIIZ string. DOS returns a numeric value, a "handle", that you use to refer to the file once you have opened it.

The older function calls that require the use of file control blocks and do not utilize ASCIIZ strings and handles are supported in MS-DOS 2.11 to provide upward compatibility. Use the newer function calls whenever possible. See Chapter 7 for details of function calls.

ASCIIZ String Format

An ASCIIZ string, also known as a pathname string, has the following format: an optional drive specifier, followed by a directory path, and where applicable, a filename. The last byte must be binary zeroes. For example:

A:\LEVELA\LEVELB\FILEA

(followed by a byte of zeroes)

Either back slash (\) or forward slash (/) are valid path-separator characters.

Handles

Several of the new function calls that support files or devices use an identifier known as a "handle" (also known as a "token"). When you create or open a file or device with these function calls, a 16-bit value is returned in register AX. Use this handle to refer to the file after it has been opened.

The following handles are pre-defined by DOS for your use. You need not open them before using them:

0000	Standard input device
0001	Standard output device
0002	Standard error output device
0003	Standard auxiliary device
0004	Standard printer device

Note:
See your MS-DOS User's Guide for information on redirecting I/O for the first two handles.

Diskette Layout

The DOS area of the diskette is formatted as follows:

Reserved Area — variable size
First copy of file allocation table — variable size
Second copy of file allocation table — variable size
Root directory — variable size
File data area

Clusters

Space for a file in the data area is not preallocated. The space is allocated one "cluster" at a time. A cluster consists of one or more consecutive sectors; all of the clusters for a file are "chained" together in the File Allocation Table (FAT). On diskettes formatted by MS-DOS 2.11, there are two copies of the FAT kept, for consistency. Should the disk develop a bad sector in the middle of the first FAT, the second is used as a backup.

Diskette Directory

The FORMAT command builds the root directory for all disks. Its location on disk and the maximum number of entries are dependent on the media.

Since directories other than the root directory are regarded as files by MS-DOS, there is no limit to the number of files they may contain.

All directory entries are 32 bytes in length, and are in the following format (byte offsets are in hexadecimal):

Directory Format

0-7 Filename. Eight characters, left aligned and padded, if necessary, with blanks. If this is not currently a file directory entry, the first byte of this field indicates the status as follows:

00H The directory entry has never been used. This is used to mark the end of the allocated directory and limit the length of directory searches, for performance reasons.

2EH The entry is for a directory (2EH is the ASCII code for the dot '.' character used to represent a directory). If the second byte is also 2EH (i.e., the entry is '..'), then the cluster field contains the cluster number of this directory's parent directory (0000H if the parent directory is the root directory). Otherwise, bytes 01H through 0AH are all spaces (i.e., the entry is '.') and the cluster field contains the cluster number of this directory.

E5H The file was used, but it has been erased.

Any other character is the first character of a filename.

Directory Format (cont'd)

8-0A	Filename extension.
0B	File attribute. The attribute byte is mapped as follows (values are in hex):

01	File is marked read-only. An attempt to open the file for writing using the Open a File system call (Function Request 3DH) results in an error code being returned. This value can be used along with other values below. Attempts to delete the file with the Delete File system call (13H) or Delete a Directory Entry (41H) will also fail.
02	Hidden file. The file is excluded from normal directory searches.
04	System file. The file is excluded from normal directory searches.
08	The entry contains the volume label in the first 11 bytes. The entry contains no other usable information (except date and time of creation), and may exist only in the root directory.
10	The entry defines a sub-directory, and is excluded from normal directory searches.
20	Archive bit. The bit is set to "on" whenever the file has been written to and closed. It is used by BACKUP and RESTORE commands for determining whether or not a file has changed since its last backup. The BACKUP command clears this attribute on all files backed up.

Directory Format (cont'd)

The system files (IO.SYS and MSDOS.SYS) are marked as read-only, hidden, and system files. Files can be marked hidden when they are created. Also, the read-only, hidden, system, and archive attributes may be changed through the Change Attributes system call (Function Request 43H).

0C-15 Reserved.

16-17 Time the file was created or last updated. The hour, minutes, and seconds are mapped into two bytes as follows:

Offset 17H

H	H	H	H	H	M	M	M
7	6	5	4	3	2	1	0

Offset 16H

M	M	M	S	S	S	S	S
7	6	5	4	3	2	1	0

H is the number of hours (0-23)
M is the number of minutes (0-59)
S is the number of two-second increments

The time is stored with the least significant bit first.

Directory Format (cont'd)

18-19 Date the file was created or last updated. The year, month, and day are mapped into two bytes as follows:

Offset 19H

Y	Y	Y	Y	Y	Y	Y	M
7	6	5	4	3	2	1	0

Offset 18H

M	M	M	D	D	D	D	D
7	6	5	4	3	2	1	0

Y is the number of years since 1980, 0-119 (1980-2099)
M is the month number 1-12
D is the day of the month 1-31

The date is stored with its least significant byte first.

1A-1B The cluster number of the first cluster in the file. The first cluster for data space on all disks is cluster 002.

The cluster number is stored with the least significant byte first.

Note:

Refer to "How to Use the File Allocation Table" for details on converting cluster numbers to logical sector numbers.

1C-1F File size in bytes. The first word of this four-byte field is the low-order part of the size.

File Allocation Table (FAT)

The following information is included primarily for system programmers who are writing installable device drivers. This section explains how MS-DOS uses the File Allocation Table to convert the clusters of a file to logical sector numbers. The driver program is then responsible for locating the logical sector on disk. If you are writing a system utility, use the MS-DOS file management function calls for accessing files; programs that access the FAT directly are not guaranteed to be upwardly-compatible with future releases of MS-DOS.

FAT Entries

The File Allocation Table is an array of 12-bit entries (1.5 bytes) for each cluster on the disk.

- The first two FAT entries map a portion of the directory; these FAT entries indicate the size and format of the disk.
- The second and third bytes currently always contain FFH.
- The third FAT entry, which starts at byte offset 4, begins the mapping of the data area (cluster 002). Files in the data area are not always written sequentially on the disk. The data area is allocated one cluster at a time, skipping over clusters already allocated.The first free cluster found will be the next cluster allocated, regardless of its physical location on the disk. This permits the most efficient utilization of disk space because clusters made available by erasing files can be allocated for new files.

Each FAT entry contains three hexadecimal characters:

- 000 — If the cluster is unused and available.
- FF7 — The cluster has a bad sector in it. MS-DOS will not allocate such a cluster. CHKDSK counts the number of bad clusters for its report. These bad clusters are not part of any allocation chain.
- FF8-FFF — Indicates the last cluster of a file.
- XXX — The cluster number of the next cluster in the file. The cluster number of the first cluster in the file is kept in the file's directory entry.

The File Allocation Table always begins on the first sector after the reserved sectors. If the FAT is larger than one sector, the sectors are contiguous. Two copies of the FAT are usually written for data integrity. The FAT is read into one of the MS-DOS buffers as needed (open, read, write, etc.). For performance reasons, this buffer is given a high priority so that it stays in memory as long as possible.

How to Use the File Allocation Table

Use the directory entry to find the starting cluster of the file. Next, to locate each subsequent cluster of the file:

1. Multiply the cluster number just used by 1.5 (each FAT entry is 1.5 bytes long).

2. The whole part of the product is an offset into the FAT, pointing to the entry that maps the cluster just used. That entry contains the cluster number of the next cluster of the file.

3. Use a MOV instruction to move the word at the calculated FAT offset into a register.

4. If the last cluster used was an even number, keep the low-order 12 bits of the register by ANDing it with FFF; otherwise, keep the high-order 12 bits by shifting the register right 4 bits with a SHR instruction.

5. If the resultant 12 bits are FF8H-FFFH, the file contains no more clusters. Otherwise, the 12 bits contain the cluster number of the next cluster in the file.

To convert the cluster to a logical sector number (relative sector, such as that used by Interrupts 25H and 26H and by DEBUG):

1. Subtract 2 from the cluster number.
2. Multiply the result by the number of sectors per cluster.
3. Add to this result the logical sector number of the beginning of the data area.

Diskette Formats

On an MS-DOS disk, the clusters are arranged on diskette to minimize head movement for multi-sided media. All of the space on a track (or cylinder) is allocated before moving on to the next track. This is accomplished by using the sequential sectors on the lowest-numbered head, then all the sectors on the next head, and so on until all sectors on all heads of the track are used. The next sector to be used will be sector 1 on head 0 of the next track.

The first byte of the FAT, called the Media Descriptor Byte, can sometimes be used to determine the format of the disk. The following formats have been defined for the AT&T Personal Computer 6300, based on values of the first byte of the FAT.

MS-DOS Standard Diskette Formats

No. sides	1	1	2	2
Tracks/side	40	40	40	40
Bytes/sector	512	512	512	512
Sectors/track	8	9	8	9
Sectors/cluster	1	1	2	2
Reserved sectors	1	1	1	1
No. FATs	2	2	2	2
Root directory entries	64	64	112	112
No. sectors	320	360	640	720
Media Descriptor Byte	FE	FC	FF	FD
Sectors for 1 FAT	1	2	1	2

6 Program Structure and Loading

- Overview
- Pros and Cons for Selecting a Program File Format
- EXE2BIN
- File Header Format
- Relocation Process for .EXE Files
- Program Segment Prefix
- Program Loading Process

Overview

This chapter describes the MS-DOS program file formats and procedures for loading them into memory. MS-DOS supports two main program file formats: .EXE and .COM.

The .EXE format is the more flexible program type. An .EXE file is limited in size only by the amount of user memory installed in your system.

Programs linked by MS-LINK are output in .EXE format. .EXE files can be executed either by COMMAND.COM or by an EXEC system call (Function Request 4BH) in your program.

A .COM program file cannot exceed 64K bytes in length. However, because it does not have the same lengthy header that an .EXE file does, a .COM file takes up less diskette storage space and loads into memory more quickly than an .EXE file.

After assembling and linking your program, it must be converted to .COM format. The easiest way to do this is with the EXE2BIN utility provided on your MS-DOS Supplemental Programs diskette.

Pros and Cons for Selecting a Program File Format

This section is concerned with the pros and cons of selecting between a .EXE program format and the .COM program type.

PROS for .EXE

- Can be larger than 64 K
- Can cross segment boundaries
- Can run .EXE immediately after linking, i.e., you need not take the extra step of running EXE2BIN
- Can declare a stack segment in the assembly program

CONS for .EXE

- Disk file has large "header" containing relocation information. .EXE therefore takes more space on disk and takes longer to load into memory at execution time.

PRO for .COM

- .COM files are smaller and faster loading because .COM does not have a file header containing relocation information.

CONS for .COM

- .COM files can be no larger than one 64K segment.

- .COM is segment-relocatable; the segment can be relocated at run time. However, all of the addresses in the program must be relative to the same segment address.

EXE2BIN

EXE2BIN

EXE2BIN is an executable program available on your MS-DOS system diskette. It converts programs that are in .EXE format (as they are after having been linked) into the .COM format.

EXE2BIN can generate two types of .COM files: relocatable and non-relocatable.

Syntax

EXE2BIN <input filename> <output file name>

Both file names are in the form:

[d:][path][filename[.ext]]

Example

EXE2BIN B:PROG.EXE B:PROG.COM

Discussion

In specifying the input file, everything except the file name is optional. If you do not specify a drive, the default is used. If you do not specify a path, the default path is used. If you do not specify an extension, the default is .EXE. The input file is converted to .COM file format (memory image of the program) and placed in the output file.

You are not required to enter any part of the output file specification. If you do not specify a drive, the drive of the input file will be used. If you do not specify an output path or filename, the input path or filename will be used. If you do not specify a filename extension in the output filename, the new file will be given an extension of .BIN.

The input file must be in valid .EXE format produced by the linker. The resident, or actual code and data part of the file must be less than 64K. There must be no STACK segment.

Two kinds of conversions are possible, depending on the initial CS:IP (Code Segment: Instruction Pointer) specified in the .EXE file:

1 If CS:IP is specified as 0000:100H, it is assumed that the file is to be run as a .COM file with the location pointer set at 100H by the assembler statement ORG; the first 100H bytes of the file are deleted. No segment address fixups (that is, instructions that contain a reference to an absolute segment address) are allowed, as .COM files must be segment relocatable. Once the conversion is complete, rename the resulting file with a .COM extension. The command processor can load and execute the program in the same way as the .COM programs supplied on your MS-DOS diskettes.

2 If CS:IP is not specified in the .EXE file, a pure binary conversion is assumed. If segment fixups are necessary (i.e., the program contains instructions requiring a segment address), you are prompted for the fixup value. This value is the absolute segment at which the program is to be loaded. The resulting program is usable only when loaded at the absolute memory address specified by your application. The command processor is not capable of properly loading the program. This is the case when writing a .BIN program to use in an application such as a device driver that is always loaded at the same absolute address.

EXE2BIN Messages

Amount read less than size in header

The program portion of the file was smaller than indicated in the file's header. You should reassemble and relink your program.

File cannot be converted

CS:IP does not meet either of the criteria specified above, or it meets the .COM file criterion but has segment fixups. This message is also displayed if the file is not a valid executable file.

File creation error

EXE2BIN cannot create the output file. Run CHKDSK to determine if the directory is full or if some other condition caused the error.

File not found

The file is not on the diskette specified.

Fixups needed - base segment (hex):

The source (.EXE) file contained information indicating that a load segment is required for the file. Specify the absolute segment address at which the finished module is to be located.

Insufficient disk space

There is not enough disk space to create a new file.

Insufficient memory

There is not enough memory to run EXE2BIN.

WARNING - Read error in EXE file

Amount read less than size in header. This is a warning message only. However, it is usually a good idea to reassemble and relink your source program when this message appears.

File Header Format

The .EXE files produced by MS-LINK consist of two parts:

- Control and relocation information
- The load module

The control and relocation information is at the beginning of the file in an area called the header. The load module immediately follows the header.

Note:
.COM files do not have file headers.

The header is formatted as follows (offsets are in hexadecimal):

Offset	Contents
00-01	Must contain 4DH, 5AH.
02-03	Number of bytes contained in last page; used for reading overlays.
04-05	Size of the file in 512-byte pages, including the header.
06-07	Number of relocation entries in table.
08-09	Size of the header in 16-byte paragraphs. This is used to locate the beginning of the load module in the file.
0A-0B	Minimum number of 16-byte paragraphs required above the end of the loaded program (minalloc).
0C-0D	Maximum number of 16-byte paragraphs required above the end of the loaded program (maxalloc). If both minalloc and maxalloc are 0, then the program will be loaded as high as possible.

0E-0F	Initial value to be loaded into stack segment before starting program execution. This must be adjusted by relocation.
10-11	Value to be loaded into the SP register before starting program execution.
12-13	Negative sum of all the words in the file.
14-15	Initial value to be loaded into the IP register before starting program execution.
16-17	Initial value to be loaded into the CS register before starting program execution. This must be adjusted by relocation.
18-19	Relative byte offset from beginning of run file to relocation table.
1A-1B	The number of the overlays generated by MS-LINK.

This is followed by the relocation table. The table consists of a variable number of relocation items. Each relocation item contains two fields: a two-byte offset value, followed by a two-byte segment value. These two fields contain the offset into the load module of a word which requires modification before the module is given control.

Relocation Process for .EXE Files

The following steps describe the relocation process:

1 The formatted part of the header is read into memory. Its size is 1BH.

2 A portion of memory is allocated depending on the size of the load module and the allocation numbers (0A-0B and 0C-0D). MS-DOS attempts to allocate FFFFH paragraphs. This will always fail, returning the size of the largest free block. If this block is smaller than minalloc and loadsize, then there will be a no memory error. If this block is larger than maxalloc and loadsize, MS-DOS will allocate (maxalloc + loadsize). Otherwise, MS-DOS will allocate the largest free block of memory.

3 A Program Segment Prefix is built in the lowest part of the allocated memory.

4 The load module size is calculated by subtracting the header size from the file size. Offsets 04-05 and 08-09 can be used for this calculation. The actual size is downward-adjusted based on the contents of offsets 02-03. Based on the setting of the high/low loader switch, an appropriate segment is determined at which to load the load module. This segment is called the start segment.

5 The load module is read into memory beginning with the start segment.

6 The relocation table items are read into a work area.

7 Each relocation table item segment value is added to the start segment value. This calculated segment, plus the relocation item offset value, points to a word in the load module to which is added the start segment value. The result is placed back into the word in the load module.

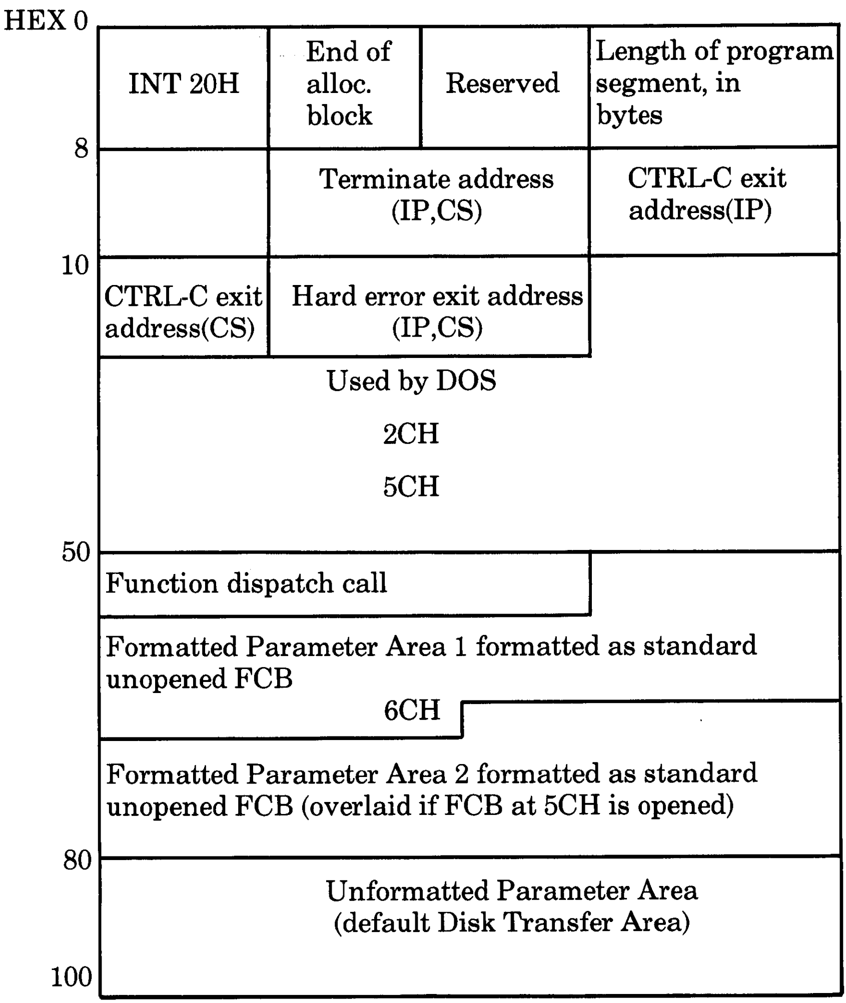
8 Once all relocation items have been processed, the SS and SP registers are set from the values in the header. Then, the start segment value is added to SS. The ES and DS registers are set to the segment address of the Program Segment Prefix. The start segment value is added to the header CS register value. The result, along with the header IP value, is the initial CS:IP to transfer to before starting execution of the program.

Program Segment Prefix

Unless you specify otherwise when linking your program, DOS loads your program in the lowest memory address available, immediately following the DOS code. This occurs whether the program loads as a result of your entering its name at the DOS prompt or through your use of the EXEC (4BH) function call. The area into which your program is loaded is called the Program Segment.

DOS requires control information for each running program: it builds a Program Segment Prefix and places it at offset 0 within the program segment. The Program Segment Prefix is hex 100 bytes long, so your program is loaded at relative address 100H.

PSP Format


HEX 0
INT 20H
End of alloc. block
Reserved
Length of program segment, in bytes
8
Terminate address (IP,CS)
CTRL-C exit address(IP)
10
CTRL-C exit address(CS)
Hard error exit address (IP,CS)
Used by DOS
2CH
5CH
50
Function dispatch call
Formatted Parameter Area 1 formatted as standard unopened FCB
6CH
Formatted Parameter Area 2 formatted as standard unopened FCB (overlaid if FCB at 5CH is opened)
80
Unformatted Parameter Area (default Disk Transfer Area)
100

HEX OFFSET	CONTENTS	
0	Return address used by interrupt hex 20	
2	Segment address of allocatable memory following this program (If this program calls a memory management function to get more memory, this is its starting address.)	
4	Reserved	
6	Number of bytes in this program segment (2 byte value)	
8	Not used	
A	Terminate address : IP	
C	Terminate address : CS	
E	Ctrl break exit : IP	
10	Ctrl break exit : CS	
12	Critical error exit : IP	
14	Critical error exit : CS	
2C	Segment address of the environment	USED by DOS
50	Code to call function dispatcher for DOS (INT 21H) interrupts	
5C	Formatted parameter area 1: formatted as standard, unopened FCB	
6C	Formatted parameter area 2: formatted as standard, unopened FCB	
80	Count of argument characters that follow the command name.	These comprise the default DTA: Disk Transfer Area(80H - FFH)
81	The argument characters themselves.	

Program Loading Process

PSP Conditions upon Program Initiation

When a program receives control, the following conditions are in effect:

- The segment address of the passed environment is contained at offset 2CH in the Program Segment Prefix. The environment is a series of ASCII strings (totaling less than 32K) in the form:

  NAME = parameter

  Each string is terminated by a byte of zeros, and the set of strings is terminated by another byte of zeros. The environment built by the command processor contains at least a COMSPEC= string (the parameters on COMSPEC define the path used by MS-DOS to locate COMMAND.COM on disk). The last PATH and PROMPT commands issued will also be in the environment, along with any environment strings defined with the MS-DOS SET command.

  The environment that is passed is a copy of the invoking process environment. If your application uses a "keep process" concept, be aware that the copy of the environment passed to you is static. That is, it will not change even if subsequent SET, PATH, or PROMPT commands are issued.

- Offset 50H in the Program Segment Prefix contains code to call the MS-DOS function dispatcher. After correctly loading the registers, a program can issue a far call to offset 50H to invoke an MS-DOS function, rather than issuing an Interrupt 21H. Since this is a call and not an interrupt, MS-DOS may place any code appropriate to making a system call at this position. This makes the process of calling the system portable.

- The Disk Transfer Address (DTA) is set to 80H (default DTA in the Program Segment Prefix).

- File control blocks at 5CH and 6CH are formatted from the first two parameters typed when the command was entered. If either parameter contains a pathname, then the corresponding FCB contains only the valid drive number. The filename field will not be valid.

- An unformatted parameter area at 81H contains all the characters typed after the command (including leading and imbedded delimiters), with the byte at 80H set to the number of characters. If the < or > parameters were typed on the command line, they (and the filenames associated with them) do not appear in this area or in the character count; redirection of standard input and output is transparent to applications.

- Offset 6 (one word) contains the number of bytes available in the segment.

- Register AX indicates whether or not the drive specifiers (entered with the first two parameters) are valid, as follows:

  AL=FF if the first parameter contained an invalid drive specifier (otherwise AL=00)

  AH=FF if the second parameter contained an invalid drive specifier (otherwise AH=00)

- Offset 2 (one word) contains the segment address of the first byte of unavailable memory. Programs must not modify addresses beyond this point unless they were obtained by allocating memory via the Allocate Memory system call (Function Request 48H).

Initial Conditions for .EXE Programs

- DS and ES registers are set to point to the Program Segment Prefix.

- CS, IP, SS, and SP registers are set to the values passed by MS-LINK.

Initial Conditions for .COM Programs

- All four segment registers contain the segment address of the initial allocation block that starts with the Program Segment Prefix control block.
- The Instruction Pointer (IP) is set to 100H.
- The Stack Pointer register is set to the end of the program's segment. The segment size at offset 6 is reduced by 100H to allow for a stack of that size.
- A word of zeros is placed on top of the stack. This allows your program to exit to COMMAND.COM by doing a RET instruction last. Make sure, however, to maintain your stack and code segments.

Other Uses of the Program Segment Prefix

In MS-DOS versions prior to 2.0, the PSP contained the mechanism for program termination. One of these four techniques had to be used to terminate your programs:

1. A long jump to offset 0 in the Program Segment Prefix.
2. By issuing an INT 20H with CS:0 pointing at the PSP.
3. By issuing an INT 21H with register AH = 0 and CS:0 pointing at the PSP.
4. By a long call to location 50H in the Program Segment Prefix with AH = 0 and CS:0 pointing at the PSP.

It is the responsibility of all programs to ensure that the CS register contains the segment address of the Program Segment Prefix when terminating via any of these methods.

However, with the 2.0 Terminate a Process system call (Function Request 4CH), the CS register need not point to the Program Segment Prefix. For this reason, Function Request 4CH is the preferred method. It may be invoked by loading the AH register with 4CH and issuing an INT 21H (or a long call to offset 50H in the Program Segment Prefix).

7 System Calls

- Quick Reference: Functions and Interrupts
- Overview
- Programming Considerations
- Interrupts
- Functions
- System Call Descriptions

Functions

Number	Function Name
00H	Terminate Program
01H	Read Keyboard and Echo
02H	Display Character
03H	Auxiliary Input
04H	Auxiliary Output
05H	Print Character
06H	Direct Console I/O
07H	Direct Console Input
08H	Read Keyboard
09H	Display String
0AH	Buffered Keyboard Input
0BH	Check Keyboard Status
0CH	Flush Buffer, Read Keyboard
0DH	Disk Reset
0EH	Select Disk
0FH	Open File
10H	Close File
11H	Search for First Entry
12H	Search for Next Entry
13H	Delete File
14H	Sequential Read
15H	Sequential Write
16H	Create File
17H	Rename File
19H	Current Disk
1AH	Set Disk Transfer Address
21H	Random Read
22H	Random Write
23H	File Size
24H	Set Relative Record
25H	Set Vector
27H	Random Block Read
28H	Random Block Write
29H	Parse File Name
2AH	Get Date
2BH	Set Date
2CH	Get Time
2DH	Set Time
2EH	Set/Reset Verify Flag
2FH	Get Disk Transfer Address

Number	Function Name
30H	Get DOS Version Number
31H	Keep Process
33H	<CTRL C> Check
35H	Get Interrupt Vector
36H	Get Disk Free Space
38H	Return Country-Dependent Info.
39H	Create Sub-Directory
3AH	Remove a Directory
3BH	Change the Current Directory
3CH	Create a File
3DH	Open a File Handle
3EH	Close a File Handle
3FH	Read From File/Device
40H	Write to a File/Device
41H	Delete a Directory Entry
42H	Move a File Pointer
43H	Change Attributes
44H	I/O Control for Devices
45H	Duplicate a File Handle
46H	Force a Duplicate of a Handle
47H	Return Name of Current Directory
48H	Allocate Memory
49H	Free Allocated Memory
4AH	Modify Allocated Memory Blocks
4BH	Load and Execute a Program (EXEC)
4CH	Terminate a Process
4DH	Retrieve the Return Code of a Child
4EH	Find Match File
4FH	Step Through a Directory Matching Files
54H	Return Current Setting of Verify
56H	Move a Directory Entry
57H	Get/Set Date/Time of File

Function Name	Number	Function Name	Number
Allocate Memory	48H	Modify Allocated Memory Blocks	4AH
Auxiliary Input	03H	Move a Directory Entry	56H
Auxiliary Output	04H	Move a File Pointer	42H
Buffered Keyboard Input	0AH	Open a File Handle	3DH
Change Attributes	43H	Open File	0FH
Change the Current Directory	3BH	Parse File Name	29H
Check Keyboard Status	0BH	Print Character	05H
Close a File Handle	3EH	Random Block Read	27H
Close File	10H	Random Block Write	28H
CTRL C Check	33H	Random Read	21H
Create a File	3CH	Random Write	22H
Create File	16H	Read From File/Device	3FH
Create Sub-Directory	39H	Read Keyboard	08H
Current Disk	19H	Read Keyboard and Echo	01H
Delete a Directory Entry	41H	Remove a Directory	3AH
Delete File	13H	Rename File	17H
Direct Console Input	07H	Retrieve the Return Code of a Child	4DH
Direct Console I/O	06H	Return Current Setting of Verify	54H
Disk Reset	0DH	Return Country-Dependent Info.	38H
Display Character	02H	Return Name of Current Directory	47H
Display String	09H	Search for First Entry	11H
Duplicate a File Handle	45H	Search for Next Entry	12H
File Size	23H	Select Disk	0EH
Find Match File	4EH	Sequential Read	14H
Flush Buffer, Read Keyboard	0CH	Sequential Write	15H
Force a Duplicate of a Handle	46H	Set Date	2BH
Free Allocated Memory	49H	Set Disk Transfer Address	1AH
Get Date	2AH	Set Relative Record	24H
Get Disk Free Space	36H	Set Time	2DH
Get Disk Transfer Address	2FH	Set Vector	25H
Get DOS Version Number	30H	Set/Reset Verify Flag	2EH
Get Interrupt Vector	35H	Step Through a Directory Matching	4FH
Get Time	2CH	Terminate a Process	4CH
Get/Set Date/Time of File	57H	Terminate Program	00H
I/O Control for Devices	44H	Write to a File/Device	40H
Keep Process	31H		
Load and Execute a Program (EXEC)	4BH		

Interrupts

Interrupts (in Numerical Order)

Interrupt (Hex)	Interrupt (Decimal)	Description
20H	32	Program Terminate
21H	33	Function Request
22H	34	Terminate Address
23H	35	<CTRL C> Exit Address
24H	36	Fatal Error Abort Address
25H	37	Absolute Disk Read
26H	38	Absolute Disk Write
27H	39	Terminate But Stay Resident
28-40H	40-64	RESERVED — DO NOT USE

Interrupts in Alphabetical Order

Description	Interrupt in Hex	Interrupt in Dec
Absolute Disk Read	25H	37
Absolute Disk Write	26H	38
<CTRL C> Exit Address	23H	35
Fatal Error Abort Address	24H	36
Function Request	21H	33
Program Terminate	20H	32
RESERVED — DO NOT USE	28-40H	40-64
Terminate Address	22H	34
Terminate But Stay Resident	27H	39

Overview

System Calls are procedures used to interface with I/O or to manage memory. They can be accessed from utility programs written in assembly language, and from some high level languages. Their use frees the programmer from having to perform primitive functions, and makes it easier to write machine-independent programs.

MS-DOS provides two types of system calls: interrupts and function requests. This chapter describes the environments from which these routines can be called, how to call them, and the processing performed by each.

Programming Considerations

System calls can be invoked from Assembly Language, from GW BASIC, or from high-level languages like PASCAL and FORTRAN. This section describes the techniques for invoking calls and for returning control to MS-DOS.

Calling from Assembly Language

The system calls can be invoked from Assembly Language simply by moving any required data into registers and issuing an interrupt. Some of the calls destroy registers, so you may have to save registers before using a system call.

Calling from GW BASIC

The BLOAD and BSAVE commands are used for loading and saving machine language programs. These are then called, using the CALL statement.

The USR function calls an indicated machine language subroutine. The starting address of the subroutine must first be specified in a DEF USR statement.

Interrupts

MS-DOS reserves interrupts 20H through 3FH for its own use. The table of interrupt routine addresses (vectors) is maintained in locations 80H-FCH. User programs should only issue Interrupts 20H, 21H, 25H, 26H, and 27H. (Functions Requests 4CH and 31H are the preferred method for Interrupts 20H and 27H for versions of MS-DOS that are 2.0 and higher.

Interrupts 22H, 23H, and 24H are not interrupts that can be issued by user programs; they are simply locations where a segment and offset address are stored. For a discussion, see the section on Address Interrupts in this chapter.

Functions

Requirements Most of the MS-DOS function calls require input to be passed to them in registers. After setting the proper register values, the function may be invoked in one of the following ways:

- Place the function number in AH and execute a long call to offset 50H in your Program Segment Prefix. Note that programs using this method will not operate correctly on versions of MS-DOS that are lower than 2.0.

- Place the function number in AH and issue Interrupt 21H. All of the examples in this chapter use this method.

- An additional method exists for programs that were written with different calling conventions. This method should be avoided for all new programs. The function number is placed in the CL register and other registers are set according to the function specification. Then, an intrasegment call is made to location 5 in the current code segment. That location contains a long call to the MS-DOS function dispatcher. Register AX is always destroyed if this method is used; otherwise, it is the same as normal function calls. Note that this method is valid only for Function Requests 00H through 024H.

This chapter provides the following type of information for each DOS interrupt and function call:

- a description of the register contents required before the system call
- a description of the register contents after the system call
- a description of the processing performed
- an example of its use.

Registers

When MS-DOS takes control after a function call, it switches to an internal stack. Registers not used to return information (except AX) are preserved. The calling program's stack must be large enough to accommodate the interrupt system — at least 128 bytes in addition to other needs.

Note

The macro definitions and extended example for MS-DOS system calls 00H through 2EH can be found at the end of this chapter.

System Call Descriptions

Interrupts

The following are not true interrupts but rather storage locations for a segment and offset address:

- Terminate Address (Interrupt 22H)
- **CTRL C** Exit Address (Interrupt 23H)
- Fatal Error Abort Address (Interrupt 24H)

The interrupts are issued by MS-DOS under the specified circumstance. You can change any of these addresses with Function Request 25H (Set Vector) if you prefer to write your own interrupt handlers.

Programming Examples

A macro is defined for most system calls, then used in some examples. In addition, a few other macros are defined for use in the examples. The use of macros allows the examples to be more complete programs, rather than isolated uses of the system calls. All macro definitions are listed at the end of the chapter.

The examples are not intended to represent good programming practice. In particular, error checking and good human interface design have been sacrificed to conserve space. You may, however, find the macros a convenient way to include system calls in your assembly language programs.

A detailed description of each system call follows. They are listed in numeric order; the interrupts are described first, then the function requests.

Note

Unless otherwise stated, all numbers in the system call descriptions, both text and code, are in hex.

20H
Program Terminate

Call

CS
Segment address of Program Segment Prefix

Return

None

Remarks

All open file handles are closed and the disk cache is cleaned. The current process is terminated and control returns to the parent process. This interrupt is almost always used in old .COM files for termination.

The CS register must contain the segment address of the Program Segment Prefix before you call this interrupt.

The following exit addresses are restored from the Program Segment Prefix:

Exit Address	Offset
Program Terminate	0AH
<CTRL C>	0EH
Critical Error	12H

All file buffers are flushed to disk.

Note

Close all files that have changed in length before issuing this interrupt. If a changed file is not closed, its length is not recorded correctly in the directory. See Functions 10H and 3EH for a description of the Close File system calls.

Interrupt 20H is provided for compatibility with versions of MS-DOS prior to 2.0. New programs should use Function Request 4CH, Terminate a Process.

Macro

```
terminate macro
  int   20H
  endm
```

Example

```
;CS must be equal to PSP values given at program
;start
;(ES and DS values)
  INT 20H
;There is no return from this interrupt
```

21H
Function Request

Call

AH
  Function number
Other registers as specified in
  individual function

Return

As specified in individual function

Remarks

The AH register must contain the number of the system function. See the following section on Function Requests, in this chapter, for a description of the MS-DOS system functions.

Note

No macro is defined for this interrupt, because all function request descriptions in this chapter that define a macro include Interrupt 21H.

Example

To call the Get Time function:

```
mov  ah,2CH    ;Get Time is Function 2CH
int  21H       ;THIS INTERRUPT
```

22H
Terminate Address

When a program terminates, control transfers to the address at offset 0AH of the Program Segment Prefix. This address is copied into the Program Segment Prefix, from the Interrupt 22H vector, when the segment is created. Interrupt 22H, then, is just a storage location for an address rather than a true interrupt.

23H
<CTRL C> Exit Address

If the user types **CTRL C** during keyboard input or display output, control transfers to the INT 23H vector in the interrupt table. This address is copied into the Program Segment Prefix, from the Interrupt 23H vector, when the segment is created.

If the **CTRL C** routine preserves all registers, it can end with an IRET instruction (return from interrupt) to continue program execution. When the interrupt occurs, all registers are set to the value they had when the original call to MS-DOS was made. There are no restrictions on what a **CTRL C** handler can do — including MS-DOS function calls — so long as the registers are unchanged if IRET is used.

If Function 09H or 0AH (Display String or Buffered Keyboard Input) is interrupted by **CTRL C**, the three-byte sequence 03H-0DH-0AH (ETX-CR-LF) is sent to the display and the function resumes at the beginning of the next line.

If the program creates a new segment and loads a second program that changes the **CTRL C** address, termination of the second program restores the **CTRL C** address to its value before execution of the second program.

Like INT 22H, this is really not a true interrupt, but a storage location.

24H
Fatal Error Abort Address

Call

If a fatal disk error occurs during execution of one of the disk I/O function calls, control transfers to the INT 24H vector in the vector table. This address is copied into the Program Segment Prefix, from the Interrupt 24H vector, when the segment is created.

Return

BP:SI contains the address of a Device Header Control Block from which additional information can be retrieved.

Note

Interrupt 24H is not issued if the failure occurs during execution of Interrupt 25H (Absolute Disk Read) or Interrupt 26H (Absolute Disk Write). These errors are usually handled by the MS-DOS error routine in COMMAND.COM that retries the disk operation, then gives the user the choice of aborting, retrying the operation, or ignoring the error. The following topics give you the information you need about interpreting the error codes, managing the registers and stack, and controlling the system's response to the error in order to write your own error-handling routines.

Error Codes

When an error-handling program gains control from Interrupt 24H, the AX and DI registers can contain codes that describe the error. If Bit 7 of AH is 1, the error is either a bad image of the File Allocation Table or an error which has occurred on a character device. The device header passed in BP:SI can be examined to determine which case exists. If the attribute byte high order bit indicates a block device, then the error was a bad FAT. Otherwise, the error is on a character device.

The following are error codes for Interrupt 24H:

Error Code	Description
0	Attempt to write on write-protected disk
1	Unknown unit
2	Drive not ready
3	Unknown command
4	Data error
5	Bad request structure length
6	Seek error
7	Unknown media type
8	Sector not found
9	Printer out of paper
A	Write fault
B	Read fault
C	General failure

The user stack will be in effect (the first item described below is at the top of the stack), and will contain the following from top to bottom:

IP	MS-DOS registers from
CS	issuing INT 24H
FLAGS	
AX	User registers at time of original
BX	INT 21H request
CX	
DX	
SI	
DI	
BP	
DS	
ES	
IP	From the original INT 21H
CS	from the user to MS-DOS
FLAGS	

The registers are set such that if an IRET is executed, MS-DOS will respond according to (AL) as follows:

(AL) = 0 ignore the error
= 1 retry the operation
= 2 terminate the program via INT 23H

Note

- Before giving this routine control for disk errors, MS-DOS performs five retries.

- For disk errors, this exit is taken only for errors occurring during an Interrupt 21H. It is not used for errors during Interrupts 25H or 26H.

- This routine is entered in a disabled state.

- The SS,SP,DS,ES,BX,CX, and DX registers must be preserved.

- The interrupt handler should refrain from using MS-DOS function calls. If necessary, it may use calls 01H through 0CH. Use of any other call will destroy the MS-DOS stack and will leave MS-DOS in an unpredictable state.

- The interrupt handler must not change the contents of the device header.

- If the interrupt handler will handle errors rather than returning to MS-DOS, it should restore the application program's registers from the stack, remove all but the last three words on the stack, then issue an IRET. This will return to the program immediately after the INT 21H that experienced the error. Note that if this is done, MS-DOS will be in an unstable state until a function call higher than 0CH is issued.

25H Absolute Disk Read

Call

AL
  Drive number (0 = A, 1 = B, etc.)
DS:BX
  Disk Transfer Address
CX
  Number of sectors
DX
  Beginning relative sector

Return

Flags
  CF = 0 if successful
     = 1 if not successful
  AL
    Error code if CF = 1

Remarks

This interrupt transfers control to the MS-DOS BIOS. The number of sectors specified in CX is read from the disk to the Disk Transfer Address. Its requirements and processing are identical to Interrupt 26H, except data is read rather than written.

Note

All registers except the segment registers are destroyed by this call. Be sure to save any registers your program uses before issuing the interrupt.

The system pushes the flags at the time of the call; they are still there upon return. (This is necessary because data is passed back in the flags). Be sure to pop the stack upon return to restore your stack pointer at the point of invocation.

If the disk operation was successful, the Carry Flag (CF) is 0. If the disk operation was not successful, CF is 1 and AL contains the MS-DOS error code (see Interrupt 24H earlier in this section for the codes and their meaning).

Macro

```
abs_disk_read macro disk,buffer,num_sectors, start
   mov  al,disk
   mov  bx,offset buffer
   mov  cx,num_sectors
   mov  dx,start
   int  25H
   endm
```

Example

The following program copies the contents of a single-sided disk in drive A: to the disk in drive B:. It uses a buffer of 32K bytes:

```
prompt   db  "Source in A, target in B",13,10
         db  "Any key to start. $"
start    dw  0
buffer   db  64 dup (512 dup (?)) ;64 sectors
         .
         .
int_25H:    display prompt ;see Function 09H
            read_kbd       ;see Function 08H

        mov  cx,5                         ;copy 5 groups of
                                          ;64 sectors
copy:   push  cx                          ;save the loop
                                          ;counter
        abs_disk_read 0,buffer, 64,start  ;THIS
                                          ;INTERRUPT
        abs_disk_write 1,buffer, 64,start ;see INT
                                          ;26H
        add start,64                      ;do the next 64
                                          ;sectors
        pop cx                            ;restore the loop
                                          ;counter
        loop copy
```

26H
Absolute Disk Write

Call

AL
Drive number (0 = A, 1 = B, etc.)
DS:BX
Disk Transfer Address
CX
Number of sectors
DX
Beginning relative sector

Return

FLAGS
CF = 0 if successful
= 1 if not successful
AL
Error code if CF = 1

Remarks

This interrupt transfers control to the MS-DOS BIOS. The number of sectors specified in CX is written from the Disk Transfer Address to the disk. Its requirements and processing are identical to Interrupt 25H, except data is written to the disk rather than read from it.

Note

All registers except the segment registers are destroyed by this call. Be sure to save any registers your program uses before issuing the interrupt.

The system pushes the flags at the time of the call; they are still there upon return. (This is necessary because data is passed back in the flags). Be sure to pop the stack upon return to restore your stack pointer at the point of invocation.

If the disk operation was successful, the Carry Flag (CF) is 0. If the disk operation was not successful, CF is 1 and AL contains the MS-DOS error code (see Interrupt 24H earlier in this section for the codes and their meaning).

Macro

```
abs_disk_write macro disk,buffer,num_sectors, start
   mov  al,disk
   mov  bx,offset buffer
   mov  cx,num_sectors
   mov  dx,start
   int  26H
   endm
```

Example

The following program copies the contents of a single-sided disk in drive A: to the disk in drive B:, verifying each write. It uses a buffer of 32K bytes:

```
off       equ 0
on        equ 1
          .
          .
prompt    db  "Source in A, target in B", 13,10
          db  "Any key to start. $"
start     dw  0

buffer    db 64 dup (512 dup (?)) ; 64 sectors
          .
          .
int_26H:  display prompt                  ;see Function 09H
          read_kbd                        ;see Function 08H
          verify on                       ;see Function 2EH
          mov cx,5                        ;copy 5 groups of
                                          ;64 sectors
copy:     push  cx                        ;save the loop
                                          ;counter
          abs_disk_read 0,buffer, 64,start  ;see INT
                                          ;25H
          abs_disk_write 1,buffer, 64,start  ;THIS
                                          ;INTERRUPT
          add start,64                    ;do the next 64
                                          ;sectors
          pop cx                          ;restore the loop
                                          ;counter
          loop copy
          verify off                      ;see Function 2EH
```

27H
Terminate But Stay Resident

Call

CCS:DX
First byte following
last byte of code

Return

None

Remarks

The Terminate But Stay Resident call is used to make a piece of code remain resident in the system after its termination. Typically, this call is used in .COM files to allow some device-specific interrupt handler to remain resident to process asynchronous interrupts.

DX must contain the number of bytes in the CS segment to be reserved. When Interrupt 27H is executed, the program terminates but is treated as an extension of MS-DOS; it remains resident and is not overlaid by other programs when it terminates.

If an executable file whose extension is .COM or .EXE ends with this interrupt, it becomes a resident operating system command.

This interrupt is provided for compatibility with versions of MS-DOS prior to 2.0. New programs should use Function 31H, Keep Process.

Macro

```
stay_resident macro last_instruc
   mov   dx,offset last_instruc
   inc   dx
   int   27H
   endm
```

Example

```
;CS must be equal to PSP values given at program
;start
;(ES and DS values)
;the variable Last Address must be equal
;to the offset of the last byte in the
;program.
   mov   DX,LastAddress
   inc   dx
   int   27H
;There is no return from this interrupt
```

00H
Terminate Program

Call

AH = 00H
CS
Segment address of
Program Segment Prefix

Return

None

Remarks

Function 00H is called by Interrupt 20H; it performs the same processing.

The CS register must contain the segment address of the Program Segment Prefix before you call this interrupt.

The following exit addresses are restored from the specified offsets in the Program Segment Prefix:

Program terminate	0AH
<CTRL C>	0EH
Critical error	12H

All file buffers are flushed to disk.

Warning

Close all files that have changed in length before calling this function. If a changed file is not closed, its length is not recorded correctly in the directory. See Function 10H for a description of the Close File system call.

00H
Terminate Program

Macro

```
terminate_program macro
   xor  ah,ah
   int  21H
   endm
```

Example

```
;CS must be equal to PSP values given at program start
;(ES and DS values)
    mov   ah,0
    int    21H
;There are no returns from this interrupt
```

01H
Read Keyboard and Echo

Call AH = 01H

Return AL
Character typed

Remarks Function 01H waits for a character to be typed at the keyboard, then echoes the character to the display and returns it in AL. If the character is **CTRL C,** Interrupt 23H is executed.

Macro

```
read_kdb_and_echo macro
  mov    ah,01H
  int    21H
  endm
```

Example The following program both displays and prints characters as they are typed. If **CR** is pressed, the program sends Line Feed-Carriage Return to both the display and the printer:

```
func_01H: read_kbd_and_ echo          ;THIS FUNCTION
          print_char      al          ;see Function 05H
          cmp             al,0DH      ;is it a CR?
          jne             func_01H    ;no, print it
          print_char      10          ;see Function 05H
          display_char    10          ;see Function 02H
          jmp             func_01H    ;get another character
```

02H
Display Character

Call AH = 02H
DL
Character to be displayed

Return None

Remarks If **CTRL C** is typed, Interrupt 23H is issued.

Macro

```
display_char macro  character
   mov   dl,character
   mov   ah,02H
   int   21H
   endm
```

Example The following program converts lowercase characters to uppercase before displaying them:

```
func_02H:   read_kbd              ;see FUNCTION 08H
            cmp   al,"a"
            jl    uppercase       ;don't convert
            cmp   al,"z"
            jg    uppercase       ;don't convert
            sub   al,20H          ;convert to ASCII code
                                  ;for uppercase
uppercase:  display_char al       ;THIS FUNCTION
            jmp   func_02H        ;get another character
```

03H
Auxiliary Input

Call AH = 03H

Return AL
Character from auxiliary device

Remarks Function 03H waits for a character from the auxiliary input device, then returns the character in AL.

This system call does not return a status or error code.

If a **CTRL C** has been typed at console input, Interrupt 23H is issued.

Macro

```
aux_input macro
  mov  ah,03H
  int  21H
  endm
```

Example The following program prints characters as they are received from the auxiliary device. It stops printing when an end-of-file character (ASCII 26, or **CTRL Z**) is received:

```
func_03H:  aux_input               ;THIS FUNCTION
           cmp        al,1AH       ;end of file?
           je         continue     ;yes, all done
           print_char al           ;see Function 05H
           jmp        func_03H     ;get another character
continue:  .
           .
```

04H
Auxiliary Output

Call

AH = 04H
DL
  Character for auxiliary device

Return

None

Remarks

This system call does not return a status or error code.

If a **CTRL C** has been typed at console input, Interrupt 23H is issued.

Macro

```
aux_output macro character
  mov   dl,character
  mov   ah,04H
  int   21H
  endm
```

Example

The following program gets a series of strings of up to 80 bytes from the keyboard, sending each to the auxiliary device. It stops when a null string (CR only) is typed:

```
string     db      82 dup(?)               ;see Function 0AH
           .
           .
func_04H:  get_string 80,string            ;see Function 0AH
           cmp     string[1],0             ;null string?
           je      continue                ;yes, all done
           xor     ch,ch                   ;zero high byte
           mov     cl, byte ptr string[1]  ;get string length
           mov     bx,0                    ;set index to 0
send_it:   aux_output string[bx+2]         ;THIS FUNCTION
           inc     bx                      ;bump index
           loop    send_it                 ;send another character
           jmp     func_04H                ;get another string
continue:  .
           .
```

Call

AH = 05H
DL
Character for printer

Return

None

Remarks

If **CTRL C** has been typed at console input, Interrupt 23H is issued.

Macro

```
print_char macro  character
  mov  dl,character
  mov  ah,05H
  int  21H
  endm
```

Example

The following program prints a walking test pattern on the printer. It stops if **CTRL C** is pressed.

```
line_num   db        0
           .
           .
func_05H:  mov       bl,33          ;first printable ASCII
                                    ;character (!)
           add       bl,line_num    ;to offset next character
           mov       cx,80          ;loop counter for line
print_it:  print_char bl            ;THIS FUNCTION
           inc       bl             ;move to next ASCII character
           cmp       bl,126         ;last printable ASCII
                                    ;character (~)
           jle       no_reset       ;not there yet
           mov       bl,33          ;start over with (!)

no_reset:  loop      print_it       ;print another character
           print_char 13            ;carriage return
           print_char 10            ;line feed
           inc       line_num       ;to offset 1st char. of line
           cmp       line_num, 93   ;end of cycle?
           jle       func_05H       ;nope, not yet
           mov       line_num, 0    ;reset char offset
           jmp       func_05H       ;continue
```

06H
Direct Console I/O

Call

AH = 06H
DL
  See Text

Return

AL
  If DL = FFH before call
    If Zero flag not set:
      Character from keyboard
    If Zero flag set:
      No character input

Remarks

Processing depends on the value in DL when the function is called:

DL is FFH.
  If a character has been typed at the keyboard, it is returned in AL and the Zero flag is 0; if a character has not been typed, the Zero flag is 1.

DL is not FFH.
  The character in DL is displayed.

This function does not check for **CTRL C.**

Macro

```
dir_console_io macro switch
   mov   dl,switch
   mov   ah,06H
   int   21H
   endm
```

Example

The following program acts as a stopwatch. When a character is typed, it sets the system clock to zero and begins to continuously display the time. When a second character is typed the system stops updating the time display.

```
time         db"00:00:00.00",13,"$"
ten          db 10
             .
             .
func_06H:    dir_console_io  0FFH     ;THIS FUNCTION
             jz       func_06H        ;wait for keystroke
             set_time        0,0,0,0  ;see Function 2DH
read_clock:  get_time                 ;see Function 2CH
             convert ch,ten,time      ;see end of chapter
             convert cl,ten,time[3]   ;see end of chapter
             convert dh,ten,time[6]   ;see end of chapter
             convert dl,ten,time[9]   ;see end of chapter
             display time             ;see Function 09H
             dir_console_io  0FFH     ;THIS FUNCTION
             jz       read_clock      ;no char, keep updating
continue:    .
             .
             .
```

07H
Direct Console Input

Call AH = 07H

Return AL
Character from keyboard

Remarks This function does not echo the character or check for **CTRL C.** (For a keyboard input function that echoes or checks for **CTRL C,** see Functions 01H or 08H.)

Macro

```
dir_console_input macro
                  mov   ah,07H
                  int   21H
                  endm
```

Example The following program fragment prompts for a password (8 characters maximum) and places the characters into a string without echoing them:

```
password    db    8 dup(?)
prompt      db    "Password: $"      ;see Function 09H for
                                     ;explanation of $
            .
            .
func_07H:   display prompt           ;see Function 09H
            mov   cx,8               ;maximum length of password
            xor   bx,bx              ;so BL can be used as index
get_pass:   dir_console_input        ;THIS FUNCTION
            cmp   al,0DH             ;was it a CR?
            je    continue           ;yes, all done
            mov   password[bx],al    ;no, put character in string
            inc   bx                 ;bump index
            loop  get_pass           ;get another character
continue:   .                        ;BX has length of password
            .
```

08H
Read Keyboard

Call AH = 08H

Return AL
Character from keyboard

Remarks If **CTRL C** is pressed, Interrupt 23H is executed. This function does not echo the character. (For a keyboard input function that echoes the character or does not check for **CTRL C,** see Functions 01H or 07H.)

Macro

```
read_kbd macro
        mov   ah,08H
        int   21H
        endm
```

Example The following program fragment prompts for a password (8 characters maximum) and places the characters into a string without echoing them:

```
password   db     8 dup(?)
prompt     db     "Password: $"      ;see Function 09H
                                     ;for explanation of $
           .
           .
func_08H:  display prompt            ;see Function 09H
           mov    cx,8               ;maximum length of password
           xor    bx,bx              ;BL can be an index
get_pass:  read_kbd;                 ;THIS FUNCTION
           cmp    al,0DH             ;was it a CR?
           je     continue           ;yes, all done
           mov    password[bx],al    ;no, put char. in string
           inc    bx                 ;bump index
           loop   get_pass           ;get another character
continue:  .                         ;BX has length of password
           .
```

Call

AH = 09H
DS:DX
String to be displayed

Return

None

Remarks

DX must contain the offset (from the segment address in DS) of a string that ends with "$". The string is displayed (the $ is not displayed).

Macro

```
display macro string
  mov   dx,offset string
  mov   ah,09H
  int   21H
  endm
```

Example

The following program displays the hexadecimal code of the key that is typed:

```
table      db "0123456789ABCDEF"
sixteen    db 16
result     db "-00H",13,10,"$"          ;see text for
                                        ;explanation of $
           .
           .
func_09H:  read_kbd_and_echo            ;see Function 01H
           convert al,sixteen,result[1] ;see end of chapter
           display result               ;THIS FUNCTION
           jmp  func_09H                ;do it again
```

0AH
Buffered Keyboard Input

Call	AH = 0AH DS:DX Input buffer
Return	None

Remarks

DX must contain the offset (from the segment address in DS) of an input buffer of the following form:

Byte	Contents
1	Maximum number of characters in buffer, including the CR (you must set this value).
2	Actual number of characters typed, not counting the CR (the function sets this value).
3-n	Buffer; must be at least as long as the number in byte 1.

This function waits for characters to be typed. Characters are read from the keyboard and placed in the buffer beginning at the third byte until **CR** is pressed. If the buffer fills to one less than the maximum, additional characters typed are ignored and ASCII 7 (BEL) is sent to the display until **CR** is pressed. The string can be edited as it is being entered. If **CTRL C** is typed, Interrupt 23H is issued.

The second byte of the buffer is set to the number of characters entered (not counting the **CR**).

Macro

```
get_string macro limit,string
   mov   dx,offset string
   mov   string,limit
   mov   ah,0AH
   int   21H
   endm
```

Example The following program gets a 16-byte (maximum) string from the keyboard and fills a 24-line by 80-character screen with it:

```
buffer            label      byte
max_length        db         ?                    ;maximum length
chars_entered     db         ?                    ;number of chars.
string            db         17 dup (?)           ;16 chars + CR
strings_per_line  dw         0                    ;how many strings
                                                  ;fit on line
crlf              db         13,10,"$"
                  .
                  .
func_0AH:         get_string 17,buffer            ;THIS FUNCTION
                  xor        bx,bx                ;so byte can be
                                                  ;used as index
                  mov        bl,chars_entered     ;get string length
                  mov        buffer[bx+2],"$"     ;see Function 09H
                  mov        al,50H               ;columns per line
                  cbw
                  div        chars_entered        ;times string fits
                                                  ;on line
                  xor        ah,ah                ;clear remainder
                  mov        strings_per_line,ax  ;save col. counter
                  mov        cx,24                ;row counter
display_screen:   push cx                         ;save it
                  mov        cx,strings_per_line  ;get col. counter
display_line:     display    string               ;see Function 09H
                  loop       display_line
                  display    crlf                 ;see Function 09H
                  pop        cx                   ;get line counter
                  loop       display_screen       ;display 1 more line
```

0BH
Check Keyboard Status

Call AH = 0BH

Return AL
FFH = characters in type-ahead buffer
0 = no characters in type-ahead buffer

Remarks Checks whether there are characters in the type-ahead buffer. If so, AL returns FFH; if not, AL returns 0. If **CTRL C** is in the buffer, Interrupt 23H is executed.

Macro

```
check_kbd_status macro
                 mov   ah,0BH
                 int   21H
                 endm
```

Example The following program fragment continuously displays the time until any key is pressed.

```
time        db   "00:00:00.00",13,10,"$"
ten         db   10
            .
            .
func_0BH:   get_time                 ;see Function 2CH
            convert ch,ten,time      ;see end of chapter
            convert cl,ten,time[3]   ;see end of chapter
            convert dh,ten,time[6]   ;see end of chapter
            convert dl,ten,time[9]   ;see end of chapter
            display time             ;see Function 09H
            check_kbd_status         ;THIS FUNCTION
            cmp   al,0FFH            ;has a key been typed
            je    all_done           ;yes, go home
            jmp   func_0BH           ;no, keep displaying
                                     ;time
all_done:   .
            .
```

Call

AH = 0CH
AL
1, 6, 7, 8, or 0AH = The corresponding function is called. Any other value = return from function.

Return

AL
0 = Type-ahead buffer was flushed; no other processing performed.

Remarks

The keyboard type-ahead buffer is emptied. Further processing depends on the value in AL when the function is called:

1, 6, 7, 8, or A:
The corresponding MS-DOS function is executed.

Any other value:
No further processing; AL returns 0.

Macro

```
flush_and_read_kbd  macro switch
                    mov   al,switch
                    mov   ah,0CH
                    int   21H
                    endm
```

Example

The following program both displays and prints characters as they are typed. If **CR** is pressed, the program sends Carriage Return-Line Feed to both the display and the printer.

```
func_0CH:  flush_and_read_kbd 1       ;THIS FUNCTION
           print_char     al          ;see Function 05H
           cmp            al,0DH      ;is it a CR?
           jne            func_0CH    ;no, print it
           print_char     10          ;see Function 05H
           display_char   10          ;see Function 02H
           print_char     13          ;see Function 05H
           display_char   13          ;see Function 02H
           jmp            func_0CH    ;get another character
```

0DH
Disk Reset

Call AH = 0DH

Return None

Remarks

Function 0DH is used to ensure that the internal buffer cache matches the disks in the drives. If buffers have been modified, but not yet written to disk, this function writes them out and marks all buffers in the internal cache as free.

Function 0DH flushes (frees) all file buffers. It does not update directory entries; you must close files that have changed to update their directory entries (see Function 10H, Close File). This function need not be called before a disk change if all files that changed were closed. It is generally used to force a known state of the system; **CTRL C** interrupt handlers should call this function.

Macro

```
disk_reset macro disk
  mov   ah,0DH
  int   21H
  endm
```

Example

```
  mov   ah,0DH
  int   21H
;There are no errors returned by this call.
```

0EH
Select Disk

Call

AH = 0EH
DL
Drive number
(0 = A:, 1 = B:, etc.)

Return

AL
Number of logical drives

Remarks

The drive specified in DL (0 = A:, 1 = B:, etc.) is selected as the default disk. The number of drives is returned in AL.

Macro

```
select_disk macro disk
    mov   dl,disk[-65]    ;ASCII offset
    mov   ah,0EH
    int   21H
    endm
```

Example

The following program fragment selects the drive not currently selected in a 2-drive system:

```
func_0EH:   current_disk      ;see Function 19H
            cmp  al,00H       ;drive A: selected?
            je   select_b     ;yes, select B
            select_disk "A"   ;THIS FUNCTION
            jmp  continue
select_b:   select_disk "B"   ;THIS FUNCTION
continue:   .
            .
```

0FH
Open File

Call

AH = 0FH
DS:DX
Unopened FCB

Return

AL
0 = Directory entry found
FFH = No directory entry found

Remarks

DX must contain the offset (from the segment address in DS) of an unopened File Control Block (FCB). The disk directory is searched for the named file.

If a directory entry for the file is found, AL returns 0 and the FCB is filled as follows:

- If the drive code was 0 (default disk), it is changed to the actual disk used (1 = A:, 2 = B:, etc.). This lets you change the default disk without interfering with subsequent operations on this file.
- The Current Block field (offset 0CH) is set to zero.
- The Record Size (offset 0EH) is set to the system default of 128.
- The File Size (offset 10H), Date of Last Write (offset 14H), and Time of Last Write (offset 16H) are set from the directory entry.

Before performing a sequential disk operation on the file, you must set the Current Record field (offset 20H). Before performing a random disk operation on the file, you must set the Relative Record field (offset 21H). If the default record size (128 bytes) is not correct, set it to the correct length.

If a directory entry for the file is not found, AL returns FFH.

Macro

```
open macro fcb
  mov   dx,offset fcb
  mov   ah,0FH
  int   21H
  endm
```

Example

The following program prints the file named TEXTFILE. ASC that is on the disk in drive B:. If a partial record is in the buffer at end-of-file, the routine that prints the partial record prints characters until it encounters an end-of-file mark (ASCII 26, or **CTRL Z**):

```
fcb          db     2,"TEXTFILEASC"
             db     25 dup (?)
buffer       db     128 dup (?)
             .
             .
func_0FH:    set_dta buffer          ;see Function 1AH
             open fcb                ;THIS FUNCTION
read_line:   read_seq fcb            ;see Function 14H
             cmp    al,01H           ;end of file?
             je     all_done         ;yes, go home
             cmp    al,00H           ;more to come?
             jg     check_more       ;no, check for partial
                                     ;record
             mov    cx,128           ;yes, print the buffer
             xor    si,si            ;set index to 0
print_it:    print_char buffer[si]   ;see Function 05H
             inc    si               ;bump index
             loop   print_it         ;print next character
             jmp    read_line        ;read another record
check_more:  cmp    al,03H           ;part. record to print?
             jne    all_done         ;no
```

```
find_eof:   mov    cx,128           ;yes, print it
            xor    si,si            ;set index to 0
            cmp    buffer[si],26    ;end-of-file mark?
            je     all_done         ;yes
            print_char buffer[si]   ;see Function 05H
            inc    si               ;bump index to next
                                    ;character
            loop   find_eof
all_done:   close  fcb              ;see Function 10H
```

10H
Close File

Call

AH = 10H
DS:DX
Opened FCB

Return

AL
0 = Directory entry found
FFH = No directory entry found

Remarks

DX must contain the offset (to the segment address in DS) of an opened FCB. The disk directory is searched for the file named in the FCB. This function must be called after a file is changed to update the directory entry.

If a directory entry for the file is found, the entry is compared with the corresponding entries in the FCB. The directory entry is updated, if necessary, to match the FCB, and AL returns 0.

If a directory entry for the file is not found, AL returns FFH.

Macro

```
close macro fcb
  mov   dx,offset fcb
  mov   ah,10H
  int   21H
  endm
```

Example

The following program checks the first byte of the file named MOD1.BAS in drive B: to see if it is FFH, and prints a message if it is:

```
message     db   "Not saved in ASCII format",13,10,"$"
fcb         db   2,"MOD1    BAS"
            db   25 dup (?)
buffer      db   128 dup (?)
            .
            .
func_10H:   set_dta buffer       ;see Function 1AH
            open    fcb          ;see Function 0FH
            read_seq fcb         ;see Function 14H
            cmp     buffer,0FFH  ;is first byte FFH?
            jne     all_done     ;no
            display message      ;see Function 09H
all_done    close   fcb          ;THIS FUNCTION
```

11H
Search for First Entry

Call

AH = 11H
DS:DX
  Unopened FCB

Return

AL
  0 = Directory entry found
  FFH = No directory entry found

Remarks

DX must contain the offset (from the segment address in DS) of an unopened FCB. The disk directory is searched for the first matching name. The name can have the ? wild card character to match any character. To search for hidden or system files, DX must point to the first byte of the extended FCB prefix.

If a directory entry for the filename in the FCB is found, AL returns 0 and an unopened FCB of the same type (normal or extended) is created at the Disk Transfer Address.

If a directory entry for the filename in the FCB is not found, AL returns FFH.

Note

If an extended FCB is used, the following search pattern is used:

- If the FCB attribute is zero, only normal file entries are found. Entries for volume label, subdirectories, hidden, and system files will not be returned.

- If the attribute field is set for hidden or system files, or directory entries, it is to be considered as an inclusive search. All normal file entries plus all entries matching the specified attributes are returned. To look at all directory entries except the volume label, the attribute byte may be set to hidden + system + directory (all 3 bits on).

- If the attribute field is set for the volume label, it is considered an exclusive search, and only the volume label entry is returned.

Macro

```
search_first macro fcb
   mov   dx,offset fcb
   mov   ah,11H
   int   21H
   endm
```

Example

The following program verifies the existence of a file named REPORT. ASM on the disk in drive B::

```
yes          db   "FILE EXISTS.$"
no           db   "FILE DOES NOT EXIST.$"
fcb          db   2,"REPORT  ASM"
             db   25 dup (?)
buffer       db   128 dup (?)
crlf         db   13,10 "$"
             .
func_11H:    set_dta buffer      ;see Function 1AH
             search_first fcb    ;THIS FUNCTION
             cmp    al,0FFH      ;directory entry found?
             je     not_there    ;no
             display yes         ;see Function 09H
             jmp    continue
not_there:   display no          ;see Function 09H
continue:    display crlf        ;see Function 09H
             .
             .
```

12H
Search for Next Entry

Call

AH = 12H
DS:DX
  Unopened FCB

Return

AL
  0 = Directory entry found
  FFH = No directory entry found

Remarks

DX must contain the offset (from the segment address in DS) of an FCB previously specified in a call to Function 11H (Search for First Entry). Function 12H is used after Function 11H to find additional directory entries that match a filename that contains wild card characters. The disk directory is searched for the next matching name.

If a directory entry for the filename in the FCB is found, AL returns 0 and an unopened FCB of the same type (normal or extended) is created at the Disk Transfer Address.

If a directory entry for the filename in the FCB is not found, AL returns FFH.

Macro

```
search_next macro fcb
  mov   dx,offset fcb
  mov   ah,12H
  int   21H
  endm
```

Example The following program displays the number of files on the disk in drive B:

```
message     db   "No files",10,13,"$"
files       db   0
ten         db   10
fcb         db   2,"???????????"
            db   25dup (?)
buffer      db   128 dup (?)
            .
            .
func_12H:   set_dta buffer              ;see Function 1AH
            search_first fcb            ;see Function 11H
            cmp   al,0FFH               ;directory entry found?
            je    all_done              ;no, no files on disk
            inc   files                 ;yes, increment file
                                        ;counter
search_dir: search_next fcb             ;THIS FUNCTION
            cmp   al,0FFH               ;directory entry found?
            je    done                  ;no
            inc   files                 ;yes, increment file
                                        ;counter
            jmp   search_dir            ;check again
done:       convert files,ten,message   ;see end of chapter
all_done:   display message             ;see Function 09H
```

13H
Delete File

Call

AH = 13H
DS:DX
  Unopened FCB

Return

AL
  0 = Directory entry found
  FFH = No directory entry found

Remarks

DX must contain the offset (from the segment address in DS) of an unopened FCB. The directory is searched for a matching filename. The filename in the FCB can contain the ? wild card character to match any character.

If a matching directory entry is found, it is deleted from the directory. If the ? wild card character is used in the filename, all matching directory entries are deleted. AL returns 0.

If no matching directory entry is found, AL returns FFH.

Macro

```
delete macro fcb
  mov  dx,offset fcb
  mov  ah,13H
  int  21H
  endm
```

Example The following deletes each file on the disk in drive B: that was last written before June 30, 1984:

```
year        dw   1984
month       db   6
day         db   30
files       db   0
ten         db   10
message     db   "NO FILES DELETED.",13,10,"$"
                                          ;see Function 09H for
                                          ;explanation of $
fcb         db   2,"???????????"
            db   25 dup(?)

buffer      db   128 dup (?)
            .
            .
func_13H:   set_dta  buffer               ;see Function 1AH
            search_first fcb              ;see Function 11H
            cmp    al,0FFH                ;directory entry found?
            je     all_done               ;no, no files on disk

compare:    convert_date buffer           ;see end of chapter
            cmp    cx,year                ;next several lines
            jg     next                   ;check date in directory
            cmp    dl,month               ;entry against date
            jg     next                   ;above & check next file
            cmp    dh,day                 ;if date in directory
            jge    next                   ;entry isn't earlier.
            delete buffer                 ;THIS FUNCTION
            inc    files                  ;bump deleted-files
                                          ;counter
next:       search_next fcb               ;see Function 12H
            cmp    al,00H                 ;directory entry found?
            je     compare                ;yes, check date
            cmp    files,0                ;any files deleted?
            je     all_done               ;no, display NO FILES
```

```
                                                ;message.
                convert files,ten,message       ;see end of chapter
    all_done:   display message                 ;see Function 09H
```

14H Sequential Read

Call

AH = 14H
DS:DX
Opened FCB

Return

AL
0 = Read completed successfully
1 = EOF
2 = DTA too small
3 = EOF, partial record

Remarks

DX must contain the offset (from the segment address in DS) of an opened FCB. The record pointed to by the current block (offset 0CH) and Current Record (offset 20H) fields is loaded at the Disk Transfer Address, then the Current Record and, if necessary, the Current Block fields are incremented.

The record size is set to the value at offset 0EH in the FCB.

AL returns a code that describes the processing:

Code	Meaning
0	Read completed successfully.
1	End-of-file, no data in the record.
2	Not enough room at the Disk Transfer Address to read one record; read canceled.
3	End-of-file; a partial record was read and padded to the record length with zeros.

Macro

```
read_seq macro fcb
   mov   dx,offset fcb
   mov   ah,14H
   int   21H
   endm
```

Example

The following program displays the file named TEXTFILE.ASC that is on the disk in drive B:; its function is similar to the MS-DOS TYPE command. If a partial record is in the buffer at end of file, the routine that displays the partial record displays characters until it encounters an end-of-file mark (ASCII 26, or **CTRL Z**):

```
fcb          db   2,"TEXTFILEASC"
             db   25 dup (?)
buffer       db   128 dup (?),"$"
             .
             .
func_14H:    set_dta buffer           ;see Function 1AH
             open   fcb               ;see Function 0FH
read_line:   read_seq fcb             ;THIS FUNCTION
             cmp    al,01H            ;end-of-file?
             je     all_done          ;yes
             cmp    al,00H            ;more to come?
             jg     check_more        ;no, check for partial record
             display buffer           ;see Function 09H
             jmp    read_line         ;get another record
check_more:  cmp    al,03H            ;partial record in buffer?
             jne    all_done          ;no, go home
             xor    si,si             ;set index to 0
find_eof:    cmp    buffer[si],26     ;is character EOF?
             je     all_done          ;yes, no more to display
             display_char buffer[si]  ;see Function 02H
             inc    si                ;bump index to next
```

```
                                          ;character
                jmp     find_eof          ;check next character
all_done:       close   fcb               ;see Function 10H
```

15H
Sequential Write

Call

AH = 15H
DS:DX
Opened FCB

Return

AL
00H = Write completed successfully
01H = Disk full
02H = DTA too small

Remarks

DX must contain the offset (from the segment address in DS) of an opened FCB. The record pointed to by Current Block (offset 0CH) and Current Record (offset 20H) fields is written from the Disk Transfer Address, then the fields are incremented as necessary.

The record size is set to the value at offset 0EH in the FCB. If the Record Size is less than a sector, the data at the Disk Transfer Address is written to a buffer; the buffer is written to disk when it contains a full sector of data, or the file is closed, or a Reset Disk system call (Function 0DH) is issued.

AL returns a code that describes the processing:

Code	Meaning
0	transfer completed successfully
1	disk full; write canceled
2	write canceled; the area beginning at the Disk Transfer Address is too small to hold a record of data without overflowing or wrapping around a segment boundary.

Macro

```
write_seq macro fcb
  mov   dx,offset fcb
  mov   ah,15H
  int   21H
  endm
```

Example

The following program creates a file named DIR.TMP on the disk in drive B: that contains the disk number (0 = A:, 1 = B:, etc.) and filename from each directory entry on the disk:

```
record_size  equ  14                          ;offset of Record Size
                                              ;field in FCB
             .
             .
fcb1         db   2,"DIR  TMP"
             db   25 dup (?)
fcb2         db   2,"???????????"
             db   25 dup (?)
buffer       db   128 dup (?)
             .
             .
func_15H:    set_dta       buffer     ;see Function 1AH
             search_first fcb2        ;see Function 11H
             cmp           al,0FFH    ;directory entry found?
             je            all_done   ;no, no files on disk
             create        fcb1       ;see Function 16H
             mov           fcb1[record_size],12
                                      ;set record size to 12
write_it:    write_seq     fcb1       ;THIS FUNCTION
             search_next   fcb2       ;see Function 12H
             cmp           al,0FFH    ;directory entry found?
             je            all_done   ;no, go home
             jmp           write_it   ;yes, write the record
all_done:    close         fcb1       ;see Function 10H
```

16H
Create File

Call

AH = 16H
DS:DX
Unopened FCB

Return

AL
00H = Empty directory entry found
FFH = No empty entry directory available

Remarks

DX must contain the offset (from the segment address in DS) of an unopened FCB. The directory is searched for an empty entry or an existing entry for the specified filename.

If an empty directory entry is found, it is initialized to a zero-length file, the Open File system call (Function 0FH) is called, and AL returns 0. You can create a hidden file by using an extended FCB with the attribute byte (offset FCB-1) set to 2.

If an entry is found for the specified filename, all data in the file is released, making a zero-length file, and the Open File system call (Function 0FH) is issued for the filename (in other words, if you try to create a file that already exists, the existing file is erased, and a new, empty file is created).

If an empty directory entry is not found and there is no entry for the specified filename, AL returns FFH.

Macro

```
create macro fcb
   mov   dx,offset fcb
   mov   ah,16H
   int   21H
   endm
```

Example

The following program creates a file named DIR.TMP on the disk in drive B: that contains the disk number (0 = A:, 1 = B:, etc.) and filename from each directory entry on the disk:

```
record_size  equu  14                        ;offset of Record Size
                                             ;field of FCB
             .
             .
fcb1         db   2,"DIR  TMP"
             db   25 dup (?)
fcb2         db   2,"???????????"
             db   25 dup (?)
buffer       db   128 dup (?)
             .
             .
func_16H:    set_dta        buffer     ;see Function 1AH
             search_first   fcb2       ;see Function 11H
             cmp            al,0FFH    ;directory entry found?
             je             all_done   ;no, no files on disk
             create         fcb1       ;THIS FUNCTION
             mov            fcb1[record_size],12
                                       ;set record size to 12
write_it:    write_seq      fcb1       ;see Function 15H
             search_next    fcb2       ;see Function 12H
             cmp            al,0FFH    ;directory entry found?
             je             all_done   ;no, go home
             jmp            write_it   ;yes, write the record
all_done:    close          fcb1       ;see Function 10H
```

17H
Rename File

Call

AH = 17H
DS:DX
Modified FCB

Return

AL
00H = Directory entry found
FFH = No directory entry found or destination already exists

Remarks

DX must contain the offset (from the segment address in DS) of an FCB with the drive number and filename filled in, followed by a second filename at offset 11H. The disk directory is searched for an entry that matches the first filename, which can contain the ? wild card character.

If a matching directory entry is found, the filename in the directory entry is changed to match the second filename in the modified FCB (the two filenames cannot be the same name). If the ? wild card character is used in the second filename, the corresponding characters in the filename of the directory entry are not changed. AL returns 0.

If a matching directory entry is not found or an entry is found for the second filename, AL returns FFH.

Macro

```
rename macro special_fcb
   mov   dx,offset special_fcb
   mov   ah,17H
   int   21H
   endm
```

Example

The following program prompts for the name of a file and a new name, then renames the file:

```
fcb         db   37 dup (?)
prompt1     db   "Filename: $"
prompt2     db   "New name: $"
reply       db   17 dup(?)
crlf        db   13,10,"$"
                 .
                 .
func_17H:   display prompt1       ;see Function 09H
            get_string 15,reply   ;see Function 0AH
            display   crlf        ;see Function 09H
            parse     reply[2],fcb ;see Function 29H
            display   prompt2     ;see Function 09H
            get_string 15,reply   ;see Function 0AH
            display   crlf        ;see Function 09H
            parse     reply[2],fcb[16]
                                  ;see Function 29H
            rename    fcb         ;THIS FUNCTION
```

19H
Current Disk

Call AH = 19H

Return AL
Currently selected drive
(0 = A:, 1 = B:, etc.)

Macro

```
current_disk macro
   mov   ah,19H
   int   21H
   endm
```

Example The following program displays the currently selected (default) drive:

```
message    db   "Current disk is $"   ;see Function 09H
                                      ;for explanation of $
crlf       db   13,10,"$"
           .
           .
func_19H:  display message            ;see Function 09H
           current_disk               ;THIS FUNCTION
           add   al, 41H              ;ASCII offset
           display_char al            ;see function 02H
           display crlf               ;see function 09H
```

1AH
Set Disk Transfer Address

Call

AH = 1AH
DS:DX
Disk Transfer Address

Return

None

Remarks

DX must contain the offset (from the segment address in DS) of the Disk Transfer Address. Disk transfers cannot wrap around from the end of the segment to the beginning, nor can they overflow into another segment.

Note:

If you do not set the Disk Transfer Address, MS-DOS defaults to offset 80H in the Program Segment Prefix.

The size of the buffer that the DTA points to must be greater than or equal to the record size at open file time.

Macro

```
set_dta macro buffer
  mov   dx,offset buffer
  mov   ah,1AH
  int   21H
  endm
```

Example The following program prompts for a letter, converts the letter to its alphabetic sequence (A = 1, B = 2, etc.), then reads and displays the corresponding record from a file named ALPHABET.-DAT on the disk in drive B:. The file contains 26 records; each record is 28 bytes long:

```
record_size      equ  14         ;offset of Record Size
                                 ;field of FCB
relative_record  equ  33         ;offset of Relative Record
                                 ;field of FCB
                  .
                  .
    fcb           db  2,"ALPHABETDAT"
                  db  25 dup (?)
    buffer        db  34 dup(?),"$"
    prompt        db  "Enter letter: $"
    crlf          db  13,10,"$"
                  .
                  .
    func_1AH:     set_dta buffer            ;THIS FUNCTION
                  open    fcb               ;see Function 0FH
                  mov     fcb[record_size],28 ;set record size
    get_char:     display prompt            ;see Function 09H
                  read_kbd_and_echo         ;see Function 01H
                  cmp     al,0DH            ;just a CR?
                  je      all_done          ;yes, go home
                  sub     al,41H            ;convert ASCII
                                            ;code to record #
                  mov     fcb[relative_record],al
                                            ;set relative record
                  display crlf              ;see Function 09H
                  read_ran fcb              ;see Function 21H
                  display buffer            ;see Function 09H
                  display crlf              ;see Function 09H
                  jmp     get_char          ;get another character
    all_done:     close   fcb               ;see Function 10H
```

Call

AH = 21H
DS:DX
  Opened FCB

Return

AL
  00H = Read completed successfully
  01H = EOF
  02H = DTA too small
  03H = EOF, partial record

Remarks

DX must contain the offset (from the segment address in DS) of an opened FCB. The Current Block (offset 0CH) and Current Record (offset 20H) fields are set to agree with the Relative Record field (offset 21H), then the record addressed by these fields is loaded at the Disk Transfer Address.

AL returns a code that describes the processing:

Code	Meaning
0	read completed successfully
1	End-of-file; no data in the record
2	not enough room at the Disk Transfer Address to read one record; read canceled
3	End-of-file; a partial record was read and padded to the record length with zeros.

Macro

```
read_ran macro fcb
  mov   dx,offset fcb
  mov   ah,21H
  int   21H
  endm
```

Example The following program prompts for a letter, converts the letter to its alphabetic sequence (A = 1, B = 2, etc.), then reads and displays the corresponding record from a file named ALPHABET.-DAT on the disk in drive B:. The file contains 26 records; each record is 28 bytes long:

```
record_size      equu  14          ;offset of Record Size
                                   ;field of FCB
relative_record  equ   33          ;offset of Relative Record
                                   ;field of FCB
                 .
                 .
fcb              db   2,"ALPHABETDAT"
                 db   25 dup (?)
buffer           db   34 dup(?),"$"
prompt           db   "Enter letter: $"
crlf             db   13,10,"$"
                 .
                 .
func_21H:        set_dta buffer            ;see Function 1AH
                 open    fcb               ;see Function 0FH
                 mov     fcb[record_size],28 ;set record size
get_char:        display prompt            ;see Function 09H
                 read_kbd_and_echo         ;see Function 01H
                 cmp     al,0DH            ;just a CR?
                 je      all_done          ;yes, go home
                 sub     al,41H            ;convert ASCII
                                           ;code to record #
                 mov     fcb[relative_record],al
                                           ;set relative record
                 display crlf              ;see Function 09H
                 read_ran fcb              ;THIS FUNCTION
                 display buffer            ;see Function 09H
                 display crlf              ;see Function 09H
                 jmp     get_char          ;get another character
all_done:        close   fcb               ;see Function 10H
```

Call

AH = 22H
DS:DX
  Opened FCB

Return

AL
  00H = Write completed successfully
  01H = Disk full
  02H = DTA too small

Remarks

DX must contain the offset from the segment address in DS of an opened FCB. The Current Block (offset 0CH) and Current Record (offset 20H) fields are set to agree with the Relative Record field (offset 21H), then the record addressed by these fields is written from the Disk Transfer Address. If the record size is smaller than a sector (512 bytes), the records are buffered until a sector is ready to write.

AL returns a code that describes the processing:

Code	Meaning
0	Write completed successfully
1	disk is full
2	write canceled; the area beginning at the Disk Transfer Area is too small to hold a record of data without overflowing or wrapping around a segment boundary.

Macro

```
write_ran macro fcb
  mov   dx,offset fcb
  mov   ah,22H
  int   21H
  endm
```

Example The following program prompts for a letter, converts the letter to its alphabetic sequence (A = 1, B = 2, etc.), then reads and displays the corresponding record from a file named ALPHABET.-DAT on the disk in drive B:. After displaying the record, it prompts the user to enter a changed record. If the user types a new record, it is written to the file; if the user just presses **CR**, the record is not replaced. The file contains 26 records; each record is 28 bytes long:

```
record_size     equ  14          ;offset of Record Size
                                 ;field of FCB
relative_record equ  33          ;offset of Relative Record
                                 ;field of FCB
                .
                .
fcb             db   2,"ALPHABETDAT"
                db   25 dup (?)
buffer          db   28 dup(?),13,10,"$"
prompt1         db   "Enter letter: $"
prompt2         db   "New record(CR for no change): $"
crlf            db   13,10,"$"
reply           db   30 dup (32)
blanks          db   28 dup (32)
                .
                .
func_22H:       set_dta buffer              ;see Function 1AH
                open    fcb                 ;see Function 0FH
                mov     fcb[record_size],28 ;set record size
get_char:       display prompt1             ;see Function 09H
                read_kbd_and_echo           ;see Function 01H
                cmp     al,0DH              ;just a CR?
                je      all_done            ;yes, go home
                sub     al,41H              ;convert ASCII
                                            ;code to record #
                mov     fcb[relative_record],al
                                            ;set relative record
```

```
                display crlf                   ;see Function 09H
                read_ran fcb                   ;THIS FUNCTION
                display buffer                 ;see Function 09H
                display crlf                   ;see Function 09H
                display prompt2                ;see Function 09H
                get_string 29, reply           ;see Function 09H
                display crlf                   ;see Function 09H
                cmp     reply[1],0             ;was anything typed
                                               ;besides cr?
                je      get_char               ;no
                                               ;get another character
                xor     bx,bx                  ;to load a byte
                mov     bl,reply[1]            ;use reply length as
                                               ;counter
                move_string blanks, buffer, 28
                                               ;see chapter end
                move_string reply[2], buffer, bx
                                               ;see chapter end
                write_ran fcb                  ;THIS FUNCTION
                jmp     get_char               ;get another character
all_done:       close   fcb                    ;see Function 10H
```

23H
File Size

Call

AH = 23H
DS:DX
Unopened FCB

Return

AL
00H = Directory entry found
FFH = No directory entry found

Remarks

DX must contain the offset (from the segment address in DS) of an unopened FCB. You must set the Record Size field (offset 0EH) to the proper value before calling this function. The disk directory is searched for the first matching entry.

If a matching directory entry is found, the Relative Record field (offset 21H) is set to the number of records in the file, calculated from the total file size in the directory entry (offset 10H) and the Record Size field of the FCB (offset 0EH). AL returns 00.

If no matching directory entry is found, AL returns FFH.

Note

If the value of the Record Size field of the FCB (offset 0EH) doesn't match the actual number of characters in a record, this function may not return the correct file size. If the default record size (128) is not correct, you must set the Record Size field to the correct value before using this function.

Macro

```
file_size macro fcb
   mov   dx,offset fcb
   mov   ah,23H
   int   21H
   endm
```

Example

The following program prompts for the name of a file, opens the file to set the Record Size field of the FCB to 80H, issues a File Size system call, and displays the file size and number of records in hexadecimal:

```
fcb          db             37 dup(?)
prompt       db             "File name: $"
msg1         db             "Record length:  ",13,10,"$"
msg2         db             "Records:  ",13,10,"$"
crlf         db             13,10,"$"
reply        db             17 dup(?)
sixteen      db             16
             .
             .
func_23H:    display prompt               ;see Function 09H
             get_string 17,reply          ;see Function 0AH
             cmp            reply[1],0    ;just a CR?
             jne            get_length    ;no, keep going
             jmp            all_done      ;yes, go home
get_length:  display crlf                 ;see Function 09H
             parse          reply[2],fcb  ;see Function 29H
             open           fcb           ;see Function 0FH
             file_size fcb                ;THIS FUNCTION
             mov            si,33         ;offset to Relative
                                          ;Record field
             mov            di,9          ;reply in msg2
convert_it:  cmp            fcb[si],0     ;digit to convert?
             je             show_it       ;no, prepare message
```

```
           convert fcb[si],sixteen,msg2[di]
           inc        si           ;bump n-o-r index
           inc        di           ;bump message index
           jmp        convert_it   ;check for a digit
show_it:   convert fcb[14],sixteen,msg1[15]
           display msg1            ;see Function 09H
           display msg2            ;see Function 09H
           jmp        func23H      ;get a filename
all_done:  close      fcb          ;see Function 10H
```

24H
Set Relative Record

Call

AH = 24H
DS:DX
  Opened FCB

Return

None

Remarks

DX must contain the offset (from the segment address in DS) of an opened FCB. The Relative Record field (offset 21H) is set to the same file address as the Current Block (offset 0CH) and Current Record (offset 20H) fields.

Macro

```
set_relative_record macro fcb
   mov   dx,offset fcb
   mov   ah,24H
   int   21H
   endm
```

Example

The following program copies a file using the Random Block Read and Random Block Write system calls. It speeds the copy by setting the record length equal to the file size and the record count to 1, and using a buffer of 32K bytes. It positions the file pointer by setting the Current Record field (offset 20H) to 0 and using Set Relative Record to make the Relative Record field (offset 21H) point to the same record as the combination of the Current Block (offset 0CH) and Current Record (offset 20H) fields:

```
current_record equ  32                      ;offset of Current Record
                                            ;field of FCB
fsize          equu 16                      ;offset of File Size
                                            ;field of FCB
               .
               .
fcb            db   37 dup (?)
filename       db   17 dup(?)
prompt1        db   "File to copy: $"       ;see Function 09H for
prompt2        db   "Name of copy: $"       ;explanation of $
crlf           db   13,10,"$"

file_length    dw   ?
buffer         db   32767 dup(?)
               .
               .
func_24H:      set_dta      buffer          ;see Function 1AH
               display prompt1              ;see Function 09H
               get_string 15,filename       ;see Function 0AH
               display crlf                 ;see Function 09H
               parse filename[2],fcb        ;see Function 29H
               open         fcb             ;see Function 0FH
               mov fcb[current_record],0    ;set Current Record
                                            ;field
               set_relative_record fcb      ;THIS FUNCTION
               mov ax,word ptr fcb[fsize]   ;get file size
               mov          file_length,ax  ;save it for
                                            ;ran_block_write
               ran_block_read fcb,1,ax      ;see Function 27H
               display      prompt2         ;see Function 09H
               get_string 15,filename       ;see Function 0AH
               display      crlf            ;see Function 09H
               parse filename[2],fcb        ;see Function 29H
               set_relative_record fcb      ;THIS FUNCTION
```

```
mov            ax,file_length        ;get original file
                                     ;length
ran_block_write fcb,1,ax             ;see Function 28H
close          fcb                   ;see Function 10H
```

25H
Set Vector

Call

AH = 25H
AL
  Interrupt number
DS:DX
  Interrupt-handling routine

Return

None

Remarks

Function 25H should be used to set a particular interrupt vector. The operating system can then manage the interrupts on a per-process basis. Note that programs should never set interrupt vectors by writing them directly in the low memory vector table.

DX must contain the offset (to the segment address in DS) of an interrupt-handling routine. AL must contain the number of the interrupt handled by the routine. The address in the vector table for the specified interrupt is set to DS:DX.

Macro

```
set_vector macro interrupt, seg_addr,off_addr
   push  ds
   mov   ax,seg_addr
   mov   ds,ax
   mov   dx,off_addr
   mov   al,interrupt
   mov   ah,25H
   int   21H
   pop   ds
   endm
```

Example

```
lds   dx,intvector
mov   ah,25H
mov   al,intnumber
int   21H
;There are no errors returned
```

Call

AH = 27H
DS:DX
  Opened FCB
CX
  Number of blocks to read

Return

AL
  00H = Read completed successfully
  01H = EOF
  02H = End of segment
  03H = EOF, partial record
CX
  Number of blocks read

Remarks

DX must contain the offset (to the segment address in DS) of an opened FCB. CX must contain the number of records to read; if it contains 0, the function returns without reading any records (no operation). The specified number of records, calculated from the Record Size field (offset 0EH), is read starting at the record specified by the Relative Record field (offset 21H). The records are placed at the Disk Transfer Address.

AL returns a code that describes the processing:

Code	Meaning
0	Read completed successfully
1	End-of-file; no data in the record
2	not enough room at the Disk Transfer Address to read one record without overflowing a segment boundary; read cancelled
3	End-of-file; a partial record was read and padded to the record length with zeros

CX returns the number of records read; the Current Block (offset 0CH), Current Record (offset 20H), and Relative Record (offset 21H) fields are set to address the next record.

Macro

```
ran_block_read macro fcb,count,rec_size
               mov   dx,offset fcb
               mov   cx,count
               mov   word ptr fcb[14],rec_size
               mov   ah,27H
               int   21H
               endm
```

Example

The following program copies a file using the Random Block Read system call. It speeds the copy by specifying a record count of 1 and a record length equal to the file size, and using a buffer of 32K bytes; the file is read as a single record (compare to the sample program for Function 28H that specifies a record length of 1 and a record count equal to the file size):

```
current_record  equ  32                  ;offset of Current Record
                                         ;field
fsize           equ  16                  ;offset of File Size
                                         ;field
                .
                .
fcb             db   37 dup (?)
filename        db   17 dup(?)
prompt1         db   "File to copy: $"   ;see Function 09H for
prompt2         db   "Name of copy: $"   ;explanation of $
crlf            db   13,10,"$"

file_length     dw   ?
buffer          db   32767 dup(?)
                .
                .
func_27H:       set_dta buffer           ;see Function 1AH
```

```
display prompt1                ;see Function 09H
get_string 15,filename         ;see Function 0AH
display crlf                   ;see Function 09H
parse filename[2],fcb          ;see Function 29H
open    fcb                    ;see Function 0FH
mov fcb[current_record],0      ;set Current Record
                               ;field
set_relative_record fcb        ;see Function 24H
mov ax,word ptr fcb[fsize]     ;get file size
mov     file_length,ax         ;save it for
                               ;ran_block_write
ran_block_read fcb,1,ax        ;THIS FUNCTION
display prompt2                ;see Function 09H
get_string 15,filename         ;see Function 0AH
display crlf                   ;see Function 09H
parse filename[2],fcb          ;see Function 29H
create  fcb                    ;see Function 16H
mov fcb[current_record],0      ;set Current Record
                               ;field
set_relative_record fcb        ;see Function 24H
mov     ax,file_length         ;get original file
                               ;size
ran_block_write fcb,1,ax       ;see Function 28H
close   fcb
```

28H
Random Block Write

Call

AH = 28H
DS:DX
  Opened FCB
CX
  Number of blocks to write
  (0 = set File Size field)

Return

AL
  00H = Write completed successfully
  01H = Disk full
  02H = End of segment
CX
  Number of blocks written

Remarks

DX must contain the offset (to the segment address in DS) of an opened FCB; CX must contain either the number of records to write or 0. The specified number of records (calculated from the Record Size field, offset 0EH) is written from the Disk Transfer Address. The records are written to the file starting at the record specified in the Relative Record field (offset 21H) of the FCB. If CX is 0, no records are written, but the File Size field of the directory entry (offset 10H) is set to the number of records specified by the Relative Record field of the FCB (offset 21H); allocation units are allocated or released, as required.

AL returns a code that describes the processing:

Code	Meaning
0	Write completed successfully
1	Disk full. No records written.
2	Not enough room at the Disk Transfer Address to write one record without overflowing a segment boundary; write canceled.

CX returns the number of records written; the current block (offset 0CH), Current Record (offset 20H), and Relative Record (offset 21H) fields are set to address the next record.

Macro

```
ran_block_write macro fcb,count,rec_size
                mov   dx,offset fcb
                mov   cx,count
                mov   word ptr fcb[14],rec_size
                mov   ah,28H
                int   21H
                endm
```

Example

The following program copies a file using the Random Block Read and Random Block Write system calls. It copies by specifying a record count equal to the file size and a record length of 1, and using a buffer of 32K bytes; the file is copied with one disk access each to read and write (compare to the sample program of Function 27H, that specifies a record count of 1 and a record length equal to file size):

```
current_record equu  32                  ;offset of Current Record
                                         ;field
fsize          equ   16                  ;offset of File Size
                                         ;field
               .
               .
fcb            db   37 dup (?)
filename       db   17 dup(?)
prompt1        db   "File to copy: $"    ;see Function 09H for
prompt2        db   "Name of copy: $"    ;explanation of $
crlf           db   13,10,"$"

num_recs       dw   ?
buffer         db   32767 dup(?)
```

```
func_28H: set_dta   buffer                  ;see Function 1AH
          display   prompt1                 ;see Function 09H
          get_string 15,filename            ;see Function 0AH
          display   crlf                    ;see Function 09H
          parse     filename[2],fcb         ;see Function 29H
          open      fcb                     ;see Function 0FH
          mov       fcb[current_record],0   ;set Current Record
                                            ;field
          set_relative_record fcb           ;see Function 24H
          mov       ax,word ptr fcb[fsize]  ;get file size
          mov       num_recs, ax            ;save it for
                                            ;ran_block_write
          ran_block_read fcb,num_recs,1     ;see Function 27H
          display   prompt2                 ;see Function 09H
          get_string 15,filename            ;see Function 0AH
          display   crlf                    ;see Function 09H
          parse     filename[2],fcb         ;see Function 29H
          create    fcb                     ;see Function 16H
          mov       fcb[current_record],0   ;set Current Record
                                            ;field
          set_relative_record fcb           ;see Function 24H
          ran_block_write fcb,num_recs,1    ;THIS FUNCTION
          close     fcb                     ;see Function 10H
```

29H
Parse File Name

Call

AH = 29H
AL
  Controls parsing (see text)
DS:SI
  String to parse
ES:DI
  Unopened FCB

Return

AL
  00H = No wild card characters
  01H = Wild-card characters used
  FFH = Drive letter invalid
DS:SI
  First byte past string that was parsed
ES:DI
  Unopened FCB

Remarks

SI must contain the offset (to the segment address in DS) of a string (command line) to parse; DI must contain the offset (to the segment address in ES) of an unopened FCB. The string is parsed for a filename of the form d:filename.ext; if one is found, a corresponding unopened FCB is created at ES:DI.

Bits 0-3 of AL control the parsing and processing. Bits 4-7 are ignored:

Bit	Value	Meaning
0	0	All parsing stops if a file separator is encountered.
	1	Leading separators are ignored.
1	0	The drive number in the FCB is set to 0 (default drive) if the string does not contain a drive number.
	1	The drive number in the FCB is not changed if the string does not contain a drive number.
2	0	The filename in the FCB is set to 8 blanks if the string does not contain a filename.
	1	The filename in the FCB is not changed if the string does not contain a filename.
3	0	The extension in the FCB is set to 3 blanks if the string does not contain an extension.
	1	The extension in the FCB is not changed if the string does not contain an extension.

If the filename or extension includes an asterisk (*), all remaining characters in the name or extension are set to question mark (?).

Filename separators:

: . ; , = + / " [ ] \ < > | space tab

Filename terminators include all the filename separators plus any control character. A filename cannot contain a filename terminator; if one is encountered, parsing stops.

If the string contains a valid filename:

- AL returns 1 if the filename or extension contains a wild card character (* or ?); AL returns 0 if neither the filename nor extension contains a wild card character.
- DS:SI point to the first character following the string that was parsed.
- ES:DI point to the first byte of the unopened FCB.

If the drive letter is invalid, AL returns FFH. If the string does not contain a valid filename, ES:DI+1 points to a blank (ASCII 20H).

Macro

```
parse macro string, fcb
        mov   si, offset string
        mov   di, offset fcb
        push  es
        push  ds
        pop   es
        mov   al,0FH  ;bits 0, 1, 2, 3 on
        mov   ah,29H
        int   21H
        pop   es
        endm
```

Example

The following program verifies the existence of the file named in reply to the prompt:

```
fcb         db   37 dup (?)
prompt      db   "Filename: $"
reply       db   17 dup (?)
yes         db   "FILE EXISTS", 13,10,"$"
no          db   "FILE DOES NOT EXIST", 13,10,"$"
crlf        db   13,14,"$"
            .
            .
func_29H:   display       prompt         ;see Function 09H
            get_string    15,reply       ;see Function 0AH
            display       crlf           ;see Function 09H
            parse         reply[2],fcb   ;THIS FUNCTION
            search_first  fcb            ;see Function 11H
            cmp           al,0FFH        ;dir. entry found?
            je            not_there      ;no
            display       yes            ;see Function 09H
            jmp           continue
not_there:  display       no
continue:   .
            .
```

Call AH = 2AH

Return

CX
  Year (1980-2099)
DH
  Month (1-12)
DL
  Day (1-31)
AL
  Day of week (0 = Sunday, 6 = Saturday)

Remarks

This function returns the current date set in the operating system as binary numbers in CX and DX:

CX	Year (1980-2099)
DH	Month (1 = January, 2 = February, etc.)
DL	Day (1-31)
AL	Day of week (0 = Sunday, 1 = Monday, etc.)

Macro

```
get_date macro
     mov   ah,2AH
     int   21H
     endm
```

Example

The following program gets the date, increments the day, increments the month or year, if necessary, and sets the new date:

```
month      db        31,28,31,30,31,30,31,31,30,31,30,31
           .
           .
func_2AH:  get_date                 ;see above
           inc       dl             ;increment day
           xor       bx,bx          ;so BL can be used as index
           mov       bl,dh          ;move month to index register
           dec       bx             ;month table starts with 0
           cmp       dl,month[bx]   ;past end of month?
           jle       month_ok       ;no, set the new date
           mov       dl,1           ;yes; set day to 1
           inc       dh             ;and increment month
           cmp       dh,12          ;past end of year?
           jle       month_ok       ;no, set the new date
           mov       dh,1           ;yes, set the month to 1
           inc       cx             ;increment year
month_ok:  set_date cx,dh,dl        ;THIS FUNCTION
```

Call

AH = 2BH
CX
  Year (1980-2099)
DH
  Month (1-12)
DL
  Day (1-31)

Return

AL
  00H = Date was valid
  FFH = Date was invalid

Remarks

Registers CX and DX must contain a valid date in binary:

CX	Year (1980-2099)
DH	Month (1 = January, 2 = February, etc.)
DL	Day (1-31)

If the date is valid, the date is set and AL returns 0. If the date is not valid, the function is canceled and AL returns FFH.

Macro

```
set_date macro year,month,day
  mov   cx,year
  mov   dh,month
  mov   dl,day
  mov   ah,2BH
  int   21H
  endm
```

Example The following program gets the date, increments the day, increments the month or year, if necessary, and sets the new date:

```
month        db            31,28,31,30,31,30,31,31,30,31,30,31
             .
             .
func_2BH:    get_date                    ;see Function 2AH
             inc           dl            ;increment day
             xor           bx,bx         ;so BL can be used as index
             mov           bl,dh         ;move month to index register
             dec           bx            ;month table starts with 0
             cmp           dl,month[bx]  ;past end of month?
             jle           month_ok      ;no, set the new date
             mov           dl,1          ;yes, set day to 1
             inc           dh            ;and increment month
             cmp           dh,12         ;past end of year?
             jle           month_ok      ;no, set the new date
             mov           dh,1          ;yes, set the month to 1
             inc           cx            ;increment year
month_ok:    set date cx,dh,dl           ;THIS FUNCTION
```

Call AH = 2CH

Return

CH
  Hour (0-23)
CL
  Minutes (0-59)
DH
  Seconds (0-59)
DL
  Hundredths (0-99)

Remarks This function returns the current time set in the operating system as binary numbers in CX and DX:

CH	Hour (0-23)
CL	Minutes (0-59)
DH	Seconds (0-59)
DL	Hundredths of a second (0-99)

Macro

```
get_time  macro
  mov     ah,2CH
  int     21H
  endm
```

Example The following program continuously displays the time until any key is pressed:

```
time        db      "00:00:00.00",13,"$"
ten         db      10
                .
                .
func_2CH:   get_time                    ;THIS FUNCTION
            convert ch,ten,time         ;see end of chapter
            convert cl,ten,time[3]      ;see end of chapter
            convert dh,ten,time[6]      ;see end of chapter
            convert dl,ten,time[9]      ;see end of chapter
            display time                ;see Function 09H
            check_kbd_status            ;see Function 0BH
            cmp     al,0FFH             ;has a key been pressed?
            je      all_done            ;yes, terminate
            jmp     func_2CH            ;no, display time
all_done:   .
            .
```

Call

AH = 2DH
CH
  Hour (0-23)
CL
  Minutes (0-59)
DH
  Seconds (0-59)
DL
  Hundredths (0-99)

Return

AL
  00H = Time was valid
  FFH (255) = Time was invalid

Remarks

Registers CX and DX must contain a valid time in binary:

CH	Hour (0-23)
CL	Minutes (0-59)
DH	Seconds (0-59)
DL	Hundredths of a second (0-99)

If the time is valid, the time is set and AL returns 0. If the time is not valid, the function is canceled and AL returns FFH (255).

Macro

```
set_time macro hour,minutes,seconds,hundredths
   mov    ch,hour
   mov    cl,minutes
   mov    dh,seconds
   mov    dl,hundredths
   mov    ah,2DH
   int    21H
   endm
```

Example

The following program acts as a stopwatch. When a character is typed, it sets the system clock to zero and begins to continuously display the time. When a second character is typed the system stops updating the time display.

```
time        db              "00:00:00.00",13,"$"
ten         db              10
            .
            .
func_2DH:   dir_console_io  0FFH            ;see Function 06H
            jz              func_2DH        ;wait for keystroke
            set_time        0,0,0,0         ;THIS FUNCTION
read_clock: get_time                        ;see Function 2CH
            convert ch,ten,time             ;see end of chapter
            convert cl,ten,time[3]          ;see end of chapter
            convert dh,ten,time[6]          ;see end of chapter
            convert dl,ten,time[9]          ;see end of chapter
            display time                    ;see Function 09H
            dir_console_io 0FFH             ;THIS FUNCTION
            jz              read_clock      ;no char, keep updating
continue:   .
            .
```

Call

AH = 2EH
AL
  00H = Do not verify
  01H = Verify

Return

None

Remarks

AL must be either 1 (verify after each disk write) or 0 (write without verifying). MS-DOS checks this flag each time it writes to a disk.

The flag is normally off; you may wish to turn it on when writing critical data to disk. Because disk errors are rare and verification slows writing, you will probably want to leave it off at other times.

Macro

```
verify macro switch
   mov   al,switch
   mov   ah,2EH
   int   21H
   endm
```

Example

The following program copies the contents of a single-sided disk in drive A: to the disk in drive B:, verifying each write. It uses a buffer of 32K bytes:

```
on          equ     1
off         equ     0
                .
                .
prompt      db      "Source in A, target in B",13,10
            db      "Any key to start. $"
start       dw      0
buffer      db      64 dup (512 dup(?)) ;64 sectors
```

```
func_2EH:   display prompt                    ;see Function 09H
            read_kbd                          ;see Function 08H
            verify on                         ;THIS FUNCTION
            mov         cx,5                  ;copy 64 sectors
                                              ;5 times
copy:       push        cx                    ;save counter
            abs_disk_read 0,buffer,64,start   ;see Interrupt 25H
            abs_disk_write 1,buffer,64,start  ;see Interrupt 26H
            add         start,64              ;do next 64 sectors
            pop         cx                    ;restore counter
            loop        copy                  ;do it again
            verify off                        ;THIS FUNCTION
```

Call AH = 2FH

Return ES:BX
Points to Disk Transfer Address

Macro

```
;
get_dta macro
   mov   2h,2fh
   int   21h
   endm
```

30H
Get DOS Version Number

Call

AH = 30H

Return

AL
  Major version number
AH
  Minor version number
BH
  OEM number
BL:CX
  User number (24 bits)

Remarks

On return, AL.AH will be the two-part version designation; i.e., for MS-DOS 1.28, AL would be 1 and AH would be 28. For pre-1.28 DOS AL = 0. Note that version 1.1 is the same as 1.10, not the same as 1.01.

Macro

```
;
get_version_num macro
  mov   ah,30h
  int   21h
  endm
```

Call

AH = 31H
AL
  Exit code
DX
  Memory size, in paragraphs

Return

None

Remarks

This call terminates the current process and attempts to set the initial allocation block to a specific size in paragraphs. It will not free up any other allocation blocks belonging to that process. The exit code passed in AX is retrievable by the parent via Function 4DH.

This method is preferred over Interrupt 27H and has the advantage of allowing more than 64K to be kept.

Macro

```
;
keep_process macro exitcode,parasize
   mov   al,exitcode
   mov   dx,parasize
   mov   ah,31h
   int   21h
   endm
```

33H
<CTRL C> Check

Call

AH = 33H
AL
  Function
  00H = Request current state
  01H = Set state
DL (if setting state)
  00H = Off
  01H = On

Return

DL
  00H = Off
  01H = On

Remarks

MS-DOS ordinarily checks for a **CTRL C** on the controlling device only when doing function call operations 01H-0CH to that device. Function 33H allows the user to expand this checking to include any system call. For example, with the **CTRL C** trapping off, all disk I/O will proceed without interruption; with **CTRL C** trapping on, the **CTRL C** interrupt is given at the system call that initiates the disk operation.

Note

Programs that wish to use calls 06H or 07H to read **CTRL C**'s as data must ensure that the **CTRL C** check is off.

Error Returns

AL = FF

The function passed in AL was not in the range 0:1.

Macro

```
;
ctrl_c_check macro switch,val
   mov  dl,val
   mov  al,switch
   mov  ah,33h
   int  21h
   endm
```

35H
Get Interrupt Vector

Call

AH = 35H
AL
  Interrupt number

Return

ES:BX
  Pointer to interrupt routine

Remarks

This function returns the interrupt vector associated with an interrupt. Note that programs should never get an interrupt vector by reading the low memory vector table directly.

Macro

```
;
get_vector macro interrupt
  mov   al,interrupt
  mov   ah,35h
  int   21h
  endm
```

36H
Get Disk Free Space

Call

AH = 36H
DL
  Drive (0 = Default, 1 = A, etc.)

Return

AX
  FFFF if drive number is invalid; otherwise sectors per cluster
BX
  Available clusters
CX
  Bytes per sector
DX
  Clusters per drive

Remarks

This function returns free space on a disk along with additional information about the disk.

Error Returns

AX = FFFF

The drive number given in DL was invalid.

Macro

```
;
get_disk_space macro drive
          mov   dl,drive
          mov   ah,36h
          int   21h
          endm
```

38H
Return Country-Dependent Information

Call

AH = 38H
DS:DX
  Pointer to 32-byte memory area
AL
  Function code.

Return

Carry set:
AX
  2 = file not found
Carry not set:
  DX:DS filled in with country data

Remarks

The value passed in AL is either 0 (for current country) or a country code. Country codes are typically the international telephone prefix code for the country.

If DX = -1, then the call sets the current country (as returned by the AL = 0 call) to the country code in AL. If the country code is not found, the current country is not changed.

Note

Applications must assume 32 bytes of information. This means the buffer pointed to by DS:DX must be able to accommodate 32 bytes.

This function is fully supported only in versions of MS-DOS 2.01 and higher. It exists in MS-DOS 2.0, but is not fully implemented.

This function returns, in the block of memory pointed to by DS:DX, information pertinent to international applications. The contents of the block are shown in the following table.

WORD Date/time format
5 BYTE ASCIIZ string currency symbol
2 BYTE ASCIIZ string thousands separator
2 BYTE ASCIIZ string decimal separator
2 BYTE ASCIIZ string date separator
2 BYTE ASCIIZ string time separator
1 BYTE Bit field
1 BYTE Currency places
1 BYTE time format
DWORD Case Mapping call
2 BYTE ASCIIZ string data list separator

The format of most of the entries is ASCIIZ (a NUL terminated ASCII string), but a fixed size is allocated for each field for easy indexing into the table.

The date/time format (see table) has the following values:

0 — USA standard	h:m:s m/d/y
1 — Europe standard	h:m:s d/m/y
2 — Japan standard	y/m/d h:m:s

The bit field contains 8 bit values. Any bit not currently defined must be assumed to have a random value.

Bit 0 = 0 If currency symbol precedes the currency amount.

= 1 If currency symbol comes after the currency amount.

Bit 1 = 0 If the currency symbol is directly adjacent to the currency amount.

= 1 If there is a space between the currency symbol and the amount.

The time format has the following values:

0 - 12 hour time
1 - 24 hour time

The currency places field indicates the number of places which appear after the decimal point on currency amounts.

The Case Mapping call is a FAR procedure which will perform country specific lower-to-uppercase mapping on character values from 80H to FFH. It is called with the character to be mapped in AL. It returns the correct upper case code for that character, if any, in AL. AL and the FLAGS are the only registers altered. It is allowable to pass this routine codes below 80H; however nothing is done to characters in this range. In the case where there is no mapping, AL is not altered.

Error Returns

AX
2 = file not found

The country passed in AL was not found (no table for specified country).

Macro

```
;
get_country_info macro buffer, country
   mov   dx,offset buffer
   mov   al,country           ;country = 0
   mov   ah,38h
   int   21h
   endm
```

39H Create Sub-Directory

Call

AH = 39H
DS:DX
  Pointer to path name

Return

Carry set:
AX
  3 = path not found
  5 = access denied
Carry not set:
  No error

Remarks

Given a pointer to an ASCIIZ name, this function creates a new directory entry at the end.

Error Returns

AX
  3 = path not found

    The path specified was invalid or not found.

  5 = access denied

    The directory could not be created (no room in parent directory), the directory/file already existed or a device name was specified.

Macro

```
;
mkdir macro name
   mov   dx,offset name
   mov   ah,39h
   int   21h
   endm
```

3AH
Remove a Directory

Call

AH = 3AH
DS:DX
Pointer to path name

Return

Carry set:
AX
3 = path not found
5 = access denied
16 = current directory
Carry not set:
No error

Remarks

Function 3AH is given an ASCIIZ name of a directory. That directory is removed from its parent directory.

Error Returns

AX
3 = path not found

The path specified was invalid or not found.

5 = access denied

The path specified was not empty, not a directory, the root directory, or contained invalid information.

16 = current directory

The path specified was the current directory on a drive.

Macro

```
;
rmdir macro name
  mov  dx,offset name
  mov  ah,3ah
  int  21h
  endm
```

3BH
Change the Current Directory

Call

AH = 3BH
DS:DX
Pointer to path name

Return

Carry set:
AX
3 = path not found
Carry not set:
No error

Remarks

Function 3BH is given the ASCIIZ name of the directory which is to become the current directory. If any member of the specified pathname does not exist, then the current directory is unchanged. Otherwise, the current directory is set to the string.

Error Returns

AX
3 = path not found

The path specified in DS:DX either indicated a file or the path was invalid.

Macro

```
;
chdir macro name
  mov   dx,offset name
  mov   ah,3bh
  int   21h
  endm
```

3CH
Create a File

Call	AH = 3CH
	DS:DX
	  Pointer to path name
	CX
	  File attribute

Return	Carry set:
	AX
	  3 = path not found
	  4 = too many open files
	  5 = access denied
	Carry not set:
	  AX is handle number

Remarks	Function 3CH creates a new file or truncates an old file to zero length in preparation for writing. DS:DX must point to an ASCIIZ path to the file. If the file did not exist, then the file is created in the appropriate directory and the file is given the attribute found in CX. The given attribute byte is placed at offset 0BH in the file's directory entry. See the section on "Diskette Directory" in chapter 5 for details about the attribute byte. The file handle returned has been opened for read/write access.

Error Returns

AX

3 = path not found

The path specified was invalid.

4 = too many open files

5 = access denied

The attributes specified in CX contained one that could not be created (directory, volume ID), a file already existed with a more inclusive set of attributes, or a directory existed with the same name.

The file was created with the specified attributes, but there were no free handles available for the process, or the internal system tables were full.

Macro

```
;
create_file macro name,attrib
   mov   dx,offset name
   mov   cx,attrib
   mov   ah,3ch
   int   21h
   endm
```

3DH
Open a File Handle

Call

AH = 3DH
AL
  Access
  0 = file opened for reading
  1 = file opened for writing
  2 = file opened for both reading and writing
DS:DX
  pointer to pathname

Return

Carry set:
AX
  2 = file not found
  4 = too many open files
  5 = access denied
  12 = invalid access
Carry not set
  AX is handle number

Remarks

Function 3DH associates a 16-bit handle with a file.

The following values are allowed:

ACCESS	Function
0	opened for reading
1	opened for writing
2	opened for both reading and writing.

DS:DX point to an ASCIIZ name of the file to be opened.

The read/write pointer is set at the first byte of the file and the record size of the file is 1 byte. The returned file handle must be used for subsequent I/O to the file.

Error Returns

AX

2 = file not found

The path specified was invalid or not found.

4 = too many open files

There were no free handles available in the current process or the internal system tables were full.

5 = access denied

The user attempted to open a directory or volume-id, or open a read-only file for writing.

12 = invalid access

The access specified in AL was not in the range 0:2.

Macro

```
;
open_handle macro name, access
        mov   dx,offset name
        mov   al,access
        mov   ah,3dh
        int   21h
        endm
```

3EH
Close a File Handle

Call

AH = 3EH
BX
  File handle

Return

Carry set:
AX
  6 = invalid handle
Carry not set:
  No error

Remarks

If BX is passed a file handle (like that returned by Functions 3CH, 3DH, or 45H), Function 3EH closes the associated file. Internal buffers are flushed to disk.

Error Returns

AX
  6 = invalid handle

The handle passed in BX was not currently open.

Macro

```
;
close_handle macro handle
        mov   bx,handle
        mov   ah,3eh
        int   21h
        endm
```

3FH
Read From File/Device

Call

AH = 3FH
DS:DX
  Pointer to buffer
CX
  Bytes to read
BX
  File handle

Return

Carry set:
AX
  5 = error set:
  6 = invalid handle
Carry not set:
  AX = number of bytes read

Remarks

Function 3FH transfers a specified number of bytes from a file into a buffer location. It is not guaranteed that the number of bytes requested will be read; for example, reading from the keyboard will read at most one line of text. If the returned value is zero, then the program has tried to read from the end of file.

All I/O is done using normalized pointers; no segment wraparound will occur.

Error Returns

AX
  5 = access denied

    The handle passed in BX was opened in a mode that did not allow reading.

  6 = invalid handle

    The handle passed in BX was not currently open.

3FH
Read From File/Device

Macro

```
;
read_from_handle macro buffer,bytes,handle
                 mov   dx,offset buffer
                 mov   cx,bytes
                 mov   bx,handle
                 mov   ah,3fh
                 int   21h
                 endm
```

40H
Write to a File or Device

Call

AH = 40H
DS:DX
  Pointer to buffer
CX
  Bytes to write
BX
  File handle

Return

Carry set:
AX
  5 = access denied
  6 = invalid handle
Carry not set:
AX = number of bytes written

Remarks

Function 40H transfers a specified number of bytes from a buffer into a file. It should be regarded as an error if the number of bytes written is not the same as the number requested.

The write system call with a count of zero (CX = 0) will set the file size to the current position. Allocation units are allocated or released as required.

All I/O is done using normalized pointers; no segment wraparound will occur.

Error Returns

AX

5 = access denied

The handle was not opened in a mode that allowed writing.

6 = invalid handle

The handle passed in BX was not currently open.

Macro

```
;
write_to_handle macro buffer,bytes,handle
        mov   dx,offset buffer
        mov   cx, bytes
        mov   bx,handle
        mov   ah,40h
        int   21h
        endm
```

41H
Delete a Directory Entry

Call

AH = 41H
DS:DX
  Pointer to path name

Return

Carry set:
AX
  2 = file not found
  5 = access denied
Carry not set:
  No error

Remarks

Function 41H deletes the file named in the ASCIIZ string pointed to by DS:DX.

Error Returns

AX
  2 = file not found

    The path specified was invalid or not found.

  5 = access denied

    The path specified was a directory or read-only.

Macro

```
;
erase macro name
  mov   dx,offset name
  mov   ah,41h
  int   21h
  endm
```

42H
Move File Pointer

Call

AH = 42H
CX:DX
  Distance to move, in bytes
AL
  Method of moving:
  (see text)
BX
  File handle

Return

Carry set:
AX
  1 = invalid function
  6 = invalid handle
Carry not set:
  DX:AX = new pointer location

Remarks

Function 42H moves the read/write pointer according to one of the following methods:

Method	Function
0	the pointer is moved to offset bytes from the beginning of the file
1	the pointer is moved to the current location plus offset
2	the pointer is moved to the end of file plus offset

Offset should be regarded as a 32-bit integer with CX occupying the most significant 16 bits.

Error Returns

AX
1 = invalid function

The function passed in AL was not in the range 0:2.

6 = invalid handle

The handle passed in BX was not currently open.

Macro

```
;
move_pointer macro highword,lowword,switch,
                   handle
         mov    dx,lowword
         mov    cx,highword
         mov    al,switch
         mov    bx,handle
         mov    ah,42h
         int    21h
         endm
```

43H
Change Attributes

Call

AH = 43H
DS:DX
  Pointer to path name
AL
  Function
  00 Return in CX
  01 Set to CX
CX (if AL = 01)
  Attribute to be set

Return

Carry set:
AX
  1 = invalid function
  3 = path not found
  5 = access denied
Carry not set:
  CX attributes (if AL = 00)

Remarks

Given an ASCIIZ name pointed to by DS:DX, Function 42H will set/get the attributes of the file to those given in CX. See the section on "Diskette Directory" in chapter 5 for a description of the attribute byte.

A function code is passed in AL:

AL	Function
0	return the attributes of the file in CX
1	set the attributes of the file to those in CX

Error Returns

AX

1 = invalid function

The function passed in AL was not in the range 0:1.

3 = path not found

The path specified was invalid.

5 = access denied

The attributes specified in CX contained one that could not be changed (directory, volume ID).

Macro

```
;
change_attrib macro name,attrib,switch
        mov   dx,offset name
        mov   cx,attrib
        mov   al,switch
        mov   ah,43h
        int   21h
        endm
```

44H
I/O Control for Devices

Call

AH = 44H
BX
  Handle
BL
  Drive (for function codes 4 and 5;
  0 = default, 1 = A:, etc.)
DS:DX
  Data or buffer
CX
  Bytes to read or write
AL
  Function code; see text

Return

Carry set:
AX
  1 = invalid function
  5 = access denied
  6 = invalid handle
  13 = invalid data
Carry not set:
Function Code = 2,3,4,5
  AX = Count transferred
Function Code = 6,7
  AL
    00 = Not ready
    FF = Ready

Remarks

Function 44H sets or gets device information associated with an open handle, or sends/ receives a control string to a device handle or device.

The inputs to AL are function numbers, for which there are returns. The function number values and functions are discussed below.

The following values are allowed in AL as function codes:

Call	Function
0	get device information (returned in DX)
1	set device information (as determined by DX)
2	read CX number of bytes into DS:DX from device control channel
3	write CX number of bytes from DS:DX to device control channel
4	read CX number of bytes into DS:DX from disk (drive number in BL)
5	write CX number of bytes from DS:DX to disk (drive number in BL)
6	get input status
7	get output status

This function can be used to get information about device channels. Calls can be made on regular files, but only calls 0,6 and 7 are defined in that case (AL = 0,6,7). All other calls return an invalid function error.

Calls 0,1: The bits of DX are defined as follows for calls AL = 0 and AL = 1. Note that the upper byte MUST be zero on a set call.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res	CTRL	Reserved						ISDEV	EOF	RAW	SPECL	ISCLK	ISNUL	ISCOT	ISCIN

ISDEV = 1 if this channel is a device
= 0 if this channel is a disk file
(Bits 8-15 = 0 in this case)

If ISDEV = 1

EOF = 0 if End Of File on input
RAW = 1 if this device is in Raw mode
= 0 if this device is cooked
SPECL = 1 if this device is special
ISCLK = 1 if this device is the clock device
ISNUL = 1 if this device is the null device
ISCOT = 1 if this device is the console output
ISCIN = 1 if this device is the console input

CTRL = 0 if this device cannot do control strings via calls AL = 2 and AL = 3
CTRL = 1 if this device can process control strings via calls AL = 2 and AL = 3.
NOTE that this bit cannot be set.

If ISDEV = 0

EOF = 0 if channel has been written
Bits 0-5 are the block device number for the channel (0 = A:, 1 = B:, ...)

NOTE: Bits 15,8-13,4 are reserved and should not be altered.

Calls 2..5: These four calls allow arbitrary control strings to be sent or received from a device. The call syntax is the same as the read and write system calls, except for 4 and 5, which take a drive number in BL instead of a handle in BX.

An invalid function error is returned if the CTRL bit (see above) is 0.

An access denied error is returned by calls AL = 4,5 if the drive number is invalid.

Calls 6,7: These two calls allow the user to check if a file handle is ready for input or output. Status of handles open to a device is the intended use of these calls, but status of a handle open to a disk file is allowed, and is defined as follows:

For input:

- Always ready (AL = FF) until EOF reached, then always not ready (AL = 0) unless current position changed via Function Request 42H (LSEEK).

For output:

- Always ready (even if disk full).

The status is defined at the time the system is CALLED. On future versions, by the time control is returned to the user from the system, the status returned may NOT correctly reflect the true current state of the device or file.

Error Returns

AX

1 = invalid function

The function passed in AL was not in the range 0:7.

5 = access denied (calls AL = 4,5)

6 = invalid handle

The handle passed in BX was not currently open.

13 = invalid data

Macro

```
;
io_ctrl_dev macro handle,buffer,bytes,switch
   mov   bx,handle           ;or 8-bit drive number
   mov   dx,offset buffer
   mov   cx,bytes
   mov   al,switch
   mov   2h,44h
   int   21h
   endm
```

45H
Duplicate a File Handle

Call

AH = 45H
BX
  File handle

Return

Carry set:
AX
  4 = too many open files
  6 = invalid handle
Carry not set:
AX = new file handle

Remarks

Function 45H takes an already opened file handle and returns a new handle that refers to the same file at the same position.

Error Returns

AX
  4 = too many open files

There were no free handles available in the current process or the internal system tables were full.

  6 = invalid handle

The handle passed in BX was not currently open.

Example

```
  mov   bx,fh
  mov   ah,45H
  int   21H
;ax has the returned handle
```

46H
Force a Duplicate of a Handle

Call

AH = 46H
BX
  Existing file handle
CX
  New file handle

Return

Carry set:
AX
  4 = too many open files
  6 = invalid handle
Carry not set:
  No error

Remarks

Function 46H takes an already opened file handle and returns a new handle that refers to the same file at the same position. If there was already a file open on handle CX, it is closed first.

Error Returns

AX
  4 = too many open files

  The internal system tables were full.

  6 = invalid handle

  The handle passed in BX was not currently open.

Example

```
mov    bx,fh
mov    cx,newfh
mov    ah,46H
int    21H
```

Return Name of Current Directory

Call

AH = 47H
DS:SI
  Pointer to 64-byte memory area
DL
  Drive number

Return

Carry set:
AX
  15 = invalid drive
Carry not set:
  No error

Remarks

Function 47H returns an ASCIIZ string giving the name of the current directory for a particular drive. The directory is root-relative and does not contain the drive specifier or leading path separator. The drive code passed in DL is 0 = default, 1 = A:, 2 = B:, etc.

Error Returns

AX
15 = invalid drive

The drive specified in DL was invalid.

Macro

```
;
duplicate_handle macro handle
  mov   bx,handle
  mov   ah,45h
  int   21h
  endm
```

48H
Allocate Memory

Call

AH = 48H
BX
Size of memory to be allocated in paragraphs

Return

Carry set:
AX
7 = arena trashed
8 = not enough memory
BX
Maximum size that could be allocated
Carry not set:
AX:0
Pointer to the allocated memory

Remarks

Function 48H returns a pointer to a free block of memory that has the requested size in paragraphs.

Error Returns

AX
7 = arena trashed

The internal consistency of the memory arena has been destroyed. This is due to a user program changing memory that does not belong to it, thus destroying the memory manager allocation marks.

8 = not enough memory

The largest available free block is smaller than that requested or there is no free block.

Macro

```
;
force_handle macro old,new
   mov   bx,old
   mov   cx,new
   mov   ah,46h
   int   21h
   endm
```

49H
Free Allocated Memory

Call

AH = 49H
ES
  Segment address of memory
  area to be freed

Return

Carry set:
AX
  7 = arena trashed
  9 = invalid block
Carry not set:
  No error

Remarks

Function 49H returns a piece of previously allocated memory to the system pool.

Error Returns

AX
  7 = arena trashed

  The internal consistency of the memory arena has been destroyed. This is due to a user program changing memory that does not belong to it, thus destroying the memory manager allocation marks.

  9 = invalid block

  The block passed in ES is not one allocated via Function Request 48H.

Macro

```
;
cur_dir_name macro buffer,drive
  mov   si,offset buffer
  mov   dl,drive
  mov   ah,47h
  int   21h
  endm
```

4AH
Modify Allocated Memory Blocks

Call

AH = 4AH
ES
  Segment address of memory area
BX
  Requested memory area size

Return

Carry set:
AX
  7 = arena trashed
  8 = not enough memory
  9 = invalid block
BX
  Maximum size possible
Carry not set:
  No error

Remarks

Function 4AH will attempt to grow/shrink an allocated block of memory.

Error Returns

AX

7 = arena trashed

The internal consistency of the memory arena has been destroyed. This is due to a user program changing memory that does not belong to it, thus destroying the memory manager allocation marks.

8 = not enough memory

There was not enough free memory after the specified block to satisfy the grow request.

9 = invalid block

The block passed in ES is not one allocated via this function.

Macro

```
;
alloc_mem macro size
   mov   bx,size
   mov   ah,48h
   int   21h
   endm
```

Load and Execute a Program (EXEC)

Call

AH = 4BH
DS:DX
  Pointer to pathname
ES:BX
  Pointer to parameter block
AL
  00 = Load and execute program
  03 = Load program

Return

Carry set:
AX
  1 = invalid function
  2 = file not found
  8 = not enough memory
  10 = bad environment
  11 = bad format
Carry not set:
  No error

Remarks

This function allows a program to load another program into memory and optionally begin execution of it. DS:DX points to the ASCIIZ name of the file to be loaded. ES:BX points to a parameter block for the load.

A function code is passed in AL:

AL	Function
0	load and execute the program. A program header is established for the program and the terminate and CTRL C addresses are set to the instruction after the EXEC system call.
3	load (do not create) the program header, and do not begin execution. This is useful in loading program overlays.

For each value of AL, the block has the format shown in the following table.

AL = 0 - load/execute program

WORD segment address of environment.
DWORD pointer to command line at 80H of Program Segment Prefix
DWORD pointer to default FCB to be passed at 5CH of PSP
DWORD pointer to default FCB to be passed at 6CH of PSP

AL = 3 - load overlay

WORD segment address where file will be loaded.
WORD relocation factor to be applied to the image.

Note that all open files of a process are duplicated in the child process after an EXEC. This is extremely powerful; the parent process has control over the meanings of stdin, stdout, stderr, stdaux and stdprn. The parent could, for example, write a series of records to a file, open the file as standard input, open a listing file as standard output and then EXEC a sort program that takes its input from stdin and writes to stdout.

Also inherited (or passed from the parent) is an "environment." This is a block of text strings (less than 32K bytes total) that convey various configurations parameters. The format of the environment is as follows:

(paragraph boundary)

BYTE ASCIIZ string 1
BYTE ASCIIZ string 2
...
BYTE ASCIIZ string n
BYTE of zero

Typically the environment strings have the form:

parameter = value

For example, COMMAND.COM might pass its execution search path as:

PATH=A:\BIN;B:\BASIC\LIB

A zero value of the environment address causes the child process to inherit the parent's environment unchanged.

Error Returns

AX

1 = invalid function

The function passed in AL was not 0 or 3.

2 = file not found

The path specified was invalid or not found.

8 = not enough memory

There was not enough memory for the process to be created.

10 = bad environment

The environment was larger than 32Kb.

11 = bad format

The file pointed to by DS:DX was in .EXE format and contained information that was internally inconsistent.

Macro

```
;
free_memory macro address
   mov   ax,address
   mov   es,ax
   mov   ah,49h
   int   21h
   endm
```

Call

AH = 4CH
AL = Return code

Return

None

Remarks

Function 4CH terminates the current process and transfers control to the invoking process. In addition, a return code may be sent. All files open at the time are closed.

This method is preferred over all others (Interrupt 20H, JMP 0) and has the advantage that CS:0 does not have to point to the Program Header Prefix.

Macro

```
;
modify_memory macro address,size
  mov   ax,address
  mov   es,ax
  mov   bx,size
  mov   ah,4ah
  int   21h
  endm
```

4DH
Retrieve the Return Code of a Child

Call AH = 4DH

Return AX
Exit code

Remarks Function 4DH returns the Exit code specified by a child process. It returns this Exit code only once. The low byte of this code is that sent by the Exit routine. The high byte is one of the following:

0 - Terminate/abort
1 - CTRL C
2 - Hard error
3 - Terminate and stay resident

Macro

```
;
exec macro path,param,switch
   mov  dx,offset path
   mov  bx,offset param
   mov  al,switch
   mov  ah,4bh
   int  21h
   endm
```

Call

AH = 4EH
DS:DX
  Pointer to pathname
CX
  Search attributes

Return

Carry set:
AX
  2 = file not found
  18 = no more files
Carry not set:
  No error

Remarks

Function 4EH takes a pathname with wild card characters in the last component (passed in an ASCIIZ string pointed to by DS:DX) along with a set of attributes (passed in CX) and attempts to find all files that match the pathname and have a subset of the required attributes. A datablock at the current DTA is written that contains information in the following form:

```
find_buf_reserved  DB 21 DUP (?); Reserved*
find_buf_attr      DB ? ;attribute found
find_buf_time      DW ? ;time
find_buf_date      DW ? ;date
find_buf_size_l    DW ? ;low(size)
find_buf_size_h    DW ? ;high(size)
find_buf_pname     DB 13 DUP (?) ;packed name
find_buf ENDS
```

*Reserved for MS-DOS internal use on subsequent find_nexts

To obtain the subsequent matches of the pathname, see the description of Function 4FH.

Error Returns

AX

2 = file not found

The path specified in DS:DX was an invalid path.

18 = no more files

There were no files matching this specification.

Macro

```
;
terminate_process macro code
   mov   al,code
   mov   ah,4ch
   int   21h
   endm
```

4FH
Step Through a Directory Matching Files

Call AH = 4FH

Return Carry set:
AX
18 = no more files
Carry not set:
No error

Remarks The current DTA address must point at a block returned by Function 4EH (see Function 4EH).

Error Returns AX
18 = no more files

There are no more files matching this pattern.

Macro

```
;
retrieve_code macro
  mov   ah,4dh
  int   21h
  endm
```

54H
Return Current Setting of Verify After Write Flag

Call AH = 54H

Return AL
Current verify flag value

Remarks The current value of the verify flag is returned in AL.

Macro

```
;
find_match macro name,attrib
   mov   dx,offset name
   mov   cx,attrib
   mov   ah,4eh
   int   21h
   endm
```

Move a Directory Entry

Call

AH = 56H
DS:DX
Pointer to pathname of existing file
ES:DI
Pointer to new pathname

Return

Carry set:
AX
2 = file not found
5 = access denied
17 = not same device
Carry not set:
No error

Remarks

Function 56H attempts to rename a file into another path. The paths must be on the same device.

Error Returns

AX
2 = file not found

The file name specifed by DS:DX was not found.

5 = access denied

The path specified in DS:DX was a directory or the file specified by ES:DI already exists or the destination directory entry could not be created.

17 = not same device

The source and destination are on different drives.

56H
Move a Directory Entry

Macro

```
;
step_match macro
   mov   ah,4fh
   int   21h
   endm
```

57H
Get/Set Date/Time of File

Call

AH = 57H
AL
  00 = get date and time
  01 = set date and time
BX
  File handle
CX (if AL = 01)
  Time to be set
DX (if AL = 01)
  Date to be set

Return

Carry set:
AX
  1 = invalid function
  6 = invalid handle
Carry not set:
No error
CX/DX set if function 0

Remarks

Function 57H returns or sets the last-write time for a handle. These times are not recorded until the file is closed.

A function code is passed in AL:

AL	Function
0	return the time/date of the handle in CX/DX
1	set the time/date of the handle to CX/DX

The format for the date and time is the same as the date and time fields for a directory entry, except that the individual bytes in each word are reversed. The high order portion of the time is in CL, and the high order portion of the date is in DL.

Error Returns

AX
1 = invalid function

The function passed in AL was not in the range 0:1.

6 = invalid handle

The handle passed in BX was not currently open.

Macro

```
;
check_verify_flag macro
          mov   ah,54h
          int   21h
          endm
```

Macro

Note These macro definitions apply to system call examples 00H through 57H.

```
;
;*******************
; Interrupts
;*******************
;
                                        ;ABS_DISK_READ
abs_disk_read macro disk, buffer, num_sectors, first_sector
    mov     al,disk
    mov     bx,offset buffer
    mov     cx,num_sectors
    mov     dx,first_sector
    int     25H                         ;interrupt 25H
    popf
    endm
;
                                        ;ABS_DISK_WRITE
abs_disk_write macro disk, buffer, num_sectors, first_sector
    mov     al,disk
    mov     bx,offset buffer
    mov     cx,num_sectors
    mov     dx,first_sector
    int     26H                         ;interrupt 26H
    popf
    endm
;
stay_resident macro last_instruc        ;STAY_RESIDENT
    mov     dx,offset last_instruc
    inc     dx
    int     27H                         ;interrupt 27H
    endm

;*******************
; Functions
;*******************
;
```

```
;
read_kbd_and_echo macro                 ;READ_KBD_AND_ECHO
    mov     ah,1                        ;function 1
    int     21H
    endm
;
display_char macro character            ;DISPLAY_CHAR
    mov     dl,character
    mov     ah,2                        ;function 2
    int     21H
    endm
;
aux_input macro                         ;AUX_INPUT
    mov     ah,3                        ;function 3
    int     21H
    endm
;
aux_output macro                        ;AUX_OUTPUT
    mov     ah,4                        ;function 4
    int     21H
    endm
;
print_char macro character              ;PRINT_CHAR
    mov     dl,character
    mov     ah,5                        ;function 5
    int     21H
    endm
dir_console_io macro switch             ;DIR_CONSOLE_IO
    mov     dl,switch
    mov     ah,6                        ;function 6
    int     21H
    endm
;
dir_console_input macro                 ;DIR_CONSOLE_INPUT
    mov     ah,7                        ;function 7
    int     21H
    endm
;
read_kbd macro                          ;READ_KBD
    mov     ah,8                        ;function 8
    int     21H
    endm
```

```
;
display macro string                    ;DISPLAY
    mov     dx,offset string
    mov     ah,9                        ;function 9
    int     21H
    endm
;
get_string macro limit,string           ;GET_STRING
    mov     string,limit
    mov     dx,offset string
    mov     ah,0AH                      ;function 0AH
    int     21H
    endm
;
check_kbd_status macro                  ;CHECK_KBD_STATUS
    mov     ah,0BH                      ;function 0BH
    int     21H
    endm
;
flush_and_read_kbd macro switch ;FLUSH_AND_READ_KBD
    mov     al,switch
    mov     ah,0CH                      ;function 0CH
    int     21H
    endm
;
reset_disk macro                        ;RESET DISK
    mov     ah,0DH                      ;function 0DH
    int     21H
    endm
;
select_disk macro disk                  ;SELECT_DISK
    mov     dl,disk[-65]
    mov     ah,0EH                      ;function 0EH
    int     21H
    endm
;
open macro fcb                          ;OPEN
    mov     dx,offset fcb
    mov     ah,0FH                      ;function 0FH
    int     21H
    endm
```

```
;
close macro fcb                     ;CLOSE
    mov     dx,offset fcb
    mov     ah,10H                  ;function 10H
    int     21H
    endm
;
search_first macro fcb              ;SEARCH_FIRST
    mov     dx,offset fcb
    mov     ah,11H                  ;function 11H
    int     21H
    endm
;
search_next macro fcb               ;SEARCH_NEXT
    mov     dx,offset fcb
    mov     ah,12H                  ;function 12H
    int     21H
    endm
;
delete macro fcb                    ;DELETE
    mov     dx,offset fcb
    mov     ah,13H                  ;function 13H
    int     21H
    endm
;
read_seq macro fcb                  ;READ_SEQ
    mov     dx,offset fcb
    mov     ah,14H                  ;function 14H
    int     21H
    endm
;
write_seq macro fcb                 ;WRITE_SEQ
    mov     dx,offset fcb
    mov     ah,15H                  ;function 15H
    int     21H
    endm
;
create macro fcb                    ;CREATE
    mov     dx,offset fcb
    mov     ah,16H                  ;function 16H
    int     21H
    endm
```

```
;
rename macro special_fcb           ;RENAME
   mov     dx,offset special_fcb
   mov     ah,17H                  ;function 17H
   int     21H
   endm
;
current_disk macro                 ;CURRENT_DISK
   mov     ah,19H                  ;function 19H
   int     21H
   endm
;
set_dta macro buffer               ;SET_DTA
   mov     dx,offset buffer
   mov     ah,1AH                  ;function 1AH
   int     21H
   endm
;
read_ran macro fcb                 ;READ_RAN
   mov     dx,offset fcb
   mov     ah,21H                  ;function 21H
   int     21H
   endm
;
write_ran macro fcb                ;WRITE_RAN
   mov     dx,offset fcb
   mov     ah,22H                  ;function 22H
   int     21H
   endm
;
file_size macro fcb                ;FILE_SIZE
   mov     dx,offset fcb
   mov     ah,23H                  ;function 23H
   int     21H
   endm
```

```
;
set_relative_record macro fcb          ;SET_RELATIVE_RECORD
   mov     dx,offset fcb
   mov     ah,24H                       ;function 24H
   int     21H
   endm
;
set_vector macro interrupt,seg_addr,off_addr ;SET_VECTOR
   push    ds
   mov     ax,seg_addr
   mov     ds,ax
   mov     dx,off_addr
   mov     al,interrupt
   mov     ah,25H                       ;function 25H
   int     21H
   endm
;
ran_block_read macro fcb,count,rec_size;RAN_BLOCK_READ
   mov     dx,offset fcb
   mov     cx,count
   mov     word ptr fcb[14], rec_size
   mov     ah,27H                       ;function 27H
   int     21H
   endm
;
ran_block_write macro fcb,count,rec_size;RAN_BLOCK_WRITE
   mov     dx,offset fcb
   mov     cx,count
   mov     word ptr fcb[14], rec_size
   mov     ah,28H                       ;function 28H
   int     21H
   endm
```

```
;
parse macro filename, fcb            ;PARSE
    mov     si,offset filename
    mov     di,offset fcb
    push    es
    push    ds
    pop     es
    mov     al,0FH
    mov     ah,29H                   ;function 29H
    int     21H
    pop     es
    endm
;
get_date macro                       ;GET_DATE
    mov     ah,2AH                   ;function 2AH
    int     21H
    endm
;
set_date macro year,month,day        ;SET_DATE
    mov     cx,year
    mov     dh,month
    mov     dl,day
    mov     ah,2BH                   ;function 2BH
    int     21H
    endm
;
get_time macro                       ;GET_TIME
    mov     ah,2CH                   ;function 2CH
    int     21H
    endm
;                                    ;SET_TIME
set_time macro hour, minutes, seconds, hundredths
    mov     ch,hour
    mov     cl,minutes
    mov     dh,seconds
    mov     dl,hundredths
    mov     ah,2DH                   ;function 2DH
    int     21H
    endm
```

```
;
verify macro switch                        ;VERIFY
   mov          al,switch
   mov          ah,2EH                     ;function 2EH
   int          21H
   endm
;
get_dta macro                              ;GET_DTA
   mov          ah,2FH
   int          21H
   endm
;
get_version_num macro                      ;GET_VERSION_NUM
   mov          ah,30H
   int          21H
   endm
;
keep_process macro exitcode,parasize       ;KEEP_PROCESS
   mov          al,exitcode
   mov          dx,parasize
   mov          ah,31H
   int          21H
   endm
;
ctrl_c_check macro switch,val              ;CTRL_C_CHECK
   mov          dl,val
   mov          al,switch
   mov          ah,33H
   int          21H
   endm
;
get_vector macro interrupt                 ;GET_VECTOR
   mov          al,interrupt
   mov          ah,35H
   int          21H
   endm
;
get_disk_space macro drive                 ;GET_DISK_SPACE
   mov          dl,drive
   mov          2h,36H
   int          21H
   endm
```

```
;
get_country_info macro buffer,country    ;GET_COUNTRY_INFO
   mov         dx,offset buffer
   mov         al,country                ;country = 0
   mov         ah,38H
   int         21H
   endm
;
mkdir macro name                         ;MKDIR
   mov         dx,offset name
   mov         ah,39H
   int         21H
   endm
;
rmdir macro name                         ;RMDIR
   mov         dx,offset name
   mov         ah,3AH
   int         21H
   endm
;
chdir macro name                         ;CHDIR
   mov         dx,offset name
   mov         ah,3BH
   int         21H
   endm
;
create_file macro name,attrib            ;CREATE_FILE
   mov         dx,offset name
   mov         cx,attrib
   mov         ah,3CH
   int         21H
   endm
;
open_handle macro name,access            ;OPEN_HANDLE
   mov         dx,offset name
   mov         al,access
   mov         2h,3DH
   int         21H
   endm
```

```
;
close_handle macro handle                    ;CLOSE_HANDLE
   mov          bx,handle
   mov          2h,3EH
   int          21H
   endm
;
read_from_handle macro buffer,bytes,handle
                                             ;READ_FROM_HANDLE
   mov          dx,offset buffer
   mov          cx,bytes
   mov          bx,handle
   mov          ah,3FH
   int          21H
   endm
;
write_to_handle macro buffer,bytes,handle
                                             ;WRITE_TO_HANDLE
   mov          dx,offset buffer
   mov          cx,bytes
   mov          bx,handle
   mov          ah,40H
   int          21H
   endm
;
erase macro name                             ;ERASE
   mov          dx,offset name
   mov          ah,41H
   int          21H
   endm
;
move_pointer macro highword,lowword,switch,handle
                                             ;MOVE_POINTER
   mov          dx,lowword
   mov          cx,highword
   mov          al,switch
   mov          bx,handle
   mov          ah,42H
   int          21H
   endm
```

```
;
change_attrib macro name,attrib,switch  ;CHANGE_ATTRIB
    mov         dx,offset name
    mov         cx,attrib
    mov         al,switch
    mov         2h,43H
    int         21H
    endm
;
io_ctrl_dev macro handle,buffer,bytes,switch
                                        ;IO_CTRL_DEV
    mov         bx,handle               ;or 8-bit drive number
    mov         dx,offset buffer
    mov         cx,bytes
    mov         al,switch
    mov         ah,44H
    int         21H
    endm
;
duplicate_handle macro handle           ;DUPLICATE_HANDLE
    mov         bx,handle
    mov         ah,45H
    int         21H
    endm
;
force_handle macro old,new              ;FORCE_HANDLE
    mov         bx,old
    mov         cx,new
    mov         ah,46H
    int         21H
    endm
;
cur_dir_name macro buffer,drive         ;CUR_DIR_NAME
    mov         si,offset buffer
    mov         dl,drive
    mov         ah,47H
    int         21H
    endm
```

```
;
alloc_mem macro size                    ;ALLOC_MEM
   mov          bx,size
   mov          ah,48H
   int          21H
   endm
;
free_memory macro address               ;FREE_MEMORY
   mov          ax,address
   mov          es,ax
   mov          ah,49H
   int          21H
   endm
;
modify_memory macro address,size        ;MODIFY_MEMORY
   mov          ax,address
   mov          es,ax
   mov          bx,size
   mov          ah,4AH
   int          21H
   endm
;
exec macro path,param,switch            ;EXEC
   mov          dx,offset path
   mov          bx,offset param
   mov          al,switch
   mov          ah,4BH
   int          21H
   endm
;
terminate_process macro code            ;TERMINATE_PROCESS
   mov          al,code
   mov          ah,4CH
   int          21H
   endm
;
retrieve_code macro                     ;RETRIEVE_CODE
   mov          ah,4DH
   int          21H
   endm
```

```
;
find_match macro name,attrib                    ;FIND_MATCH
    mov         dx,offset name
    mov         cx,attrib
    mov         ah,4EH
    int         21H
    endm
;
step_match macro                                ;STEP_MATCH
    mov         ah,4FH
    int         21H
    endm
;
check_verify_flag macro                         ;CHECK_VERIFY_FLAG
    mov         ah,54H
    int         21H
    endm
;
rename macro old,new                            ;RENAME
    mov         dx,offset old
    mov         di,offset new
    mov         ah,56H
    int         21H
    endm
;
date_time_of_file macro switch,handle,date,time
                                                ;DATE_TIME_OF_FILE
    mov         al,switch
    mov         bx,handle
    mov         cx,time
    mov         dx,date
    mov         ah,57H
    int         21H
    endm
```

```
;********************
; General
;********************
move_string macro source, destination, num_bytes
                                              ;MOVE_STRING
    push        es
    mov         ax,ds
    mov         es,ax
    mov         si,offset source
    mov         di,offset destination
    mov         cx,num_bytes
rep movs        es:destination,source
    pop         es
    endm
;
;
convert macro value, base, destination ;CONVERT
        local   table,start
        jmp     start
table   db      "0123456789ABCDEF"
start:  mov     al,value
        xor     ah,ah
        xor     bx,bx
        div     base
        mov     bl,al
        mov     al,cs: table[bx]
        mov     destination,al
        mov     bl,ah
        mov     al,cs: table[bx]
        mov     destination[1],al
        endm
```

```
;
convert_to_binary macro string, number, value
                                        ;CONVERT_TO_BINARY
            local       ten,start,calc,mult,no_mult
            jmp         start
ten         db          10
start:      mov         value,0
            xor         cx,cx
            mov         cl,number
            xor         si,si
calc:       xor         ax,ax
            mov         al,string[si]
            sub         al,48
            cmp         cx,2
            jl          no_mult
            push        cx
            dec         cx
mult:       mul         cs:ten
            loop        mult
            pop         cx
no_mult:    add         value,ax
            inc         si
            loop        calc
            endm
;
convert_date macro dir_entry
            mov         dx,word ptr dir_entry[25]
            mov         cl,5
            shr         dl,cl
            mov         dh,dir_entry[25]
            and         dh,1fh
            xor         cx,cx
            mov         cl,dir_entry[26]
shr         cl,1
add         cx,1980
endm
;
```

8 ROM BIOS Service Routines

- Overview
- Conventions
- Interrupt Vector Listing
- Video Control
- Diskette Services
- Communications Services
- Keyboard Handling
- Printer Routines
- Miscellaneous ROM BIOS Services
- Bypassing the BIOS
- CONFIG.SYS

Overview

This chapter describes the ROM BIOS service routines that are provided to perform the more low-level functions that you may need in your assembly language programs. Because these are low-level routines, they provide more direct access to the hardware than the DOS routines. However, they do not provide some of the protection and conveniences that the DOS routines give. Be sure to check the chapter on "System Calls" to make your choice between similar DOS and BIOS calls.

Conventions

Access to the BIOS service routines is through the 8086 software interrupts. The routines are called with conventions that are very similar to the conventions for calling DOS routines.

To issue a BIOS interrupt, use the Interrupt statement to select the desired interrupt:

INT 11H

Some interrupts, like Interrupt 11H (Equipment List), perform only one function. Others, like Interrupt 13H (Diskette Services), have several sub-functions that you can call. To select a sub-function, move the number of the sub-function into the AH register.

This chapter describes the register usage for each of the BIOS service routines. It is usually wise to save all important registers before calling a BIOS service routine.

Interrupt Vector List

Interrupt Number (Hex)	Name
5	Print Screen
10	Video
11	Equipment Check
12	Determine Memory Size
13	Diskette
14	Communications
16	Keyboard
17	Printer
19	Bootstrap

Video Control

Introduction

The video controller on the standard AT&T Personal Computer 6300 supports both monochrome and color monitors and produces text or graphics for both color and monochrome. Interrupt 10H has the BIOS services to support all of these modes. This section describes the details that pertain to each major type of video access.

Monochrome Text Mode

The monochrome text modes are mode 0 — 40x25 characters and mode 2 — 80x25 characters. The monochrome text mode uses 32K starting at B8000H. For each screen position, there are two bytes in memory. The first byte is the ASCII code for the character to be displayed. The second byte is the "attribute" that specifies how the character is to be displayed. This attribute byte controls brightness, underlining, and blinking.

The low order nybble of the attribute byte governs the character being displayed according to the following table:

Value	Meaning
0	Character is black
1	Character is normal (white) intensity, underlined
7	Character is normal (white) intensity
F	Character is high intensity white

Any other value for the low nybble selects a particular gray character intensity.

The high order nybble of the attribute byte governs the character background and blinking. A displayed character will blink if the high order bit of its attribute byte is set. The remaining three bits select the gray scale of the background — again, 000 is black and 111 is white. Note that inverse video can be obtained by forcing a black character on a white background, i.e. an attribute byte of 70H.

The first two bytes in the display memory control the character in the top left corner of the screen. The next two bytes control the character in the top row, in the second column position, and so on.

At the end of each line, the display memory returns to the first column of the next line. There are no gaps in the display storage, and no boundaries between one line and the next.

Eight pages of memory are used to build up to eight separate screens. Only one page is active at any time, but you can switch the active page number and thereby display screens very rapidly.

The display pages are numbered 0 - 7 for 40x25 mode and 0 - 3 for 80x25 mode. Page 0 starts at memory location B8000H. For 40 column mode, the pages occur at 2K intervals; for 80 column, at 4K intervals. A total of 32K of memory is used.

Color Text Modes

The color text modes are mode 1 — 40x25 color mode and mode 3 — 80x25 color.

Memory usage for the color text modes is similar to the method used for monochrome text. Two bytes of memory are used for each character position: the first is the ASCII code for the character and the second is the attribute byte. The attribute byte specifies blinking, brightness, and color.

The attribute bytes in color text mode operate much the same way as they do in monochrome text modes with two major differences:

- Instead of bits 0-3 and 4-7 selecting the gray scale of the foreground and background, they select foreground and background colors according to the following chart:

Bit d	c	b	a	Color
0	0	0	0	Black
0	0	0	1	Blue
0	0	1	0	Green
0	0	1	1	Cyan
0	1	0	0	Red
0	1	0	1	Magenta
0	1	1	0	Brown
0	1	1	1	White
1	0	0	0	Grey
1	0	0	1	Lt. blue
1	0	1	0	Lt. green
1	0	1	1	Lt. cyan
1	1	0	0	Lt. red
1	1	0	1	Lt. magenta
1	1	1	0	Yellow
1	1	1	1	High intensity white

Note that since background color is determined by a three-bit value, only the first eight colors apply to that field.

- There is no underline attribute possible in color mode. As can be seen by the chart above, attribute settings that produce an underline in monochrome mode produce a blue character in color mode.

The display memory maps to the character positions exactly as it does in monochrome text mode.

Color Graphics Mode

There is one color graphics mode: mode 4 — medium resolution (320x200) color graphics. For any color display, you can use up to four colors. You select from one of two "palettes," each of which provides three colors. You select a "background" color to be used as the fourth color.

Palette 0 contains green, yellow, and red. Palette 1 contains cyan (light blue), magenta, and white.

320 pixels can be displayed on each of 200 lines. Each line takes 80 bytes or 640 bits of display memory. Each color pixel use two bits of memory. Since two bits give you four possible combinations, for each pixel you specify either the background color or one of the three colors in the current palette. The leftmost pixels are represented by the high order bits in the byte.

Display memory for color graphics mode starts at location B8000H and is divided into four 8K blocks. Starting at B8000, the first 8000D bytes contain the pixel data for the even scan lines on page zero. That is, the first 80 bytes describe line 0, the next 80H describe line 2, and so on through line 198. The odd lines are described in the 8K block starting at BA000. The same pattern is repeated for page one in the next 16K block, with the even lines starting at BC000 and the odd lines starting at BE000.

High Resolution Monochrome Graphics

Memory for high resolution 640x200 monochrome graphics is handled similarly to 320x200 color graphics. The only difference is that instead of memory containing two bits of color information per pixel, each pixel can only be on or off and is thus represented by one bit. In this way eight pixels can be represented in a byte instead of four, so that a scan line takes as many bytes as in color graphics mode even though it contains twice as many pixels. As in color graphics mode, the leftmost pixels are represented in the high order bits of each byte. Line mapping is exactly as described above for color graphics mode. In high resolution monochrome graphics the background color is always black and the foreground color is chosen by bits 0-3 of the color select register.

Super High Resolution Monochrome Graphics

Super high resolution 640x400 monochrome graphics mode maps one bit per pixel with the leftmost pixel represented at the high end of the byte, just like high resolution 640x200 mode. Also like high resolution mode, super high mode maps onto a black background with a foreground color chosen by the color select register. The memory mapping, however, takes up all 32K of display memory for a single page. Memory is broken up into four 8K segments, with each segment containing the data for every fourth scan line. Thus display memory looks like this:

Memory location	Contains pixels for line numbers
B8000	0, 4, 8, ... 396
B9F3F	Not used.
BA000	1, 5, 9, ... 397
BBF3F	Not used.
BC000	2, 6, 10, ... 398
BDF3F	Not used.
BE000	3, 7, 11, ... 399
BFF3F	Not used.

Set Mode and Clear Screen

Input:
(AH) = 0
(AL) contains the CRT mode value

Text Modes:
(AL) = 0 40x25 monochrome
(AL) = 1 40x25 color
(AL) = 2 80x25 monochrome
(AL) = 3 80x25 color

Graphics modes:
(AL) = 4 320x200(medium resolution), color
(AL) = 5 320x200(medium resolution), monochrome
(AL) = 6 640x200 black/white (high resolution)
(AL) = 40H graphics 640x400 monochrome super high resolution
(AL) = 48H graphics 640x400 monochrome tiny text (80x50 text)

Set Cursor Type

Input:
(AH) = 1
Low order 5 bits of (CH) = start line for cursor.

Note
Do not set the high bits of CH: unpredictable results will occur.

Low order 5 bits of (CL) = end line for cursor.

Set Cursor Position

Input:
(AH) = 2
(DH,DL) = Row,Column (Position 0,0 is upper left.)
(BH) = page number (must be 0 for super-res graphics mode.)

Read Cursor Position

Input:
(AH) = 3
(BH) = page number (must be 0 for super-res
graphics mode.)

Output:
(DH,DL) = row, column of current cursor
(CH,CL) = current cursor start and end lines

Read Light Pen Position

Input:
(AH) = 4

Output:
(AH) = 0 light pen switch not triggered
(AH) = 1 valid light pen value obtained:
(DH,DL) = row, column of character
light pen position
(CH) = raster line (0-199)
(BX) = pixel column (0-319 for medium
resolution, 0-639 for high
resolution.)

Select Active Page Number

Valid only for modes (0 - 6)

Input:
(AH) = 5
(AL) = 0-15 for modes 0, 1
= 0-7 for modes 2, 3
= 0-1 for modes 4, 6

Scroll Active Page up

Input:
(AH) = 6
(AL) = number of lines blanked at bottom of window by scrolling up. AL = 0 means blank entire window.
(CH,CL) = row, column of upper left corner of scroll
(DH,DL) = row, column of lower right corner of scroll
(BH) = attribute to be used on blank line(s).

Scroll Active Page Down

Input:
(AH) = 7
(AL) = number of lines blanked at top of window by scrolling down. AL = 0 means blank entire window.
(CH,CL) = row, column of upper left corner of scroll
(DH,DL) = row, column of lower right corner of scroll
(BH) = attribute to be used on blank line(s).

Character Handling

The next three video services perform character input/output for the CRT. If your program displays characters to the screen while in graphics modes, the characters are formed from a character generator image that is maintained in the ROM. However, only the first 128 characters are encoded there. If you want to create your own characters, either for the purposes of doing character graphics or implementing a foreign language alphabet, you must set up a table of code points for 128 new characters and initialize the pointer at interrupt 1F (address 0007CH) to point to the 1K table. These codes can then be accessed by referring to characters 128-255.

When you write characters to the screen in text mode, if you send more characters to be written than will fit on one line, the extra characters automatically wrap around to the beginning to the next line. In graphics mode, the character handling routines only produce correct results for characters contained on the same row (continuation to succeeding lines does not work.)

Read Attribute or Character at Current Cursor Position

Input:
(AH) = 8
(BH) = current display page

Output:
(AL) = character read
(AH) = attribute of character read

Write Attribute and Character at Current Cursor Position

Input:
(AH) = 9
(BH) = current display page
(CX) = count of characters to write
(AL) = character to write
(BL) = attribute of character (if text mode)
= color of character (if graphics mode)

Note
If bit 7 of BL = 1, the color value is exclusive OR'd with the current contents of the dot.

Write Character Only at Current Cursor Position

Input:
(AH) = OAH
(BH) = current display page
(CX) = count of characters to write
(AL) = character to write

Set Color Pallette

Input:
(AH) = OBH
(BH) = color palette ID (0-127)
(BL) = color value to be used with that color ID

Color ID = 0 selects the background color (0-15)
Color ID = 1 selects the palette to be used:
0 = green/red/yellow
1 = cyan/magenta/white

Write Dot

Input:
(AH) = OCH
(DX) = row number
(CX) = column number
(AL) = color value. If bit 7 of AL = 1, the color value is exclusive OR'd with the current contents of the dot.

Read Dot

Input:
(AH) = ODH
(DX) = row number
(CX) = column number

Output:
(AL) = the dot read

Write Teletype

This routine is used by the "TYPE" command and other DOS commands to display data on the screen.

Input:
(AH) = OEH
(AL) = character to write
(BL) = foreground color in graphics mode

Note
Screen width is controlled by previous mode set.

Current Video State

Input:
(AH) = OFH

Output:
(AL) = current mode
(AH) = number of character columns on screen
(BH) = current active display page

Diskette Services

Introduction

Interrupt 13H is the BIOS routine for diskette services. There are six services provided by INT 13H.

Input

(AH) = 0 Reset Diskette System
(AH) = 1 Read status of diskette system into AL
(AH) = 2 Read sectors into memory
(AH) = 3 Write sectors from memory to diskette
(AH) = 4 Verify the specified sectors
(AH) = 5 Format a track

Additional settings for read, write, verify, and format:

(DL) = drive number (0 - 3 allowed, value checked)
(DH) = head number (0 - 1 allowed, value not checked)
(CH) = track number (0-39 allowed, value not checked)

Additional settings for read, write, and verify:

(CL) = sector number (1-9, value not checked)
(AL) = number of sectors (max = 9, value not checked)

ES:BX = address of buffer (not required for verify) For the format operation, ES:BX points to the collection of address fields for the track. There must be one of these fields for every sector on the track. Each field has four bytes:

Offset 0 = track number
1 = head number
2 = sector number
3 = number of bytes/ sector
(00 = 128, 01 = 256, 02 = 512, 03 = 1024)

Output

(AH) = Status of operation:

01 bad command
02 address mark not found
03 write was requested on write-protected disk
04 requested sector not found
08 DMA overrun
09 DMA transfer crossed a 64K boundary
10 read data error detected by CRC
20 diskette controller chip failed
40 seek to desired track failed
80 device timeout

(CY) = 0 successful operation
(CY) = 1 unsuccessful operation (AH has details)

For read, write, and verify these registers are preserved: DS, BX, DX, CH, and CL.

(AL) = number of sectors read; this value may be incorrect if a timeout occurred.

NOTE

If an error is reported by the diskette, reset the diskette, then retry the operation. On read operations, no motor start delay is taken, so your code should retry three times to make sure that a read error is not caused by motor start-up.

Communications Services

Introduction

This set of routines performs serial, RS232C communications through the communications port. You should use a polling technique in your communications; this is not interrupt-driven I/O. All functions are accessed through BIOS interrupt 14H.

Initialize the Communications Port

Input:
(AH) = 0
(DX) = selection of RS-232 channel (0 or 1)
(AL) = parameters for initialization in the following form:

7 6 5	4 3	2	1 0
—Baud Rate—	—Parity—	Stopbit	—Word length—
000 - 110 baud	00 - None	0 - 1	10 - 7 bits
001 - 150	01 - Odd	1 - 2	11 - 8 bits
010 - 300	11 - Even		
011 - 600			
100 - 1200			
101 - 2400			
110 - 4800			
111 - 9600			

Output:
Condition is set according to the same conventions as in “Get Comm Port Status” (see below).

Send Character

Input:
(AH) = 1
(DX) = RS232 channel to be used (0 or 1)
(AL) = the character to be sent.

Output:
(AL) is preserved.
(AH) — if the operation was unsuccessful, bit 7 is set. The other bits in (AH) are set as they are in "Get Comm Port Status" if the operation was successful.

Receive Character

Input:
(AH) = 2
(DX) = RS232 channel to be used (0 or 1)

Output:
(AL) = the received character.
(AH) = status of operation, if (AH) = 0, the operation was successful. If the high order bit of (AH) is set, a timeout error aborted the operation and the rest of (AH) can be ignored. Any other setting of (AH) indicates errors in the receive character operation.

Get Comm Port Status

Input:
(AH) = 3
(DX) = RS232 channel to be used (0 or 1)

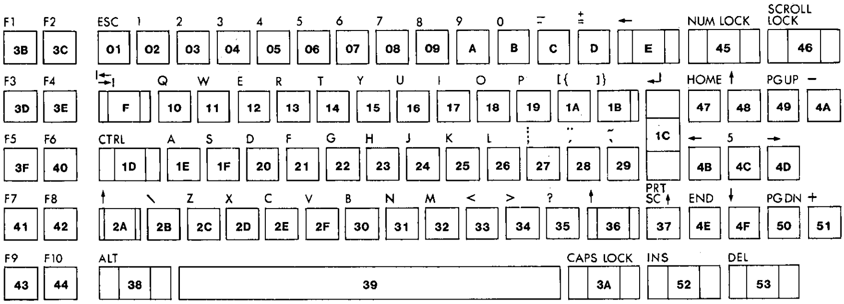
Output
(AX) = status:
(AH) = line control status
- bit 7 = timeout
- bit 6 = transmission shift reg. empty
- bit 5 = transmission holding reg. empty
- bit 4 = break detect
- bit 3 = framing error
- bit 2 = parity error
- bit 1 = overrun error
- bit 0 = data ready

(AL) = modem status
- bit 7 = received line signal detect
- bit 6 = ringing detect
- bit 5 = data set ready
- bit 4 = clear to send
- bit 3 = delta receive line signal detect
- bit 2 = trailing edge ring detected
- bit 1 = delta data set ready
- bit 0 = delta clear to send

Keyboard Handling

Introduction

Interrupt 16H provides the keyboard handling functions through three sub-functions. Most keys return two values: a scan code and a character code. The scan code is the same as the key number (see diagram below), and the character code is the ASCII superset interpretation of the key (including coincident SHIFTs or CTRLs). Check the section on "DOS Interrupts and Function Calls" to select either the BIOS keyboard routines or the DOS routines.



CHARACTER CODES

ASCII Decimal	Value Hex	Character	Control Character	ASCII Decimal	Value Hex	Character
000	00	(null)	NUL	032	20	(space)
001	01	☺	SOH	033	21	!
002	02	☻	STX	034	22	”
003	03	♥	ETX	035	23	#
004	04	♦	EOT	036	24	$
005	05	♣	ENQ	037	25	%
006	06	♠	ACK	038	26	&
007	07	(beep)	BEL	039	27	’
008	08	◘	BS	040	28	(
009	09	(tab)	HT	041	29	)
010	0A	(line feed)	LF	042	2A	*
011	0B	(home)	VT	043	2B	+
012	0C	(form feed)	FF	044	2C	,
013	0D	(carriage return)	CR	045	2D	-
014	0E	♫	SO	046	2E	.
015	0F	☼	SI	047	2F	/
016	10	►	DLE	048	30	0
017	11	◄	DC1	049	31	1
018	12	↕	DC2	050	32	2
019	13	‼	DC3	051	33	3
020	14	¶	DC4	052	34	4
021	15	§	NAK	053	35	5
022	16	▬	SYN	054	36	6
023	17	↨	ETB	055	37	7
024	18	↑	CAN	056	38	8
025	19	↓	EM	057	39	9
026	1A	→	SUB	058	3A	:
027	1B	←	ESC	059	3B	;
028	1C	(cursor right)	FS	060	3C	<
029	1D	(cursor left)	GS	061	3D	=
030	1E	(cursor up)	RS	062	3E	>
031	1F	(cursor down)	US	063	3F	?

CHARACTER CODES (Cont'd)

ASCII Decimal	Value Hex	Character	Control Character	ASCII Decimal	Value Hex	Character
064	40	@		096	60	©
065	41	A		097	61	a
066	42	B		098	62	b
067	43	C		099	63	c
068	44	D		100	64	d
069	45	E		101	65	e
070	46	F		102	66	f
071	47	G		103	67	g
072	48	H		104	68	h
073	49	I		105	69	i
074	4A	J		106	6A	j
075	4B	K		107	6B	k
076	4C	L		108	6C	l
077	4D	M		109	6D	m
078	4E	N		110	6E	n
079	4F	O		111	6F	o
080	50	P		112	70	p
081	51	Q		113	71	q
082	52	R		114	72	r
083	53	S		115	73	s
084	54	T		116	74	t
085	55	U		117	75	u
086	56	V		118	76	v
087	57	W		119	77	w
088	58	X		120	78	x
089	59	Y		121	79	y
090	5A	Z		122	7A	z
091	5B	[		123	7B	{
092	5C	\		124	7C	¦
093	5D	]		125	7D	}
094	5E	^		126	7E	~
095	5F	—		127	7F	⌂

CHARACTER CODES (Cont'd)

ASCII Decimal	Value Hex	Character	ASCII Decimal	Value Hex	Character
128	80	Ç	160	A0	á
129	81	ü	161	A1	í
130	82	é	162	A2	ó
131	83	â	163	A3	ú
132	84	ä	164	A4	ñ
133	85	à	165	A5	Ñ
134	86	å	166	A6	ª
135	87	ç	167	A7	º
136	88	ê	168	A8	¿
137	89	ë	169	A9	⌐
138	8A	è	170	AA	¬
139	8B	ï	171	AB	½
140	8C	î	172	AC	¼
141	8D	ì	173	AD	¡
142	8E	Ä	174	AE	«
143	8F	Å	175	AF	»
144	90	É	176	B0	░
145	91	æ	177	B1	▒
146	92	Æ	178	B2	▓
147	93	ô	179	B3	│
148	94	ö	180	B4	┤
149	95	ò	181	B5	╡
150	96	û	182	B6	╢
151	97	ù	183	B7	╖
152	98	ÿ	184	B8	╕
153	99	Ö	185	B9	╣
154	9A	Ü	186	BA	║
155	9B	¢	187	BB	╗
156	9C	£	188	BC	╝
157	9D	¥	189	BD	╜
158	9E	Pt	190	BE	╛
159	9F	ƒ	191	BF	┐

CHARACTER CODES (Cont'd)

ASCII Decimal	Value Hex	Character	ASCII Decimal	Value Hex	Character
192	C0	└	224	E0	α
193	C1	┴	225	E1	β
194	C2	┬	226	E2	Γ
195	C3	├	227	E3	π
196	C4	─	228	E4	Σ
197	C5	┼	229	E5	σ
198	C6	╞	230	E6	μ
199	C7	╟	231	E7	τ
200	C8	╚	232	E8	Φ
201	C9	╔	233	E9	Θ
202	CA	╩	234	EA	Ω
203	CB	╦	235	EB	δ
204	CC	╠	236	EC	∞
205	CD	═	237	ED	ø
206	CE	╬	238	EE	ε
207	CF	╧	239	EF	∩
208	D0	╨	240	F0	≡
209	D1	╤	241	F1	±
210	D2	╥	242	F2	≥
211	D3	╙	243	F3	≤
212	D4	╘	244	F4	⌠
213	D5	╒	245	F5	⌡
214	D6	╓	246	F6	÷
215	D7	╫	247	F7	≈
216	D8	╪	248	F8	°
217	D9	┘	249	F9	●
218	DA	┌	250	FA	•
219	DB	█	251	FB	√
220	DC	▄	252	FC	n
221	DD	▌	253	FD	^{2}
222	DE	▐	254	FE	■
223	DF	▀	255	FF	(blank)

Read Next ASCII Character

Input:
(AH) = 0

Output:
(AL) = character code
(AH) = scan code

Note

This routine will not return execution to the calling program until it has a keystroke to report.

Check if Keystroke Available

This routine is used to check to see if a keystroke has been entered. Use this function if you want to continue processing whether or not a key has been pressed.

Input:
(AH) = 1

Output:
Z flag = 1 — no code available
Z flag = 0 — code is available
If a character is available, it is stored in AX in the same format as for the "Read Next ASCII Character" call. However, the code also remains in the keyboard buffer, so that a "Read Next ASCII Character" call returns this character's code value again.

Get Current Shift Status

Input:
(AH) = 2

Output:
(AL) = current shift status

Shift States

Bit	Subject matter	Meaning, when bit is 1
7	Insert	state active
6	Caps-Lock	state active
5	Num-Lock	state active
4	Scroll-Lock	state active
3	Alt shift	key depressed
2	Ctrl shift	key depressed
1	left-hand shift	key depressed
0	right-hand shift	key depressed

Printer Routines

This set of BIOS routines communicates with the printer through interrupt 17H.

Print a Character

Input:
(AH) = 0
(AL) = the character to be printed
(DX) = printer port number (0-3)

Output:
(AH) = 1 if character not printed due to timeout. Otherwise, bits are set as they are in "Get Printer Status" call.

Initialize Printer Port

Input:
(AH) = 1
(DX) = printer port number (0-3)

Output:
(AH) = printer status (see below)

Get Printer Status

Input:
(AH) = 2
(DX) = printer port number (0-3)

Output:

Bit	Meaning if Set (equal to 1)
7	not busy
6	acknowledge
5	out of paper
4	selected
3	I/O error
2	not used
1	not used
0	timeout (set by software)

If a printer is connected, 10H and 90H are healthy statuses. Otherwise 30H is healthy.

Miscellaneous ROM BIOS Services

System Reset

If you issue issue interrupt 19H, the system bootstraps itself in much the same way as it does with the Ctrl-Alt-Delete key sequence. The only difference is that Ctrl-Alt-Delete causes diagnostics to be run, whereas Interrupt 19H causes an immediate system load.

Print Screen

To obtain a printed copy of what is on the screen, issue a request for interrupt 5H. This produces exactly the same result as pressing the shift and PrtSc keys. This routine works in either text or graphics modes. Unrecognizable characters are printed as blanks.

Equipment List

You can use this routine to obtain a list of the optional equipment attached to your system. Simply issue an interrupt 11H; no register set-up is necessary. Output (starting with the most significant bit of (AX)) is as follows:

	Bit	Meaning
AH	7-6	number of printer adapters (0-3)
	5	not used
	4	game adapter attached, or not
	3-1	number of communications adapters (0-7)
	0	not used
AL	7-6	number of diskette drives minus 1, if bit 0 is set.
	5-4	starting video mode: 01 — graphics card display,40 columns,b/w 10 — graphics card display,80 columns,b/w 11 —monochrome card display
	3-2	amount of memory on system board: 00 — 16KB Base 01 — 32KB Base 10 — 48KB Base 11 — 64KB Base
	1	not used
	0	diskettes are attached; refer to bits 7 and 6.

Determine Memory Size

Interrupt 12H gives the total amount of memory in the address space, up to a megabyte. No register set-up is required. The BIOS reads the switches on the system board and adds the amount of memory on a memory expansion board and returns the total in (AX). The amount of memory is expressed as the number of 1K blocks.

Bypassing the BIOS

This section explains how you can either replace one of the BIOS service routines with a program of your own or add a "front-end" so that you perform some preprocessing immediately prior to using a particular BIOS routine.

When the system is powered on, low memory is initialized with the addresses of all of the BIOS interrupt routines. To replace a BIOS routine, change the address in the interrupt table to the address of the code which you want to execute in place of the BIOS code. To perform preprocessing before handing execution on to the BIOS code:

1. replace the address of the BIOS routine with the address of your program

2. transfer execution to the BIOS routine at the end of your program.

Be sure to use the "Set Vector" system call (Function Request 25H) to replace the BIOS routine addresses instead of writing directly to low memory.

CONFIG.SYS

The special file CONFIG.SYS is processed automatically when DOS starts. As with the batch file AUTOEXEC.BAT, this processing is automatic. DOS will simply look at the root directory to see if the file is there.

CONFIG.SYS is a text file that can be edited by EDLIN or any other text editor that produces ASCII files. Five commands can be used in the CONFIG.SYS file. Each command changes a system parameter.

- **BREAK ON/OFF**
  Changes the way DOS checks for a CTRL/BREAK

- **BUFFERS=xx**
  Sets the number of data buffers that DOS uses.

- **DEVICE=[d:] [path] filename [.ext]**
  Adds a nonstandard device driver to DOS.

- **FILES=xx**
  Sets the number of files that can use ASCIIZ strings.

- **SHELL=[d:] [path] filename [.ext] [d:] [path] /P**
  Specifies an alternate command processor.

Break

Normally, DOS checks for a CTRL/BREAK only when it is doing input or output. Some programs do very little (if any) input or output for long periods of time. The BREAK ON command sets DOS to check for a CTRL/BREAK when any DOS function is called.

BREAK OFF resets the default so that DOS only checks for a CTRL/BREAK during input and output. This command can be used to override a BREAK ON that was set by CONFIG.-SYS.

Buffers

The number of buffers has an effect on both the speed of disk I/O and available memory. A larger number of buffers allocates more memory to the system for operations. This is highly desirable for systems with large amounts of RAM that will be used for database applications. It is highly undesirable for systems with minimal RAM that do little work with disk files.

The system default is BUFFERS=2, which is adequate for most purposes and requires only 1K of RAM. If your system will be doing a significant amount of data handling, especially on a hard disk, BUFFERS=4 would improve access times at a cost of only 1K of internal memory.

Device

When DOS is first started, it loads all of the standard device drivers for the keyboard, screen and so on. If your system requires a special device driver, this command tells DOS where to find it. The device driver will then be loaded as an extension to DOS.

Device drivers are .COM files with a specific structure described in Chapter 9. The command DEVICE=ANSI.SYS causes DOS to replace the standard display and keyboard device drivers with the extended screen and keyboard support that the extended functions require.

If you wish to load several special device drivers, you must use a DEVICE command for each one.

Files

Opening a file with an ASCIIZ string eliminates the traditional direct handling of a File Control Block (FCB). DOS will handle all these things for you by creating and maintaining FCBs internally. To do this, it needs memory. Each file requires 39 bytes of memory.

The FILES command sets aside memory for this operation. The default is FILES=8. The maximum is FILES=99. This is the limit for the entire system. A particular process (or program) can still have only 20 files open at once.

Shell

The COMMAND.COM file that DOS uses as its "front end" processor can be replaced by another command processor. The SHELL command specifies the file to be used and the default path for processing commands. The command processor must be able to read and execute commands, and handle interrupts 22H, 23H, and 24H.

Since COMMAND.COM handles internal commands, .BAT file execution, and .EXE file loading, these functions will be unavailable unless the new command processor duplicates them.

(This page intentionally left blank)

9 MS-DOS Device Drivers

- Overview
- MS-DOS Device Drivers
- Asynchronous Communications Element
- DMA Controller
- Floppy Diskette Interface and Controller
- Hard Disk Controller
- Keyboard Interface
- Parellel Printer Interface
- Programmable Interrupt Controller
- Programmable Interval Timer
- Real Time Clock and Calendar
- Serial Communications Controller
- Speaker
- Video Controller

Overview

Interested Audience

This section contains information on how to write device drivers. You may not use it often since many input and output capabilities are implemented by the BIOS routines discussed in Section 6. BIOS routines allow you to do general input and output without a detailed understanding of the hardware and shield your program from hardware changes.

However, there are times when BIOS routines do not perform the necessary function or do so in an inefficient fashion. Then your own driver is necessary. Implementation of operating systems, of high speed graphics packages, and of unusual keyboard mapping are examples of software which require specialized drivers.

Programmable Devices

This is a list of programmable devices.

INS	8350B	Asynchronous Communications Element
INTEL	8237A	DMA Controller
NECE	uPD765	Floppy Diskette Controller
DTC	5150BX	Hard Disk Controller
INTEL	8041	Keyboard Interface
INTEL	8259A	Programmable Interrupt Controller
INTEL	8253	Programmable Interval Timer
	58174A	Real Time Clock and Calendar
AMD	Z8530	Serial Communications Controller
		Speaker Interface
HD	6845	Video Controller

Port Addresses

A port is a place to read or write information to an I/O device. An I/O device is hardware which the CPU controls. It is both input and output peripherals as well as hardware such as the DMA controller, the Interrupt controller and the Interval Timer. Each device needs different information, but they all need some combination of control, status, and data.

Each port has an address. Sixteen of the twenty possible address lines are available for I/O addressing. This means there are 65,535 possible port addresses.

I/O Instructions

IN and OUT instructions distinguish an I/O access from a memory access. These instructions translate into control signals which define the direction and path of the data.

The port address can be specified in one of two ways — fixed or variable. In the fixed method, the absolute port address is specified in the instruction. The following instruction is an example of fixed port addressing:

```
OUT 020H, AL
```

Only 8-bit port addresses can be used in this format.

The variable method allows 16-bit port addresses to be specified. In this case, the port address is loaded into the DX register and then the DX register is used in the I/O instruction. An example of variable port addressing follows:

```
MOV DX,03F2H
OUT DX,AL
```

Interrupts

External devices do not need the CPU's attention all the time. When they need servicing, they ask for it either by interrupting or by setting a polled flag. External interrupts are ignored when the CLI instruction has cleared the interrupt enable flag and are recognized when the STI instruction has set the flag.

The first 1024 bytes of memory contain an interrupt table. This table has 255 interrupt pointers defining the start address of interrupt service routines. This is the pointer format.

INTERRUPT POINTER

IP
CS

When the CPU recognizes the interrupt, an 8-bit interrupt type identifies the device. The interrupt type is an index into the table of pointers. To obtain the interrupt pointer address, the type is multiplied by four. The CPU saves the flag register on the stack, disables interrupts and single step mode, and saves the CS and IP registers on the stack. Then the interrupt pointer is loaded into IP and CS, and control is transferred to the interrupt service routine. The stack looks like this when the service routine gets control.

Interrupt Devices

SYSTEM STACK

IP
CS
FLAGS

When the service routine begins, interrupts are disabled. Depending on the nature of the application, interrupts can be enabled immediately or just prior to releasing control.

The service routine must preserve the value of all internal registers. Therefore it saves the registers it uses on the stack. Before returning, it restores these registers from the stack.

To return control to the interrupt program, the service routine executes IRET. All the information necessary to do this was carefully placed on the stack

Interrupt Devices

INS	8250B	Asynchronous Communications Element
INTEL	8237A	DMA Controller
NEC	uPD765	Floppy Diskette Controller
DTC	5150BX	Hard Disk Controller
INTEL	8041	Keyboard Interface
INTEL	8259A	Programmable Interrupt Controller
INTEL	8253	Programmable Interval Timer
	58174A	Real Time Clock and Calendar
AMD	Z8530	Serial Communications Controller

Block Diagrams

Block diagrams are a pictorial description of the electrical connection between the CPU, the interface, and the external device. In general, the CPU's bus signals appear on the left. The middle of the diagram describes the internals of the interface. The right side of the diagram describes the electrical interface. This physically connects the interface to the external device. The connection is often through a dual inline package (DIP) of pins. Each pin carries one signal.

Individual signals are represented by lines. Busses are shown as double lines. Each of these have arrows which indicate the direction of the data. Often they are bidirectional.

There are several common CPU control bus signals. There mnemonics and definitions follow:

- A0-A19
  These are the lines used to transmit the address of memory or the address of an I/O port. Only A0-A15 are used for I/O addressing.

- CS
  This signal selects the chip. No reading or writing will occur unless the device is selected.

- D0-D7
  These are the bidirectional data lines used to exchange information with a memory location or an I/O port. D7 is the most significant bit.

- INT0-INT7
  These are the priority interrupt request lines. See the description of the Interrupt Controller.

- IORD
  This signal indicates that an input port address has been placed on the address bus. The data at the specified port is to be placed on the data bus.

- IOWR
  This signal indicates that an output port address has been placed on the address bus and the data has been placed on the data bus to be output to the specified port.

- RESET
  This signal resets the system to a predetermined state.

MS-DOS Device Drivers

What Is a Device Driver?

A device driver is binary code which manipulates hardware in the MS-DOS environment. A special header at the beginning identifies it as a driver, defines the strategy and interrupt entry points, and describes various attributes of the device. The file must have an origin of zero.

There are two kinds of devices:

- Character devices
- Block devices

Character devices perform serial character I/O like CONSOLE, AUXILIARY, and PRINTER. These devices are named and users open channels to do I/O to them.

Block devices are the disk drivers on the system. They perform random I/O in pieces called blocks. This is usually the physical sector size. These devices are not named as the character devices are, and cannot be opened directly. Instead they are identified by drive letters (A:,B:,C:, etc.).

Drive letters are assigned to device drivers based on their ordering in the CONFIG.SYS file. Starting with the letter 'A', each device driver is assigned as many consecutive alphabetic characters as the driver has units. The theoretical limit is 63, but after 26 the drive letters are non-alphabetic (such as] and ^).

Character devices cannot define multiple units because they have only one name.

Device Headers

A device header is required at the beginning of a device driver. A device header looks like this:

DWORD pointer to next device (Must be set to -1)

WORD attributes
  Bit 15 = 1 if char device, 0 if blk
  if bit 15 is 1
    Bit 0 = 1 if current sti device
    Bit 1 = 1 if current sto device
    Bit 2 = 1 if current NUL device
    Bit 3 = 1 if current CLOCK dev
    Bit 4 = 1 if special
    Bits 5-12 Reserved; must be set
        to 0
  Bit 14 is the IOCTL bit
  Bit 13 is the NON IBM FORMAT bit

WORD pointer to device strategy   entry point
WORD pointer to device interrupt   entry point
8-BYTE character device name field Character devices set a device name. For block devices the first byte is the number of units

The strategy and interrupt routines are in the same segment as this device header.

Pointer to Next Device Field

The pointer to the next device header field is a double word field, offset followed by segment. MS-DOS chains the device headers together using this field. If you have a single device header in your driver, initialize this field to -1. If you have more than one device header, the first word of the double word pointer is the offset of the next driver's Device Header.

Attribute Field

The attribute field is used to tell the system whether this device is a block or character device (Bit 15). Most other bits are used to give selected character devices special treatment and are meaningless on a block device. For example, assume that you have a new standard input and output device driver. Besides installing the driver, you must tell MS-DOS that you want this new driver to override the current standard input and standard output device. This is accomplished by setting the attributes to the desired characteristics, so you set Bits 0 and 1 to 1. Similarly, a new CLOCK device could be installed by setting the appropriate attribute. Although there is a NUL device attribute, it is reserved for MS-DOS.

The NON PC-DOS FORMAT bit applies only to block devices and affects the operation of the BUILD BPB (Bios Parameter Block) device call. The implementation of all block devices is PC-DOS software and hardware compatible.

The IOCTL bit is meaningful for both types of devices. This bit tells MS-DOS whether the device can handle control strings with the IOCTL system call, Function 44H.

If a driver cannot process control strings, this bit is 0. MS-DOS returns an error if an attempt is made to handle control strings. A device which can process control strings sets the IOCTL bit to 1. For drivers of this type, MS-DOS calls IOCTL INPUT and OUTPUT device functions to send and receive IOCTL strings.

The IOCTL functions allow data outside of the normal user's reads and writes to be sent to the driver. The interpretation of this information is up to the driver.

Strategy and Interrupt Routines

These two fields are the entry points of the strategy and interrupt routines. They are word values and they must be in the same segment as the Device Header.

Name Field

This 8-byte field contains the name of a character device or the number of units of a block device. If it is a block device, the number of units can be put in the first byte. This is optional because MS-DOS fills in this location with the value returned by the driver's INIT code.

How to Create a Device Driver

To create a device driver, write a binary file with a Device Header at the beginning of the file. The code originates at 0.

MS-DOS always processes installable device drivers before handling the default devices. To install a new CON device, simply name the device CON. Remember to set the standard input device and standard output device bits in the attribute word on the new CON device. The scan of the device list stops on the first match, so the installable device driver takes precedence.

MS-DOS installs the driver anywhere in memory; therefore, be careful with memory references. Do not expect the driver to be loaded in the same place.

Installation of Device Drivers

MS-DOS allows new device drivers to be installed dynamically at boot time. This is accomplished by INIT code in the BIOS which processes the CONFIG.SYS file.

At load time, DOS searches the root directory for a file named CONFIG.SYS. Declare the files containing your device drivers using the DEVICE command.

DEVICE = [C:] [path] filename [.ext]

DOS loads your drivers as an extension of itself. Include a separate DEVICE command for each driver to be loaded.

Request Header

When MS-DOS calls a device driver to perform a function, it passes a Request Header in ES:BX to the strategy entry point. This is a fixed length header followed by data pertinent to the operation being performed. It is the device driver's responsibility to preserve the machine state. For example, save all registers on entry and restore them on exit. There is enough room on the stack when strategy or interrupt is called to do about 20 pushes. If more stack is needed, the driver sets up its own stack.

The following figure illustrates a Request Header.

REQUEST HEADER ->

BYTE length of record (the length in bytes of pertinent data, plus the length of this Request Header)
BYTE unit code The subunit the operation is for (minor device) (no meaning on character devices)
BYTE command code
WORD status
8 bytes RESERVED

Unit Code

If your device driver has three units, then the possible values of the unit code field are 0, 1, and 2.

Command Code Field

The command code field in the Request Header can have the following values:

Command Code	Function
0	INIT
1	MEDIA CHECK (Block only, NOP for character)
2	BUILD BPB ”
3	IOCTL INPUT (Only called if device has IOCTL)
4	INPUT (Read)
5	NON-DESTRUCTIVE INPUT NO WAIT (Char devices only)
6	INPUT STATUS ”
7	INPUT FLUSH ”
8	OUTPUT (Write)
9	OUTPUT (Write) with verify
10	OUTPUT STATUS ”
11	OUTPUT FLUSH ”
12	IOCTL OUTPUT (Only called if device has IOCTL)

Status Word

The following figure illustrates the status word in the Request Header.

15	14 13 12 11 10	9	8	7 6 5 4 3 2 1 0
E R R	RESERVED	B U S	D O N	ERROR CODE (bit 15 on)

The status word is set by the driver interrupt routine.

Bit 8 is the done bit. When set, it means the operation is complete.

Bit 15 is the error bit. If it is set, then the low 8 bits indicate the error. The errors are:

0 Write protect violation
1 Unknown Unit
2 Drive not ready
3 Unknown command
4 CRC error
5 Bad drive request structure length
6 Seek error
7 Unknown media
8 Sector not found
9 Printer out of paper
A Write fault
B Read Fault
C General failure

Bit 9 is the busy bit which is set only by status calls.

- For output on character devices:
  If bit 9 is 1 on return, a write request waits for completion of a current request. If it is 0, there is no current request and a write request starts immediately.

- For input on character devices with a buffer:
  If bit 9 is 1 on return, a read request goes to the physical device. If it is 0 on return, then there are characters in the device buffer and a read returns quickly. MS-DOS assumes all character devices have an input type-ahead buffer. Devices that do not have a type-ahead buffer return busy=0 so that MS-DOS does not wait for non-existent buffer input.

Media Check and Build BPB

MEDIA CHECK and BUILD BPB are used with block devices only. MS-DOS calls MEDIA CHECK first for a drive unit and passes its current media descriptor byte. MEDIA CHECK returns one of the following results:

- Media Not Changed — current DPB and media byte are OK.
- Media Changed — Current DPB and media are wrong. MS-DOS invalidates buffers for this unit and calls the device driver to build the BPB.
- Not Sure — If there are dirty buffers for this unit, MS-DOS assumes the DPB and media byte are OK. If nothing is dirty, MS-DOS assumes the media has changed. It invalidates buffers for the unit and calls the device driver to build the BPB.
- Error — If an error occurs, MS-DOS sets the error code.

MS-DOS calls BUILD BPB under the following conditions:

- If Media Changed is returned
- If Not Sure is returned and there are no dirty buffers

Init Routine

The Init Routine is called only once when the device is installed. It returns a location DS:DX which is a pointer to the first free byte of memory after the device driver. To save space, this pointer method can be used to delete initialization code that is only used once.

Additional information that block drivers return is:

- The number of units
- A pointer to a BPB
- The media decriptor

The number of units determines the logical device names. This mapping is determined by the position of the driver in the device list and by the number of units on the device.

BPB blocks are used to build an internal MS-DOS data structure for each of the units. The driver passes MS-DOS a pointer to an array of n word BPB pointers, where n is the number of units. If all units are the same, they can share a BPB to save space. This array must be before the free space pointer since MS-DOS builds an internal DOS structure starting at this free byte. The defined sector size must be less than or equal to the maximum sector size defined at INIT time; otherwise, the install fails.

The media descriptor byte means nothing to MS-DOS. It is passed to drivers so that they know what parameters MS-DOS is currently using for a drive unit.

Block devices are either dumb or smart. A dumb device defines a unit and an internal DOS structure for each possible media drive combination. For example, unit 0 = drive 0 single side, unit 1 = drive 0 double side. In this case, media descriptor bytes mean nothing.

A smart device allows multiple media per unit. In this case, the BPB table returned by INIT defines space large enough to accommodate the largest possible media supported. Smart drivers use the media descriptor byte to pass information about the media currently in a unit.

Function Call Parameters

Strategy routines are called with ES:BX pointing to the Request Header. The interrupt routines get the pointers to the Request Header from the queue that the strategy routines store them in. The command code in the Request Header tells the driver which function to perform.

All DWORD pointers are stored offset first, then segment.

INIT

Command code = 0

INIT - ES:BX ->

13-BYTE Request Header
BYTE # of units
DWORD break address
DWORD pointer to BPB array (Not set by character devices)

The number of units, break address, and BPB pointer are set by the driver. On entry, the DWORD points to the character after the '=' on the line in CONFIG.SYS. This allows drivers to scan the CONFIG.SYS invocation line for arguments.

If there are multiple device drivers in a single .COM file, the ending address returned by the last INIT is the one MS-DOS uses. All of the device drivers in a single .COM file return the same ending address.

Media Check

Command Code = 1

MEDIA CHECK - ES:BX ->

13-BYTE	Request Header
BYTE media descriptor from DPB	
BYTE returned	

In addition to setting the status word, the driver sets the return byte to one of the following:

-1 Media has been changed
0 Don't know if media has been changed
1 Media has not been changed

If the driver can return -1 or 1 because it has a door-lock or other interlock mechanism, MS-DOS performance is enhanced because MS-DOS does not need to reread the FAT for each directory access.

Build BPB (BIOS Parameter Block)

Command code = 2

BUILD BPB - ES:BX ->

13-BYTE Request Header
BYTE media descriptor from DPB
DWORD transfer address (Points to one sector worth of scratch space or first sector of FAT depending on the value of the NON PC-DOS FORMAT bit)
DWORD pointer to BPB

If the NON PC-DOS FORMAT bit is 1, then the DWORD transfer address points to a sector scratch buffer.

If the NON PC-DOS FORMAT bit is 0, then this buffer contains the first sector of the first FAT and the driver must not alter this buffer.

The first sector of the first FAT must be located in the same sector for all media. This is because the FAT sector is read BEFORE the media is actually determined. Use this mode to read the FAT ID byte.

In addition to setting status word, the driver must set the pointer to the BPB on return.

To allow different OEMs to read each other's disks, the information relating to the BPB for the media is kept in the boot sector of the media. The format of the boot sector is:

	3 BYTE near JUMP to boot code
	8 BYTES OEM name and version
	WORD bytes per sector
	BYTE sectors per allocation unit
B	WORD reserved sectors
P	BYTE number of FATs
B	WORD number of root dir entries
	WORD number of sectors in logical image
	BYTE media descriptor
	WORD number of FAT sectors
	WORD sectors per track
	WORD number of heads
	WORD number of hidden sectors

Sectors per track, number of heads, and number of hidden sectors are optional. They are intended to help the BIOS understand the media. Sectors per track may be redundant since it can be calculated from total size of the disk. Number of heads supports different multi-head drives with the same storage capacity but a different number of surfaces. Number of hidden sectors supports drive-partitioning schemes.

Media Descriptor Byte

The last two digits of the FAT ID byte are called the media descriptor byte. Currently, the media descriptor byte has been defined for a few media types.

Bit	Meaning	
0	1 = 2 sided;	0 = not 2 sided
1	1 = 8 sector;	0 = not 8 sector
2	1 = removable;	0 = not removable
3-6	must be set to 1	
7	1 = not 80 track;	0 = 80 track

5 1/4" disks:

FEh	160K	
FCh	180K	
FFh	320K	formatted single sided
FDh	360K	formatted single sided
7Dh	720K	

HDU: F8h

Although these media bytes map directly to FAT ID bytes which must be F8-FF, media bytes can, in general, be any value in the range 0-FF.

Read or Write

Command codes = 3,4,8,9, and 12

READ or WRITE - ES:BX (Including IOCTL) ->

13-BYTE Request Header
BYTE media descriptor from DPB
DWORD transfer address
WORD byte/sector count
WORD starting sector number (Ignored on character devices)

In addition to setting the status word, the driver must set the sector count to the actual number of sectors (or bytes) transferred. No error check is performed on an IOCTL I/O call. The driver must set the return sector (byte) count to the actual number of bytes transferred.

A user program can not request an I/O of more than FFFFH bytes and cannot wrap around in the transfer segment.

Non Destructive Read No Wait

Command code = 5

NON DESTRUCTIVE READ NO WAIT -
ES:BX ->

13-BYTE Request Header
BYTE read from device

If the character device returns busy bit = 0 (characters in buffer), then the next character that would be read is returned. This character is not removed from the input buffer, hence the term Non Destructive Read. Basically, this call allows MS-DOS to look ahead one input character.

Status

Command codes = 6 and 10

STATUS Calls - ES:BX ->

13-BYTE Request Header

The driver sets the status word and the busy bit as follows:

- For output on character devices:
  If bit 9 is 1 on return, a write request waits for completion of a current request. If it is 0, there is no current request and a write request starts immediately.

- For input on character devices with a buffer:
  A return of 1 means a read request goes to the physical device. If it is 0 on return, then there are characters in the device's buffer and a read returns quickly. A return of 0 also indicates that the user has typed something. MS-DOS assumes that all character devices have an input type-ahead buffer. Devices that do not have a type-ahead buffer return busy = 0 so that the DOS does not wait for something to get into a non-existent buffer.

Flush

Command codes = 7 and 11

FLUSH Calls - ES:BX ->

13-BYTE Request Header

The FLUSH call tells the driver to terminate all pending requests. This call is used to flush the input queue on character devices.

Clock Device

One of the special enhancements for the Safari-3 is the battery-backed 58174A clock-calender chip and related driver. This chip is integrated into the system as the CLOCK device and is accessed with the TIME and DATE command.

This CLOCK device defines and performs functions like any other character device. When a read or write to this device occurs, exactly 6 bytes are transferred. The first two bytes are the count of days since 1-1-80. The third byte is minutes, the fourth, hours, the fifth, hundredths of seconds, and the sixth, seconds.

Reading the CLOCK device gets the date and time; writing to it sets the date and time.

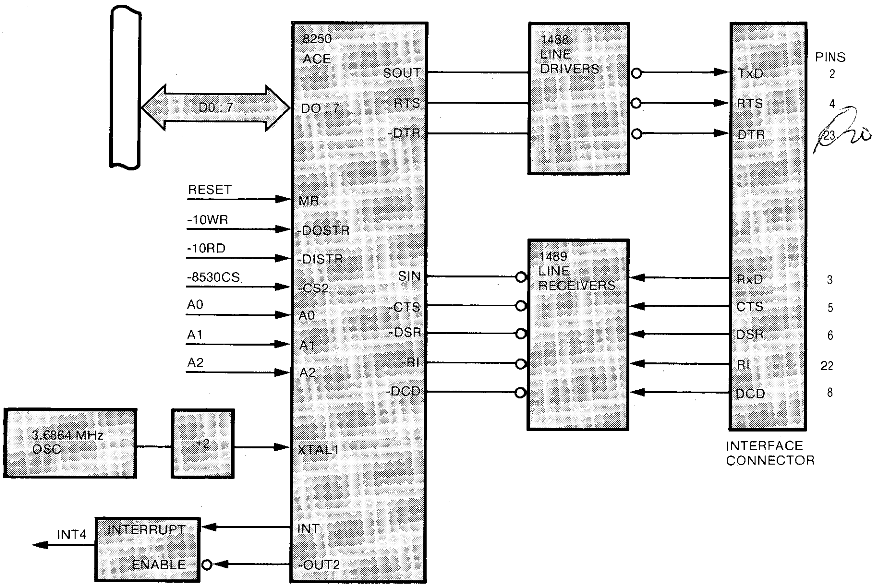
Asynchronous Communications Element

Functional Description

The Asynchronous Communications Element (ACE) performs serial-to-parallel conversion on input data characters received from a modem and parallel-to-serial conversion on output data characters received from the CPU. You can read the status of transfer operations at any time. This device gives you modem control capability.

The baud rate and serial interface characteristics are programmable. The ACE has a software-tailored interrupt system whose interrupt request is on INT4.

Block Diagram

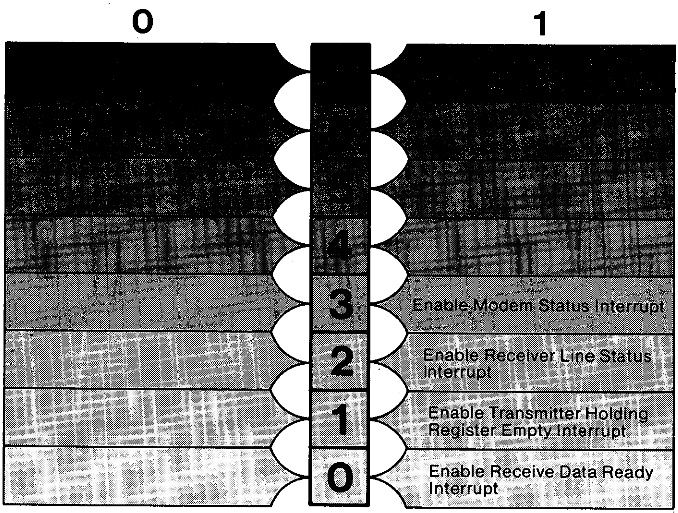


Registers

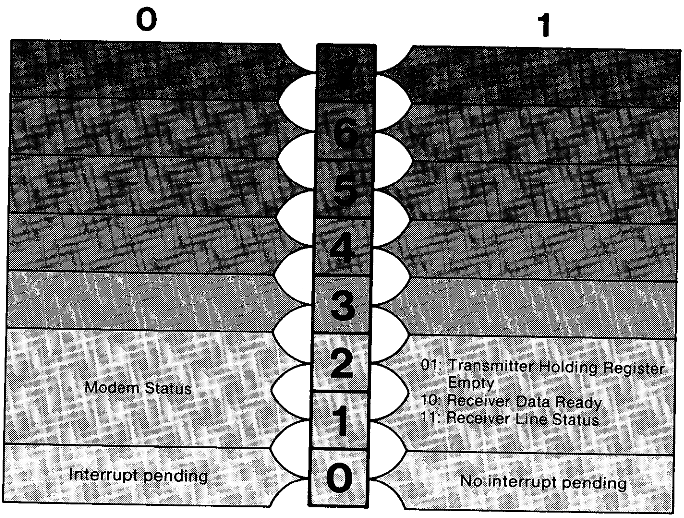
PORT A	PORT B	NAME	READ/ WRITE	DESCRIPTION
3F8	2F8	DATA	R/W W	Receive Buffer, Transmitter Holding Register
3F9	2F9	INTERRUPT ENABLE	W	See layout
3FA	2FA	INTERRUPT IDENTIFICATION	R	See layout
3FB	2FB	LINE CONTROL	R/W	See layout
3FC	2FC	MODEM CONTROL	R/W	See layout
3FD	2FD	LINE STATUS	R	See layout
3FE	2FE	MODEM STATUS	R/W	See layout
3FF	2FF	SCRATCH	R/W	Scratch pad register
3F8	2F8	DIVISOR LATCH	W	Least significant byte
3F9	2F9	DIVISOR LATCH	W	Most significant byte

Layout

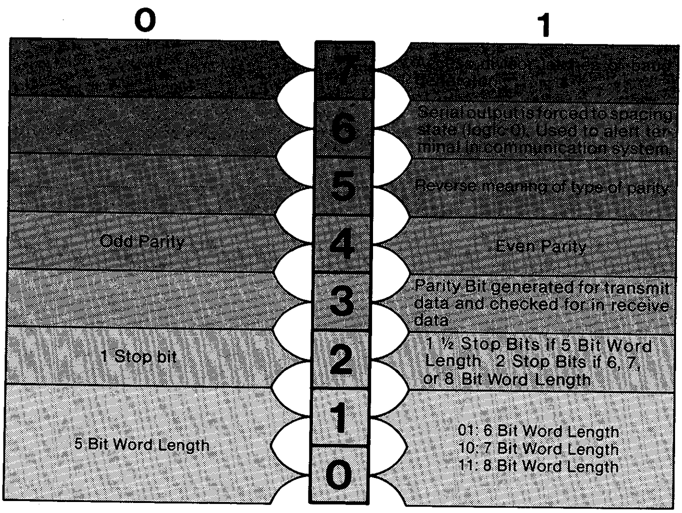
INTERRUPT ENABLE REGISTER



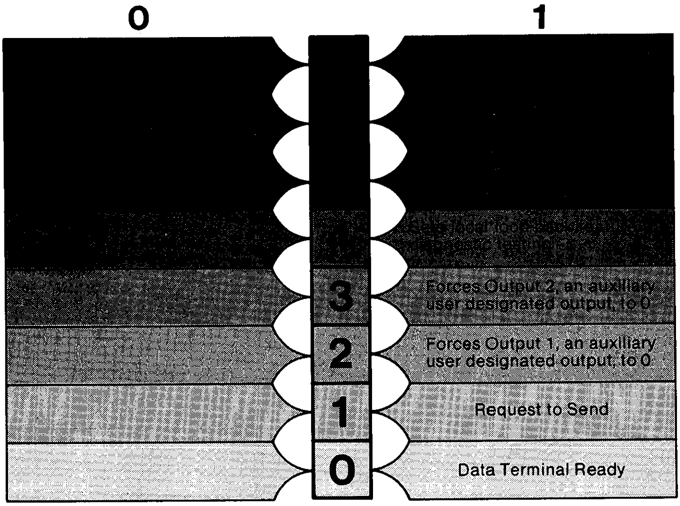
INTERRUPT IDENTIFICATION REGISTER



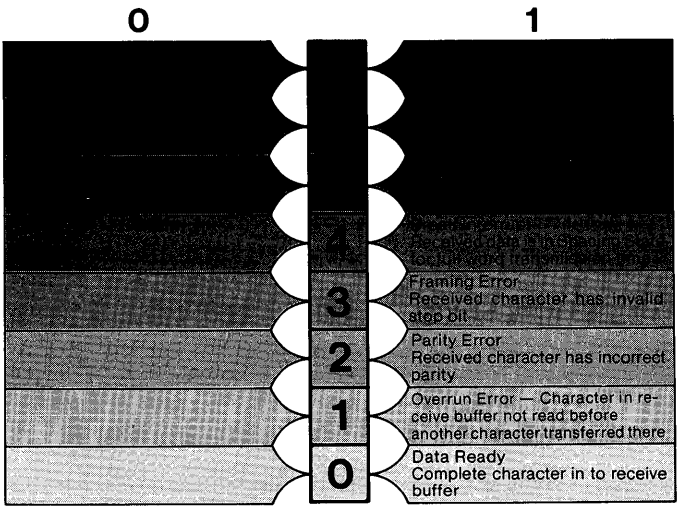
LINE CONTROL REGISTER



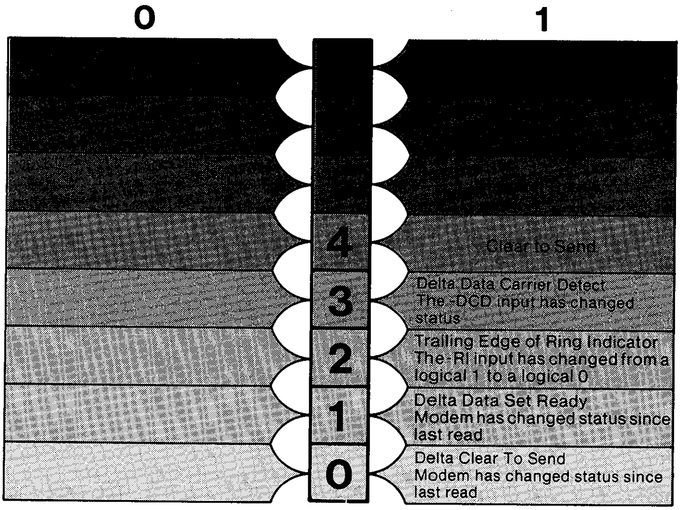
MODEM CONTROL REGISTER



LINE STATUS



MODEM STATUS REGISTER



Functions

- IDENTIFY INTERRUPTS
  INPUT: INTERRUPT IDENTIFICATION REGISTER
- READ LINE STATUS
  INPUT: LINE STATUS REGISTER
- READ MODEM STATUS
  INPUT: MODEM STATUS REGISTER
- RECEIVE CHARACTER
  INPUT: DATA
- SEND CHARACTER
  OUTPUT: DATA

- SET BAUD RATE
  To set the baud rate, you load the divisor which yields the correct rate (16 x divisor = clock frequency (1.8432 MHz)/baud rate x 16)

  OUTPUT: LINE CONTROL REGISTER
  BIT 7 = 1
  DIVISOR LATCH - Least significant byte
  DIVISOR LATCH - Most significant byte

- SET INTERRUPTS
  OUTPUT: INTERRUPT ENABLE REGISTER

- WRITE LINE CONTROL CHARACTERISTICS
  OUTPUT: LINE CONTROL REGISTER

- WRITE MODEM CONTROL CHARACTERISTICS
  OUTPUT: MODEM CONTROL REGISTER

Sequencing and Timing

To transmit a character, first issue a Request to Send and Data Terminal Ready to the Modem Control Register. Then wait for the Modem Status to have Data Set Ready and Clear to Send set. When the Transmitter Holding Register is empty as indicated in the Line Status Register, write the character to the data register.

To receive a character, set Data Terminal Ready in the Modem Control Register. Then wait for Data Set Ready in the Modem Status Register. When Data Ready in the Line Status Register is set, input the character from the data register.

The following table states the divisors to use to obtain a given baud rate.

BAUD RATES USING 1.8432 MHz CLOCK

BAUD RATE	DIVISOR
110	1047
150	768
300	384
600	192
1200	96
1800	64
2000	58
2400	48
3600	32
4800	24
7200	16
9600	12

If you wish to use the break control feature to alert a terminal in a communication system, the following sequence assures that no erroneous or extraneous characters are transmitted.

- Load an all 0's pad character into the transmitter holding register.
- Set break when the transmitter holding register is empty.
- Wait for transmitter empty (Bit 6 = 1 in Line Status Register) and clear break.

Note that the transmitter operates normally during a break sequence and can be used as a character timer to establish an accurate break length.

If the ACE is programmmed to interrupt, the interrupt is on INT4. The ACE acknowledges the highest priority interrupt as indicated in this chart. The Interrupt Identification Register states which interrupt is pending.

TYPE	SOURCE — RESET
Receiver Line Status	Overrun Error — Read Line Status Register
	Parity Error
	Framing Error
	Break Interrupt
Received Data Ready	Receive Data Ready — Read data
Transmitter Holding Register Empty	Transmitter Holding Register Empty — Write data
Modem Status	Clear To Send — Read Modem Status Register
	Data Set Ready
	Ring Indicator
	Data Carrier Detect

Sample Program

```
;This program sets the baud rate to 1200 baud
LINE_CTL   EQU 3FB
DVSR_L   EQU 3F8
DVSR_M   EQU 3F9

SET_BAUD:
   MOV AL,080H          ;access divisor
   MOV DX,LINE_CTL      ;latch
   OUT DX,AL

   MOV AX,096           ;divisor for 1200 baud
   MOV DX,DVSR_L
   OUT DX,AL            ;least significant byte

   MOV DX,DVSR_M
   MOV AL,AH
   OUT DX,AL            ;most significant byte

   XOR AL,AL            ;turn off access to latch
   MOV DX,LINE_CTL
   OUT DX,AL
   RET
```

DMA Controller

Functional Description

The DMA controller allows devices to transfer data directly to and from memory without CPU involvement. It has four channels.

Channel 0 has the highest priority and is used to refresh memory. The Interval Timer is programmed to periodically request a dummy DMA transfer. This creates a memory read cycle which refreshes memory.

Channel 1 is available on the I/O expansion bus to support high speed transfer between I/O devices and memory. Channel 3 has the lowest priority.

Channel 3 is dedicated to the hard disk controller. Channel 2 is dedicated to the floppy disk controller.

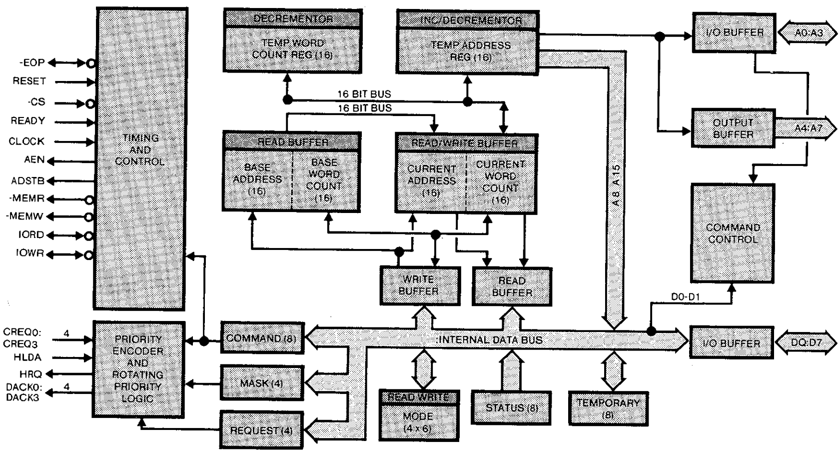
The DMA controller has four transfer modes. Single transfer mode makes only one transfer. Block transfer mode continues transferring until the count goes from 0 to FFFFH. Demand transfer allows transfers to continue until the I/O device has exhausted its capacity. The cascade mode allows more than one DMA controller to be used and is not applicable in the AT&T Personal Computer 6300.

When autoinitialize is requested, the original values of the Current Address and Current Count registers are restored at the end of the operation.

The DMA controller has two types of priority schemes. The fixed scheme bases the priority on the descending value of their numbers. In this scheme, Channel 3 has the lowest priority. The second scheme is rotating priority. The last channel to get service becomes the lowest priority channel.

Compressed Timing allows greater throughput by compressing the transfer time into two clock cycles.

Block Diagram



Registers

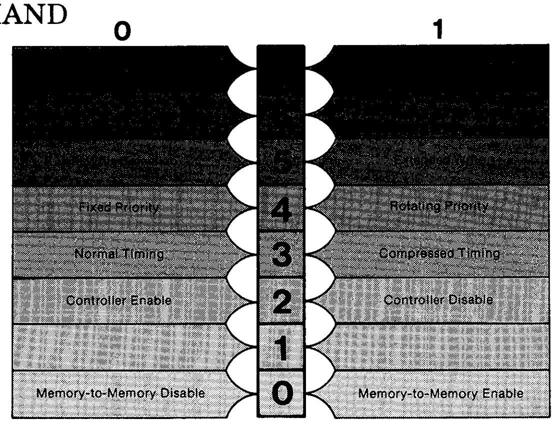
PORT	NAME	READ/ WRITE	DESCRIPTION
0	CHANNEL 0 ADDRESS	W	16 bit address
1	CHANNEL 0 COUNT	W	1's complement of # of bytes to transfer
0	CHANNEL 0 CURRENT ADDRESS	R	16 bit address
1	CHANNEL 0 CURRENT COUNT	R	1's complement of # of bytes to transfer
2	CHANNEL 1 ADDRESS	W	16 bit address
3	CHANNEL 1 COUNT	W	1's complement of # of bytes to transfer
2	CHANNEL 1 CURRENT ADDRESS	R	16 bit address
3	CHANNEL 1 CURRENT COUNT	R	1's complement of # of bytes to transfer
4	CHANNEL 2 ADDRESS	W	16 bit address
5	CHANNEL 2 COUNT	W	1's complement of # of bytes to transfer
4	CHANNEL 2 CURRENT ADDRESS	R	16 bit address
5	CHANNEL 2 CURRENT COUNT	R	1's complement of # of bytes to transfer
6	CHANNEL 3 ADDRESS	W	16 bit address
7	CHANNEL 3 COUNT	W	1's complement of # of bytes to transfer
6	CHANNEL 3 CURRENT ADDRESS	R	16 bit address
7	CHANNEL 3 CURRENT COUNT	R	1's complement of # of bytes to transfer
8	STATUS	R	See layout
8	COMMAND	W	See layout
9	REQUEST	W	See layout
A	SINGLE MASK	W	See layout
B	MODE	W	See layout
C	CLEAR FLIP/FLOP	W	Execute prior to read or write of address or count
D	TEMPORARY	R	Contains last byte of memory-to-memory transfer

Registers

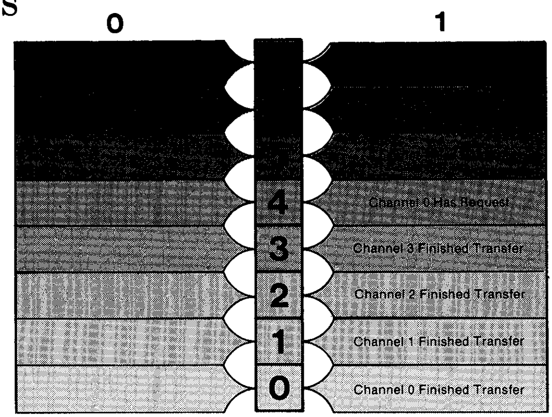
D	MASTER CLEAR	W	Any write clears controller
E	CLEAR MASK	W	Clear mask, all channels accept DMA commands
F	WRITE ALL MASK	W	See layout
80	CHANNEL 0 SEGMENT	W	Address segment nybble
82	CHANNEL 1 SEGMENT	W	Address segment nybble
81	CHANNEL 2 SEGMENT	W	Address segment nybble
83	CHANNEL 3 SEGMENT	W	Address segment nybble

Layout

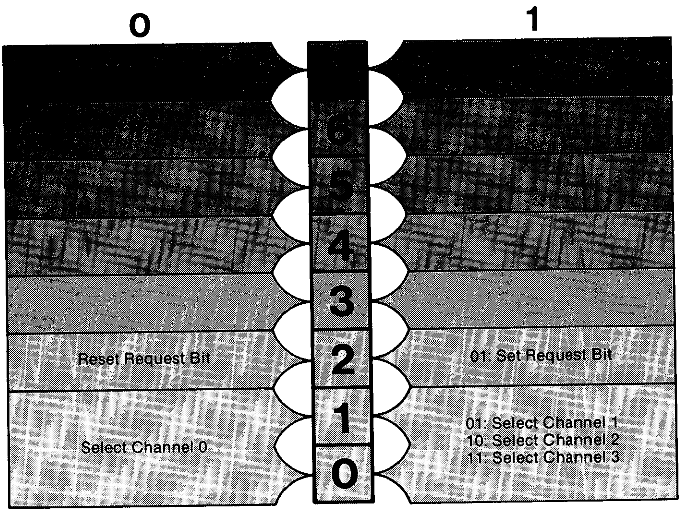
COMMAND



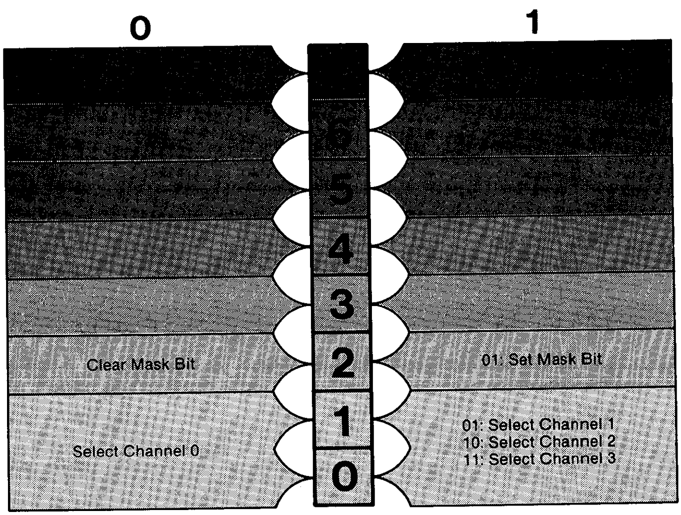
STATUS



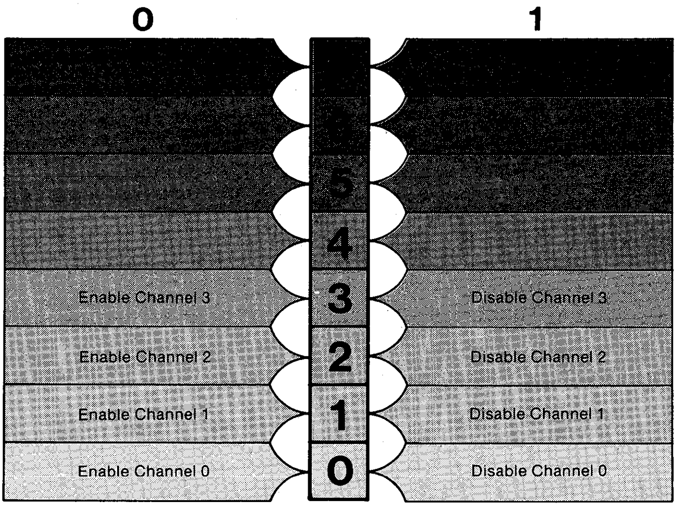
REQUEST REGISTER



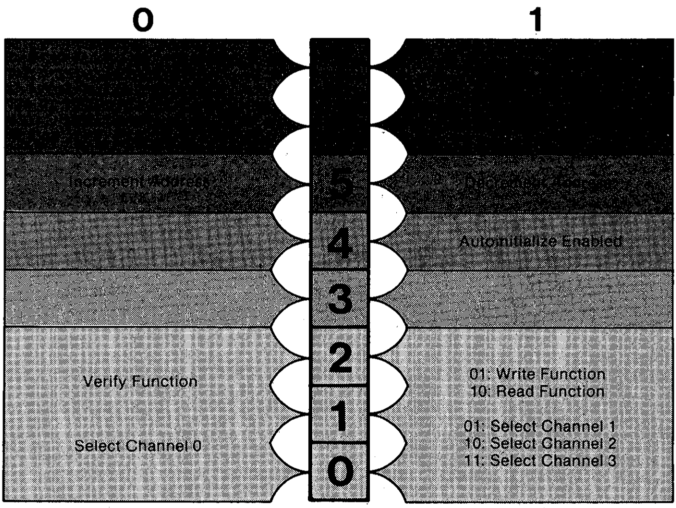
MARK REGISTER



WRITE ALL MASK REGISTER



MODE



Functions

- DISABLE CONTROLLER
  This function disables the controller.
  OUTPUT: COMMAND REGISTER
  BIT 2 = 1

- MASTER CLEAR CONTROLLER
  This function clears the controller. The Command, Status, Request, Temporary and Flip/Flop Registers are cleared.
  OUTPUT: MASTER CLEAR

- DMA READ, WRITE OR VERIFY
  This function sets up the controller to do the desired operation.
  OUTPUT: CLEAR FLIP/FLOP
  MODE REGISTER
  BITS 0-1 channel
  BITS 2-3 function
  BIT 4 autoinitialize
  BIT 5 address increment or decrement
  BITS 6-7 mode
  ADDRESS REGISTER
  16 BIT address, first output the LSB and then the MSB
  SEGMENT NYBBLE
  One nybble. The significant nybble of the segment register is in bits 12-15. Rotate to Bits 0-3 before output.
  COUNT REGISTER
  16 bit 1's complement of the # of bytes. First output the LSB and then the MSB.

- REQUEST DMA SERVICE
  This is a software request for DMA services.
  OUTPUT: REQUEST REGISTER

- READ STATUS
  This function reads the channel status.
  INPUT: STATUS REGISTER

- WRITE COMMAND REGISTER
  This function controls the operation of the DMA controller.
  OUTPUT: COMMAND REGISTER

- WRITE ALL MASK REGISTER
  This function enables and disables automatic DMA transfer for all channels.
  OUTPUT: WRITE ALL MASKS REGISTER

- WRITE MASK REGISTER
  This function enables and disables automatic DMA transfers for a channel.
  OUTPUT: MASK REGISTER

- WRITE MODE REGISTER
  This function specifies the mode for the specified channel.
  OUTPUT: MODE REGISTER

- CLEAR MASKS
  This function clears all the masks so that all channels accept DMA commands.
  OUTPUT: CLEAR MASK REGISTER

Sequencing and Timing

When the system is powered-up, it is recommended that all mode registers be set with valid data even if the channel is not used.

Before loading the address and count registers, disable the controller (BIT 2 of COMMAND REGISTER) or mask the channel. This prevents erroneous transfers before a complete address is loaded.

A write to the Clear Flip/Flop sets the controller so that an access to an address or count is to the upper and lower byte in the correct sequence.

Sample Program

```
;This program initializes an unused channel
;at start-up
COMMAND        EQU 8
MODE           EQU B
INIT_CHAN:
   MOV AL,04H            ;disable controller
   OUT COMMAND,AL
   MOV AL,041H           ;channel 1, verify, inc.
                         ;addr
   OUT MODE,AL           ;single mode
   RET                   ;now setup other
                         ;channels
```

Floppy Diskette Interface and Controller

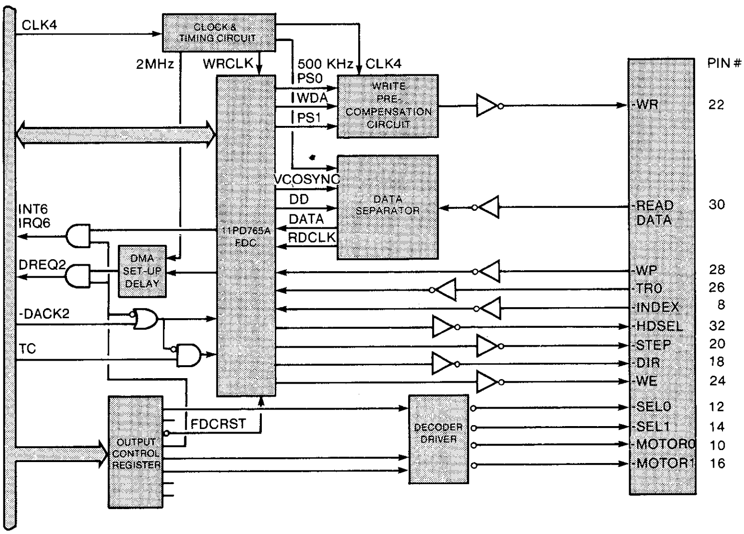
Functional Description

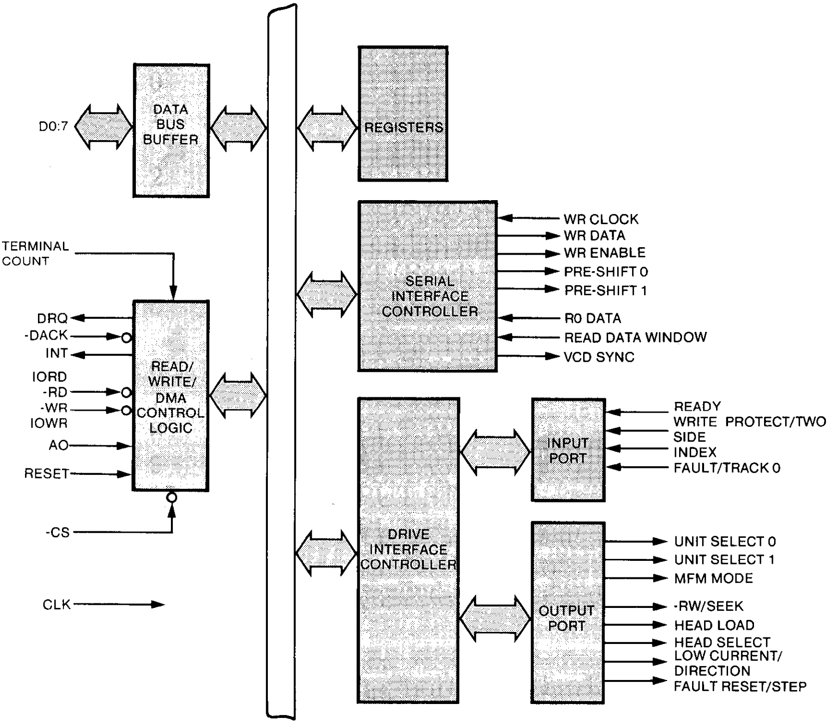
The diskette interface and NEC uPD765 controller read and write 5 1/4 inch diskettes on as many as two drives. Single density (FM) or double density (MFM) formats are supported. Single density diskettes contain 163,840 bytes and double density diskettes contain 327,680 bytes.

Each sector on the diskette contains an ID field and the Data field. The ID field contains the cylinder number, the head number, the sector number and the number of bytes per sector.

The diskette controller performs 15 separate functions. It operates in either DMA or non-DMA mode. Interrupts can be enabled on INT6.

Block Diagrams



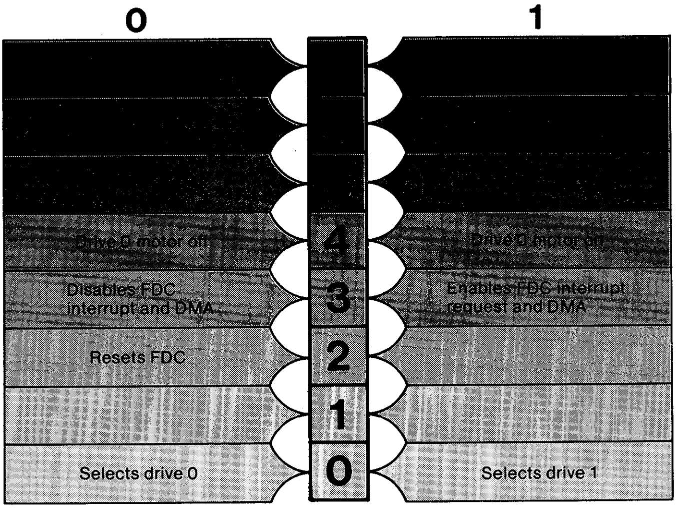


Registers

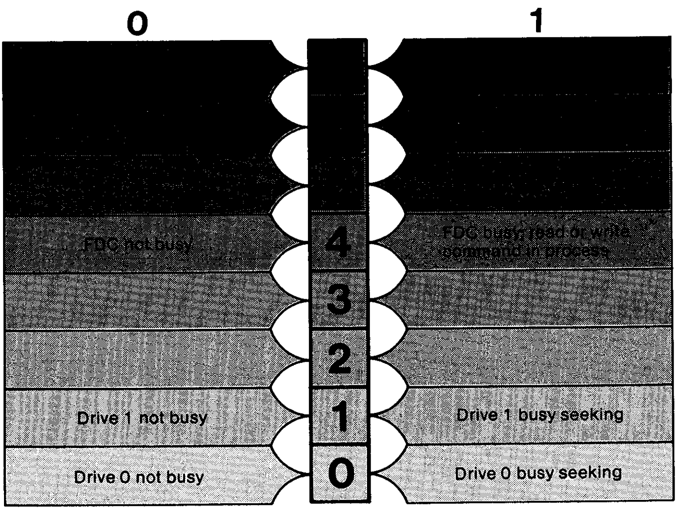
PORT	NAME	READ/ WRITE	DESCRIPTION
3F2	INTERFACE OUTPUT CONTROL	W	See layout
3F4	FDC MAIN STATUS REGISTER	R	See layout
3F5	FDC DATA	R/W	Transfers data, commands, parameters, and status

Layout

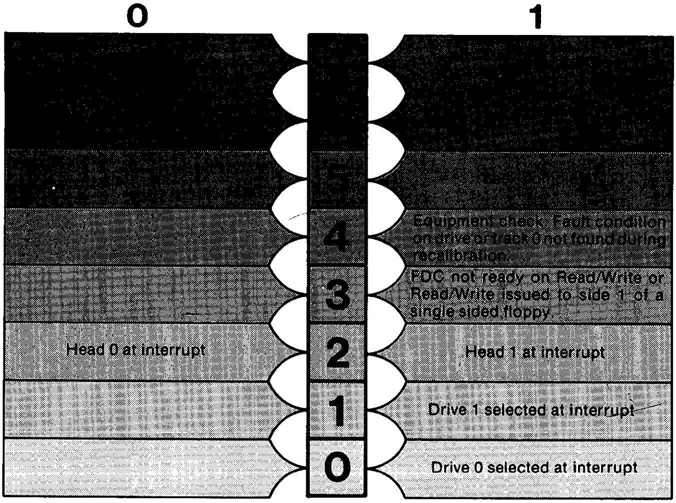
INTERFACE OUTPUT CONTROL



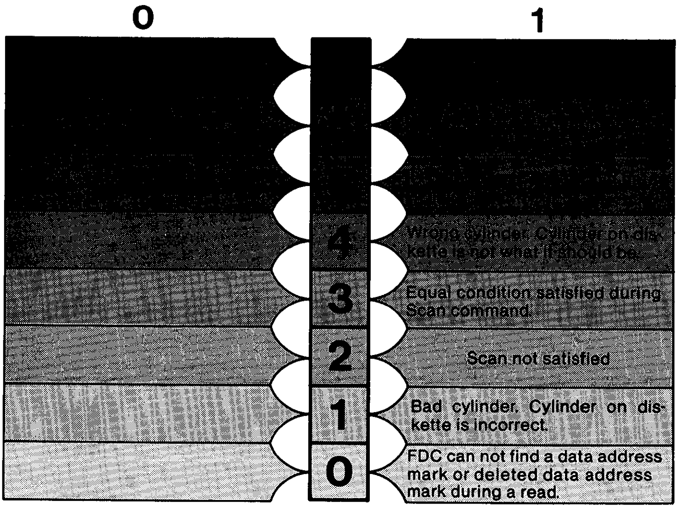
FDC MAIN STATUS REGISTER (MSR)



STATUS REGISTER 0



STATUS REGISTER 1



STATUS REGISTER 2

0	Bit	1
	[illegible]	
	5	[illegible]
[illegible]	5	Data error. CRC error in data field or ID field.
	4	Data overrun. FDC not serviced during data transfer within curtain time.
	3	No data. Data sector not found during read data, write deleted data or scan command. Failure to read sector ID during read ID command. Starting sector not found during read cylinder command.
	2	
	1	Write protect error.
	0	ID address mark not detected.

STATUS REGISTER 3

0	Bit	1
	[illegible]	
	[illegible]	[illegible]
	5	Drive ready
	4	Track 0
Side 0 Selected	3	Side 1 Selected
Head 0 selected	2	Head 1 selected
	1	Drive 1 selected
	0	Drive 0 selected

Parameters

SYMBOL	NAME	DESCRIPTION
DTL	Data Length	Only applies when there are 128 bytes per sector. If so, number of bytes to read or write. Otherwise, DTL = FF.
EOT	End of Track	Last sector number on cylinder. If there are 8 sectors per cylinder, then EOT = 8.
GPL	Gap Length	Gap 3 length between sectors. Different for format and read/write commands.
HD	Head	Selected Head number.
HLT	Head Load Time	4ms to 508ms in 4ms increments for 8Mhz clock. In this case, 4ms.
HUT	Head Unload Time	0ms to 480ms in 32ms increments for 8Mhz clock. This is the amount of time to wait after a read or write before the heads are unloaded. If a new command is issued quickly, this saves head load time.
MF	MF or MFM Mode	0 = MF, 1 = MFM
MT	Multi-track	MT = 1, multi-track operation. After completing an operation on side 0, the FDC continues on side 1.
N	Number of bytes/ sector	0 = 128 bytes, 1 = 256 bytes, 2 = 512 bytes 3 = 1024 bytes
ND	Non-DMA	0 = DMA mode, 1 = non-DMA mode
SRT	Step Rate Time	32ms to 2ms in 2 ms increments for 8Mhz clock. This is the amount of time to move the head from track to track. At 48 TPI, SRT = 6 ms. At 96 TPI, SRT = 4ms.
ST0-ST3	Status Registers	See layout
STP	Scan Test Flag	STP = 1, sector by sector compare STP = 2, alternate sectors
US0,US1	Unit Select	USx = 0, drive not selected USx = 1, drive selected

Functions • FORMAT A TRACK

This function formats an entire track. The ID Field for each sector is supplied by the programmer during the execution phase.

OUTPUT:

0	MF	0	0	1	1	0	1
0	0	0	0	0	HD	US1	US0

Number of bytes/sector
Sector/track
Gap Length
Filler Byte

EXECUTION: Sector ID Field transfer

INPUT: Status Register 0
Status Register 1
Status Register 2
Cylinder
Head
Sector
Number of bytes/sector

POSSIBLE ERRORS: Equipment check, and not ready

- READ DATA
  This function reads data from the diskette.

OUTPUT:

MT	MF	SK	0	0	1	1	0

0	0	0	0	0	HD	US1	US0

Cylinder
Head
Sector
Number of bytes/sector
End of Track
Gap Length
Data Length

EXECUTION: Data Transfer

INPUT: Status Register 0
Status Register 1
Status Register 2
Cylinder
Head
Sector
Number of bytes/sector

POSSIBLE ERRORS: No data, data error, data error in data field, and control mark

- READ DELETED DATA
  This function reads deleted data.

OUTPUT:

MT	MF	SK	0	1	1	0	0
0	0	0	0	0	HD	US1	US0

Cylinder
Head
Sector
Number of bytes/sector
End of Track
Gap Length
Data Length

EXECUTION: Data Transfer

INPUT:
Status Register 0
Status Register 1
Status Register 2
Cylinder
Head
Sector
Number of bytes/sector

- READ ID
  This function reads the first correct sector ID field.

OUTPUT:

0	MF	0	0	1	0	1	0
0	0	0	0	0	HD	US1	US0

EXECUTION:

INPUT:
Status Register 0
Status Register 1
Status Register 2
Cylinder
Head

	Sector
	Number of bytes/sector
POSSIBLE ERRORS:	Missing address mark and no data

- READ TRACK
  This function reads all data fields from the index hole to EOT. The FDC continues reading even if it finds a CRC error in the ID or data fields.

OUTPUT:

0	MF	SK	0	0	0	1	0
0	0	0	0	0	HD	US1	US0

	Cylinder
	Head
	Sector
	Number of bytes/sector
	End of Track
	Gap Length
	Data Length
EXECUTION:	Data transfer
INPUT:	Status Register 0
	Status Register 1
	Status Register 2
	Cylinder
	Head
	Sector
	Number of bytes/sector
POSSIBLE ERRORS:	No data, data error and missing address mark

- RECALIBRATE
  This function positions the head to head 0, cylinder or track 0.

OUTPUT:

0	0	0	0	0	1	1	1
0	0	0	0	0	0	US1	US0

EXECUTION: Head repositioned

POSSIBLE
ERRORS: Equipment check

- SCAN EQUAL
  This function scans for an equal data compare.

OUTPUT:

MT	MF	SK	1	0	0	0	1
0	0	0	0	0	HD	US1	US0

Cylinder
Head
Sector
Number of bytes/sector
End of track
Gap Length
Contiguous or alternate sectors

EXECUTION: Data transfer

INPUT:
Status Register 0
Status Register 1
Status Register 2
Cylinder
Head
Sector
Number of bytes/sector

- SCAN LOW OR EQUAL
  This function scans for a low or equal data compare.

OUTPUT:

MT	MF	SK	1	1	0	0	1
0	0	0	0	0	HD	US1	US0

Cylinder
Head
Sector
Number of bytes/sector
End of track
Gap Length
Contiguous or alternate sectors

EXECUTION: Data transfer

INPUT: Status Register 0
Status Register 1
Status Register 2
Cylinder
Head
Sector
Number of bytes/sector

- SCAN HIGH OR EQUAL
  This function scans for a high or equal data compare.

OUTPUT:

MT	MF	SK	1	1	1	0	1
0	0	0	0	0	HD	US1	US0

Cylinder
Head
Sector
Number of bytes/sector
End of track
Gap Length
Contiguous or alternate sectors

EXECUTION: Data transfer

INPUT: Status Register 0
Status Register 1
Status Register 2
Cylinder
Head
Sector
Number of bytes/sector

- SEEK
  This function positions the head at the requested cylinder.

OUTPUT:

0	0	0	0	1	1	1	1
0	0	0	0	0	HD	US1	US0

New cylinder number

EXECUTION: Heads repositioned

- SENSE DRIVE STATUS
  This function obtains the current drive status.

OUTPUT:

0	0	0	0	0	1	0	0
0	0	0	0	0	HD	US1	US0

INPUT: Status Register 0

- SPECIFY
  This function defines the drive parameters.

OUTPUT:

0	0	0	0	0	0	1	1
SRT				HUT			
HLT							ND

- WRITE DATA
  This function writes data.

OUTPUT:

MT	MF	0	0	0	1	0	1
0	0	0	0	0	HD	US1	US0

Cylinder
Head
Sector
Number of bytes/sector
End of Track
Gap Length
Data Length

EXECUTION: Data Transfer

INPUT: Status Register 0
Status Register 1
Status Register 2
Cylinder
Head
Sector
Number of bytes/sector

- WRITE DELETE DATA
  This function writes deleted data.

OUTPUT:

MT	MF	0	0	1	0	0	1
0	0	0	0	0	HD	US1	US0

Cylinder
Head
Sector
Number of bytes/sector
End of Track
Gap Length
Data Length

EXECUTION: Data Transfer

INPUT: Status Register 0
Status Register 1
Status Register 2
Cylinder
Head
Sector
Number of bytes/sector

Sequencing and Timing

There are three phases to each function:

- command - The programmer writes the required information to the FDC.
- execution - The FDC performs the operation.
- result - The programmer reads the FDC's status.

Before any data can be read or written to the FDC the Main Status Register (MSR) must be read to determine the status of Bit 6 and Bit 7. In the command phase, Bit 6 must be 0 and Bit 7 must be 1. In the result phase both bits must be 1. You must wait 12 usec after a data read or write before reading the MSR.

In the command phases, all output must be written. The same is true in the result stage. All status information must be read.

During the execution phase, the FDC operates in DMA mode or non-DMA mode. In DMA mode, there is one interrupt at the end of the phase. In non-DMA mode, there is an interrupt after the transfer of each byte. In the format command, the ID field information for all the sectors in a track is sent to the FDC (cylinder, head, sector and bytes/sector). In DMA mode, 4 DMA requests per sector are issued. In non-DMA mode, there are 4 interrupts per sector. If interrupts cannot be handled every 13 ms in MFM mode or every 27 ms in FM then the FDC is polled. When polling, Bit 7 in the MSR functions just like the interrupt.

When it is not executing a command, the FDC polls the drives looking for a change in drive ready. If there is a change, the FDC interrupts. You can determine the cause of the unexpected interrupt with the Sense Drive Status function.

The drive motors should be off when the drives are not in use. However, they must be on prior to a drive select.

During the execution phase of read and write commands, the following occurs:

- The heads are loaded if unloaded.
- The FDC waits for the head settle time to elapse.
- The FDC begins reading the ID address marks and ID field.
- When the requested sector number compares with the one on the diskette, the transfer begins.
- After completion of the transfer, the FDC waits the head unload time before unloading the heads.

The amount of data that can be transferred in one instruction depends on MT, MF, and N.

Multi-Track MT	MFM/FM MF	Bytes/Sector N	Maximum Transfer (Bytes/Sector)(Number of Sectors)
0	0	00	128*26 = 3,328
0	1	01	256*26 = 6,656
1	0	00	128*52 = 6,656
1	1	01	256*52 = 13,312
0	0	01	256*15 = 3,840
0	1	02	512*15 = 7,680
1	0	01	256*30 = 7,680
1	1	02	512*30 = 15,360
0	0	02	512*8 = 4,096
0	1	03	1024*8 = 8,192
1	0	02	512*16 = 8,192
1	1	03	1024*16 = 16,384

If a read or write terminates on error, then the values for cylinder, head, sector, and number of bytes per cylinder depends on the state of MT and EOT.

MT	HD	LAST SECTOR TRANSFERRED EOT	ID INFORMATION IN RESULTS C	H	S	N
0	0	Less than EOT	NC	NC	S+1	NC
0	0	Equal to EOT	C+1	NC	S=1	NC
0	1	Less than EOT	NC	NC	S+1	NC
0	1	Equal to EOT	C+1	NC	S=1	NC
1	0	Less than EOT	NC	NC	S+1	NC
1	0	Equal to EOT	NC	LSB	S=1	NC
1	1	Less than EOT	NC	NC	S+1	NC
1	1	Equal to EOT	C+1	LSB	S=1	NC

NC = No Change
LSB = Least Significant Bit

The Write Deleted Data is the same as Write Data except that the FDC writes a Deleted Data Address mark at the beginning of the Data Field instead of the normal Data Address Mark. When reading deleted data, the FDC sets the CM error in Status Register 2 and reads the data. A Read Data would not read the data. If SK = 1, then the FDC skips the sector with the Deleted Data Address mark and reads the next one.

The Gap Length is different for read, write, and format commands. This table suggests appropriate values.

FORMAT	SECTOR SIZE	N	SC	GPL(1)	GPL(2)
FM	128	00	12	07	09
FM	128	00	10	10	19
FM	256	01	08	18	30
FM	512	02	04	46	87
FM	1024	03	02	C8	FF
FM	2048	04	01	C8	FF
MFM	256	01	12	0A	0C
MFM	256	01	10	20	32
MFM	512	02	08	2A	50
MFM	1024	03	04	80	F0
MFM	2048	04	02	C8	FF
MFM	4096	05	01	C8	FF

GPL(1) - Suggested GPL in read and write commands
GPL(2) - Suggested GPL in format commands

The scan commands terminate when a scan condition is met, last sector on the track is reached, or a terminal count is received. The DMA issues the terminal count when it has no more data to send. This chart determines the result of the scan.

COMMAND	STATUS REGISTER 2 BIT 2	BIT 3	COMMENT
SCAN EQUAL	0	1	DISKETTE DATA = PROCESSOR DATA
SCAN EQUAL	1	0	DISKETTE DATA >< PROCESSOR DATA
SCAN LOW OR EQUAL	0	1	DISKETTE DATA = PROCESSOR DATA
SCAN LOW OR EQUAL	0	0	DISKETTE DATA < PROCESSOR DATA
SCAN LOW OR EQUAL	1	0	DISKETTE DATA > PROCESSOR DATA
SCAN HIGH OR EQUAL	0	1	DISKETTE DATA = PROCESSOR DATA
SCAN HIGH OR EQUAL	0	0	DISKETTE DATA > PROCESSOR DATA
SCAN HIGH OR EQUAL	1	0	DISKETTE DATA < PROCESSOR DATA

Scans allow the compare to be on contiguous sectors (STP = 1) or alternate sectors (STP = 2). However, for normal termination of the command the last sector on the track must be compared.

When a seek is requested, the FDC checks its current position and decides in which direction to move. Then step pulses are issued to move the heads. The speed of the pulse is controlled by the Step Rate Time in the Specify function. While the drive is seeking, the seek bit in the MSR is set. It must be cleared by Sense Interrupt Status at the completion interrupt. While a drive is seeking, the FDC is not busy. Another seek command to the other drive can be requested.

Interrupts occur as the result of:

1) Entering Result Phase of:
   Read Data
   Read Track
   Read Deleted Data
   Write Data
   Write Deleted Data
   Format Track
   Scans
2) The execution phase in non-DMA mode
3) The Drive Ready line changing state
4) The end of Seek or Recalibrate

When the latter two occur, a Sense Drive Status determines the cause of the interrupt. It is mandatory to follow Seek and Recalibrate functions with a Sense Drive Status. This chart shows how to interpret the results of a Sense Drive Status.

STATUS REGISTER 0			CAUSE
BIT 5	BIT 6	BIT 7	
0	1	1	Ready line changed state
1	0	0	Normal termination of Seek or Recalibrate
1	1	0	Abnormal termination of Seek or Recalibrate

The Specify command defines internal timers. Head Unload Time (HUT) is programmable from 32ms to 480ms in increments of 32ms. Therefore 1 = 32ms, 2 = 64ms, and F = 480ms. The Step Rate Time is programmable from 2ms to 32ms in increments of 2ms. In this case, F = 2ms, E = 4ms, and 1 = 32ms. The Head Load Time is programmable from 4ms to 508ms in increments of 4ms. In this case, 1 = 4ms, 2 = 8ms, 7F = 508ms.

Sample Program

```
;
; GET_RESULTS
;       This subroutine obtains a variable amount
;       of status information in the result phase.
; INPUT:  ES:DI points to the area that receives
;       the status bytes
;
NEC_STATUS        EQU 3F4

GET_RESULTS:
   MOV CX,7                 ;max. bytes in status

GET1:
   MOV DX, NEC_STATUS       ;port address of MSR
   IN AL,DX                 ;get MSR
   TEST AL,080H             ;Data register ready to
                            ;send or receive
   JZ GET1                  ;jump if not ready yet
   TEST AL,40H              ;direction bit
   JZ GET2                  ;jump if wrong
                            ;direction
   INC DX                   ;port addr of data
                            ;register
   IN AL,DX                 ;get one byte of status
   STOSB                    ;move it to status area
   DEC CX                   ;maximum number
   JNZ GET1                 ;jump if not max yet

GET2:RET
```

Hard Disk Controller

Functional Description

The DTC-5150BX hard disk controller reads and writes to a maximum of two standard 5 1/4" Winchester disk drives. A sector size of 256, 512, or 1024 bytes is selectable. The sectors can be interleaved in 16 different ways.

The hard disk controller operates in DMA or non-DMA mode. Interrupts can be enabled on INT5.

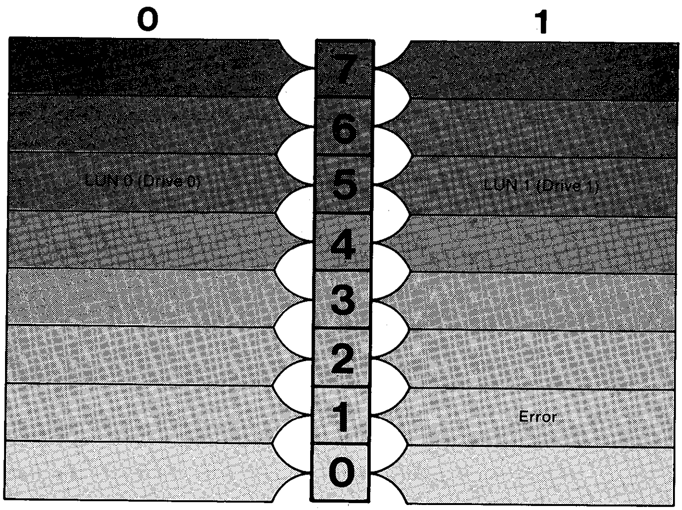
Extensive diagnostics are implemented. If a correctable data error is discovered, the error is automatically corrected using ECC.

Registers

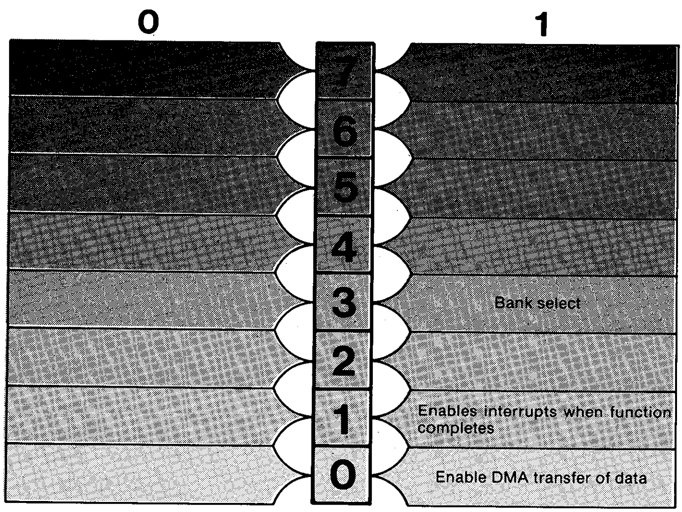
PORT #	NAME	READ/ WRITE	DESCRIPTION
320	COMPLETION STATUS REGISTER	R	See layout
320	DATA	R/W	Transfers data, function bytes, and controller sense bytes. See layout.
321	RESET CONTROLLER	W	Initialize Controller
321	STATUS	R	See layout
322	SELECT CONTROLLER	W	Select Controller
322	DRIVE TYPE	R	See layout
323	CONTROL REGISTER	W	See layout

Layout

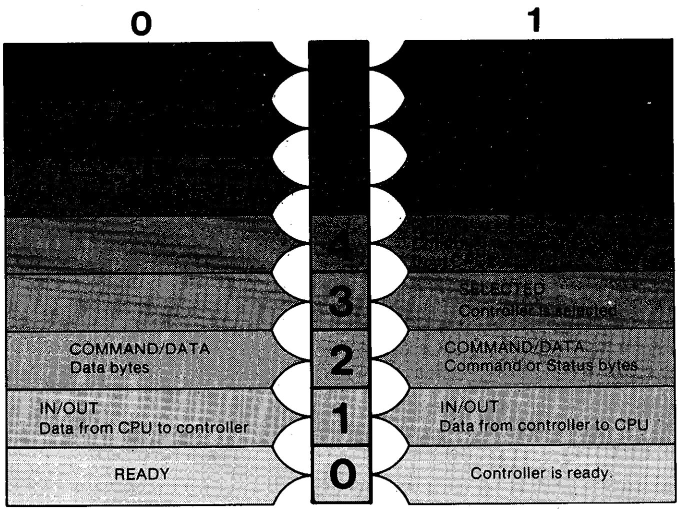
COMPLETION STATUS REGISTER



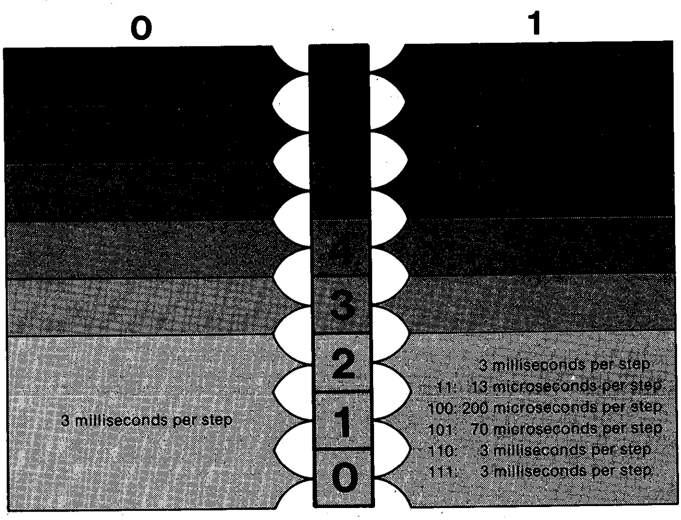
CONTROL REGISTER



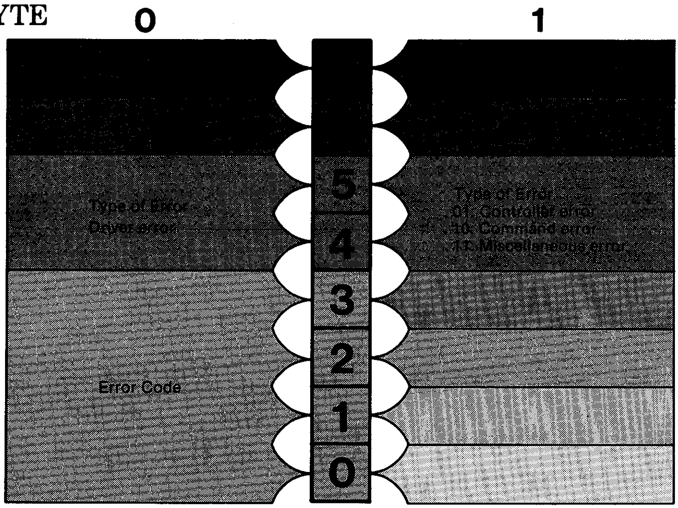
STATUS REGISTER



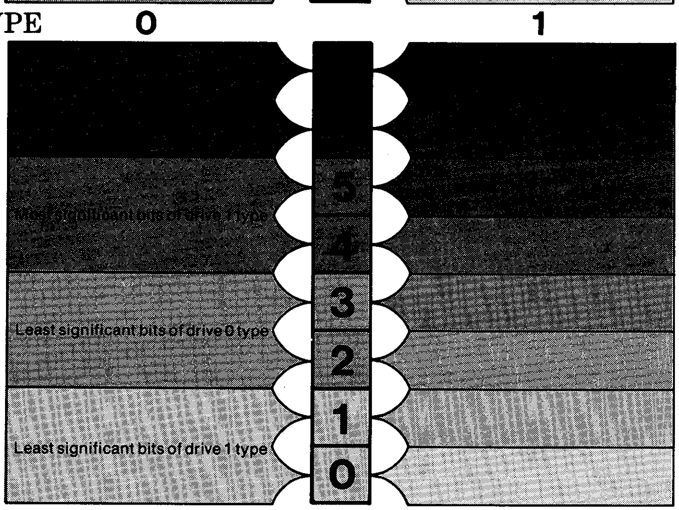
CONTROL COMMAND



SENSE BYTE



DRIVE TYPE



NUMBER	DRIVE TYPE
0	5 MB
1	24 MB
2	15 MB
3	10 MB
4	SQ306
5	CDC Wren

Error Codes

TYPE	CODE	DESCRIPTION
0	0	No error status
0	1	No index signal
0	2	No seek complete within 1.0 seconds
0	3	Write fault
0	4	Drive not ready
0	6	No track 0
0	8	Seek in progress
1	0	ID read error. ECC error in the ID field.
1	1	Uncorrectable data error during a read
1	2	Address Mark not found
1	4	Record not found. Found correct cylinder and head.
1	5	Seek error. Read/Write head positioned on wrong cylinder and/or wrong head selected.
1	8	Correctable data field error
1	9	Bad sector found
1	A	Format error. An unexpected format discovered during the Check Track function.
1	C	Unable to read the alternate track address
1	E	Attempted to directly access and alternate track
2	0	Invalid function
2	1	Illegal disk address. Address is beyond maximum.
3	0	RAM error. Data error detected during RAM diagnostic
3	1	Program Memory Checksum error
3	2	ECC Polynominal error

Function Parameters

NAME	DESCRIPTION
CONTROL COMMAND	Tells controller how to react to an error condition and defines step mode. See layout.
CYLINDER LOW	Eight least significant bits of the cylinder number
CYLINDER HIGH	Two most significant bits of the cylinder number
ECC0,ECC1,ECC2	ECC bytes of sector. ECC0 is least significant byte.
HEAD #	Head number
LUN #	Logical Unit Number. Winchester Drive 1 = Lun 0, Winchester Drive 2 = Lun 1.
SECTOR #	Sector number
SENSE BYTE	Gives detailed error information. See layout.
TYPE	0 = good track, 1 = alternate track, 2 = bad track, 3 = alternate bad track

Functions

- ASSIGN ALTERNATE TRACK
  This function formats the primary track specified in the function block with the alternated and bad track flags set in the ID fields and with the track address of the alternate track written in the data fields. The data field is written with the data in the sector buffer.

  Future read/write accesses to the primary track cause the drive to seek to the alternate track and to perform the operation there. This is transparent to the software. Alternate tracks can be assigned once. An alternate track cannot point to another alternate track.

OUTPUT:

<table>
<tr><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="8" align="right">11</td></tr>
<tr><td colspan="3">LUN</td><td colspan="5">PRIMARY HEAD #</td></tr>
<tr><td colspan="2">CYL HI</td><td colspan="6">0</td></tr>
<tr><td colspan="8">PRIMARY CYLINDER LOW</td></tr>
<tr><td colspan="8">INTERLEAVE</td></tr>
<tr><td colspan="8">CONTROL</td></tr>
</table>

DATA
OUTPUT:

<table>
<tr><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>0</td><td>0</td><td>0</td><td colspan="5">SECONDARY HEAD #</td></tr>
<tr><td colspan="2">CYL HI</td><td colspan="6">0</td></tr>
<tr><td colspan="8">SECONDARY CYLINDER LOW</td></tr>
<tr><td colspan="8">0</td></tr>
</table>

- CHECK TRACK
  This function checks the track format on the specified track for the correctness of the ID fields and the interleave of the sectors. It does not read the data.

OUTPUT:

7	6	5	4	3	2	1	0
10							
LUN			HEAD #				
CYL HI		0					
CYLINDER LOW							
INTERLEAVE							
CONTROL							

- CONTROLLER INTERNAL DIAGNOSTICS
  This function performs the controller internal diagnostics. The controller checks the internal processor, data buffer, ECC circuit and the checksum.

OUTPUT:

7	6	5	4	3	2	1	0
E4							
0							
0							
0							
0							
0							

- COPY
  This function transfers the data blocks from the source unit to the destination unit. The number of sectors to copy is specified in the number of blocks field. If the field is zero, 15,777,216 sectors are copied.

OUTPUT:

7	6	5	4	3	2	1	0
A0							
LUN/S			HEAD #/S				
CYL HI/S		SECTOR #/S					
CYLINDER LOW/S							
LUN/D			HEAD #/D				
CYL HI/D		SECTOR #/D					
CYLINDER LOW/D							
# OF BLOCKS 2							
# OF BLOCKS 1							
# OF BLOCKS 0							
0							
CONTROL							

S = Source D = Destination

- DRIVE DIAGNOSTIC
  This function performs a diagnostic on the specified unit. It reads sector 0 on sequential tracks and then reads sector 0 on 256 random tracks.

OUTPUT:

<table>
<tr><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="8" align="right">E3</td></tr>
<tr><td colspan="3">LUN</td><td colspan="5">0</td></tr>
<tr><td colspan="8">0</td></tr>
<tr><td colspan="8">0</td></tr>
<tr><td colspan="8">0</td></tr>
<tr><td colspan="8">CONTROL</td></tr>
</table>

- FORMAT BAD TRACK
  This function formats the track with the bad block flag set in all ID fields. It fills the data field with the data pattern in the sector buffer. The interleave must be the same for the entire drive.

OUTPUT:

<table>
<tr><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="8" align="right">7</td></tr>
<tr><td colspan="3">LUN</td><td colspan="5">HEAD #</td></tr>
<tr><td colspan="2">CYL HI</td><td colspan="6">0</td></tr>
<tr><td colspan="8">CYLINDER LOW</td></tr>
<tr><td colspan="8">INTERLEAVE</td></tr>
<tr><td colspan="8">CONTROL</td></tr>
</table>

- FORMAT TRACK
This function formats the specified track with no flags set in the ID fields. It fills the data field with the data pattern in the sector buffer. The interleave must be the same for the entire drive.

OUTPUT:

<table>
<tr><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="8" align="right">6</td></tr>
<tr><td colspan="3">LUN</td><td colspan="5">HEAD #</td></tr>
<tr><td colspan="3">CYL HI</td><td colspan="5">0</td></tr>
<tr><td colspan="8">CYLINDER LOW</td></tr>
<tr><td colspan="8">INTERLEAVE</td></tr>
<tr><td colspan="8">CONTROL</td></tr>
</table>

- FORMAT DRIVE
This function formats all of the tracks starting with the one specified in the function block to the end of the drive. The selected track format is used. The sectors are placed on the tracks according to the interleave code. The data fields are filled with the data pattern from the sector buffer.

OUTPUT:

<table>
<tr><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="8" align="right">4</td></tr>
<tr><td colspan="3">LUN</td><td colspan="5">HEAD #</td></tr>
<tr><td colspan="3">CYL HI</td><td colspan="5">0</td></tr>
<tr><td colspan="8">CYLINDER LOW</td></tr>
<tr><td colspan="8">INTERLEAVE</td></tr>
<tr><td colspan="8">CONTROL</td></tr>
</table>

- INITIALIZE DRIVE CHARACTERISTICS
  This function sets up the drive with different capacities and characteristics.

OUTPUT:

7	6	5	4	3	2	1	0
							C
LUN			0				
0							
0							
0							
0							

DATA
OUTPUT:

MAX # OF CYLINDERS HIGH
MAX # OF CYLINDERS LOW
MAX # OF HEADS
REDUCED WR. CUR. CYLINDER HIGH
REDUCED WR. CUR. CYLINDER LOW
WRITE PRECOMP. CYLINDER HIGH
WRITE PRECOMP. CYLINDER LOW
MAX ECC DATA BURST LENGTH

- RAM DIAGNOSTIC
  This function performs a data pattern test on the controller RAM.

  OUTPUT:

<table>
<tr><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="8" align="right">E0</td></tr>
<tr><td colspan="8" align="center">0</td></tr>
<tr><td colspan="8" align="center">0</td></tr>
<tr><td colspan="8" align="center">0</td></tr>
<tr><td colspan="8" align="center">0</td></tr>
<tr><td colspan="8" align="center">0</td></tr>
</table>

- READ
  This function reads the specified number of blocks. The function specifies the initial sector address. The data is transferred to the CPU.

  OUTPUT:

<table>
<tr><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="8" align="right">8</td></tr>
<tr><td colspan="3" align="center">LUN</td><td colspan="5" align="center">HEAD #</td></tr>
<tr><td colspan="2" align="center">CYL HI</td><td colspan="6" align="center">SECTOR #</td></tr>
<tr><td colspan="8" align="center">CYLINDER LOW</td></tr>
<tr><td colspan="8" align="center"># OF BLOCKS</td></tr>
<tr><td colspan="8" align="center">CONTROL</td></tr>
</table>

- READ ECC BURST ERROR LENGTH
  This function transfers one byte of data to the CPU. This byte contains the ECC burst length that the controller detected for the correctable ECC data error during the last read function.

OUTPUT:

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

DATA
INPUT:

ECC BURST LENGTH

- READ ID
  This function reads three bytes and three ECC bytes from the specified sector address given in the function block and transfers them to the CPU.

OUTPUT:

<table>
<tr><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="8">E2</td></tr>
<tr><td colspan="3">LUN</td><td colspan="5">HEAD #</td></tr>
<tr><td colspan="2">CYL HI</td><td colspan="6">SECTOR #</td></tr>
<tr><td colspan="8">CYLINDER LOW</td></tr>
<tr><td colspan="8">INTERLEAVE</td></tr>
<tr><td colspan="8">CONTROL</td></tr>
</table>

DATA
INPUT:

<table>
<tr><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="8">CYLINDER LOW</td></tr>
<tr><td colspan="2">TYPE</td><td colspan="3">CYL HI</td><td colspan="3">HEAD #</td></tr>
<tr><td colspan="8">SECTOR #</td></tr>
<tr><td colspan="8">ECC 2</td></tr>
<tr><td colspan="8">ECC 1</td></tr>
<tr><td colspan="8">ECC 0</td></tr>
</table>

- READ LONG
  This function reads sectors of data and ECC bytes from the disk and transfers them to the CPU. If an ECC error occurs during the read, the controller does not attempt to correct the data.

  OUTPUT:

<table>
<tr><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="8">E5</td></tr>
<tr><td colspan="3">LUN</td><td colspan="5">HEAD #</td></tr>
<tr><td colspan="2">CYL HI</td><td colspan="6">SECTOR #</td></tr>
<tr><td colspan="8">CYLINDER LOW</td></tr>
<tr><td colspan="8"># OF BLOCKS</td></tr>
<tr><td colspan="8">CONTROL</td></tr>
</table>

  DATA
  INPUT:

<table>
<tr><td>256/512/1024
BYTES OF DATA</td><td>E
C
2</td><td>E
C
1</td><td>E
C
0</td><td>0
0</td></tr>
</table>

- READ SECTOR BUFFER
  This function reads one sector from the controller sector buffer. No data transfer occurs between the controller and the drives.

OUTPUT:

<table>
<tr><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="8" align="right">E</td></tr>
<tr><td colspan="8" align="center">0</td></tr>
<tr><td colspan="8" align="center">0</td></tr>
<tr><td colspan="8" align="center">0</td></tr>
<tr><td colspan="8" align="center">0</td></tr>
<tr><td colspan="8" align="center">0</td></tr>
</table>

- READ VERIFY
  This function reads the specified number of blocks but does not transfer the data to the CPU. The function specifies the sector number where verification begins.

  OUTPUT:

<table>
<tr><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="8" align="right">5</td></tr>
<tr><td colspan="3" align="center">LUN</td><td colspan="5" align="center">HEAD #</td></tr>
<tr><td colspan="3" align="center">CYL HI</td><td colspan="5" align="center">SECTOR #</td></tr>
<tr><td colspan="8" align="center">CYLINDER LOW</td></tr>
<tr><td colspan="8" align="center"># OF BLOCKS</td></tr>
<tr><td colspan="8" align="center">CONTROL</td></tr>
</table>

- RECALIBRATE
  This function positions the read/write arm at track 0 and clears errors in the drive.

OUTPUT:

7	6	5	4	3	2	1	0
1							
LUN			0				
0							
0							
0							
CONTROL							

- REQUEST LOGOUT
  This function retrieves the four bytes of error log for the specified unit. Each device has its own error log which is incremented every time certain errors occur and is cleared after this function is executed.

OUTPUT:

7	6	5	4	3	2	1	0
E7							
LUN			0				
0							
0							
0							
0							

DATA
INPUT:

RETRY COUNT HIGH
RETRY COUNT LOW
PERMANENT ERROR HIGH
PERMANENT ERROR LOW

- REQUEST SENSE
  This function sends the four Sense Bytes to the CPU as data.

OUTPUT:

<table>
<tr><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="8" align="right">3</td></tr>
<tr><td colspan="3">LUN</td><td colspan="5">0</td></tr>
<tr><td colspan="8" align="right">0</td></tr>
<tr><td colspan="8" align="right">0</td></tr>
<tr><td colspan="8" align="right">0</td></tr>
<tr><td colspan="8" align="right">0</td></tr>
</table>

DATA
INPUT:

<table>
<tr><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="8">SENSE BYTE</td></tr>
<tr><td colspan="3">LUN</td><td colspan="5">HEAD #</td></tr>
<tr><td colspan="2">CYL HI</td><td colspan="5">SECTOR #</td><td></td></tr>
<tr><td colspan="8">CYLINDER LOW</td></tr>
</table>

- REQUEST SYNDROME
  This function returns the four bytes of the ECC syndrome to the CPU as data.

  OUTPUT:

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

  DATA
  INPUT:

7	6	5	4	3	2	1	0
MSB bit offset							
LSB bit offset							
0							
0				MASK			

- SEEK
  This function seeks to the cylinder of the specified block. For Winchester drives capable of overlap seeks, this function returns completion status before the seek is complete.

  OUTPUT:

7	6	5	4	3	2	1	0
							B
LUN			HEAD #				
CYL HI			0				
CYLINDER LOW							
0							
CONTROL							

- TEST DRIVE READY
  This function selects the specified drive and verifies the drive is ready, the seek is complete, and there are no drive faults.

  OUTPUT:

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

- WRITE
  This function writes the data starting at the initial block address given in the function.

OUTPUT:

<table>
<tr><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="8" align="right">A</td></tr>
<tr><td colspan="3">LUN</td><td colspan="5">HEAD #</td></tr>
<tr><td colspan="3">CYL HI</td><td colspan="5">SECTOR #</td></tr>
<tr><td colspan="8">CYLINDER LOW</td></tr>
<tr><td colspan="8"># OF BLOCKS</td></tr>
<tr><td colspan="8">CONTROL</td></tr>
</table>

- WRITE LONG
  This function writes blocks of data and ECC bytes from the CPU to the disk without generating ECC for the data.

OUTPUT:

<table>
<tr><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="8" align="right">E6</td></tr>
<tr><td colspan="3">LUN</td><td colspan="5">HEAD #</td></tr>
<tr><td colspan="3">CYL HI</td><td colspan="5">SECTOR #</td></tr>
<tr><td colspan="8">CYLINDER LOW</td></tr>
<tr><td colspan="8"># OF BLOCKS</td></tr>
<tr><td colspan="8">CONTROL</td></tr>
</table>

DATA
OUTPUT:

<table>
<tr><td>256/512/1024
BYTES OF DATA</td><td>E
C
2</td><td>E
C
1</td><td>E
C
0</td><td>0
0</td></tr>
</table>

- WRITE SECTOR BUFFER
  This function writes one sector's worth of data to the controller sector buffer. No data transfer occurs between the controller and the drives.

  OUTPUT:

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

Sequencing and Timing

There are three phases to each function:

- function initiation
- function execution
- function results

To initiate a function, you select the controller with the Select Controller Register. Then you wait for Ready in the Status Register to be set. The In/Out bit and the Command/Data bit should indicate function transfer to the controller. You then write six function bytes to the Data Register.

If the Ready is set after this transfer, either there was an error in the function bytes or the controller is ready to receive another group of six function bytes and/or data.

Data can be transferred in DMA or non-DMA mode. If the transfer is in DMA mode, the DMA Controller is programmed in Single Transfer mode (See DMA Controller). The count word is set to:

(number of sectors to transfer)(bytes/sector) - 1

If data is transferred in non-DMA, you use Ready, In/Out, Command/Data and Interrupt Request to time the transfer during execution.

Execution begins when the last function byte is received. In data transfer functions, the controller temporarily stores the data in the sector buffer. This prevents data overruns. When the function completes and the Completion Status byte is loaded, the controller issues interrupts if requested.

You clear the Interrupt Enable and the DMA enable bits in the Control Register after reading the Completion Status. This allows the controller to clear Interrupt Request and Data Request in the Status Register. It also clears the Selected bit.

The controller does extensive error recovery. If an error is found, four retries are attempted. If a retry is successful, the error is not reported; however, the retry count is incremented.

The following errors result in a retry:

- Seek error
- Sector not found
- Uncorrectable data error

- Correctable data error
- No data address mark
- No ID address mark
- ECC error in ID field

On a seek error, a recalibrate and reseek is done by the controller.

The following errors are accumulated in the log:

- ECC error in ID field
- Correctable error in data field
- Uncorrectable error in data field
- No ID address mark
- No data address mark
- Seek error
- Record not found

If rereads are disabled, the controller does not reread before applying the ECC correction.

When a reset is done, the controller defaults to the following characteristics:

- Maximum number of cylinders = 306
- Maximum number of heads = 4
- Starting reduced write current cylinder = 306
- Starting write precompensation cylinder = 0
- Maximum ECC data burst length = 4 bits

The interleave factor states how many physical sectors logical sectors are apart. For example, if the Interleave Factor is 6 and there are 16 sectors in a track, then a sector looks like this:

Physical Sector	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Logical Sector	0	3	6	9	12	15	1	4	7	10	13	16	2	5	8	11	14

The track layout for 256 bytes per sector, 33 sectors per track is:

13 bytes 00's	am	FE	cy1	hd	sec	ecc	00	00	13 bytes 00's	am	F8	256 bytes data	ecc	00	00	10 bytes 4E's

am, FE, cyl, hd, sec, 00, F8 = 1 byte
ecc = 3 bytes

Track capacity = 10416

$$
\begin{array}{rl}
16 & = \text{Index Gap (4E)} \\
10197 & = \text{33 sectors @ 309 bytes/sector} \\
\underline{203} & = \text{Speed Tolerance Gap (4E)} \\
10416 &
\end{array}
$$

309 bytes/sector including ID and overhead

The track layout for 512 bytes per sector, 17 sectors per track is:

13 bytes 00's	am	FE	cy1	hd	sec	ecc	00	00	13 bytes 00's	am	F8	512 bytes data	ecc	00	00	37 bytes 4E's

am, FE, cyl, hd, sec, 00, F8 = 1 byte
ecc = 3 bytes

Track capacity = 10416

16 = Index Gap (4E)
10064 = 17 sectors @ 592 bytes/sector
336 = Speed Tolerance Gap (4E)
10416

592 bytes/sector including ID and overhead

The track layout for 1024 bytes per sector, 9 sectors per track is:

13 bytes 00's	am	FE	cy1	hd	sec	ecc	00	00	13 bytes 00's	am	F8	1024 bytes data	ecc	00	00	58 bytes 4E's

am, FE, cyl, hd, sec, 00, F8 = 1 byte
ecc = 3 bytes

Track capacity = 10416

16 = Index Gap (4E)
10125 = 9 sectors @ 1125 bytes/sector
275 = Speed Tolerance Gap (4E)
10416

1125 bytes/sector including ID and overhead

Sample Program

```
; INIT_CTLR
;       This routine prepares the controller to
;       receive a function.
;          OUTPUT:  Carry set if error
SELECT          EQU 322
STATUS          EQU 321
CONTROL         EQU 323
INIT-CTLR:
   MOV DX,SELECT         ;Select Port Address
   OUT DX,AL             ;Output anything
   MOV DX,CONTROL        ;Control Port Address
   MOV AL,3H             ;Enable interrupts and
                         ;DMA
   OUT DX,AL
   MOV DX,STATUS         ;Status Port Address
INIT1:
   IN AL,DX              ;Get status
   TEST AL,1H            ;Is it ready?
   LOOPZ INIT1           ;Loop if not ready
   CMP AL,DH             ;Jump if ready for a
   JE INIT2              ;function and selected
   STC                   ;Flag error
   RET
INIT2:
   CLC
   RET
```

Keyboard Interface

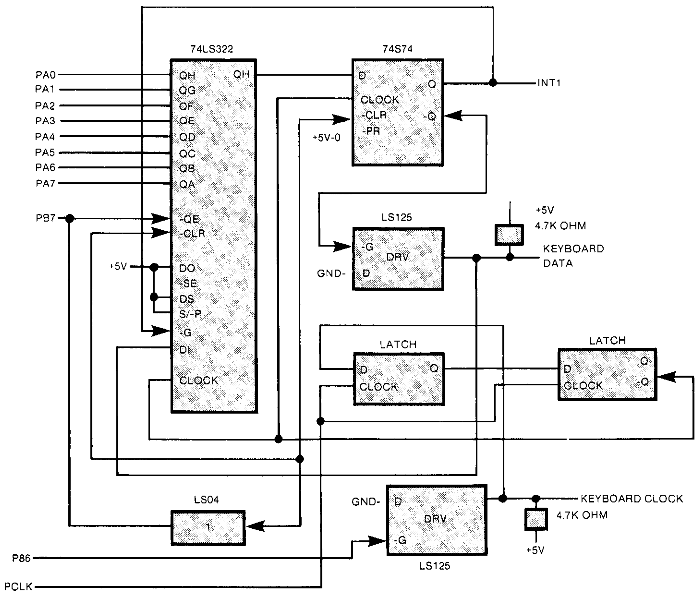
Functional Description

The keyboard interface converts the parallel data into serial data for transmission to and from the keyboard.

To provide maximum flexibility in defining keyboard operations, the keyboard uses scan codes rather than ASCII codes. In addition, all keys generate a make scan code when pressed and a break scan code when released. The break scan code is 80H plus the make scan code.

The keyboard is responsible for keeping track of the amount of time a key is depressed and for generating the repeat key signal. All the keys have this repeat function.

Block Diagram

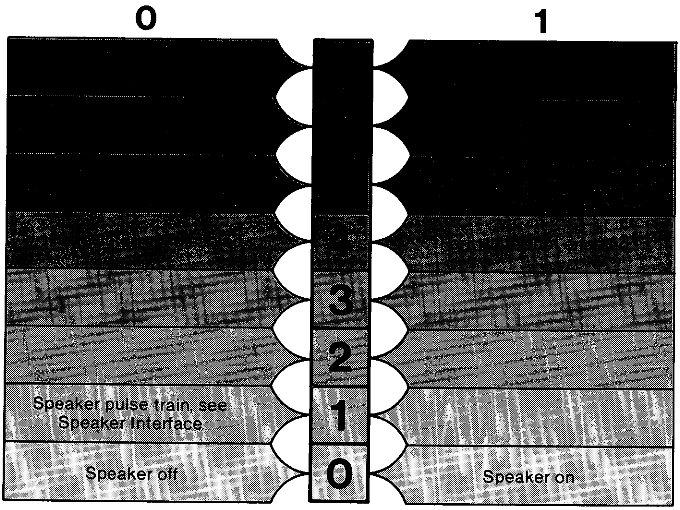


Registers

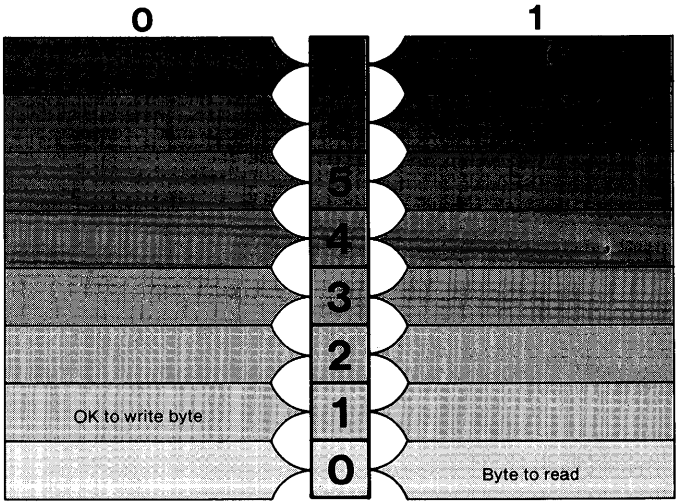
PORT	NAME	READ/ WRITE	DESCRIPTION
60	DATA	R	8 bit scan code when pressed, 8 bit scan code plus 80H when released See scan code chart.
60	DATA	W	Keyboard command codes
60	CONTROL	R/W	See layout
64	STATUS	R	See layout

Layout

CONTROL



STATUS



Functions

- READ STATUS
  This function allows you to determine when there is a character to read and when the keyboard is ready to receive a character.
  INPUT: STATUS

- READ DATA
  This function allows you to read the scan code of the key that was pressed or released.
  INPUT: DATA

- WRITE CONTROL
  This function allows you to issue control commands to the keyboard.
  OUTPUT: CONTROL

- GET KEYBOARD TYPE
  This function determines if the keyboard is an Olivetti M20 type keyboard.
  OUTPUT: DATA REGISTER
  5
  INPUT: DATA REGISTER
  1 = Olivetti M20 keyboard

- KEYBOARD LEDS
  This function controls the illuminations of the CAPS LOCK and NUM LOCK LEDS.
  OUTPUT: DATA REGISTER
  13H

Value	**Operation**
00h	No operation
01h	Cap lock LED OFF
02h	Num lock LED OFF
03h	Both LEDS OFF
80h	No operation
81h	Caps lock LED ON
82h	Num lock LED ON
83h	Both LEDS ON

Any other value will cause one of the above operations.

Scan Codes

KEY NO	SCAN CODE	KEY NO	SCAN CODE	KEY NO	SCAN CODE
1	01H	36	24H	71	47H
2	02H	37	25H	72	48H
3	03H	38	26H	73	49H
4	04H	39	27H	74	4AH
5	05H	40	28H	75	4BH
6	06H	41	29H	76	4CH
7	07H	42	2AH	77	4DH
8	08H	43	2BH	78	4EH
9	09H	44	2CH	79	4FH
10	0AH	45	2DH	80	50H
11	0BH	46	2EH	81	51H
12	0CH	47	2FH	82	52H
13	0DH	48	30H	83	53H
14	0EH	49	31H	84	54H
15	0FH	50	32H	85	55H
16	10H	51	33H	86	56H
17	11H	52	34H	87	57H
18	12H	53	35H	88	58H
19	13H	54	36H	89	59H
20	14H	55	37H	90	5AH
21	15H	56	38H	91	5BH
22	16H	57	39H	92	5CH
23	17H	58	3AH	93	5DH
24	18H	59	3BH	94	5EH
25	19H	60	3CH	95	5FH
26	1AH	61	3DH	96	60H
27	1BH	62	3EH	97	61H
28	1CH	63	3FH	98	62H
29	1DH	64	40H	99	63H
30	1EH	65	41H	100	64H
31	1FH	66	42H	101	65H
32	20H	67	43H	102	66H
33	21H	68	44H	103	67H
34	22H	69	45H	104	68H
35	23H	70	46H	-	-

Sequencing and Timing

To send a character to the keyboard, wait for Bit 1 of the Status Register to be set and write the byte to the data register.

To receive a character, wait for Bit 0 of the Status Register to be set and read the character. The keyboard interface can be programmed to interrupt on INT1 when there is a character to read.

Sample Program

```
;This program sets the CAPS LOCK LED
STATUS      EQU 64
DATA        EQU 60
SET_LED:
  IN AL,STATUS       ;read status
  TEST AL,2          ;keyboard ready to receive
  JNZ SET_LED        ;input? Jump if ready.
  MOV AL,013H        ;keyboard LED command
  OUT DATA,AL

SET1:
  IN AL,STATUS       ;read status
  TEST AL,2          ;ready to receive input?
  JNZ SET1           ;jump if not
  MOV AL,81          ;CAPS LOCK
  OUT DATA,AL        ;write out code
  RET
```

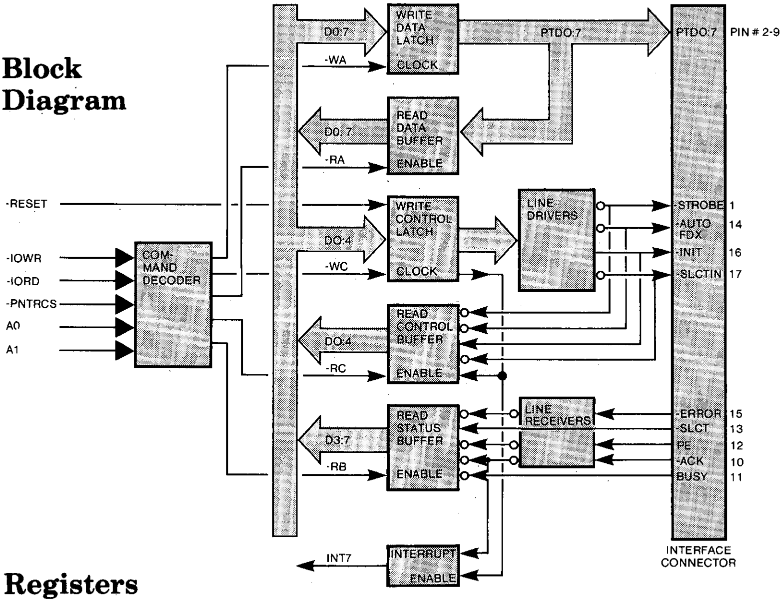
Parallel Printer Interface

Functional Description

The parallel printer interface connects to printers with a Centronics-like parallel interface or any other device with identical interface characteristics. The input and output signals are presented to the external device through a 25-pin "D" type connector.

The interface has 5 buffered outputs — data, strobe, initialize printer, automatic linefeed, and select. These can be read and written. In addition, the interface has five inputs — acknowledge, busy, paper out, error and select. An interrupt can be enabled on INT7.

Block Diagram

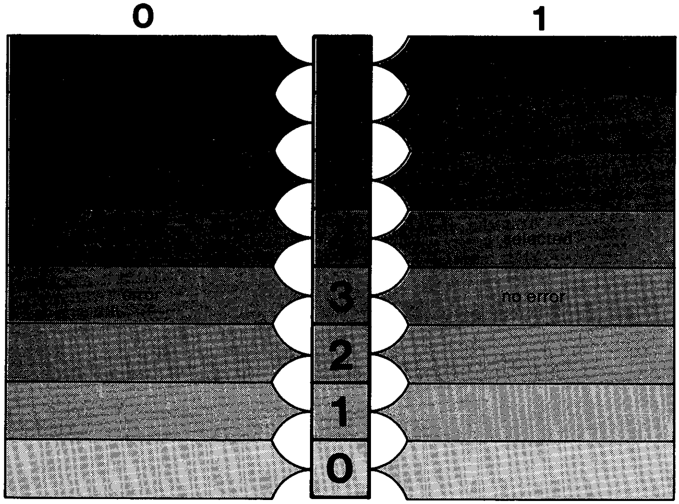


Registers

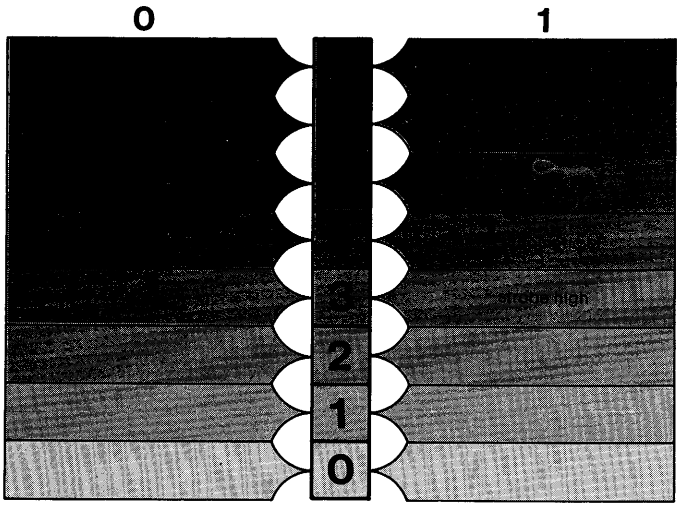
PORT #	NAME	READ/ WRITE	DESCRIPTION
378	DATA	R/W	Print character
379	STATUS	R	See layout
37A	CONTROL	R/W	See layout

Layout

STATUS



CONTROL



Functions

- RESET
  After a power on or a hardware reset, the data buffer is cleared and the control register is initialized to:

  bit 0: 0
  bit 1: 0
  bit 2: 0
  bit 3: 0
  bit 4: 0

- WRITE DATA
  This instruction enables the data on the data bus to be written to the printer data bus. The actual writing occurs when the strobe line is activated.
  OUTPUT: DATA

- WRITE CONTROL
  This instruction inverts D0, D1, and D3 on the data bus and writes the data to the control register. If D4 is a 1 then interrupts are requested.
  OUTPUT: CONTROL

- READ DATA
  This instruction enables the data on the printer data bus to be read onto the data bus. It normally is the last character written to the printer.
  INPUT: DATA

- READ CONTROL
  This instruction enables the data on the printer control lines and the interrupt control bit to be placed on the data bus.
  INPUT: CONTROL

- READ STATUS
  This instruction enables the data on the printer status lines to be placed on the data bus.
  INPUT: STATUS

Sequencing and Timing

To send a character to the printer, the character is put on the data bus. When the printer is not busy, it is ready to accept the next character. The character must be strobed into the printer by setting the strobe bit to 1 for at least 5 μ-seconds and then resetting it.

Interrupt can be enabled on INT7 by writing D4 = 1 in the control register. An interrupt will be triggered everytime Bit 6 of the status register goes from 0 to 1 (end of the acknowledge cycle from the printer).

To initialize the printer, first select it. Then issue the initialize command setting the automatic line feed and interrupt parameters.

Sample Program

```
; PRINT_CHAR
;       Send character to printer and get status
;       INPUT  AL - character to print
;       OUTPUT:  AL - status
;
DATA                EQU 378H
PRINT_CHAR:
    MOV DX,DATA                 ;get data port
    OUT DX,AL                   ;put char on data
                                ;line
    INC DX                      ;get status port
    IN AL,DX                    ;read in status
    TEST AL,080H                ;is the printer busy?
    JNZ PRINT_NOT_BUSY          ;jump if not busy
    .
    .
    .
PRINT_NOT_BUSY:
    MOV AL,0DH                  ;strobe high
    INC DX                      ;get control port
    OUT DX,AL                   ;to control register
    NOP                         ;wait
    NOP                         ;wait
    MOV AL,0CH                  ;strobe low
    OUT DX,AL                   ;to control register
    DEC DX                      ;get status port
    IN AL,DX                    ;read in status
    RET
```

Programmable Interrupt Controller

Functional Description

The Intel 8259A Programmable Interrupt Controller (PIC) manages external interrupts. It receives requests from peripheral equipment, decides which request has the highest priority, and issues an interrupt to the CPU. Each PIC handles 8 maskable priority interrupts. PIC's can be cascaded allowing up to 64 priority interrupts. However, on the AT&T Personal Computer 6300 this is not done.

Each interrupt device runs to one of eight interrupt lines (INT0-INT7). If more than one device interrupts at once, the PIC decides which device to service according to one of several schemes.

The following schemes apply to a single PIC:

- Fully Nested Mode
  This is the default mode. The interrupt requests have an ordered priority from 0 (highest) to 7 (lowest). The highest priority is acknowledged first and those of lower priority are inhibited.

- Special Mask Mode
  This mode is similar to fully nested mode except that the interrupt mask register (IMR) determines which interrupts are disabled.

- Polled
  This mode allows the CPU to poll the devices. It is selected by disabling interrupts with the CLI instruction. Periodically the CPU polls the PIC to receive the interrupt type of the highest priority device requesting service.

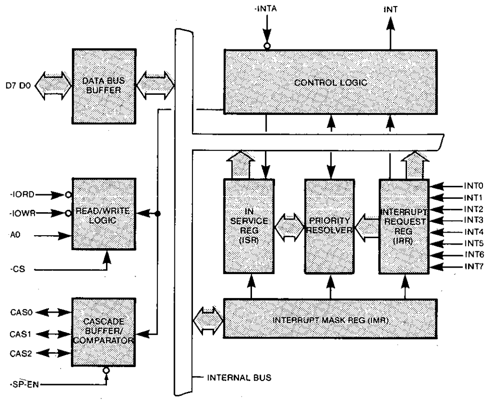
- Automatic Rotation
  In this mode, a device receives the lowest priority after it is serviced. All other devices have their priorities adjusted accordingly. The next highest interrupt line receives the highest priority.

- Programmable Rotation
  In this mode, the programmer declares the lowest priority device.

The PIC keeps track of devices that are waiting for service in the interrupt request register (IRR). If not in polled mode, the PIC notifies the CPU of the pending interrupt. When it receives an interrupt acknowlege from the CPU (INTA), it sends the interrupt type of the device to the CPU. The device is then in-service. This is noted in the in-service register (ISR). The type of INT0 is programmable. It must be a multiple of 8.

After the interrupt service routine services the interrupt, it notifies the PIC of end of interrupt (EOI). The device is then removed from the in-service register.

Block Diagram

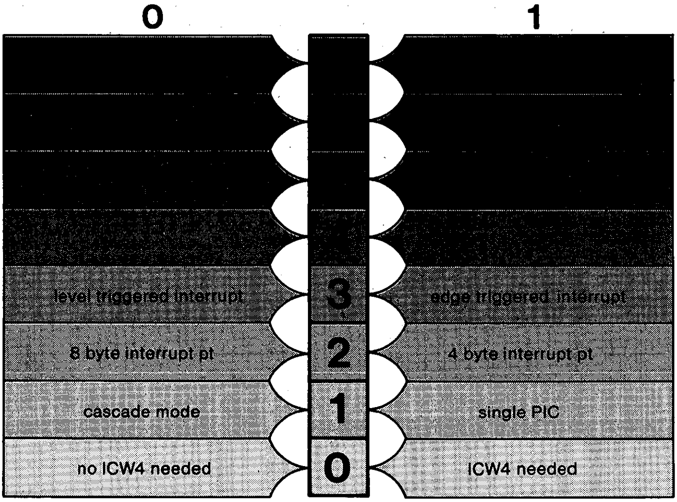


Registers

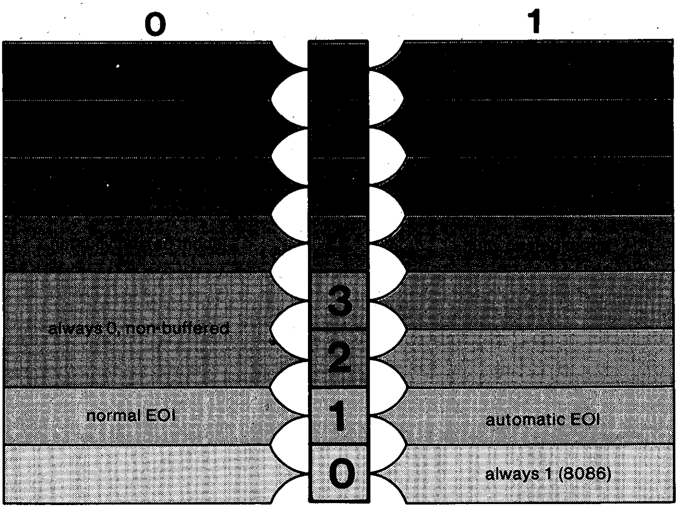
PORT #	NAME	READ/ WRITE	DESCRIPTION
20	INIT COMMAND WORD 1 (ICW1)	W	See layout
21	INIT COMMAND WORD 2 (ICW2)	W	INT0 interrupt type, multiple of 8
21	INIT COMMAND WORD 3 (ICW3)	W	Cascade mode only
21	INIT COMMAND WORD 4 (ICW4)	W	See layout
21	OPERATION COMMAND WORD 1 (OCW1, IMR)	R/W	Interrupt Mask Register See layout
20	OPERATION COMMAND WORD 2 (OCW2)	W	See layout
20	OPERATION COMMAND WORD 3 (OCW3)	W	See layout
20	IN-SERVICE REGISTER (ISR)	R	See layout
21	INTERRUPT LEVEL	R	See layout
20	INTERRUPT REQUEST REGISTER (IRR)	R	See layout

Layout

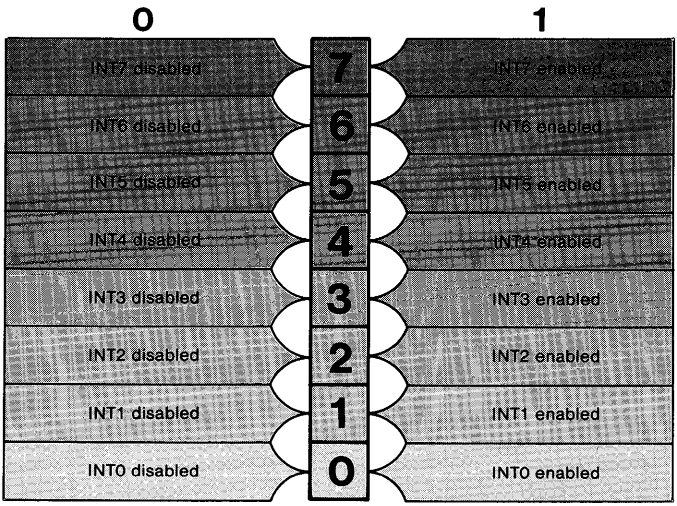
ICW1



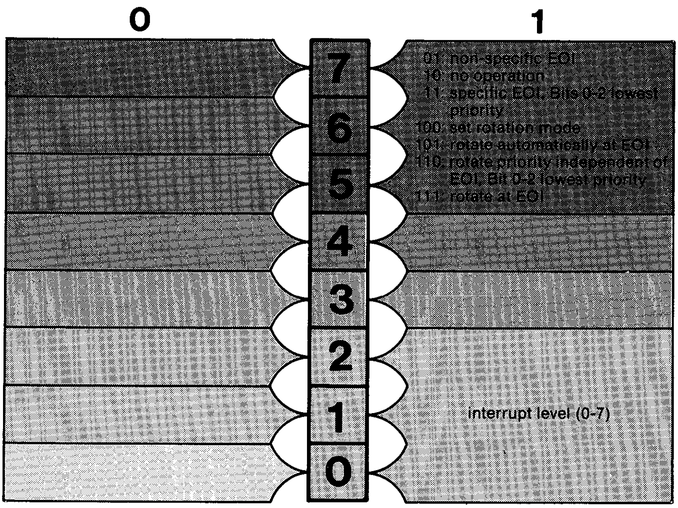
ICW4



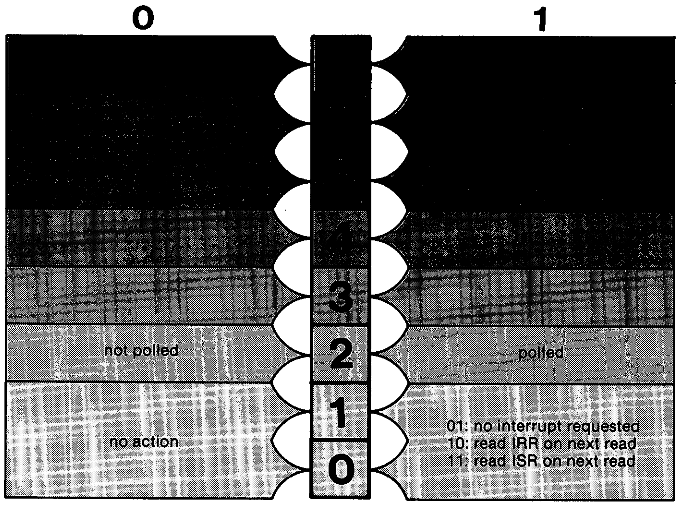
OCW1 or INTERRUPT MASK REGISTER



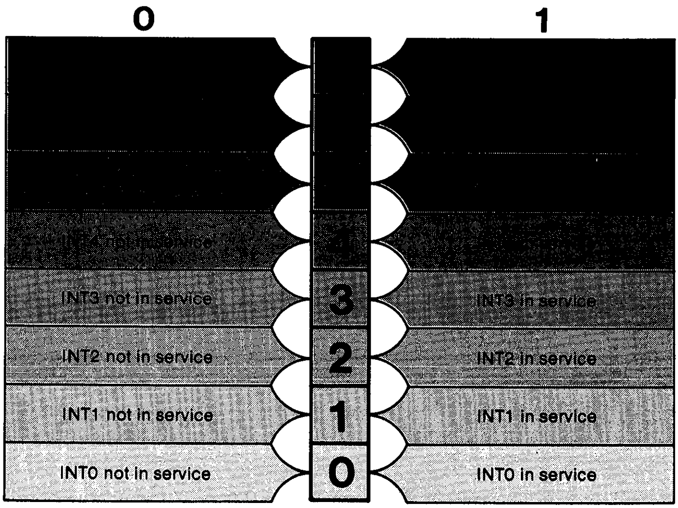
OCW2



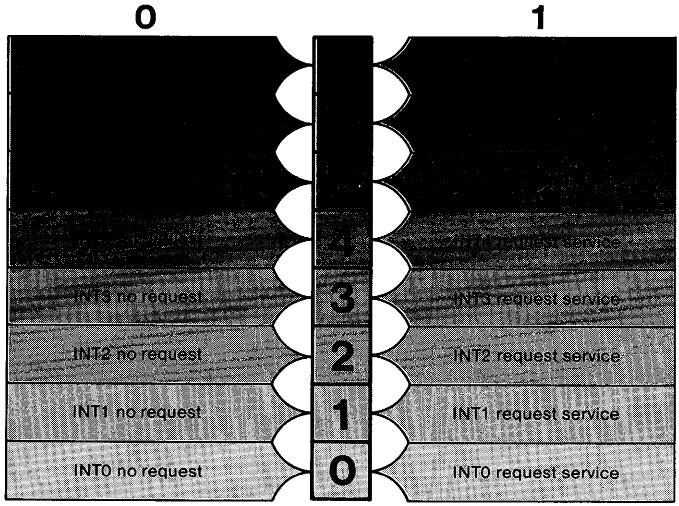
OCW3



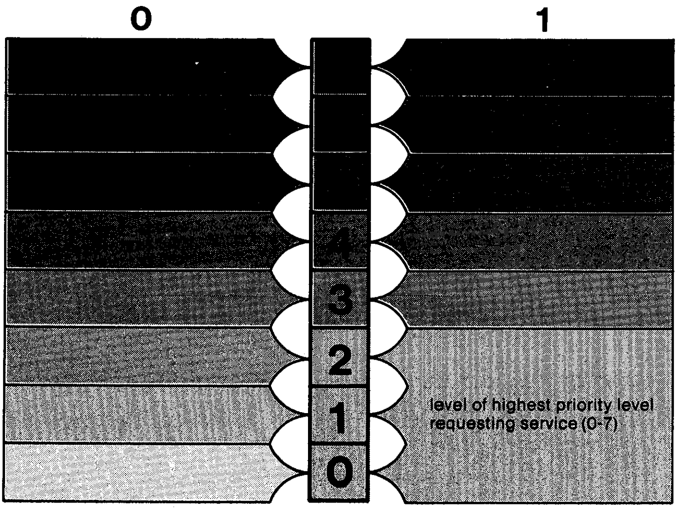
IN SERVICE REGISTER



INTERRUPT REQUEST REGISTER



INTERRUPT LEVEL



Functions

- INITIALIZATION
  This function prepares the PIC to accept interrupts by setting it to an initial state.
  OUTPUT: ICW1
  ICW2
  ICW3 (Cascaded PIC's only)
  ICW4

- SET SPECIAL MASK MODE
  This function sets the priority scheme to Special Mask Mode. The Interrupt Mask Register (IMR) defines enabled priorities.
  OUTPUT: OCW3 BITS 5-6 = 3
  IMR

- RESET SPECIAL MASK MODE
  This function resets the priority structure to the Fully Nested Mode. The IMR is ignored.
  OUTPUT: OCW3 BITS 5-6 = 1

- SET POLLED MODE
  This function sets the priority scheme to Polled Mode. The CLI instruction must be executed to disable external interrupts. The next read fetches the interrupt level.
  OUTPUT: OCW3 BIT 2 = 1
  INPUT: interrupt level

- AUTOMATIC ROTATION OF PRIORITIES AT EOI
  This function requests an automatic rotation of priorities at EOI.
  OUTPUT: ICW2 BITS 5-7 = 5

- PROGRAMMED ROTATION AT EOI
  This function requests a specific change in priorities at EOI.
  OUTPUT: ICW2 BITS 5-7 = 7
  ICW2 BITS 0-2 = level of lowest priority

- ROTATION OF PRIORITIES INDEPENDENT OF EOI
  This function requests an immediate change in priorities.
  OUTPUT: OCW2 BITS 5-7 = 6
  OCW2 BITS 0-2 = level of lowest priority

- SPECIFIC END OF INTERRUPT (SEOI)
  This function is issued in an interrupt service routine to declare end of interrupt service for the specified level.
  OUTPUT: OCW2 BITS 5-7 = 3
  BITS 0-2 = interrupt level

- NON-SPECIFIC END OF INTERRUPT (EOI)
  When the PIC is operating in Fully Nested Mode, it can determine which interrupt is completing. This function signals completion of an interrupt service routine.
  OUTPUT: OCW2 BITS 5-7 = 1

- AUTOMATIC END OF INTERRUPT
  This function requests the PIC to declare end of interrupt automatically after delivering the interrupt to the CPU.
  OUTPUT: ICW4 BIT 1 = 1

- READ IRR
  This function reads the devices requesting service.
  OUTPUT: OCW3 BITS 0-1 = 2
  INPUT: IRR

- READ ISR
  This function reads the in-service register.
  OUTPUT: OCW3 BITS 0-1 = 3
  INPUT: ISR

- READ IMR
  This function reads the interrupt mask register.
  INPUT: OCW1

Sequencing and Timing

When the ICW1 command is issued, the initialization process begins. The following automatically occurs:

- IMR is cleared.
- INT7 is assigned the lowest priority.
- Single mode is assumed.
- Special Mask Mode is cleared.
- A status read fetches IRR.
- If Bit 0 equals zero, then ICW4 functions are set to zero.

Next ICW2 is output. ICW3 is skipped in all single PIC systems. If the ICW4 was requested by ICW1 then it is output. This completes initialization.

Once the initialization process is complete, the PIC is ready to accept interrupts. Any of the functions to change the priority scheme can be executed. In addition, the IRR, IMR, and ISR can be read.

If automatic EOI is not specified then the interrupt service routine must declare EOI. Either specific or non-specific EOI can be used depending on the priority scheme in use.

When a write command is issued to the PIC, 480 nanoseconds must elapse before another command is issued. In the read case, 395 nanoseconds must elapse.

Sample Program

```
;
; SEND_SEOI
;     Send end of INT1 service at the end of the
;     interrupt service routine.
;
OCW2          EQU 20   ;port address of OCW2
COMMAND       EQU 61   ;6 = SEOI, 1 = INT1

SEND_SEOI:
   MOV AL,COMMAND      ;set AL = command
   MOV DX,OCW2         ;set DX = port address
   OUT DX,AL           ;send command
   STI                 ;enable external interrupts
   RET                 ;return to service routine
```

Programmable Interval Timer

Functional Description

The Intel 8253 Interval Timer has three identical, 16-bit, settable, decrementing counters. Each counter is totally independent. The counters have either a BCD or binary value and operate in one of five modes:

- Interrupt on Terminal Count
  The output remains low after the mode is set. It continues low after the counter is loaded until the counter counts down to zero. Then the output goes high. It remains high until a new mode is selected or a new count is loaded.

- Programmable One-Shot
  The output goes low one count following the rising edge of the gate input. The output goes high on the terminal count.

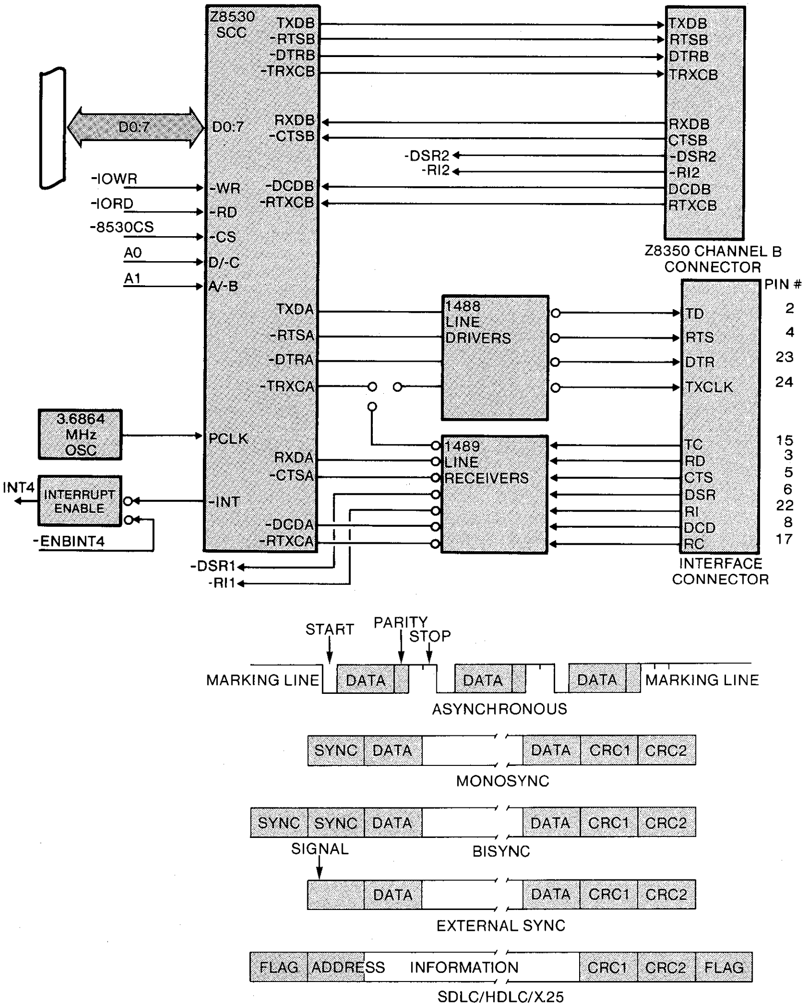
- Rate Generator
  The output is low for one period of the input clock. The period from one output pulse to the next equals the number of input counts in the count register.

- Square Wave Rate Generator
  The output remains low for one period of the input clock. The output remains high until one half the count has elapsed. If the count is odd, the output is high for (n+1)/2 and low for (n-1)/2.

- Software Triggered Strobe
  After the mode is set, the output will be high. When the count is loaded, the counter begins counting. On the terminal count,the output goes low for one clock period and then goes high.

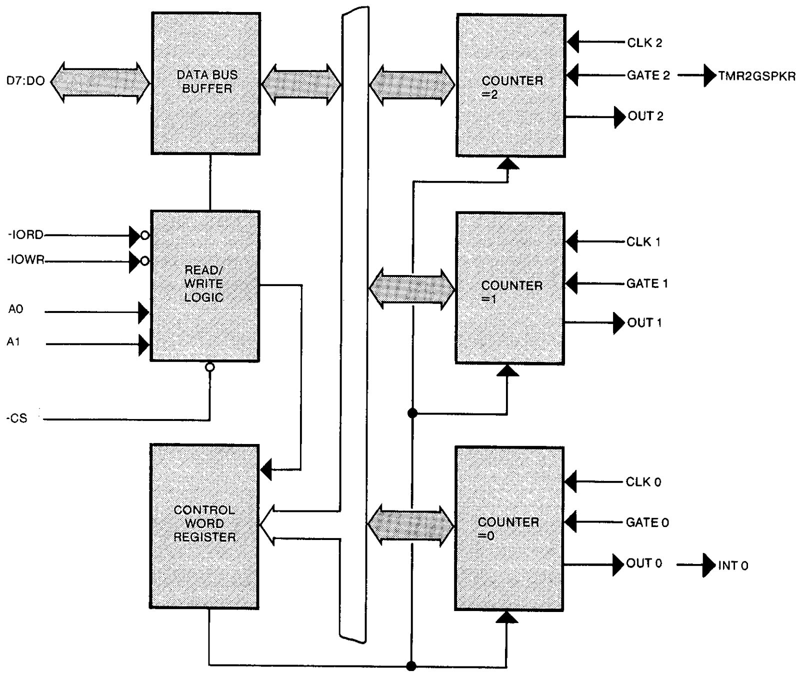
- Hardware Triggered Strobe
  The counter starts counting after the rising edge of the trigger input and goes low for one clock period when terminal count is reached.

  These timing diagrams illustrate the different modes.



In the AT&T Personal Computer 6300 system, Counter 0 provides real time interrupts on INT0, counter 1 requests memory refreshes, and counter 2 generates a pulse train for the audio speaker.

Block Diagram

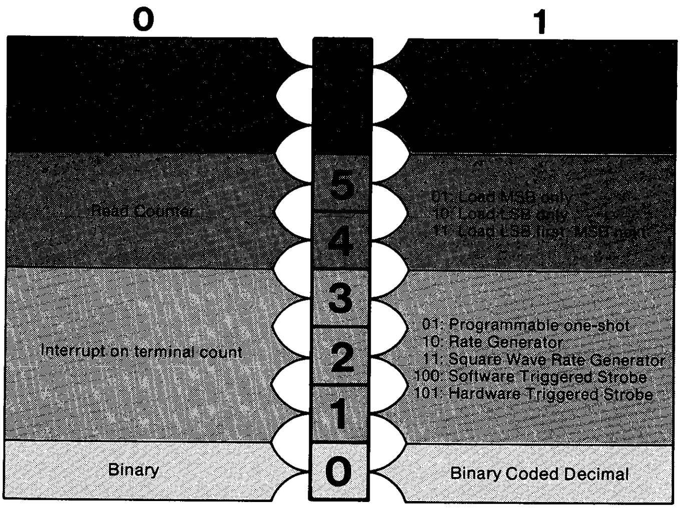


Registers

PORT #	NAME	READ/ WRITE	DESCRIPTION
40	COUNTER 0	R/W	Provides real time interrupt INT0
41	COUNTER 1	R/W	Provides signals to refresh memory
42	COUNTER 2	R/W	Generate pulse train for the audio speaker
43	CONTROL	W	See Layout

Layout

CONTROL



Functions

- LOAD COUNTER
  This function allows you to set a specific counter.

OUTPUT:	CONTROL REGISTER
	BITS 0 binary or BCD
	BITS 1-3 mode
	BITS 4-5 1,2 or 3
	BITS 6-7 counter
	8 or 16 bit count value

- READ COUNTER
  This function allows you to read a specific counter.

OUTPUT:	CONTROL REGISTER
	BITS 4-5 0
	BITS 6-7 counter

Sequencing and Timing

All counters must be initialized with the control register. The control register specifies the number of bytes which must be loaded.

Whenever a read or a load command is issued, the requested counter bytes must be read or written. In the read case, two reads are necessary, the first for the least significant byte (LSB) and the last for the most significant byte (MSB). In the write case, the control register specifies the byte to write.

1 microsecond recovery time is required between a read or a load and any other control signal.

Input to the timer is 1.2288MHz. Therefore, there are 18.75 interrupts per second. To generate a 1.00 KHz tone with the audio speaker, a square wave rate generator is used with a count of 614 (1.2288MHz/2*614 = 1KHz).

$\frac{1.2288 \times 10^6}{64 \times 1024} = 18.8$

FF →

IBM PC 1.19 MHz

≅ 3%

Sample Program

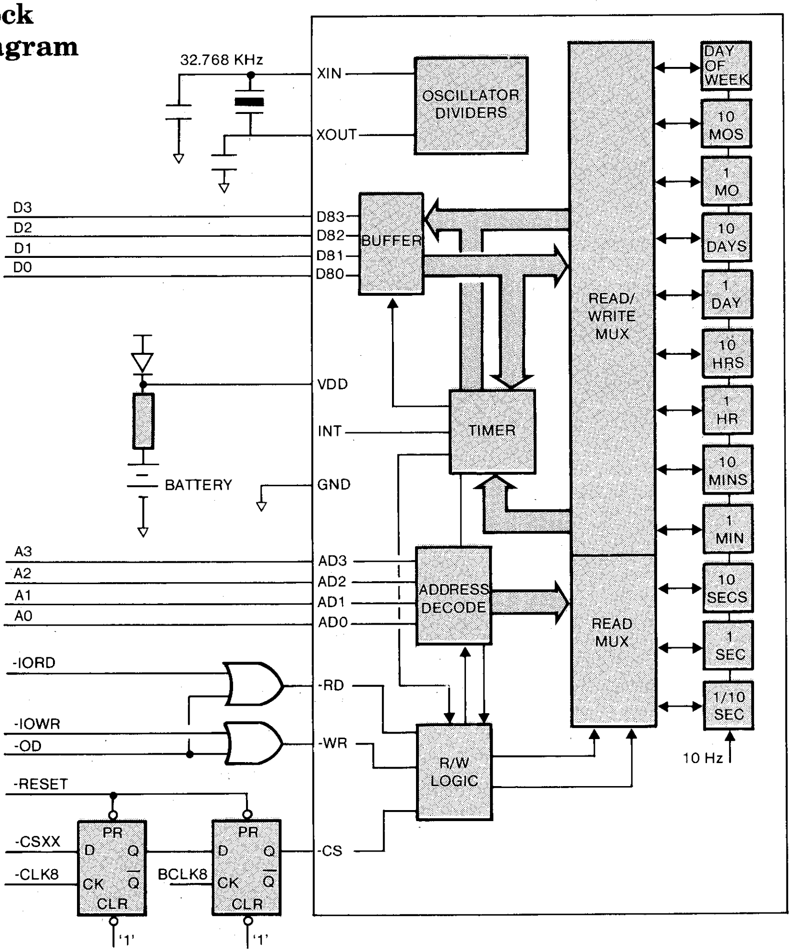
```
;
; ASK_FOR_INTR
;       This program requests an interrupt in
;       approximately 10 usec (9765.6 nsec)
;
TIMER_CONTROL   EQU 43
TIMER0          EQU 40

INT_MASK        EQU 21

ASK_FOR_INTR:
    MOV AL,0FEH                 ;allow only INT0
                                ;interrupt
    OUT INT_MASK,AL             ;send mask
    MOV AL,00110000B            ;binary counter,
                                ;interrupt on terminal
                                ;count, set counter 0
    OUT TIME_CONTROL,AL
    MOV AX,12                   ;12 * 813.8 nsec
    OUT TIMER0,AL
    MOV AL,AH                   ;output counter, LSB
                                ;then MSB
    OUT TIMER0,AL
    RET
```

Real Time Clock and Calendar

Functional Description

The real time clock and calendar keep the current data and time. All of the date and time fields can be read but the second fields cannot be written. The calendar keeps up to eight years. A rechargeable battery keeps the unit running even when the computer is turned off.

Block Diagram

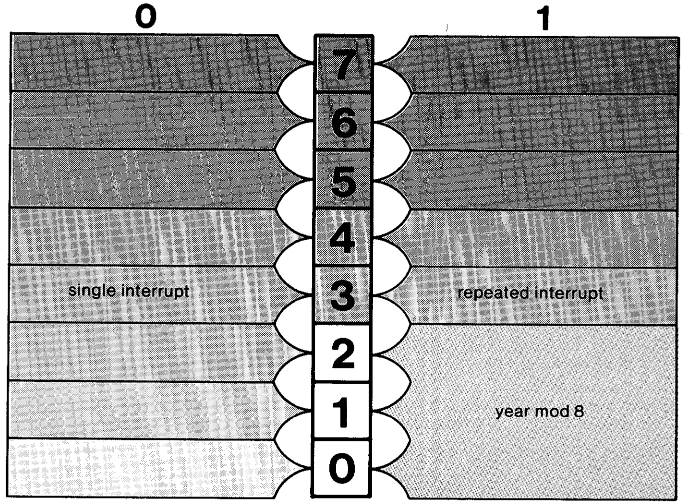


Registers

PORT #	NAME	READ/ WRITE	DESCRIPTION
70	TEST PORT	W	0=not test mode, 1=test mode
71	1/10 OF A SECOND	R	nybble, 0-9
72	UNIT SECONDS	R	nybble, 0-9
73	10'S OF SECONDS	R	nybble, 0-5
74	UNIT MINUTES	R/W	nybble, 0-9
75	10'S OF MINUTES	R/W	nybble, 0-5
76	UNIT HOURS	R/W	nybble, 0-9
77	10'S OF HOURS	R/W	nybble, 0-1
78	UNIT DAYS	R/W	nybble, 0-9
79	10'S OF DAYS	R/W	nybble, 0-3
7A	DAY OF WEEK	R/W	nybble, 1-7
7B	UNIT MONTHS	R/W	nybble, 0-9
7C	10'S OF MONTHS	R/W	nybble, 0-1
7D	LEAP YEAR	W	See layout
7E	STOP/START	W	0 = stop, FF = start
7F	INTERRUPT/YEAR MOD 8	R/W	See layout

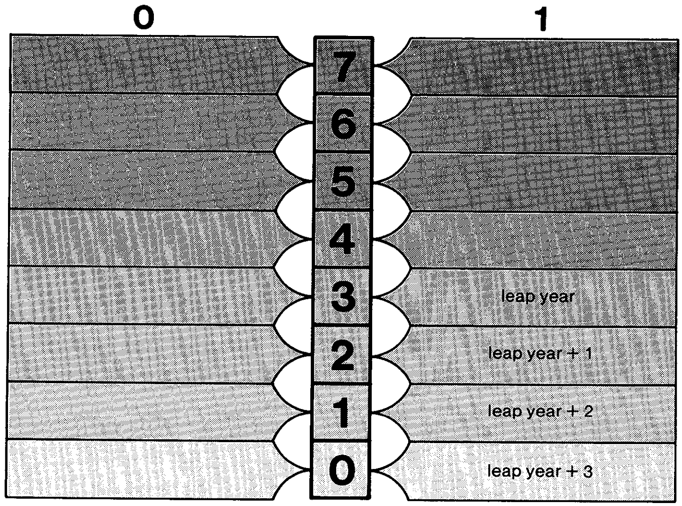
Layout

INTERRUPT/YEAR MOD 8



7F

LEAP YEAR



7D

Functions

- READ CALENDAR AND CLOCK
  All of the calendar and time registers are readable. Those from port address 71 to 7D and 7F contain a nibble of data.

- WRITE CALENDAR AND CLOCK
  All of the calendar and time registers except seconds are writable. The registers from port address 74 to 7D and 7F each contain a nibble of data.

Sequencing and Timing

To write data to the clock and time registers, the unit must be out of test mode and stopped. After writing to the clock, it must be restarted.

To initialize interrupts, set Bit 4 in the Interrupt/Year mod 8 register. Write the register once and then read it in three times.

If an update occurs while reading a register, the illegal code of F is returned.

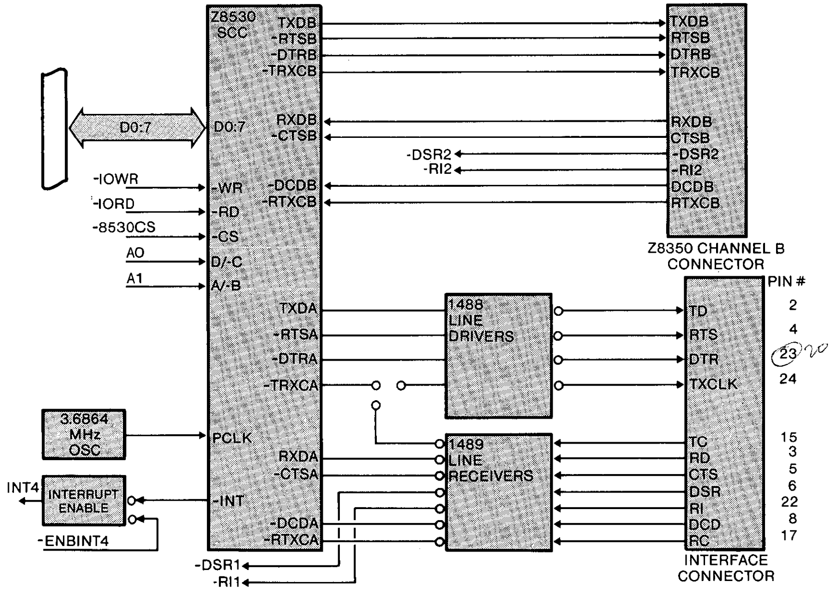
Sample Program

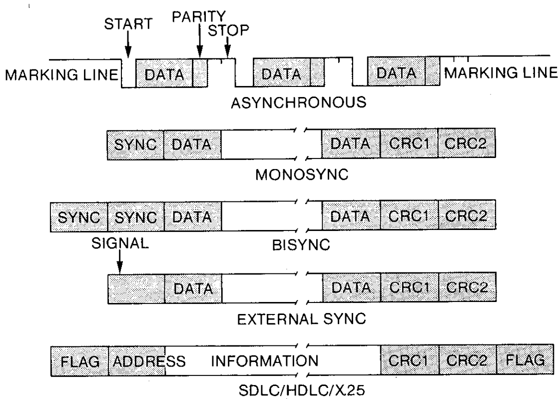
```
; SET TIME
;       This routine sets the time to 12:00 noon
;
TEST_PORT          EQU 70
STOP_START         EQU 7E
TENS_HOURS         EQU 77
WRITE_TIME:
  XOR AX,AX                  ;AX = 0
  MOV DX,TEST_PORT           ;setup dx
  OUT DX,AL                  ;take out of
                             ;test mode
  OUT STOP_START,AL          ;stop the clock

  MOV DX,TENS_HOURS          ;port addr of
                             ;10's of hours
  MOV AL,1
  OUT DX,AL                  ;ten's of hours
                             ;= 1

  DEC DX
  MOV AL,2
  OUT DX,AL                  ;unit hours = 2

  DEC DX
  XOR AL,AL
  OUT DX,AL                  ;tens of minutes
                             ;= 0

  DEC DX
  OUT DX,AL                  ;minutes = 0

  MOV AL,0FFH
  OUT STOP_START,AL          ;start the clock

  RET
```

Serial Communications Controller

Functional Description

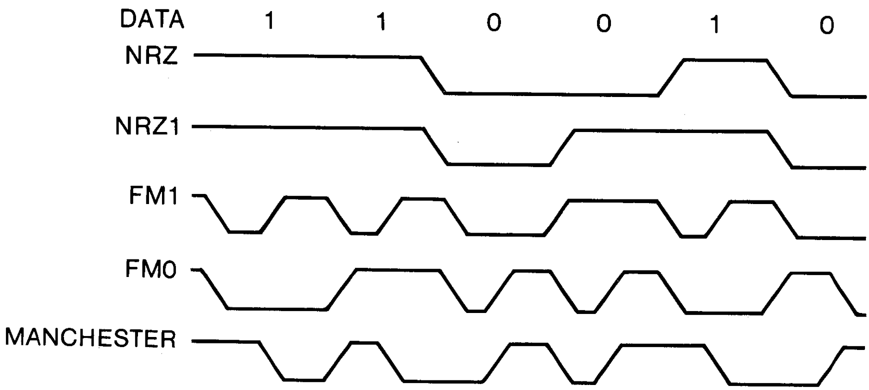
Z8530 Serial Communications Controller (SCC) performs serial-to-parallel conversion on input data characters received from a modem and parallel-to-serial conversion on output data characters received from the CPU. It supports the following common asynchronous and synchronous data communication protocols.





In addition, the SCC supports five encoding methods — NRZ, NRZI, FM1 (bi-phase mark), FM0 (bi-phase space), and Manchester (bi-phase level).

Figure 20



The SCC has the following capabilities:

Asynchronous

- 5, 6, 7, or 8 bits per character
- 1, 1 1/2, or 2 stop bits
- Odd or even parity
- Times 1, 16, 32, or 64 clock modes
- Break generation and detection
- Parity, overrun and framing error detection

Byte-oriented synchronous

- Internal or external character synchronization
- 1 or 2 sync characters in separate registers
- Automatic sync character insertion and deletion
- Cyclic redundancy check (CRC) generation/ detection
- 6- or 8-bit sync character

SDLC/HDLC

- Abort sequence generation and checking
- Automatic zero insertion and deletion
- Automatic flag insertion between messages
- Address field recognition
- I-field residue handling
- CRC generation/detection
- SDLC loop mode with EOP recognition/loop entry and exit

The baud rate is programmable for both channels.

Registers

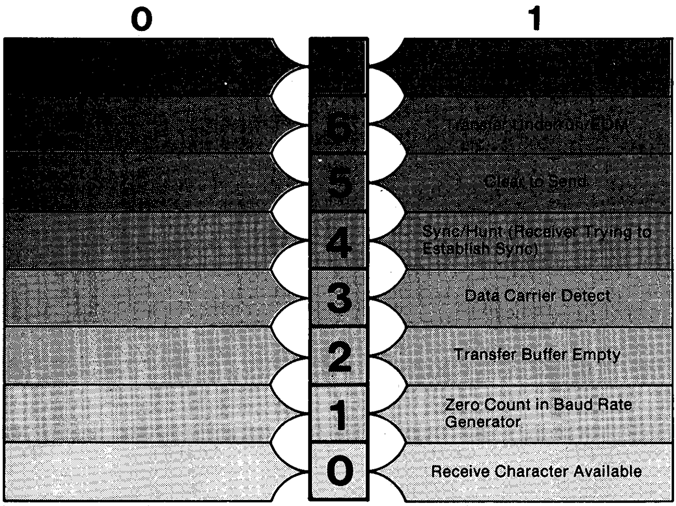
PORT # A	B	NAME	READ/ WRITE	DESCRIPTION
51	53	DATA	R/W	Transfer data
50	52	SCC REGISTER POINTER	R/W	Transfer SCC register number and SCC register data

SCC Registers

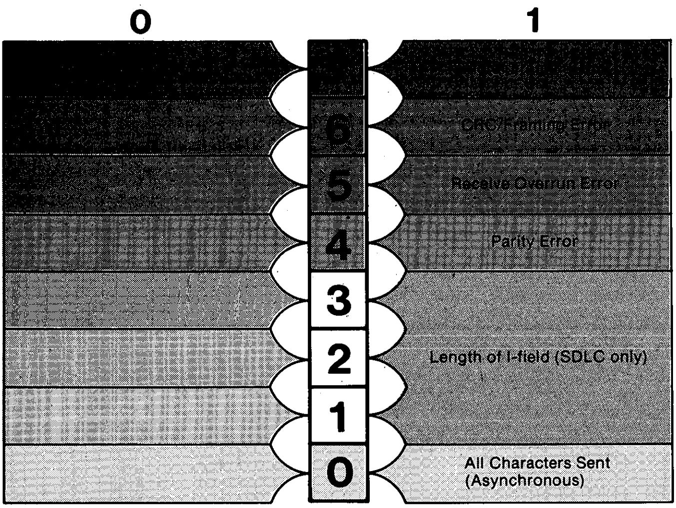
READ/ WRITE	NO	NAME	DESCRIPTION
R	0	Buffer and External Status	See layout
R	1	Special Receive Condition Status	See layout
R	2	Modified Interrupt Vector (Channel B) Unmodified Interrupt Vector (Channel A)	
R	3	Interrupt Pending Bits (Channel A)	See layout
R	8	Receive Buffer	See layout
R	10	Miscellaneous Status	See layout
R	12	Low Byte of Baud Rate Generator Constant	
R	13	High Byte of Baud Rate Generator Constant	
R	15	External/Status Interrupt Information	See layout
W	0	CRC Initialize	See layout
W	1	Transmit/Receive Interrupt and Data Transfer Mode Definition	See layout
W	2	Interrupt Vector	
W	3	Receive Parameters and Control	See layout
W	4	Transmit/Receive Miscellaneous Parameters and Modes	See layout
W	5	Transmit Parameters and Control	See layout
W	6	Sync Characters or SDLC Address Field	See layout
W	7	Sync Characters or SDLC Flag	See layout
W	8	Transmit Buffer	See layout
W	9	Master Interrupt Control and Reset	See layout
W	10	Miscellaneous Transmit/Receive Control Bits	See layout
W	11	Clock Mode Control	See layout
W	12	Low Byte of Baud Rate Generator Constant	
W	13	High Byte of Baud Rate Generator Constant	
W	14	Miscellaneous Control Bits	See layout
W	15	External/Status Interrupt Control	See layout

Layout

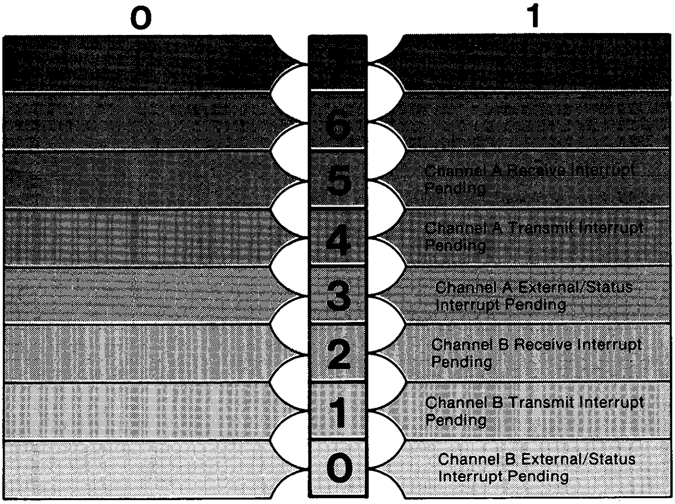
READ REGISTER 0



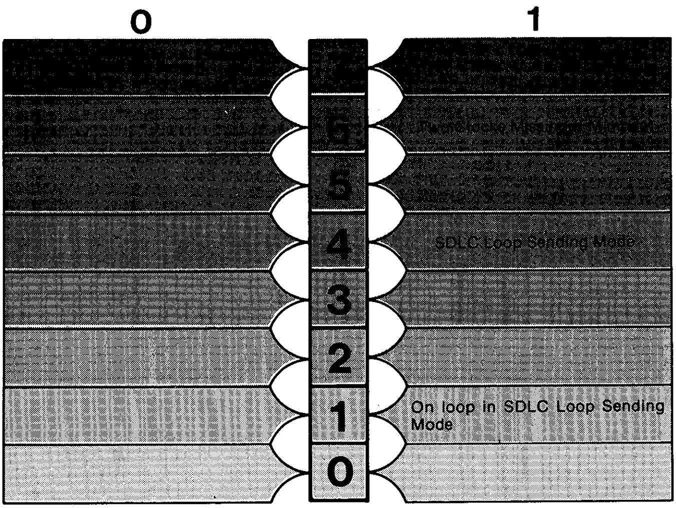
READ REGISTER 1



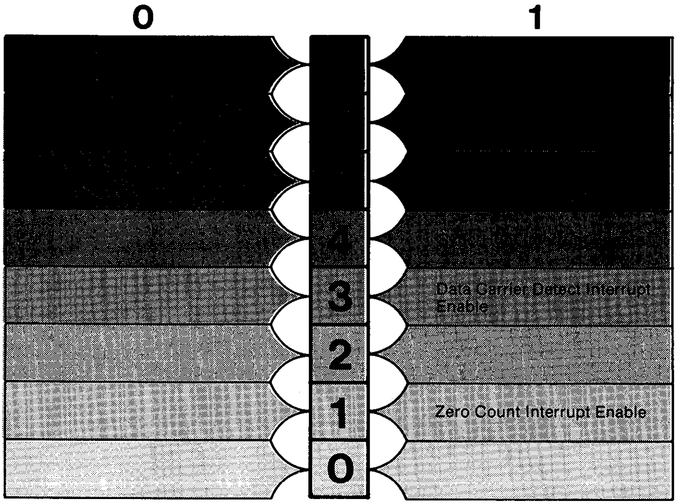
READ REGISTER 3



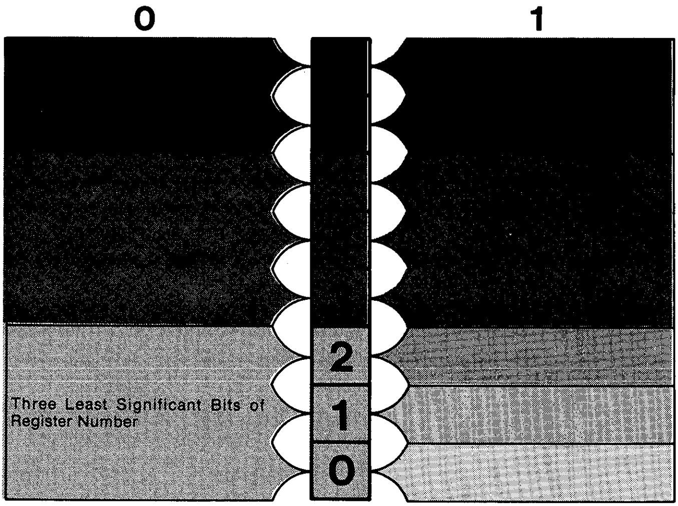
READ REGISTER 10



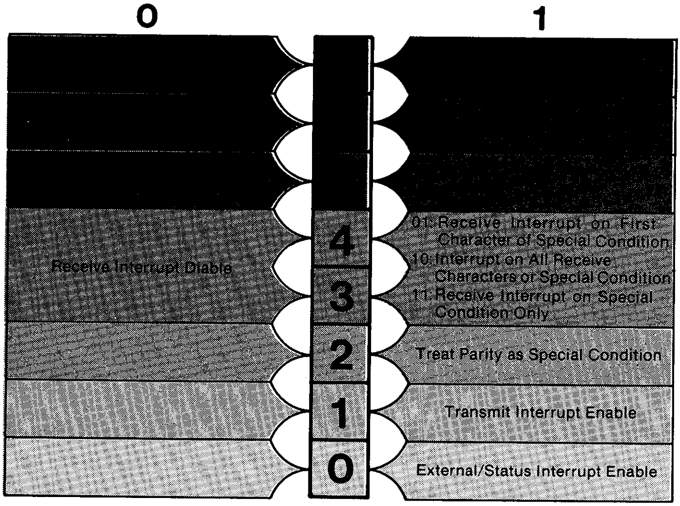
READ REGISTER 15



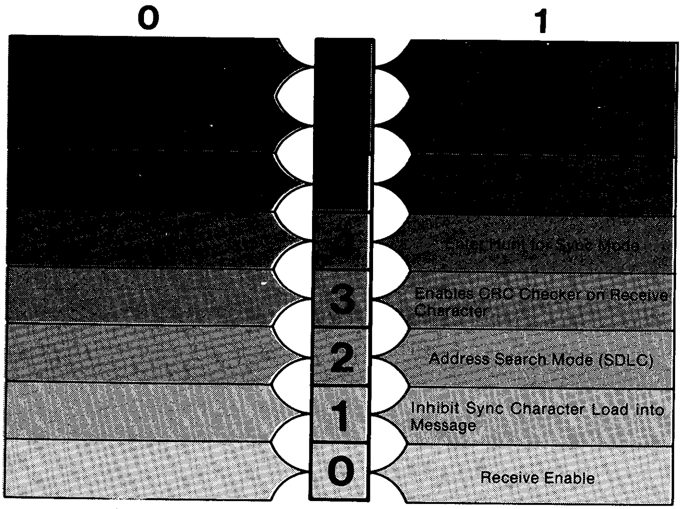
WRITE REGISTER 0



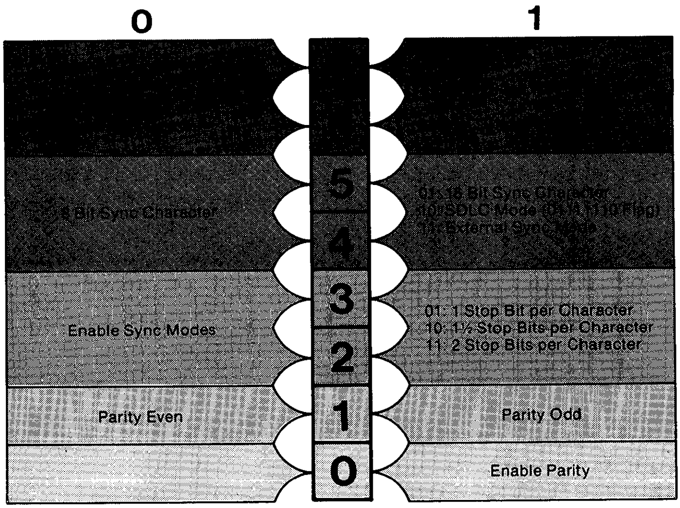
WRITE REGISTER 1



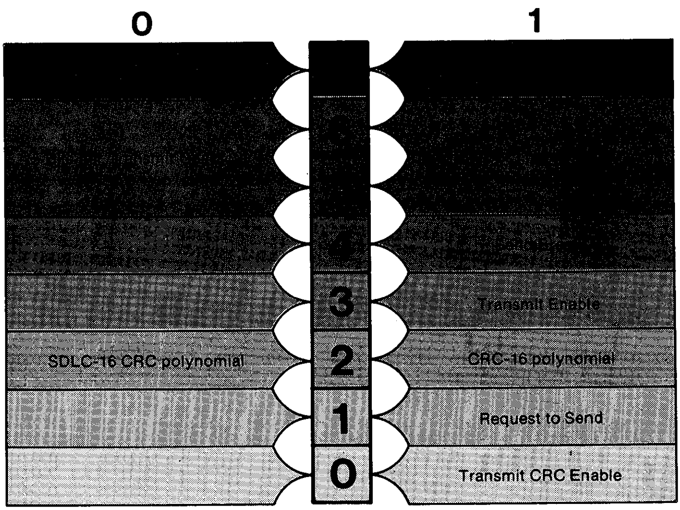
WRITE REGISTER 3



WRITE REGISTER 4



WRITE REGISTER 5



WRITE REGISTER 6

	MONO,8 BITS	MONO,16 BITS	BISYNC,16 BITS	BISYNC,12 BITS	SDLC	SDLC
0	SYNC0	SYNC0	SYNC0	1	ADR0	X
1	SYNC1	SYNC1	SYNC1	1	ADR1	X
2	SYNC2	SYNC2	SYNC2	1	ADR2	X
3	SYNC3	SYNC3	SYNC3	1	ADR3	X
4	SYNC4	SYNC4	SYNC4	SYNC0	ADR4	ADR4
5	SYNC5	SYNC5	SYNC5	SYNC1	ADR5	ADR5
6	SYNC6	SYNC0	SYNC6	SYNC2	ADR6	ADR6
7	SYNC7	SYNC1	SYNC7	SYNC3	ADR7	ADR7

WRITE REGISTER 7

	MONO,8 BITS	MONO,16 BITS	BISYNC,16 BITS	BISYNC,12 BITS	SDLC
0	SYNC0	X	SYNC8	SYNC4	0
1	SYNC1	X	SYNC9	SYNC5	1
2	SYNC2	SYNC0	SYNC10	SYNC6	1
3	SYNC3	SYNC1	SYNC11	SYNC7	1
4	SYNC4	SYNC2	SYNC12	SYNC8	1
5	SYNC5	SYNC3	SYNC13	SYNC9	1
6	SYNC6	SYNC4	SYNC14	SYNC10	1
7	SYNC7	SYNC5	SYNC15	SYNC11	0

WRITE REGISTER 9

0	Bit	1
No Reset	7, 6	01: Reset Channel B 10: Reset Channel A 11: Reset Both Channels
	5	
Status in High Bits of Vector	4	Status in Low Bits of Vector
	3	Master Interrupt Enable
	2	Disable Lower Chain
	1	No Vector
	0	Vector Includes 3 Bits of Status

WRITE REGISTER 10

0	Bit	1
[illegible] Generator Checker	7	Preset CRC Generator Checker to 1
NRZ	6, 5	01: NRZI 10: FM1 11: FMO
	4	Go Active on Poll
Transmit Sends Flags in Idle State	3	Transmit Sends Continuous 1's in Idle State
	2	On Underrun, Send Abort and Flag Instead of CRC and Flag (SDLC)
	1	Loop Mode
6-Bit Sync	0	8-Bit Sync

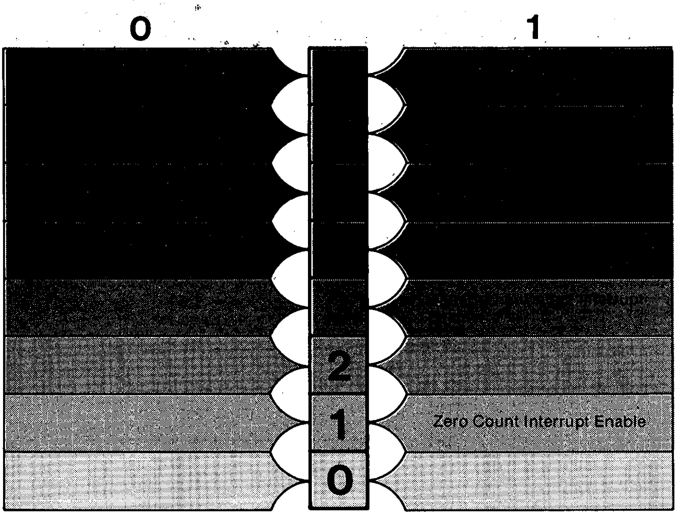
WRITE REGISTER 11

0	Bit	1
RTxC Pin Input is TTL Compatible	7	RTxC Pin Input is [illegible]
Receive Clock = -RTxC Pin	6, 5	01: Receive Clock = -TRxC Pin 10: Receive Clock = Baud Rate Generator Output 11: Receive Clock = DPLL Output
Transmit Clock = -RTxC Pin	4, 3	01: Transmit Clock = -TRxC Pin 10: Transmit Clock = Baud Rate Generator Output 11: Transmit Clock = DPLL Output
-TRxC Pin is Input	2	-TRxC Pin is Output
-TRxC Pin Out = Xtal Output	1, 0	01: -TRxC Pin Out = Transmit Clock 10: -TRxC Pin Out = Baud Rate Generator Output 11: -TRxC Pin Out = DPLL Output

WRITE REGISTER 14

0	Bit	1
Null Command	7, 6, 5	01: Enter Search Mode 10: Reset Missing Clock 11: Disable DPLL 100: Set DPLL Clock Source = Baud Rate Generator 101: Set DPLL Clock Source = -RTxC or Crystal Oscillator 110: Set DPLL FM Mode 111: Set DPLL NRZI Mode
	4	Transmitter Connected to Receiver and Transmits Out
	3	Transmitter Connected to Receiver as in Local Loopback but Receiver Listens to Input
-DTR/-REQ Pin Follows State of DTR	2	-DTR/-REQ Pin Equals 0 When Transmit Buffer Empty
	1	Baud Rate Generator Source
	0	Baud Rate Generator Enable

WRITE REGISTER 15



Functions

- READ DATA
  This function reads character data.
  INPUT: DATA

- READ SCC REGISTERS
  This function reads Read Register 0-15.
  OUTPUT: SCC REGISTER POINTER =
  register #
  INPUT: SCC REGISTER POINTER = data

- WRITE DATA
  This function writes character data.
  OUTPUT: DATA

- WRITE SCC REGISTER
  OUTPUT: SCC REGISTER POINTER =
  register #
  OUTPUT: SCC REGISTER POINTER = data

Sequencing and Timing

The SCC has direct addressing for the data register only. To access the other SCC registers, first write the register number to the SCC Register Pointer and then read or write the register using the SCC Register Pointer.

The SCC can operate in three basic modes to transfer data, status and control information:

- polling
- interrupt
- block

The block mode is not used on the AT&T Personal Computer 6300.

In the polling mode, Receive Character Available and Transmit Buffer Empty in Read Register 0 are examined before receiving and sending a character. All interrupt functions must be disable. To do this, clear the Master Bit Enable and set the No Vector bit in Write Register 9. Then clear Write Register 1. This disables specific types of interrupts.

For interrupt mode, the Master Interrupt Enable in Write Register 9 must be set. In addition, bits 3-4 in Write Register 1 specify interrupts on receive character conditions. Bit 1 in Write Register 1 enables interrupts on Transmit Buffer Empty. Bit 0 enables External/Status interrupt. This interrupt is caused by transmit underrun condition, a zero count in the baud rate generator, a break detection (Asynchronous Mode), Abort (SDLC Mode), or EOP (SDLC Loop Mode). Write Register 15 enables or disables more specific types of interrupts.

Interrupt sources have the following priority:

Receive Channel A
Transmit Channel A
External/Status Channel A
Receive Channel B
Transmit Channel B
Receive Channel B

A bit in Read Register 3 is set to indicate the highest priority device needing service. You can read the vector address of the interrupt service routine in Read Register 2 if it was programmed. If this vector is read on Channel B, it includes status bits. Vectors are initialized with Write Register 2. The interrupt service routine resets the Highest Interrupt Under Service in Write Register 0. Other interrupts are reset in Write Register 0.

To set the baud rate, first clear bits 0-1 in Write Register 14. Then load Write Register 12 and 13 with the time constant. Last set bit 0-1 in Write Register 14 to enable the baud rate generator. To use the baud rate, set the transmit and receive clocks in Write Register 11 to the baud rate generator.

To determine the time constant to use for a given baud rate, use this formula:

$$\frac{3{,}686{,}400}{(16)(2)(\text{Baud Rate})} - 2 = \text{Time Constant}$$

The following table states the divisors to use to obtain a given baud rate.

BAUD RATES USING 3.6864 MHZ CLOCK

BAUD RATE	TIME CONSTANT
110	1045
150	766
300	382
600	190
1200	94
2400	46
4800	22
9600	10

In Asynchronous mode, you initialize:

- Write Register 1 to disable DMA transfers and to enable or disable interrupts,
- Write Register 3 to set Receive Enable and the number of bits per receive character, to disable synchronous functions, and to enable or disable Auto Enable,
- Write Register 4 to set parity, stop bits, and data rate, and to disable synchronous mode,
- Write Register 5 to set Transmit Enable and the number of bits per transmit character, to enable or disable Request to Send, Data Terminal Ready and Send Break, and disable synchronous functions,
- Write Register 9 to force a reset and set interrupt parameters,
- Write Register 10 to choose an encoding method,
- Write Register 11 to select clock sources,
- Write Register 12 and 13 to set the baud rate time constant, and
- Write Register 15 to enable the baud rate generator.

To transmit a character, wait for Transmit Buffer Empty to be set in Read Register 1 of the SCC. Then write the character to the Data Register.

To receive a character, wait for Receive Character Ready in the Read Register 1 of the SCC. Then read the character from the Data Register.

In synchronous mode, you must transfer the data using interrupts. The External/Status interrupt is used to monitor the status of Clear to Send and Transmit Underrun/EOM latch.

In bisynchronous mode, you initialize:

- Write Register 0 to reset Transmit Underrun/EOM Latch, receive CRC checker, external/status interrupts, and enable interrupts on next receive character,
- Write Register 3 to enable the receiver, and to program Sync Character Load Inhibit, Enter Hunt For Sync Character, and number of bits per receive character,
- Write Register 4 to set parity, enable sync modes, number of bits per sync character, and clock mode,
- Write Register 5 to enable Transmit CRC, to request 16-bit CRC polynomial, enable transmit, transmit 8 bits per character, and to set Data Terminal Ready and Request to Send,
- Write Register 6 and 7 to set the sync bytes,
- Write Register 9 first to reset the hardware and later to set interrupts and vector variables,

- Write Register 10 to set length of sync character, the encoding method, to preset the CRC generator and to set the loop mode and go active on poll if wanted,
- Write Register 11 to set the clock sources, and
- Write Register 15 to set interrupt enable conditions.

The monosync transmitter is initialized as a bisynchronous transmitter with two exceptions:

- Only one sync character is written, and
- The 6-bit or 8-bit selection in Write Register 10 must be made.

In SDLC mode, you initialize:

- Write Register 0 to reset the transmit CRC generator after transmit enable has been done and to enable interrupts,
- Write Register 1 to enable interrupts,
- Write Register 3 to select bits per receive character, to set address search mode, to enable CRC receiver and receive enable,
- Write Register 4 to set SDLC mode before anything else is initialized and later to set clock mode,
- Write Register 5 to select the SDLC-CRC polynomial, to set Request to Send, Data Terminal Ready, transmit character length, transmit enable, and transmit CRC enable
- Write Register 6 to contain the secondary address field,

- Write Register 7 to contain flag character 01111110,
- Write Register 9 to reset the hardware and to set interrupt parameters,
- Write Register 10 to set loop mode, Go Active on Poll, Mark/Idle Flag, Abort on Underrun, the CRC preset condition, and the encoding mode,
- Write Register 11 to set clock sources,
- Write Register 14 to set the clock source for the DPLL, and
- Write Register 15 to set interrupt enable condition.

Sample Program

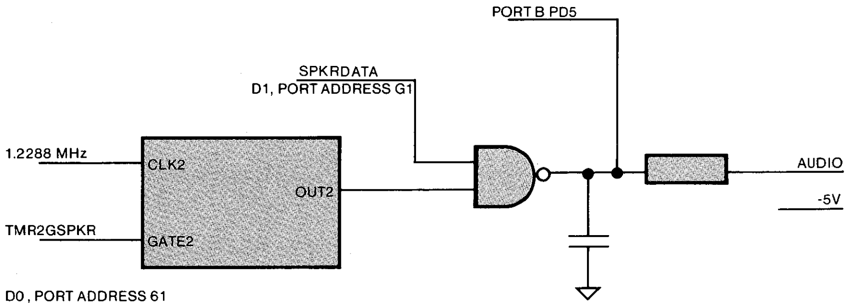
```
; RECEIVE
;       This routine receives one character
;
PTER          EQU 50
DATA          EQU 51
RECEIVE:
   MOV DX,PTER      ;Pointer Port Address
   XOR AL,AL        ;AL = 0
   OUT DX,AL        ;Select Read Register 0
REC1:
   IN AL,DX         ;Get Read Register 0
   TEST AL,1        ;Receive character available
   JZ REC1          ;Jump if no character
   MOV DX,DATA      ;Data Port Address
   IN AL,DX         ;Read character
   RET
```

Speaker

Functional Description

The speaker uses a permanent magnet speaker which is driven by one of two sources. Counter 2 of the Interval Timer can be programmed to automatically generate a pulse train. A bit in the Keyboard Control Register controls this pulse train. A bit in the Keyboard Control Register can also be programmed to manually generate a pulse train.

Block Diagram



Registers

PORT #	NAME	READ/ WRITE	DESCRIPTION
61	KEYBOARD CONTROL	W	Contains speaker enable and manual pulse train bits. See Keyboard documentation.
42	COUNTER 2	R/W	Counter for audio speaker tone generation. See Timer documentation.
43	TIMER CONTROL	W	Control register for Interval Timer. See Interval Timer documentation.

Functions

- AUTOMATIC PULSE TRAIN
  This function automatically generates an audible sound.
  OUTPUT:
  INTERVAL TIMER CONTROL REGISTER
  BITS 1-3 = 3, square wave rate generator
  BITS 4-5 = 3, set counter
  BITS 6-7 = 2, generator
  INTERVAL TIME COUNTER 2
  KEYBOARD CONTROL
  BIT 0= 1, turns on speaker

- MANUAL PULSE TRAIN
  This function manually generates an audible sound.
  OUTPUT: KEYBOARD CONTROL
  BIT 1 This bit is set and cleared to generate a pulse train.

Sequencing and Timing

Input to the timer is 1.2288MHz. To generate a 1.00 KHz tone with the audio speaker, a square wave rate generator is used with a count of 614 (1.2288MHz/2*614 = 1KHz).

Sample Program

```
;
BEEP
: This program sounds the beep manually
;
KEY_CONTROL            EQU 61
TIMER_CONTROL          EQU 43
BEEP:
   MOV DX, TIMER_CONTROL ;port address
   MOV AL, B8H           ;of timer
   OUT DX, AL            ;set channel 2
                         ;in mode 4
   MOV DX, KEY_CONTROL
   IN AL,DX
   MOV AH,AL
   OR AH, 01H            ;turn on Gate
   MOV BL, 80H
BEEP1:

   MOV AL, AH            ;restore value
   AND AL, OFDH
   OUT DX, AL
   MOV CX, 48H
   LOOP $
   MOV AL, AH
   OR AL, 02H
   OUT DX, AL
   MOVE CX, 48H
   LOOP $
   DEC BL
   JNZ BEEP1
   MOV AL,AH
   OUT DX,AL
   RET
```

Video Controller

Functional Description

The AT&T Personal Computer 6300 Display Controller interfaces the CPU to either monochrome or color displays. It uses a HD6845 CRT Controller. The Display Controller operates in two basic modes — text or all points addressable (APR) graphics. Several resolutions are available depending on the mode and display.

RESOLUTION	PC COMPATIBLE	GRAPHIC/ TEXT	COLOR/ MONOCHROME
80X25	YES	T	C/M
40X25	YES	T	C/M
640X400	NO	G	M
640X200	YES	G	M
320X200	YES	G	C

In text mode, character attributes include reverse video, blinking, highlight, hide and underline. In color mode if blinking is not requested, one of 16 colors can be chosen. Otherwise, one of 8 colors can be chosen.

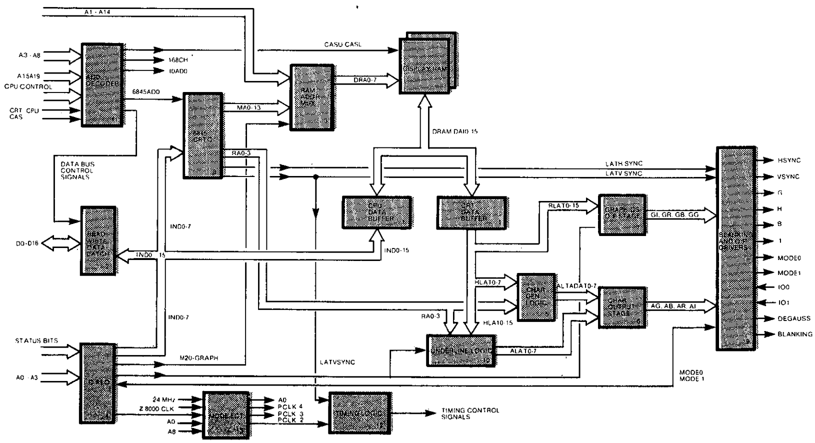
In graphic mode, each pixel on a color monitor is one of four selected colors. These four colors are from a choice of 16. In a monochrome monitor, these 16 colors are shades of gray from black to white.

The Display Controller has 32K of RAM to refresh one screen page.

The Display Controller can be upgraded with an optional board. This board gives you the following features:

- Up to three additional screen pages in RAM
- Software controlled look-up table for an international character set
- High resolution characters (16 X 16 pixels instead of 8 X 16 pixels)
- Hardware smooth scroll
- The ability to mix text and graphic modes simultaneously
- Up to 16 colors can be displayed at once

Block Diagram



Registers

PORT #	NAME	READ/ WRITE	DESCRIPTION
3D8	MODE SELECT REGISTER 1	W	See layout
3D9	COLOR SELECT REGISTER	W	See layout
3DA	STATUS REGISTER	R	See layout
3DE	MODE SELECT REGISTER 2	W	See layout
3D4	POINTER TO HD6845 REGISTER	W	
3D5	HD6845 DATA REGISTER	R/W	

HD6845 Registers

NO	NAME	READ/ WRITE	INITIALIZATION VALUE 40X25	80X25	GRAPHIC
0	HORIZONTAL TOTAL	W	38	71	38
1	HORIZONTAL DISPLAYED	W	28	50	28
2	HORIZONTAL SYNC POSITION	W	2D	5A	2D
3	HORIZONTAL SYNC WIDTH	W	06	0C	06
4	VERTICAL TOTAL	W	1F	1F	7F
5	VERTICAL TOTAL ADJUST	W	06	06	06
6	VERTICAL DISPLAYED	W	19	19	64
7	VERTICAL SYNC POSITION	W	1C	1C	70
8	INTERLACE MODE	W	02	02	02
9	MAX. SCAN LINE ADDRESS	W	07	07	01
A	CURSOR START LINE (SIZE)	W	06	06	06
B	CURSOR END LINE	W	07	07	07
C	ACTIVE PAGE START ADDR (H)	W	00	00	00
D	ACTIVE PAGE START ADDR (L)	W	00	00	00
E	CURSOR ADDRESS (H)	R/W			
F	CURSOR ADDRESS (L)	R/W			
10	LIGHT PEN (H)	R			
11	LIGHT PEN (L)	R			

Layout

MODE SELECT REGISTER 1

0	Bit	1
	7	
	6	
text mode, attribute char bit 7 defines background intensity	5	text mode, blinking character
medium resolution graphics	4	high resolution graphics
blank display	3	enable display
	2	
text mode	1	graphic mode
40 X 25 test mode	0	80 X 25 text mode

MODE SELECT REGISTER 2

0	Bit	1
enable 24 MHz clock	7	enable 19 MHz clock (Z8001)
	6	allows underline in color text
	5	
	4	
enable 16K RAM	3	enable 32K RAM
primary char set	2	alternate char set
	1	Request degauss
not 512 X 256 graphics	0	512 X 256 graphics

COLOR SELECT REGISTER 1 (Graphics Mode Only)

0	Bit	1
N/A	7	N/A
N/A	6	N/A
Select Foreground Palette (320 x 200) Cyan, Magenta, White	5	Select Foreground Palette (320 x 200) Green, Red, Yellow
Set Foreground Intensity Off (320 x 200)	4	Set Foreground Intensity On (320 x 200)
	3	Modify Color Selected By Bits 0-2
Modified Colors (Bit 3) 640 x 200 / 640 x 400 Foreground: 320 x 200 Background: 000: Dark Gray 001: Light Blue 010: Light Green 011: Light Cyan 100: Light Red 101: Light Magenta 110: Yellow 111: White	2, 1, 0	Select Color Shade 320 x 200 Background: 000: Black 001: Blue 101: Green 010: Cyan 100: Red 101: Magenta 110: Brown 111: Light Gray 640 x 200 / 640 x 400 Foreground: 001: Darkest Gray

STATUS REGISTER

0	Bit	1
Display option board present	7, 6	11: No expansion
	5, 4	01: 12" Color monitor 11: 12" Monochrome monitor
	3	First half of vertical retrace
	2	Light Pen switched off
	1	Light Pen triggered
	0	Horizontal Retracing

Text Mode

Every character position is defined by two bytes:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
I	R	G	B	I	R	G	B	ASCII CHAR CODE							

Background Foreground
Attribute Byte

If neither the underline or blinking capabilities are specified, the color choice for foreground and background with a color monitor is:

Black	Red
Blue	Magenta
Green	Brown
Cyan	Light Gray
Dark Gray	Light Red
Light Blue	Light Magenta
Light Green	Yellow
Light Cyan	White

With a monochrome monitor, the color choice is:

Black
Darkest Gray
.
Lightest Gray
.
White

The codes for common monochrome choices follow:

normal	00001111
reverse video	11110000
non-display black	00000000
non-display white	11111111

When the blinking capability is specified in Mode Select 1, then Bit 15 of the attribute byte specifies whether the character blinks.

If the underline capability is specified in the Mode Select Register 2, then Bit 11 specifies whether the character is underlined.

The first position in the left-hand corner of the screen is defined in the first two bytes of memory starting at B0000. The next position, one column to the right, is defined in the next two bytes of memory at B0002. The first character in the next row follows immediately after the definition for the last character in the first row. For 80 column X 25 rows, memory looks like this:

B0000	Row 1
B00A0	Row 2
B0140	Row 3
	.
	.
	.
	.
B0E60	Row 24
B0F00	Row 25

The 80 column display uses 4K of RAM and the 40 column display uses 2K of RAM. The rest of the 32K is used for multiple screen images called pages. There are either 16 or eight pages available.

Graphics Mode

In graphics mode, the display screen is a grid of pixels, the smallest displayable unit on a video monitor. In medium resolution, there are 640 across and either 200 or 400 down.

In high resolution, each pixel is defined by one bit. Bit 7 of each byte defines the first pixel and bit 0 defines the last pixel to be displayed. The background color is always black and is displayed when the pixel is off. When the pixel is on, the foreground color is one of 16 shades of gray as defined in the Color Select Register Bits 0-2 and 3.

In medium resolution, each pixel is defined by two bits:

7 6	5 4	3 2	1 0	
1st	2nd	3rd	4th	PIXELS

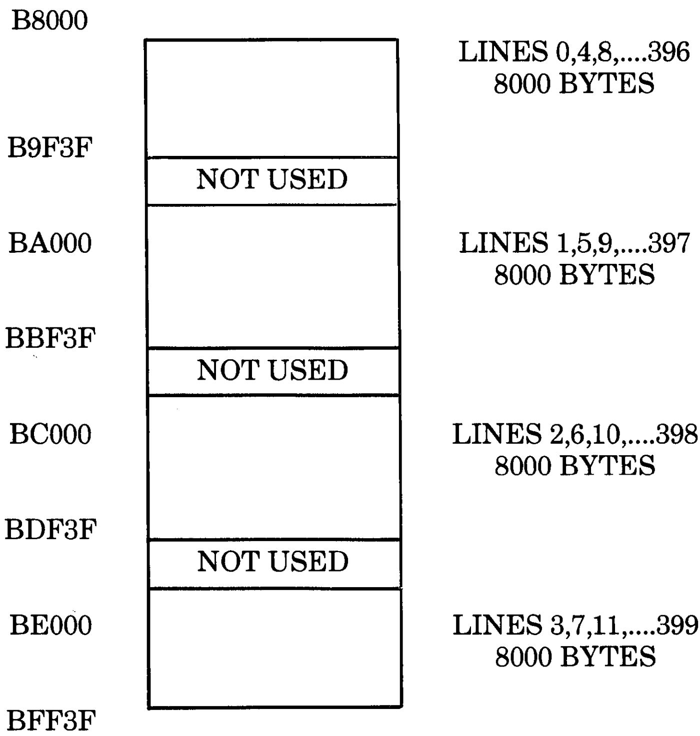
The value of two bits define one of four preselected colors.

0 — background color
1 — color 1
2 — color 2
3 — color 3

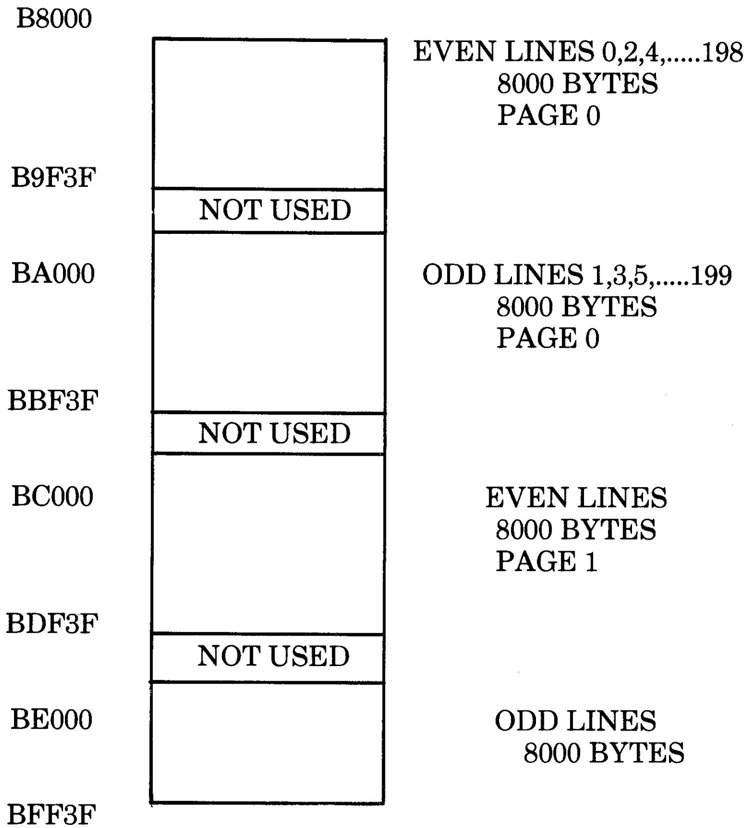
The background color is defined by Bits 0-3 in the Color Select Register. Colors 1, 2, and 3 are cyan, magenta and white if Bit 5 of the Color Select Register is zero and are green, red, and yellow if Bit 5 is one.

Unlike the text mode, rows of pixels do not follow one another in memory. The following memory maps illustrate the layouts.

MEMORY MAP (640 X 400 GRAPHICS MODE)



MEMORY MAP (320 X 200 GRAPHICS MODE)



Functions

- INITIALIZE HD6845
  This function initializes the 16 registers of the HD6845 with predetermined values.
  OUTPUT: POINTER TO HD6845 REGISTER
  Number of HD6845 register
  HD6845 DATA REGISTER
  Value of HD6845 register
  (Repeat 16 times for each register)

- SET MODE
  Set different mode characteristics such as text or graphics, type of graphics, blinking character, etc.
  OUTPUT: MODE SELECT 1
  MODE SELECT 2

- SET COLOR TYPE
  Choose the different color or shades of gray to display.
  OUTPUT: COLOR SELECT REGISTER

- SET CURSOR SIZE
  Set starting and ending line for cursor.
  OUTPUT: POINTER TO HD6845 REGISTER
  0AH
  HD6845 DATA REGISTER
  start line
  POINTER TO HD6845 REGISTER
  0BH
  HD6845 DATA REGISTER
  end line

- SET CURSOR POSITION
  Set cursor to location in memory.
  OUTPUT: POINTER TO HD6845 REGISTER
  0EH
  HD6845 DATA REGISTER
  most significant byte of address
  POINTER TO H36845 REGISTER
  0FH
  HD6845 DATA REGISTER
  least significant byte of address

- READ CURSOR POSITION
  Read the current position of the cursor.
  INPUT: POINTER TO HD6845 REGISTER
  0EH
  HD6845 DATA REGISTER
  most significant byte of address
  POINTER TO H36845 REGISTER
  0FH
  HD6845 DATA REGISTER
  least significant byte of address

- SET ACTIVE PAGE
  Set the address of the current page to display.
  OUTPUT: POINTER TO HD6845 REGISTER
  0CH
  HD6845 DATA REGISTER
  most significant byte of address
  POINTER TO H36845 REGISTER
  0DH
  HD6845 DATA REGISTER
  least significant byte of address

Sequencing and Timing

There are two methods of communicating with the video display. One is with I/O commands. This method is used to set the modes of operation, the cursor position, the cursor size or the current active page.

Scrambler ROM

For programs which do not use the regular BIOS calls, the 6845 CRT controller requires some additional circuitry to translate the data sent to it by application programs into the appropriate data for the display controller. This circuitry (called the scrambler circuitry) converts the IBM6845 register values to the values which the display controller requires. A listing of these conversions follows. For additional information see the AT&T Personal Computer 6300 reference manual.

```
SCRAM7 (USED ON REV. P1-P2A)
Unrecognized command
*DIS ..350
0000  39 39 39 39 39 39 39 39  39 39 39 39 39 39 39 39  9999999999999999
0010  39 39 39 39 39 39 39 39  39 39 39 39 39 39 39 39  9999999999999999
0020  39 39 39 39 39 39 39 39  39 39 39 39 39 39 39 39  9999999999999999
0030  39 39 39 39 39 39 39 39  39 39 39 39 39 39 39 39  9999999999999999
0040  39 39 39 39 39 39 39 39  39 39 39 39 39 39 39 39  9999999999999999
0050  39 39 39 39 39 39 73 73  73 73 73 73 73 73 73 73  999999ssssssssss
0060  73 73 73 73 73 73 73 73  73 73 73 73 73 73 73 73  ssssssssssssssss
0070  73 73 73 73 73 73 73 73  73 73 73 73 73 73 73 73  ssssssssssssssss
0080  00 01 02 03 04 05 06 07  08 09 0A 0B 0C 0D 0E 0F  ................
0090  10 11 12 13 14 15 16 17  18 19 1A 1B 1C 1D 1E 1F  ................
00A0  20 21 22 23 24 25 26 27  28 29 2A 2B 2C 2D 2E 2F   !"#$%&'()*+,-./
00B0  30 31 32 33 34 35 36 37  38 39 3A 3B 3C 3D 3E 3F  0123456789:;<=>?
00C0  40 41 42 43 44 45 46 47  48 49 4A 4B 4C 4D 4E 4F  @ABCDEFGHIJKLMNO
00D0  50 51 52 53 54 55 56 57  58 59 5A 5B 5C 5D 5E 5F  PQRSTUVWXYZ[\]^_
00E0  60 61 62 63 64 65 66 67  68 69 6A 6B 6C 6D 6E 6F  `abcdefghijklmno
00F0  70 71 72 73 74 75 76 77  78 79 7A 7B 7C 7D 7E 7F  pqrstuvwxyz{|}~.
0100  00 01 02 03 04 05 06 07  08 09 0A 0B 0C 0D 0E 0F  ................
0110  10 11 12 13 14 15 16 17  18 19 1A 1B 1C 1D 1E 1F  ................
0120  20 21 22 23 24 25 26 27  28 29 2A 2B 2C 2D 2E 2F   !"#$%&'()*+,-./
0130  30 31 32 33 34 35 36 37  38 39 3A 3B 3C 3D 3E 3F  0123456789:;<=>?
0140  40 41 42 43 44 45 46 47  48 49 4A 4B 4C 4D 4E 4F  @ABCDEFGHIJKLMNO
0150  50 51 52 53 54 55 56 57  58 59 5A 5B 5C 5D 5E 5F  PQRSTUVWXYZ[\]^_
0160  60 61 62 63 64 65 66 67  68 69 6A 6B 6C 6D 6E 6F  `abcdefghijklmno
0170  70 71 72 73 74 75 76 77  78 79 7A 7B 7C 7D 7E 7F  pqrstuvwxyz{|}~.
0180  00 01 02 03 04 05 06 07  08 09 0A 0B 0C 0D 0E 0F  ................
0190  10 11 12 13 14 15 16 17  18 19 1A 1B 1C 1D 1E 1F  ................
01A0  20 21 22 23 24 25 26 27  28 29 2A 2B 2C 2D 2E 2F   !"#$%&'()*+,-./
01B0  30 31 32 33 34 35 36 37  38 39 3A 3B 3C 3D 3E 3F  0123456789:;<=>?
01C0  40 41 42 43 44 45 46 47  48 49 4A 4B 4C 4D 4E 4F  @ABCDEFGHIJKLMNO
01D0  50 51 52 53 54 55 56 57  58 59 5A 5B 5C 5D 5E 5F  PQRSTUVWXYZ[\]^_
01E0  60 61 62 63 64 65 66 67  68 69 6A 6B 6C 6D 6E 6F  `abcdefghijklmno
01F0  70 07 72 73 74 75 76 77  78 79 7A 7B 7C 7D 7E 7F  p.rstuvwxyz{|}~.
```

```
0200  02 03 04 05 06 07 08 09  0A 0B 0C 0D 0E 0F 10 11  ................
0210  12 13 14 15 16 17 18 19  1A 1B 1C 1D 1E 1F 20 21  .............. !
0220  22 23 24 25 26 27 28 29  2A 2B 2C 2D 2E 2F 30 31  "#$%&'()*+,-./01
0230  32 33 34 35 36 37 38 39  3A 3B 3C 3D 3E 3F 40 41  23456789:;<=>?@A
0240  42 43 44 45 46 47 48 49  4A 4B 4C 4D 4E 4F 50 51  BCDEFGHIJKLMNOPQ
0250  52 53 54 55 56 57 58 59  5A 5B 5C 5D 5E 5F 60 61  RSTUVWXYZ[\]^_'a
0260  62 63 64 65 66 67 68 69  6A 6B 6C 6D 6E 6F 70 71  bcdefghijklmnopq
0270  72 73 74 75 76 77 78 79  7A 7B 7C 7D 7E 7F 80 81  rstuvwxyz{|}~...
0280  00 01 02 03 04 05 06 07  08 09 0A 0B 0C 0D 0E 0F  ................
0290  10 11 12 13 14 15 16 17  18 19 1A 1B 1C 1D 1E 1F  ................
02A0  20 21 22 23 24 25 26 27  28 29 2A 2B 2C 2D 2E 2F   !"#$%&'()*+,-./
02B0  30 31 32 33 34 35 36 37  38 39 3A 3B 3C 3D 3E 3F  0123456789:;<=>?
02C0  40 41 42 43 44 45 46 47  48 49 4A 4B 4C 4D 4E 4F  @ABCDEFGHIJKLMNO
02D0  50 51 52 53 54 55 56 57  58 59 5A 5B 5C 5D 5E 5F  PQRSTUVWXYZ[\]^_
02E0  60 61 62 63 64 65 66 67  68 69 6A 6B 6C 6D 6E 6F  'abcdefghijklmno
02F0  70 71 72 73 74 75 76 77  78 79 7A 7B 7C 7D 7E 7F  pqrstuvwxyz{|}~.
0300  06 06 06 06 06 06 06 06  06 06 09 06 0C 0C 0C 0C  ................
0310  09 09 09 09 09 09 09 09  09 09 09 09 09 09 09 09  ................
0320  09 09 09 09 09 09 09 09  09 09 09 09 09 09 09 09  ................
0330  09 09 09 09 09 09 09 09  09 09 09 09 09 09 09 09  ................
0340  09 09 09 09 09 09 09 09  09 09 09 09 09 09 09 09  ................
0350  09 09 09 09 09 09 09 09  09 09 09 09 09 09 09 09  ................
0360  09 09 09 09 09 09 09 09  09 09 09 09 09 09 09 09  ................
0370  09 09 09 09 09 09 09 09  09 09 09 09 09 09 09 09  ................
0380  09 09 09 09 09 09 09 09  09 09 09 09 09 09 09 09  ................
0390  09 09 09 09 09 09 09 09  09 09 09 09 09 09 09 09  ................
03A0  09 09 09 09 09 09 09 09  09 09 09 09 09 09 09 09  ................
03B0  09 09 09 09 09 09 09 09  09 09 09 09 09 09 09 09  ................
03C0  09 09 09 09 09 09 09 09  09 09 09 09 09 09 09 09  ................
03D0  09 09 09 09 09 09 09 09  09 09 09 09 09 09 09 09  ................
03E0  09 09 09 09 09 09 09 09  09 09 09 09 09 09 09 09  ................
03F0  09 09 09 09 09 09 09 09  09 09 09 09 09 09 09 06  ................
0400  1A 1A 1A 1A 1A 1A 1A 1A  1A 1A 1A 1A 1A 1A 1A 1A  ................
0410  1A 1A 1A 1A 1A 1A 1A 1A  1A 1A 1A 1A 1A 1A 1A 1A  ................
0420  1A 1A 1A 1A 1A 1A 1A 1A  1A 1A 1A 1A 1A 1A 1A 1A  ................
0430  1A 1A 1A 1A 1A 1A 1A 1A  1A 1A 1A 1A 1A 1A 1A 1A  ................
0440  1A 6B 6B 6B 6B 6B 6B 6B  6B 6B 6B 6B 6B 6B 6B 6B  .kkkkkkkkkkkkkkk
0450  6B 6B 6B 6B 6B 6B 6B 6B  6B 6B 6B 6B 6B 6B 6B 6B  kkkkkkkkkkkkkkkk
0460  6B 6B 6B 6B 6B 6B 6B 6B  6B 6B 6B 6B 6B 6B 6B 6B  kkkkkkkkkkkkkkkk
0470  6B 6B 6B 6B 6B 6B 6B 6B  6B 6B 6B 6B 6B 6B 6B 6B  kkkkkkkkkkkkkkkk
0480  6B 6B 6B 6B 6B 6B 6B 6B  6B 6B 6B 6B 6B 6B 6B 6B  kkkkkkkkkkkkkkkk
0490  6B 6B 6B 6B 6B 6B 6B 6B  6B 6B 6B 6B 6B 6B 6B 6B  kkkkkkkkkkkkkkkk
04A0  6B 6B 6B 6B 6B 6B 6B 6B  6B 6B 6B 6B 6B 6B 6B 6B  kkkkkkkkkkkkkkkk
04B0  6B 6B 6B 6B 6B 6B 6B 6B  6B 6B 6B 6B 6B 6B 6B 6B  kkkkkkkkkkkkkkkk
04C0  6B 6B 6B 6B 6B 6B 6B 6B  6B 6B 6B 6B 6B 6B 6B 6B  kkkkkkkkkkkkkkkk
04D0  6B 6B 6B 6B 6B 6B 6B 6B  6B 6B 6B 6B 6B 6B 6B 6B  kkkkkkkkkkkkkkkk
04E0  6B 6B 6B 6B 6B 6B 6B 6B  6B 6B 6B 6B 6B 6B 6B 6B  kkkkkkkkkkkkkkkk
04F0  6B 6B 6B 6B 6B 6B 6B 6B  6B 6B 6B 6B 6B 6B 6B 6B  kkkkkkkkkkkkkkkk
0500  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0510  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0520  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0530  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
```

```
0540  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0550  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0560  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0570  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0580  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0590  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
05A0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
05B0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
05C0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
05D0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
05E0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
05F0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0600  00 01 02 03 04 05 06 07  08 09 0A 0B 0C 0D 0E 0F  ................
0610  10 11 12 13 14 15 16 17  18 19 1A 1B 1C 1D 1E 1F  ................
0620  20 21 22 23 24 25 26 27  28 29 2A 2B 2C 2D 2E 2F   !"#$%&'()*+,-./
0630  30 31 32 33 34 35 36 37  38 39 3A 3B 3C 3D 3E 3F  0123456789:;<=>?
0640  40 41 42 43 44 45 46 47  48 49 4A 4B 4C 4D 4E 4F  @ABCDEFGHIJKLMNO
0650  50 51 52 53 54 55 56 57  58 59 5A 5B 5C 5D 5E 5F  PQRSTUVWXYZ[\]^_
0660  60 61 62 63 64 65 66 67  68 69 6A 6B 6C 6D 6E 6F  'abcdefghijklmno
0670  70 71 72 73 74 75 76 77  78 79 7A 7B 7C 7D 7E 7F  pqrstuvwxyz{|}~.
0680  80 81 82 83 84 85 86 87  88 89 8A 8B 8C 8D 8E 8F  ................
0690  90 91 92 93 94 95 96 97  98 99 9A 9B 9C 9D 9E 9F  ................
06A0  A0 A1 A2 A3 A4 A5 A6 A7  A8 A9 AA AB AC AD AE AF  ................
06B0  B0 B1 B2 B3 B4 B5 B6 B7  B8 B9 BA BB BC BD BE BF  ................
06C0  C0 C1 C2 C3 C4 C5 C6 C7  C8 C9 CA CB CC CD CE CF  ................
06D0  D0 D1 D2 D3 D4 D5 D6 D7  D8 D9 DA DB DC DD DE DF  ................
06E0  E0 E1 E2 E3 E4 E5 E6 E7  E8 E9 EA EB EC ED EE EF  ................
06F0  F0 F1 F2 F3 F4 F5 F6 F7  F8 F9 FA FB FC FD FE FF  ................
0700  19 19 19 19 19 19 19 19  19 19 19 19 19 19 19 19  ................
0710  19 19 19 19 19 19 19 19  19 19 19 19 19 19 19 19  ................
0720  19 19 19 19 19 19 19 19  19 19 19 19 19 19 19 19  ................
0730  19 19 19 19 19 19 19 19  19 19 19 19 19 19 19 19  ................
0740  19 64 64 64 64 64 64 64  64 64 64 64 64 64 64 64  .ddddddddddddddd
0750  64 64 64 64 64 64 64 64  64 64 64 64 64 64 64 64  dddddddddddddddd
0760  64 64 64 64 64 64 64 64  64 64 64 64 64 64 64 64  dddddddddddddddd
0770  64 64 64 64 64 64 64 64  64 64 64 64 64 64 64 64  dddddddddddddddd
0780  64 64 64 64 64 64 64 64  64 64 64 64 64 64 64 64  dddddddddddddddd
0790  64 64 64 64 64 64 64 64  64 64 64 64 64 64 64 64  dddddddddddddddd
07A0  64 64 64 64 64 64 64 64  64 64 64 64 64 64 64 64  dddddddddddddddd
07B0  64 64 64 64 64 64 64 64  64 64 64 64 64 64 64 64  dddddddddddddddd
07C0  64 64 64 64 64 64 64 64  64 64 64 64 64 64 64 64  dddddddddddddddd
07D0  64 64 64 64 64 64 64 64  64 64 64 64 64 64 64 64  dddddddddddddddd
07E0  64 64 64 64 64 64 64 64  64 64 64 64 64 64 64 64  dddddddddddddddd
07F0  64 64 64 64 64 64 64 64  64 64 64 64 64 64 64 64  dddddddddddddddd
0800  00 00 00 03 00 00 00 00  00 00 00 00 00 00 00 00  ................
0810  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0820  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0830  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0840  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0850  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0860  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0870  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
```

```
0880  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0890  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
08A0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
08B0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
08C0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
08D0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
08E0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
08F0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0900  01 03 05 03 09 0B 0D 0F  11 13 15 17 19 1B 1D 0F  ................
0910  21 23 25 27 29 2B 2D 2F  31 33 35 37 39 3B 3D 3F  !#%')+-/13579;=?
0920  41 43 45 47 49 4B 4D 4F  51 53 55 57 59 5B 5D 5F  ACEGIKMOQSUWY[]_
0930  61 63 65 67 69 6B 6D 6F  71 73 75 77 79 7B 7D 7F  acegikmoqsuwy{}.
0940  81 83 85 87 89 8B 8D 8F  91 93 95 97 99 9B 9D 9F  ................
0950  A1 A3 A5 A7 A9 AB AD AF  B1 B3 B5 B7 B9 BB BD BF  ................
0960  C1 C3 C5 C7 C9 CB CD CF  D1 D3 D5 D7 D9 DB DD DF  ................
0970  E1 E3 E5 E7 E9 EB ED EF  F1 F3 F5 F7 F9 FB FD FF  ................
0980  01 03 05 07 09 0B 0D 0F  11 13 15 17 19 1B 1D 1F  ................
0990  21 23 25 27 29 2B 2D 2F  31 33 35 37 39 3B 3D 3F  !#%')+-/13579;=?
09A0  41 43 45 47 49 4B 4D 4F  51 53 55 57 59 5B 5D 5F  ACEGIKMOQSUWY[]_
09B0  61 63 65 67 69 6B 6D 6F  71 73 75 77 79 7B 7D 7F  acegikmoqsuwy{}.
09C0  81 83 85 87 89 8B 8D 8F  91 93 95 97 99 9B 9D 9F  ................
09D0  A1 A3 A5 A7 A9 AB AD AF  B1 B3 B5 B7 B9 BB BD BF  ................
09E0  C1 C3 C5 C7 C9 CB CD CF  D1 D3 D5 D7 D9 DB DD DF  ................
09F0  E1 E3 E5 E7 E9 EB ED EF  F1 F3 F5 F7 F9 FB FD FF  ................
0A00  00 02 04 06 08 0A 0C 0E  10 10 10 10 10 10 10 10  ................
0A10  10 10 10 10 10 10 10 10  10 10 10 10 10 10 10 10  ................
0A20  20 22 24 26 28 2A 2C 2E  30 30 30 30 30 30 30 30   "$&(*,.00000000
0A30  30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  0000000000000000
0A40  40 42 44 46 48 4A 4C 4E  50 50 50 50 50 50 50 50  @BDFHJLNPPPPPPPP
0A50  50 50 50 50 50 50 50 50  50 50 50 50 50 50 50 50  PPPPPPPPPPPPPPPP
0A60  60 62 64 66 68 6A 6C 6E  70 70 70 70 70 70 70 70  'bdfhjlnpppppppp
0A70  70 70 70 70 70 70 70 70  70 70 70 70 70 70 70 70  pppppppppppppppp
0A80  00 02 04 06 08 0A 0C 0E  10 10 10 10 10 10 10 10  ................
0A90  10 10 10 10 10 10 10 10  10 10 10 10 10 10 10 10  ................
0AA0  20 22 24 26 28 2A 2C 2E  30 30 30 30 30 30 30 30   "$&(*,.00000000
0AB0  30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  0000000000000000
0AC0  40 42 44 46 48 4A 4C 4E  50 50 50 50 50 50 50 50  @BDFHJLNPPPPPPPP
0AD0  50 50 50 50 50 50 50 50  50 50 50 50 50 50 50 50  PPPPPPPPPPPPPPPP
0AE0  60 62 64 66 68 6A 6C 6E  70 70 70 70 70 70 70 70  'bdfhjlnpppppppp
0AF0  70 70 70 70 70 70 70 70  70 70 70 70 70 70 70 70  pppppppppppppppp
0B00  00 02 04 06 08 0A 0C 0E  10 12 14 16 18 1A 1C 1E  ................
0B10  20 22 24 26 28 2A 2C 2E  30 32 34 36 38 3A 3C 3E   "$&(*,.02468:<>
0B20  40 42 44 46 48 4A 4C 4E  50 52 54 56 58 5A 5C 5E  @BDFHJLNPRTVXZ\
0B30  60 62 64 66 68 6A 6C 6E  70 72 74 76 78 7A 7C 7E  'bdfhjlnprtvxz ~
0B40  80 82 84 86 88 8A 8C 8E  90 92 94 96 98 9A 9C 9E  ................
0B50  A0 A2 A4 A6 A8 AA AC AE  B0 B2 B4 B6 B8 BA BC BE  ................
0B60  C0 C2 C4 C6 C8 CA CC CE  D0 D2 D4 D6 D8 DA DC DE  ................
0B70  E0 E2 E4 E6 E8 EA EC EE  F0 F2 F4 F6 F8 FA FC FE  ................
0B80  00 02 04 06 08 0A 0C 0E  10 12 14 16 18 1A 1C 1E  ................
0B90  20 22 24 26 28 2A 2C 2E  30 32 34 36 38 3A 3C 3E   "$&(*,.02468:<>
0BA0  40 42 44 46 48 4A 4C 4E  50 52 54 56 58 5A 5C 5E  @BDFHJLNPRTVXZ\
0BB0  60 62 64 66 68 6A 6C 6E  70 72 74 76 78 7A 7B 7E  'bdfhjlnprtvxz ~
```

```
0BC0  80 82 84 86 88 8A 8C 8E  90 92 94 96 98 9A 9C 9E  ................
0BD0  A0 A2 A4 A6 A8 AA AC AE  B0 B2 B4 B6 B8 BA BC BE  ................
0BE0  C0 C2 C4 C6 C8 CA CC CE  D0 D2 D4 D6 D8 DA DC DE  ................
0BF0  E0 E2 E4 E6 E8 EA EC EE  F0 F2 F4 F6 F8 FA FC FE  ................
0C00  00 01 02 03 04 05 06 07  08 09 0A 0B 0C 0D 0E 0F  ................
0C10  10 11 12 13 14 15 16 17  18 19 1A 1B 1C 1D 1E 1F  ................
0C20  20 21 22 23 24 25 26 27  28 29 2A 2B 2C 2D 2E 2F   !"#$%&'()*+,-./
0C30  30 31 32 33 34 35 36 37  38 39 3A 3B 3C 3D 3E 3F  0123456789:;<=>?
0C40  40 41 42 43 44 45 46 47  48 49 4A 4B 4C 4D 4E 4F  @ABCDEFGHIJKLMNO
0C50  50 51 52 53 54 55 56 57  58 59 5A 5B 5C 5D 5E 5F  PQRSTUVWXYZ[\]^_
0C60  60 61 62 63 64 65 66 67  68 69 6A 6B 6C 6D 6E 6F  'abcdefghijklmno
0C70  70 71 72 73 74 75 76 77  78 79 7A 7B 7C 7D 7E 7F  pqrstuvwxyz{|}~.
0C80  80 81 82 83 84 85 86 87  88 89 8A 8B 8C 8D 8E 8F  ................
0C90  90 91 92 93 94 95 96 97  98 99 9A 9B 9C 9D 9E 9F  ................
0CA0  A0 A1 A2 A3 A4 A5 A6 A7  A8 A9 AA AB AC AD AE AF  ................
0CB0  B0 B1 B2 B3 B4 B5 B6 B7  B8 B9 BA BB BC BD BE BF  ................
0CC0  C0 C1 C2 C3 C4 C5 C6 C7  C8 C9 CA CB CC CD CE CF  ................
0CD0  D0 D1 D2 D3 D4 D5 D6 D7  D8 D9 DA DB DC DD DE DF  ................
0CE0  E0 E1 E2 E3 E4 E5 E6 E7  E8 E9 EA EB EC ED EE EF  ................
0CF0  F0 F1 F2 F3 F4 F5 F6 F7  F8 F9 FA FB FC FD FE FF  ................
0D00  00 01 02 03 04 05 06 07  08 09 0A 0B 0C 0D 0E 0F  ................
0D10  10 11 12 13 14 15 16 17  18 19 1A 1B 1C 1D 1E 1F  ................
0D20  20 21 22 23 24 25 26 27  28 29 2A 2B 2C 2D 2E 2F   !"#$%&'()*+,-./
0D30  30 31 32 33 34 35 36 37  38 39 3A 3B 3C 3D 3E 3F  0123456789:;<=>?
0D40  40 41 42 43 44 45 46 47  48 49 4A 4B 4C 4D 4E 4F  @ABCDEFGHIJKLMNO
0D50  50 51 52 53 54 55 56 57  58 59 5A 5B 5C 5D 5E 5F  PQRSTUVWXYZ[\]^_
0D60  60 61 62 63 64 65 66 67  68 69 6A 6B 6C 6D 6E 6F  'abcdefghijklmno
0D70  70 71 72 73 74 75 76 77  78 79 7A 7B 7C 7D 7E 7F  pqrstuvwxyz{|}~.
0D80  80 81 82 83 84 85 86 87  88 89 8A 8B 8C 8D 8E 8F  ................
0D90  90 91 92 93 94 95 96 97  98 99 9A 9B 9C 9D 9E 9F  ................
0DA0  A0 A1 A2 A3 A4 A5 A6 A7  A8 A9 AA AB AC AD AE AF  ................
0DB0  B0 B1 B2 B3 B4 B5 B6 B7  B8 B9 BA BB BC BD BE BF  ................
0DC0  C0 C1 C2 C3 C4 C5 C6 C7  C8 C9 CA CB CC CD CE CF  ................
0DD0  D0 D1 D2 D3 D4 D5 D6 D7  D8 D9 DA DB DC DD DE DF  ................
0DE0  E0 E1 E2 E3 E4 E5 E6 E7  E8 E9 EA EB EC ED EE EF  ................
0DF0  F0 F1 F2 F3 F4 F5 F6 F7  F8 F9 FA FB FC FD FE FF  ................
0E00  00 01 02 03 04 05 06 07  08 09 0A 0B 0C 0D 0E 0F  ................
0E10  10 11 12 13 14 15 16 17  18 19 1A 1B 1C 1D 1E 1F  ................
0E20  20 21 22 23 24 25 26 27  28 29 2A 2B 2C 2D 2E 2F   !"#$%&'()*+,-./
0E30  30 31 32 33 34 35 36 37  38 39 3A 3B 3C 3D 3E 3F  0123456789:;<=>?
0E40  40 41 42 43 44 45 46 47  48 49 4A 4B 4C 4D 4E 4F  @ABCDEFGHIJKLMNO
0E50  50 51 52 53 54 55 56 57  58 59 5A 5B 5C 5D 5E 5F  PQRSTUVWXYZ[\]^_
0E60  60 61 62 63 64 65 66 67  68 69 6A 6B 6C 6D 6E 6F  'abcdefghijklmno
0E70  70 71 72 73 74 75 76 77  78 79 7A 7B 7C 7D 7E 7F  pqrstuvwxyz{|}~.
0E80  80 81 82 83 84 85 86 87  88 89 8A 8B 8C 8D 8E 8F  ................
0E90  90 91 92 93 94 95 96 97  98 99 9A 9B 9C 9D 9E 9F  ................
0EA0  A0 A1 A2 A3 A4 A5 A6 A7  A8 A9 AA AB AC AD AE AF  ................
0EB0  B0 B1 B2 B3 B4 B5 B6 B7  B8 B9 BA BB BC BD BE BF  ................
0EC0  C0 C1 C2 C3 C4 C5 C6 C7  C8 C9 CA CB CC CD CE CF  ................
0ED0  D0 D1 D2 D3 D4 D5 D6 D7  D8 D9 DA DB DC DD DE DF  ................
0EE0  E0 E1 E2 E3 E4 E5 E6 E7  E8 E9 EA EB EC ED EE EF  ................
0EF0  F0 F1 F2 F3 F4 F5 F6 F7  F8 F9 FA FB FC FD FE FF  ................
```

```
0F00   00 01 02 03 04 05 06 07   08 09 0A 0B 0C 0D 0E 0F   ................
0F10   10 11 12 13 14 15 16 17   18 19 1A 1B 1C 1D 1E 1F   ................
0F20   20 21 22 23 24 25 26 27   28 29 2A 2B 2C 2D 2E 2F    !"#$%&'()*+,-./
0F30   30 31 32 33 34 35 36 37   38 39 3A 3B 3C 3D 3E 3F   0123456789:;<=>?
0F40   40 41 42 43 44 45 46 47   48 49 4A 4B 4C 4D 4E 4F   @ABCDEFGHIJKLMNO
0F50   50 51 52 53 54 55 56 57   58 59 5A 5B 5C 5D 5E 5F   PQRSTUVWXYZ[\]^_
0F60   60 61 62 63 64 65 66 67   68 69 6A 6B 6C 6D 6E 6F   'abcdefghijklmno
0F70   70 71 72 73 74 75 76 77   78 79 7A 7B 7C 7D 7E 7F   pqrstuvwxyz{|}~.
0F80   80 81 82 83 84 85 86 87   88 89 8A 8B 8C 8D 8E 8F   ................
0F90   90 91 92 93 94 95 96 97   98 99 9A 9B 9C 9D 9E 9F   ................
0FA0   A0 A1 A2 A3 A4 A5 A6 A7   A8 A9 AA AB AC AD AE AF   ................
0FB0   B0 B1 B2 B3 B4 B5 B6 B7   B8 B9 BA BB BC BD BE BF   ................
0FC0   C0 C1 C2 C3 C4 C5 C6 C7   C8 C9 CA CB CC CD CE CF   ................
0FD0   D0 D1 D2 D3 D4 D5 D6 D7   D8 D9 DA DB DC DD DE DF   ................
0FE0   E0 E1 E2 E3 E4 E5 E6 E7   E8 E9 EA EB EC ED EE EF   ................
0FF0   F0 F1 F2 F3 F4 F5 F6 F7   F8 F9 FA FB FC FD FE FF   ................
At end of current range
*
```

```
SCRAM8B (USED ON REV. P3 AND LATER)
Unrecognized command
*DIS ..350
0000   39 39 39 39 39 39 39 39   39 39 39 39 39 39 39 39   9999999999999999
0010   39 39 39 39 39 39 39 39   39 39 39 39 39 39 39 39   9999999999999999
0020   39 39 39 39 39 39 39 39   39 39 39 39 39 39 39 39   9999999999999999
0030   39 39 39 39 39 39 39 39   39 39 39 39 39 39 39 39   9999999999999999
0040   39 39 39 39 39 39 39 39   39 39 39 39 39 39 39 39   9999999999999999
0050   39 39 39 39 39 39 73 73   73 73 73 73 73 73 73 73   999999ssssssssss
0060   73 73 73 73 73 73 73 73   73 73 73 73 73 73 73 73   ssssssssssssssss
0070   73 73 73 73 73 73 73 73   73 73 73 73 73 73 73 73   ssssssssssssssss
0080   00 01 02 03 04 05 06 07   08 09 0A 0B 0C 0D 0E 0F   ................
0090   10 11 12 13 14 15 16 17   18 19 1A 1B 1C 1D 1E 1F   ................
00A0   20 21 22 23 24 25 26 27   28 29 2A 2B 2C 2D 2E 2F    !"#$%&'()*+,-./
00B0   30 31 32 33 34 35 36 37   38 39 3A 3B 3C 3D 3E 3F   0123456789:;<=>?
00C0   40 41 42 43 44 45 46 47   48 49 4A 4B 4C 4D 4E 4F   @ABCDEFGHIJKLMNO
00D0   50 51 52 53 54 55 56 57   58 59 5A 5B 5C 5D 5E 5F   PQRSTUVWXYZ[\]^_
00E0   60 61 62 63 64 65 66 67   68 69 6A 6B 6C 6D 6E 6F   'abcdefghijklmno
00F0   70 71 72 73 74 75 76 77   78 79 7A 7B 7C 7D 7E 7F   pqrstuvwxyz{|}~.
0100   00 01 02 03 04 05 06 07   08 09 0A 0B 0C 0D 0E 0F   ................
0110   10 11 12 13 14 15 16 17   18 19 1A 1B 1C 1D 1E 1F   ................
0120   20 21 22 23 24 25 26 27   28 29 2A 2B 2C 2D 2E 2F    !"#$%&'()*+,-./
0130   30 31 32 33 34 35 36 37   38 39 3A 3B 3C 3D 3E 3F   0123456789:;<=>?
0140   40 41 42 43 44 45 46 47   48 49 4A 4B 4C 4D 4E 4F   @ABCDEFGHIJKLMNO
0150   50 51 52 53 54 55 56 57   58 59 5A 5B 5C 5D 5E 5F   PQRSTUVWXYZ[\]^_
0160   60 61 62 63 64 65 66 67   68 69 6A 6B 6C 6D 6E 6F   'abcdefghijklmno
0170   70 71 72 73 74 75 76 77   78 79 7A 7B 7C 7D 7E 7F   pqrstuvwxyz{|}~.
0180   80 81 82 83 84 85 86 87   88 89 8A 8B 8C 8D 8E 8F   ................
0190   90 91 92 93 94 95 96 97   98 99 9A 9B 9C 9D 9E 9F   ................
01A0   A0 A1 A2 A3 A4 A5 A6 A7   A8 A9 AA AB AC AD AE AF   ................
01B0   B0 B1 B2 B3 B4 B5 B6 B7   B8 B9 BA BB BC BD BE BF   ................
```

```
01C0  C0 C1 C2 C3 C4 C5 C6 C7  C8 C9 CA CB CC CD CE CF  ................
01D0  D0 D1 D2 D3 D4 D5 D6 D7  D8 D9 DA DB DC DD DE DF  ................
01E0  E0 E1 E2 E3 E4 E5 E6 E7  E8 E9 EA EB EC ED EE EF  ................
01F0  F0 F1 F2 F3 F4 F5 F6 F7  F8 F9 FA FB FC FD FE FF  ................
0200  02 03 04 05 06 07 08 09  0A 0B 0C 0D 0E 0F 10 11  ................
0210  12 13 14 15 16 17 18 19  1A 1B 1C 1D 1E 1F 20 21  ................
0220  22 23 24 25 26 27 28 29  2A 2B 2C 2D 2E 2F 30 31  "#$%&'()*+,-./01
0230  32 33 34 35 36 37 38 39  3A 3B 3C 3D 3E 3F 40 41  23456789:;<=>?@A
0240  42 43 44 45 46 47 48 49  4A 4B 4C 4D 4E 4F 50 51  BCDEFGHIJKLMNOPQ
0250  52 53 54 55 56 57 58 59  5A 5B 5C 5D 5E 5F 60 61  RSTUVWXYZ[\]^_'a
0260  62 63 64 65 66 67 68 69  6A 6B 6C 6D 6E 6F 70 71  bcdefghijklmnopq
0270  72 73 74 75 76 77 78 79  7A 7B 7C 7D 7E 7F 80 81  rstuvwxyz{|}~...
0280  00 01 02 03 04 05 06 07  08 09 0A 0B 0C 0D 0E 0F  ................
0290  10 11 12 13 14 15 16 17  18 19 1A 1B 1C 1D 1E 1F  ................
02A0  20 21 22 23 24 25 26 27  28 29 2A 2B 2C 2D 2E 2F   !"#$%&'()*+,-./
02B0  30 31 32 33 34 35 36 37  38 39 3A 3B 3C 3D 3E 3F  0123456789:;<=>?
02C0  40 41 42 43 44 45 46 47  48 49 4A 4B 4C 4D 4E 4F  @ABCDEFGHIJKLMNO
02D0  50 51 52 53 54 55 56 57  58 59 5A 5B 5C 5D 5E 5F  PQRSTUVWXYZ[\]^_
02E0  60 61 62 63 64 65 66 67  68 69 6A 6B 6C 6D 6E 6F  'abcdefghijklmno
02F0  70 71 72 73 74 75 76 77  78 79 7A 7B 7C 7D 7E 7F  pqrstuvwxyz{|}~.
0300  0E 0E 0E 0E 0E 0E 0E 0E  0E 0E 0E 0E 0E 0E 0E 0E  ................
0310  0E 0E 0E 0E 0E 0E 0E 0E  0E 0E 0E 0E 0E 0E 0E 0E  ................
0320  0E 0E 0E 0E 0E 0E 0E 0E  0E 0E 0E 0E 0E 0E 0E 0E  ................
0330  0E 0E 0E 0E 0E 0E 0E 0E  0E 0E 0E 0E 0E 0E 0E 0E  ................
0340  0E 0E 0E 0E 0E 0E 0E 0E  0E 0E 0E 0E 0E 0E 0E 0E  ................
0350  0E 0E 0E 0E 0E 0E 0E 0E  0E 0E 0E 0E 0E 0E 0E 0E  ................
0360  0E 0E 0E 0E 0E 0E 0E 0E  0E 0E 0E 0E 0E 0E 0E 0E  ................
0370  0E 0E 0E 0E 0E 0E 0E 0E  0E 0E 0E 0E 0E 0E 0E 0E  ................
0380  0E 0E 0E 0E 0E 0E 0E 0E  0E 0E 0E 0E 0E 0E 0E 0E  ................
0390  0E 0E 0E 0E 0E 0E 0E 0E  0E 0E 0E 0E 0E 0E 0E 0E  ................
03A0  0E 0E 0E 0E 0E 0E 0E 0E  0E 0E 0E 0E 0E 0E 0E 0E  ................
03B0  0E 0E 0E 0E 0E 0E 0E 0E  0E 0E 0E 0E 0E 0E 0E 0E  ................
03C0  0E 0E 0E 0E 0E 0E 0E 0E  0E 0E 0E 0E 0E 0E 0E 0E  ................
03D0  0E 0E 0E 0E 0E 0E 0E 0E  0E 0E 0E 0E 0E 0E 0E 0E  ................
03E0  0E 0E 0E 0E 0E 0E 0E 0E  0E 0E 0E 0E 0E 0E 0E 0E  ................
03F0  0E 0E 0E 0E 0E 0E 0E 0E  0E 0E 0E 0E 0E 0E 0E 0E  ................
0400  1A 1A 1A 1A 1A 1A 1A 1A  1A 1A 1A 1A 1A 1A 1A 1A  ................
0410  1A 1A 1A 1A 1A 1A 1A 1A  1A 1A 1A 1A 1A 1A 1A 1A  ................
0420  1A 1A 1A 1A 1A 1A 1A 1A  1A 1A 1A 1A 1A 1A 1A 1A  ................
0430  1A 1A 1A 1A 1A 1A 1A 1A  1A 1A 1A 1A 1A 1A 1A 1A  ................
0440  1A 6B 6B 6B 6B 6B 6B 6B  6B 6B 6B 6B 6B 6B 6B 6B  .kkkkkkkkkkkkkkk
0450  6B 6B 6B 6B 6B 6B 6B 6B  6B 6B 6B 6B 6B 6B 6B 6B  kkkkkkkkkkkkkkkk
0460  6B 6B 6B 6B 6B 6B 6B 6B  6B 6B 6B 6B 6B 6B 6B 6B  kkkkkkkkkkkkkkkk
0470  6B 6B 6B 6B 6B 6B 6B 6B  6B 6B 6B 6B 6B 6B 6B 6B  kkkkkkkkkkkkkkkk
0480  00 01 02 03 04 05 06 07  08 09 0A 0B 0C 0D 0E 0F  ................
0490  10 11 12 13 14 15 16 17  18 19 1A 1B 1C 1D 1E 1F  ................
04A0  20 21 22 23 24 25 26 27  28 29 2A 2B 2C 2D 2E 2F   !"#$%&'()*+,-./
04B0  30 31 32 33 34 35 36 37  38 39 3A 3B 3C 3D 3E 3F  0123456789:;<=>?
04C0  40 41 42 43 44 45 46 47  48 49 4A 4B 4C 4D 4E 4F  @ABCDEFGHIJKLMNO
04D0  50 51 52 53 54 55 56 57  58 59 5A 5B 5C 5D 5E 5F  PQRSTUVWXYZ[\]^_
04E0  60 61 62 63 64 65 66 67  68 69 6A 6B 6C 6D 6E 6F  'abcdefghijklmno
04F0  70 71 72 73 74 75 76 77  78 79 7A 7B 7C 7D 7E 7F  pqrstuvwxyz{|}~.
```

```
0500  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0510  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0520  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0530  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0540  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0550  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0560  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0570  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0580  00 01 02 03 04 05 06 07  08 09 0A 0B 0C 0D 0E 0F  ................
0590  10 11 12 13 14 15 16 17  18 19 1A 1B 1C 1D 1E 1F  ................
05A0  20 21 22 23 24 25 26 27  28 29 2A 2B 2C 2D 2E 2F   !"#$%&'()*+,-./
05B0  30 31 32 33 34 35 36 37  38 39 3A 3B 3C 3D 3E 3F  0123456789:;<=>?
05C0  40 41 42 43 44 45 46 47  48 49 4A 4B 4C 4D 4E 4F  @ABCDEFGHIJKLMNO
05D0  50 51 52 53 54 55 56 57  58 59 5A 5B 5C 5D 5E 5F  PQRSTUVWXYZ[\]^_
05E0  60 61 62 63 64 65 66 67  68 69 6A 6B 6C 6D 6E 6F  `abcdefghijklmno
05F0  70 71 72 73 74 75 76 77  78 79 7A 7B 7C 7D 7E 7F  pqrstuvwxyz{|}~.
0600  00 01 02 03 04 05 06 07  08 09 0A 0B 0C 0D 0E 0F  ................
0610  10 11 12 13 14 15 16 17  18 19 1A 1B 1C 1D 1E 1F  ................
0620  20 21 22 23 24 25 26 27  28 29 2A 2B 2C 2D 2E 2F   !"#$%&'()*+,-./
0630  30 31 32 33 34 35 36 37  38 39 3A 3B 3C 3D 3E 3F  0123456789:;<=>?
0640  40 41 42 43 44 45 46 47  48 49 4A 4B 4C 4D 4E 4F  @ABCDEFGHIJKLMNO
0650  50 51 52 53 54 55 56 57  58 59 5A 5B 5C 5D 5E 5F  PQRSTUVWXYZ[\]^_
0660  60 61 62 63 64 65 66 67  68 69 6A 6B 6C 6D 6E 6F  `abcdefghijklmno
0670  70 71 72 73 74 75 76 77  78 79 7A 7B 7C 7D 7E 7F  pqrstuvwxyz{|}~.
0680  80 81 82 83 84 85 86 87  88 89 8A 8B 8C 8D 8E 8F  ................
0690  90 91 92 93 94 95 96 97  98 99 9A 9B 9C 9D 9E 9F  ................
06A0  A0 A1 A2 A3 A4 A5 A6 A7  A8 A9 AA AB AC AD AE AF  ................
06B0  B0 B1 B2 B3 B4 B5 B6 B7  B8 B9 BA BB BC BD BE BF  ................
06C0  C0 C1 C2 C3 C4 C5 C6 C7  C8 C9 CA CB CC CD CE CF  ................
06D0  D0 D1 D2 D3 D4 D5 D6 D7  D8 D9 DA DB DC DD DE DF  ................
06E0  E0 E1 E2 E3 E4 E5 E6 E7  E8 E9 EA EB EC ED EE EF  ................
06F0  F0 F1 F2 F3 F4 F5 F6 F7  F8 F9 FA FB FC FD FE FF  ................
0700  19 19 19 19 19 19 19 19  19 19 19 19 19 19 19 19  ................
0710  19 19 19 19 19 19 19 19  19 19 19 19 19 19 19 19  ................
0720  19 19 19 19 19 19 19 19  19 19 19 19 19 19 19 19  ................
0730  19 19 19 19 19 19 19 19  19 19 19 19 19 19 19 19  ................
0740  19 64 64 64 64 64 64 64  64 64 64 64 64 64 64 64  .ddddddddddddddd
0750  64 64 64 64 64 64 64 64  64 64 64 64 64 64 64 64  dddddddddddddddd
0760  64 64 64 64 64 64 64 64  64 64 64 64 64 64 64 64  dddddddddddddddd
0770  64 64 64 64 64 64 64 64  64 64 64 64 64 64 64 64  dddddddddddddddd
0780  00 01 02 03 04 05 06 07  08 09 0A 0B 0C 0D 0E 0F  ................
0790  10 11 12 13 14 15 16 17  18 19 1A 1B 1C 1D 1E 1F  ................
07A0  20 21 22 23 24 25 26 27  28 29 2A 2B 2C 2D 2E 2F   !"#$%&'()*+,-./
07B0  30 31 32 33 34 35 36 37  38 39 3A 3B 3C 3D 3E 3F  0123456789:;<=>?
07C0  40 41 42 43 44 45 46 47  48 49 4A 4B 4C 4D 4E 4F  @ABCDEFGHIJKLMNO
07D0  50 51 52 53 54 55 56 57  58 59 5A 5B 5C 5D 5E 5F  PQRSTUVWXYZ[\]^_
07E0  60 61 62 63 64 65 66 67  68 69 6A 6B 6C 6D 6E 6F  `abcdefghijklmno
07F0  70 71 72 73 74 75 76 77  78 79 7A 7B 7C 7D 7E 7F  pqrstuvwxyz{|}~.
0800  00 00 00 03 00 00 00 00  00 00 00 00 00 00 00 00  ................
0810  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0820  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0830  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
```

```
0840  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0850  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0860  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0870  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0880  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0890  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
08A0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
08B0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
08C0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
08D0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
08E0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
08F0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0900  01 03 05 03 09 0B 0D 0F  11 13 15 17 19 1B 1D 0F  ................
0910  21 23 25 27 29 2B 2D 2F  31 33 35 37 39 3B 3D 3F  !#%')+-/13579;=?
0920  41 43 45 47 49 4B 4D 4F  51 53 55 57 59 5B 5D 5F  ACEGIKMOQSUWY[]_
0930  61 63 65 67 69 6B 6D 6F  71 73 75 77 79 7B 7D 7F  acegikmoqsuwy{}.
0940  81 83 85 87 89 8B 8D 8F  91 93 95 97 99 9B 9D 9F  ................
0950  A1 A3 A5 A7 A9 AB AD AF  B1 B3 B5 B7 B9 BB BD BF  ................
0960  C1 C3 C5 C7 C9 CB CD CF  D1 D3 D5 D7 D9 DB DD DF  ................
0970  E1 E3 E5 E7 E9 EB ED EF  F1 F3 F5 F7 F9 FB FD FF  ................
0980  00 01 02 03 04 05 06 07  08 09 0A 0B 0C 0D 0E 0F  ................
0990  10 11 12 13 14 15 16 17  18 19 1A 1B 1C 1D 1E 1F  ................
09A0  20 21 22 23 24 25 26 27  28 29 2A 2B 2C 2D 2E 2F   !"#$%&'()*+,-./
09B0  30 31 32 33 34 35 36 37  38 39 3A 3B 3C 3D 3E 3F  0123456789:;<=>?
09C0  40 41 42 43 44 45 46 47  48 49 4A 4B 4C 4D 4E 4F  @ABCDEFGHIJKLMNO
09D0  50 51 52 53 54 55 56 57  58 59 5A 5B 5C 5D 5E 5F  PQRSTUVWXYZ[\]^_
09E0  60 61 62 63 64 65 66 67  68 69 6A 6B 6C 6D 6E 6F  'abcdefghijklmno
09F0  70 71 72 73 74 75 76 77  78 79 7A 7B 7C 7D 7E 7F  pqrstuvwxyz{|}~.
0A00  00 02 04 06 08 0A 0C 0E  10 10 10 10 10 10 10 10  ................
0A10  10 10 10 10 10 10 10 10  10 10 10 10 10 10 10 10  ................
0A20  20 22 24 26 28 2A 2C 2E  30 30 30 30 30 30 30 30   "$&(*,.00000000
0A30  30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  0000000000000000
0A40  40 42 44 46 48 4A 4C 4E  50 50 50 50 50 50 50 50  @BDFHJLNPPPPPPPP
0A50  50 50 50 50 50 50 50 50  50 50 50 50 50 50 50 50  PPPPPPPPPPPPPPPP
0A60  60 62 64 66 68 6A 6C 6E  70 70 70 70 70 70 70 70  'bdfhjlnpppppppp
0A70  70 70 70 70 70 70 70 70  70 70 70 70 70 70 70 70  pppppppppppppppp
0A80  00 01 02 03 04 05 06 07  08 09 0A 0B 0C 0D 0E 0F  ................
0A90  10 11 12 13 14 15 16 17  18 19 1A 1B 1C 1D 1E 1F  ................
0AA0  20 21 22 23 24 25 26 27  28 29 2A 2B 2C 2D 2E 2F   !"#$%&'()*+,-./
0AB0  30 31 32 33 34 35 36 37  38 39 3A 3B 3C 3D 3E 3F  0123456789:;<=>?
0AC0  40 41 42 43 44 45 46 47  48 49 4A 4B 4C 4D 4E 4F  @ABCDEFGHIJKLMNO
0AD0  50 51 52 53 54 55 56 57  58 59 5A 5B 5C 5D 5E 5F  PQRSTUVWXYZ[\]^_
0AE0  60 61 62 63 64 65 66 67  68 69 6A 6B 6C 6D 6E 6F  'abcdefghijklmno
0AF0  70 71 72 73 74 75 76 77  78 79 7A 7B 7C 7D 7E 7F  pqrstuvwxyz{|}~.
0B00  00 02 04 06 08 0A 0C 0E  10 12 14 16 18 1A 1C 1E  ................
0B10  20 22 24 26 28 2A 2C 2E  30 32 34 36 38 3A 3C 3E   "$&(*,.02468:<>
0B20  40 42 44 46 48 4A 4C 4E  50 52 54 56 58 5A 5C 5E  @BDFHJLNPRTVXZ\
0B30  60 62 64 66 68 6A 6C 6E  70 72 74 76 78 7A 7B 7E  'bdfhjlnprtvxz ~
0B40  80 82 84 86 88 8A 8C 8E  90 92 94 96 98 9A 9C 9E  ................
0B50  A0 A2 A4 A6 A8 AA AC AE  B0 B2 B4 B6 B8 BA BC BE  ................
0B60  C0 C2 C4 C6 C8 CA CC CE  D0 D2 D4 D6 D8 DA DC DE  ................
0B70  E0 E2 E4 E6 E8 EA EC EE  F0 F2 F4 F6 F8 FA FC FE  ................
```

```
0B80  00 01 02 03 04 05 06 07  08 09 0A 0B 0C 0D 0E 0F  ................
0B90  10 11 12 13 14 15 16 17  18 19 1A 1B 1C 1D 1E 1F  ................
0BA0  20 21 22 23 24 25 26 27  28 29 2A 2B 2C 2D 2E 2F   !"#$%&'()*+,-./
0BB0  30 31 32 33 34 35 36 37  38 39 3A 3B 3C 3D 3E 3F  0123456789:;<=>?
0BC0  40 41 42 43 44 45 46 47  48 49 4A 4B 4C 4D 4E 4F  @ABCDEFGHIJKLMNO
0BD0  50 51 52 53 54 55 56 57  58 59 5A 5B 5C 5D 5E 5F  PQRSTUVWXYZ[\]^_
0BE0  60 61 62 63 64 65 66 67  68 69 6A 6B 6C 6D 6E 6F  `abcdefghijklmno
0BF0  70 71 72 73 74 75 76 77  78 79 7A 7B 7C 7D 7E 7F  pqrstuvwxyz{|}~.
0C00  00 01 02 03 04 05 06 07  08 09 0A 0B 0C 0D 0E 0F  ................
0C10  10 11 12 13 14 15 16 17  18 19 1A 1B 1C 1D 1E 1F  ................
0C20  20 21 22 23 24 25 26 27  28 29 2A 2B 2C 2D 2E 2F   !"#$%&'()*+,-./
0C30  30 31 32 33 34 35 36 37  38 39 3A 3B 3C 3D 3E 3F  0123456789:;<=>?
0C40  40 41 42 43 44 45 46 47  48 49 4A 4B 4C 4D 4E 4F  @ABCDEFGHIJKLMNO
0C50  50 51 52 53 54 55 56 57  58 59 5A 5B 5C 5D 5E 5F  PQRSTUVWXYZ[\]^_
0C60  60 61 62 63 64 65 66 67  68 69 6A 6B 6C 6D 6E 6F  `abcdefghijklmno
0C70  70 71 72 73 74 75 76 77  78 79 7A 7B 7C 7D 7E 7F  pqrstuvwxyz{|}~.
0C80  80 81 82 83 84 85 86 87  88 89 8A 8B 8C 8D 8E 8F  ................
0C90  90 91 92 93 94 95 96 97  98 99 9A 9B 9C 9D 9E 9F  ................
0CA0  A0 A1 A2 A3 A4 A5 A6 A7  A8 A9 AA AB AC AD AE AF  ................
0CB0  B0 B1 B2 B3 B4 B5 B6 B7  B8 B9 BA BB BC BD BE BF  ................
0CC0  C0 C1 C2 C3 C4 C5 C6 C7  C8 C9 CA CB CC CD CE CF  ................
0CD0  D0 D1 D2 D3 D4 D5 D6 D7  D8 D9 DA DB DC DD DE DF  ................
0CE0  E0 E1 E2 E3 E4 E5 E6 E7  E8 E9 EA EB EC ED EE EF  ................
0CF0  F0 F1 F2 F3 F4 F5 F6 F7  F8 F9 FA FB FC FD FE FF  ................
0D00  00 01 02 03 04 05 06 07  08 09 0A 0B 0C 0D 0E 0F  ................
0D10  10 11 12 13 14 15 16 17  18 19 1A 1B 1C 1D 1E 1F  ................
0D20  20 21 22 23 24 25 26 27  28 29 2A 2B 2C 2D 2E 2F   !"#$%&'()*+,-./
0D30  30 31 32 33 34 35 36 37  38 39 3A 3B 3C 3D 3E 3F  0123456789:;<=>?
0D40  40 41 42 43 44 45 46 47  48 49 4A 4B 4C 4D 4E 4F  @ABCDEFGHIJKLMNO
0D50  50 51 52 53 54 55 56 57  58 59 5A 5B 5C 5D 5E 5F  PQRSTUVWXYZ[\]^_
0D60  60 61 62 63 64 65 66 67  68 69 6A 6B 6C 6D 6E 6F  `abcdefghijklmno
0D70  70 71 72 73 74 75 76 77  78 79 7A 7B 7C 7D 7E 7F  pqrstuvwxyz{|}~.
0D80  80 81 82 83 84 85 86 87  88 89 8A 8B 8C 8D 8E 8F  ................
0D90  90 91 92 93 94 95 96 97  98 99 9A 9B 9C 9D 9E 9F  ................
0DA0  A0 A1 A2 A3 A4 A5 A6 A7  A8 A9 AA AB AC AD AE AF  ................
0DB0  B0 B1 B2 B3 B4 B5 B6 B7  B8 B9 BA BB BC BD BE BF  ................
0DC0  C0 C1 C2 C3 C4 C5 C6 C7  C8 C9 CA CB CC CD CE CF  ................
0DD0  D0 D1 D2 D3 D4 D5 D6 D7  D8 D9 DA DB DC DD DE DF  ................
0DE0  E0 E1 E2 E3 E4 E5 E6 E7  E8 E9 EA EB EC ED EE EF  ................
0DF0  F0 F1 F2 F3 F4 F5 F6 F7  F8 F9 FA FB FC FD FE FF  ................
0E00  00 01 02 03 04 05 06 07  08 09 0A 0B 0C 0D 0E 0F  ................
0E10  10 11 12 13 14 15 16 17  18 19 1A 1B 1C 1D 1E 1F  ................
0E20  20 21 22 23 24 25 26 27  28 29 2A 2B 2C 2D 2E 2F   !"#$%&'()*+,-./
0E30  30 31 32 33 34 35 36 37  38 39 3A 3B 3C 3D 3E 3F  0123456789:;<=>?
0E40  40 41 42 43 44 45 46 47  48 49 4A 4B 4C 4D 4E 4F  @ABCDEFGHIJKLMNO
0E50  50 51 52 53 54 55 56 57  58 59 5A 5B 5C 5D 5E 5F  PQRSTUVWXYZ[\]^_
0E60  60 61 62 63 64 65 66 67  68 69 6A 6B 6C 6D 6E 6F  `abcdefghijklmno
0E70  70 71 72 73 74 75 76 77  78 79 7A 7B 7C 7D 7E 7F  pqrstuvwxyz{|}~.
0E80  80 81 82 83 84 85 86 87  88 89 8A 8B 8C 8D 8E 8F  ................
0E90  90 91 92 93 94 95 96 97  98 99 9A 9B 9C 9D 9E 9F  ................
0EA0  A0 A1 A2 A3 A4 A5 A6 A7  A8 A9 AA AB AC AD AE AF  ................
0EB0  B0 B1 B2 B3 B4 B5 B6 B7  B8 B9 BA BB BC BD BE BF  ................
```

```
0EC0  C0 C1 C2 C3 C4 C5 C6 C7  C8 C9 CA CB CC CD CE CF  ................
0ED0  D0 D1 D2 D3 D4 D5 D6 D7  D8 D9 DA DB DC DD DE DF  ................
0EE0  E0 E1 E2 E3 E4 E5 E6 E7  E8 E9 EA EB EC ED EE EF  ................
0EF0  F0 F1 F2 F3 F4 F5 F6 F7  F8 F9 FA FB FC FD FE FF  ................
0F00  00 01 02 03 04 05 06 07  08 09 0A 0B 0C 0D 0E 0F  ................
0F10  10 11 12 13 14 15 16 17  18 19 1A 1B 1C 1D 1E 1F  ................
0F20  20 21 22 23 24 25 26 27  28 29 2A 2B 2C 2D 2E 2F   !"#$%&'()*+,-./
0F30  30 31 32 33 34 35 36 37  38 39 3A 3B 3C 3D 3E 3F  0123456789:;<=>?
0F40  40 41 42 43 44 45 46 47  48 49 4A 4B 4C 4D 4E 4F  @ABCDEFGHIJKLMNO
0F50  50 51 52 53 54 55 56 57  58 59 5A 5B 5C 5D 5E 5F  PQRSTUVWXYZ[\]^_
0F60  60 61 62 63 64 65 66 67  68 69 6A 6B 6C 6D 6E 6F  'abcdefghijklmno
0F70  70 71 72 73 74 75 76 77  78 79 7A 7B 7C 7D 7E 7F  pqrstuvwxyz{¦}~.
0F80  80 81 82 83 84 85 86 87  88 89 8A 8B 8C 8D 8E 8F  ................
0F90  90 91 92 93 94 95 96 97  98 99 9A 9B 9C 9D 9E 9F  ................
0FA0  A0 A1 A2 A3 A4 A5 A6 A7  A8 A9 AA AB AC AD AE AF  ................
0FB0  B0 B1 B2 B3 B4 B5 B6 B7  B8 B9 BA BB BC BD BE BF  ................
0FC0  C0 C1 C2 C3 C4 C5 C6 C7  C8 C9 CA CB CC CD CE CF  ................
0FD0  D0 D1 D2 D3 D4 D5 D6 D7  D8 D9 DA DB DC DD DE DF  ................
0FE0  E0 E1 E2 E3 E4 E5 E6 E7  E8 E9 EA EB EC ED EE EF  ................
0FF0  F0 F1 F2 F3 F4 F5 F6 F7  F8 F9 FA FB FC FD FE FF  ................
At end of current range
*
```

Index

A

A (assemble) DEBUG command 3-9 to 3-12
Absolute disk read (INT 25H) 7-19 to 7-21
Absolute disk write (INT 26H) 7-22 to 7-24
Addressing 4-1 to 4-14
Aligned words 4-5
Allocate memory (Function 48H) 7-140 to 7-141
ASCII 3-15, 8-22 to 8-27
ASCIIZ strings 5-11 to 5-12
Assembler 1-4
  modules 2-7 to 2-8, 2-12
  in DEBUG 3-9 to 3-12
  DEBUG list 3-39 to 3-40
  using system calls 7-6
Attribute (see File attribute and Video control)
ARF (see Automatic response file)
Asynchronous communications (see communications)
Automatic response file 2-9, 2-19 to 2-20
Auxiliary input (Function 03H) 7-31
Auxiliary output (Function 04H) 7-32

B

BASIC 1-4, 7-6
BIOS
  Memory usage 5-4, 5-5
  Service routines 8-1 to 8-36
Block devices 9-8
Bootstrap loader (INT 19H) 8-4, 8-30
Break ON/OFF Command 8-33
Breakpoints 3-21 to 3-23
Buffered keyboard input (Function 0AH) 7-40 to 7-41
Buffers Command 8-34
Bus lines 9-6 to 9-7

C

C (compare) DEBUG command 3-13 to 3-14
Calendar (see Clock)
Calculated addressing 4-14
Change file attributes (Function 43H) 7-130 to 7-131
Change current directory (Function 3BH) 7-117
Character devices 9-8
Control-C check (Function 33H) 7-106 to 7-107
Check keyboard status (Function 0BH) 7-42
Class 2-7
CLOCK
  device 9-26, 9-123 to 9-127
  (also see Time and Date)
Close file
  (Function 10H) 7-50 to 7-51
  (Function 3EH) 7-122
Clusters 5-13 to 5-23, 7-109
COM files 3-5, 3-26, 3-30, 6-1 to 6-7, 6-10, 6-21
Command line
  MS-LINK 2-9, 2-18 to 2-19
  DEBUG 3-3, 3-7 to 3-8
  direct commands 6-18

COMMUNICATIONS
BIOS routines (INT 14H) 8-4, 8-19 to 8-21
device drivers 9-27 to 9-35, 9-128 to 9-146
Compilers 1-4
COMSPEC 6-18
CON device (see Console)
CONFIG.SYS
Break ON/OFF 8-33
Buffer 8-34
Device 8-34
Files 8-35
Shell 8-35
Console 2-12
Direct I/O 7-35 to 7-36
Direct input 7-37
(also see Keyboard and Video)
Control blocks (see File control block)
Control-C check (Function 33H) 7-106 to 7-107
Country-dependent information 7-110 to 7-114
Create file
(Function 16H) 7-64 to 7-65
(Function 3CH) 7-118 to 7-119
Create sub-directory (Function 39H) 7-115
Current disk (Function 19H) 7-68

D

D (display) DEBUG command 3-15 to 3-16
Date
read 7-93 to 7-94, 7-157 to 7-158
set 7-95 to 7-96, 7-157 to 7-158
DEBUG 3-1 to 3-44
Commands:
A (assemble) 3-9 to 3-12
C (compare) 3-13 to 3-14
D (display) 3-15 to 3-16
E (enter) 3-17 to 3-19
F (fill) 3-20
G (go) 3-21 to 3-23
H (hexarithmetic) 3-24
I (input) 3-25
L (load) 3-26
M (move) 3-28
N (name) 3-5, 3-29 to 3-31
O (output) 3-32
Q (quit) 3-33
R (register) 3-34 to 3-36
S (search) 3-37
T (trace) 3-38
U (unassemble) 3-39 to 3-40
W (write) 3-41 to 3-43
Delete directory entry (Function 41H) 7-127
Delete file (Function 13H) 7-56 to 7-58
Determine memory size (INT 12H) 8-4, 8-31
DEVICES
character 9-8
drivers 1-4, 9-4 to 9-161
installation 9-12
read 7-123 to 7-124
Device header 9-9 to 9-12
Direct addressing 4-13

Direct console I/O (Function 06H) 7-35 to 7-36
Direct console input (Function 07H) 7-37
Directory 5-14 to 5-17, 5-23
  Change 7-117
  Create sub-directory 7-115
  Delete entry 7-127
  Delete file 7-56 to 7-58
  Move entry 7-115 to 7-156
  Remove directory 7-116
  Return current 7-139
  Step through 7-153
Diskette
  allocation 5-13 to 5-23
  BIOS routines (INT 13H) 8-4, 8-17 to 8-18
Disk operations:
  (also see Files)
  Create sub-directory 7-115
  Current disk 7-68
  Select 7-46
  Current directory 7-139
  Delete directory entry 7-127
  Duplicate file handle 7-137
  Move directory entry 7-155 to 7-156
  Read absolute 7-19 to 7-21
  Remove directory 7-116
  Reset 7-45
  Space 7-109
  Step through directory 7-153
  Transfer address 7-69 to 7-70, 7-103
  Verify flag 7-101 to 7-102, 7-154
  Write absolute 7-22 to 7-24
Display character (Function 02H) 7-30
Display string (Function 09H) 7-39
DMA controller 9-36 to 9-44
DOS version number 7-104
DSALLOCATE link switch 2-14
Duplicate file handle (Function 45H) 7-137

E

E (enter) DEBUG command 3-17 to 3-19
Equipment check (INT 11H) 8-4, 8-31
Error messages
  MS-LINK 2-24 to 2-27
  DEBUG 3-22, 3-44
  EXE2BIN 6-8 to 6-9
EXE files (see Run files)
EXE2BIN 6-2, 6-5 to 6-9
EXEC (see Load and execute program)
Exit address (INT 23H) 7-15

F

F (fill) DEBUG command 3-20
FAT (see File Allocation Table)
Fatal error abort address (INT 24H) 7-16 to 7-18
FCB (see File control block)
FILES
  (also see Disk operations)
  allocation table 5-13, 5-18 to 5-21, 5-22, 5-23
  attribute 5-15, 7-53

change 7-130 to 7-131
close file 7-50 to 7-51, 7-122
control block 3-30, 5-6 to 5-12, 6-16 to 6-19
create file 7-64 to 7-65, 7-118 to 7-119
delete file 7-56 to 7-58
disk transfer address 7-69 to 7-70, 7-103
duplicate file handle 7-137
forced 7-138
find 7-151 to 7-152, 7-52 to 7-53, 7-54 to 7-55
header (Run files) 6-10 to 6-14
load and execute program file 7-145 to 7-148
names 7-89 to 7-92, 7-66 to 7-67
open file 7-47 to 7-49, 7-120 to 7-121
pointer 7-128 to 7-129
random operations
block read 7-83 to 7-85
block write 7-86 to 7-88
record read 7-71 to 7-72
record write 7-73 to 7-75
set relative record 7-79 to 7-81
read 7-123 to 7-124
sequential operations
read 7-59 to 7-61
write 7-62 to 7-63
size 7-76 to 7-78
usage 2-3 to 2-6
write 7-125 to 7-126
Files Command 8-35
Find file (Function 4EH) 7-151 to 7-152
Flags 4-8 to 4-9
Floppy diskette interface controller 9-45 to 9-65
Flush buffer, read keyboard (Function 0CH) 7-43 to 7-44
Force duplicate of handle (Function 46H) 7-138
Free allocated memory (Function 49H) 7-142
Function request (INT 21H) 7-13, 7-29 to 7-173
Functions (DOS) 7-2 to 7-3, 7-8 to 7-9, 7-13, 7-29 to 7-173, 7-160 to 7-173
FORTRAN 2-14, 7-6

G

G (go) DEBUG command 3-21 to 3-23
Get date
(Function 2AH) 7-93 to 7-94
(Function 57H) 7-157 to 7-158
Get disk free space
(Function 36H) 7-109
Get disk transfer address
(Function 2FH) 7-103
Get DOS version number
(Function 30H) 7-104
Get interrupt vector
(Function 35H) 7-108
Get time
(Function 2CH) 7-97 to 7-98
(Function 57H) 7-157 to 7-158

Global symbols (see Public symbols)
Group 2-8, 2-14

H

H (hexarithmetic) DEBUG command 3-24
Handles 5-12
Hard disk controller 9-66 to 9-93
HEX files 3-4, 3-27
HIGH Link switch 2-14

I

I (input) DEBUG command 3-25
I/O control for devices (Function 44H) 7-132 to 7-136
I/O ports 9-3
Immediate addressing 4-12
Indirect addressing 4-13
INPUT
  (also see files)
  auxiliary device 7-31
  control devices 7-132 to 7-136
  file or device 7-123 to 7-124
  keyboard
    buffered 7-40 to 7-41
    direct 7-38
    echo 7-29
    flush buffer 7-43 to 7-44
Installing a device driver 9-12
Interrupts
  In general 5-3, 5-4, 7-4, 7-7, 7-10 to 7-28, 7-159, 8-1 to 8-32, 9-4 to 9-5
  Get vector 7-108
  Set vector 7-82
Interrupt controller 9-105 to 9-115
Interval timer 9-116 to 9-122

K

Keep process (Function 31H) 7-105
KEYBOARD
  (also see Console)
  BIOS routines 8-4, 8-22 to 8-28
  Buffered input 7-40 to 7-41
  Control-C check 7-106 to 7-107
  Device driver 9-94 to 9-99
  Flush buffer read 7-43 to 7-44
  Status check 7-42

L

L (load) DEBUG command 3-5
LIB files (see Libraries)
Libraries 2-4, 2-10, 2-12, 2-17, 2-18, 2-19, 2-22
LINENUMBERS link switch 2-15, 2-21
Linking object modules (see MS-LINK)
List files (see MAP files)
Load module (see Run files)
Loading programs 6-18 to 6-22, 7-145 to 7-148
Load and execute program (Function 4BH) 7-145 to 7-148
Logic lines (see Bus lines)

M

M (move) DEBUG command 3-28
Macro assembler (see Assembler)
MAP files 2-4, 2-10, 2-11, 2-15, 2-16, 2-18, 2-19, 2-23
MAP link switch 2-15, 2-21, 2-23
Media check 2-19, 9-19
Memory
  addressing 4-1 to 4-14
  allocation 7-142 to 7-144
  maps 5-1 to 5-5
  size 8-4, 8-31
Messages (also see Error messages) 1-3, 3-5, 3-6, 3-35
Modifications to BIOS 8-32 to 8-36
Modify allocated memory blocks (Function 4AH) 7-143 to 7-144
Move directory entry (Function 56H) 7-155 to 7-156
Move file pointer (Function 42H) 7-128 to 7-129
MS-LINK 1-4, 2-1 to 2-27

N

N (name) DEBUG command 3-5, 3-29 to 3-31
NO link switch 2-17
Non-aligned words 4-5
Notation 1-3

O

O (output) DEBUG command 3-32
OBJ files (see Object modules)
Object modules 2-2, 2-4, 2-10, 2-11, 2-18, 2-19, 2-20
Open file
  (Function 0FH) 7-47 to 7-49
  (Function 3DH) 7-120 to 7-121
OUTPUT
  Auxiliary device 7-32
  Character 7-30, 7-33 to 7-34
  Control devices 7-132 to 7-136
  File or device 7-125 to 7-126
  Printer BIOS routines 8-4, 8-29
  Screen dump 8-4, 8-30
  String 7-39

P

Parallel printer interface 9-100 to 9-104
Parse file name (Function 29H) 7-89 to 7-92
Pascal 2-14, 2-17, 7-6
PATH command 6-18
Pathname string (see ASCIIZ strings)
PAUSE link switch 2-15 to 2-16, 2-20
Print character (Function 05H) 7-33 to 7-34
Print screen (INT 05H) 8-4, 8-30
Printer routines (INT 17H) 8-4, 8-29

Printer 2-12, 2-21, 8-4
(also see Output)
PRN device (see Printer)
Process control (also see
Memory allocation)
  Keep process 7-105
  Load and execute program
  7-145 to 7-148
  Return code 7-150
  Terminate 7-11 to 7-12,
  7-25 to 7-28, 7-149
Program files (also see Run
files and COM files)
  Load and execute 7-145 to
  7-148
  Structure 6-1 to 6-22
Program segment prefix 3-5,
6-15 to 6-17, 6-19, 6-21, 7-11
Program terminate (INT 20H)
7-11 to 7-12
PROMPT command 6-18
Prompts 1-3, 2-10, 2-12, 3-4, 3-12
Public symbols 2-15, 2-20, 2-23

Q

Q (quit) DEBUG command
3-33

R

R (register) DEBUG command
3-34 to 3-36
Random block read (Function
27H) 7-83 to 7-85
Random block write (Function
28H) 7-86 to 7-88
Random Read (Function 21H)
7-71 to 7-72
Random Write (Function 22H)
7-73 to 7-75
Read from file or device
(Function 3FH) 7-123 to 7-124
Read keyboard (Function 08H)
7-38
Read keyboard and echo
(Function 01H) 7-29
Registers 4-6 to 4-10, 7-9
Register addressing 4-12
Remove directory (Function
3AH) 7-116
Rename file (Function 17H)
7-66 to 7-67
Request header 9-13 to 9-15,
9-18 to 9-28
Reset disks (Function 0DH)
7-45
Reset verify flag (Function
2EH) 7-101 to 7-102
Retrieve return code of child
(Function 4DH) 7-150
Return country-dependent info.
(Function 38H) 7-110 to 7-114
Return current verify flag
(Function 54H) 7-154
Return current directory
(Function 47H) 7-139
Run files 2-4, 2-11, 2-14, 2-15,
2-18, 2-19, 2-21, 3-4, 3-5, 3-27, 6-1
to 6-3, 6-5 to 6-7, 6-13 to 6-14,
6-20

S

S (search) DEBUG command
3-37
Scrambler ROM 9-162

Search for first entry (Function 11H) 7-52 to 7-53
Search for next entry (Function 12H) 7-54 to 7-55
Sector 5-21, 5-22, 5-23, 7-109, 8-17
Segment 2-7, 2-14, 2-21, 3-7, 3-8, 3-10, 3-21, 3-28, 3-37, 4-3 to 4-4, 4-7, 4-11, 6-6, 6-7, 6-16, 6-18, 6-21
Select disk (Function 0EH) 7-46
Sequential read (Function 14H) 7-59 to 7-61
Sequential write (Function 15H) 7-62 to 7-63
SET command 6-18
Set date
  (Function 2BH) 7-95 to 7-96
  (Function 57H) 7-157 to 7-158
Set time
  (Function 2DH) 7-99 to 7-100
  (Function 57H) 7-157 to 7-158
Set disk transfer address (Function 1AH) 7-69 to 7-70
Set relative record (Function 24H) 7-79 to 7-81
Set interrupt vector (Function 25H) 7-82
Set verify flag (Function 2EH) 7-101 to 7-102
Shell command 8-35
Speaker 9-147 to 9-149
STACK link switch 2-16
Status byte 5-5
Step through directory files (Function 4FH) 7-153
Switches 2-13 to 2-17, 2-19
Syntax (general) 1-3
System calls 7-1 to 7-173

T

T (trace) DEBUG command 3-38
Terminate address (INT 22H) 7-14
Terminate but stay resident (INT 27H) 7-25 to 7-26
Terminate process (Function 4CH) 7-149
Terminate program (Function 00H) 7-27 to 7-28
Time
  get 7-97 to 7-98, 7-157 to 7-158
  set 7-99 to 7-100, 7-157 to 7-158
Token (see Handle)

U

U (unassemble) DEBUG command 3-39 to 3-40
Utility programs 1-4

V

Video control (also see Console and Output)
  BIOS routine (INT 10H) 8-4, 8-5 to 8-16
Video controller 9-150 to 9-161
VM.TMP file 2-5, 2-6, 2-16

W

W (write) DEBUG command 3-41 to 3-43
Write to file or device (Function 41H) 7-125 to 7-126

The Display Enhancement Board

Table of Contents

1

DEB Capabilities

Introduction 1-2
The DEB Driver 1-4
16-Color Graphics 1-5
Look-Up Table (LUT) 1-7
Overlay Modes 1-8

2

Programming Tips

Presence of Hardware/Software 2-2
Hardware/Software Compatibility 2-3
Setup 2-4

3

How to Program the DEB

Overview 3-2
Mode Setting 3-3
Setting Colors and Effects 3-5
Displaying Graphics Images 3-6

4 Interrupt 10H Functions

Introduction 4-2
Functions 4-4

5 Programming the LUT

Overview 5-2
16-Color Graphics LUT Programming 5-3
Overlay Modes LUT Programming 5-23
Programming the Bit Planes 5-34

1 DEB Capabilities

- Introduction
- The DEB Driver
- 16-Color Graphics
- Look-Up Table (LUT)
- Overlay Modes

Introduction

The Display Enhancement Board (DEB) option adds improved color and graphics functionality to your AT&T PC 6300. When you use the DEB with the PC 6300 color monitor, you can display graphics in up to 16 color combinations simultaneously or treat the screen as two screens in one and overlay one screen treatment on top of the other. When you use the DEB with the PC 6300 monochrome monitor, you have the same capabilities you have with the color monitor, except that colors are displayed as "shades of green."

The DEB is compatible with existing software, so all the programs you have already can be used now as if the DEB were not installed. Of course, these programs may not have access to any of the new capabilities.

This supplement describes the functionality of the DEB device driver. Although it is not necessary to use the driver in order to use the DEB, the driver is designed to work with MS-DOS, GWBASIC, and other AT&T software products. If you wish to program the DEB hardware directly, you must consult the *AT&T Technical Reference Manual.* Such programming is considered a circumvention of the AT&T operating system and we advise against it.

This supplement assumes that you are familiar with video programming through the Interrupt 10H interface and with INTEL® 8086 assembler programming. Information on the Interrupt 10H interface can be found in the *System Programmer's Guide,* in the section on the ROM BIOS Service Routines.

Before you begin writing programs for the DEB, follow the procedures in the DEB Installation Manual for installing the DEB hardware and device driver software.

The DEB is an optional hardware component for the AT&T PC 6300 that works in conjunction with the PC 6300's built-in Video Display Controller (VDC) to provide improved color and graphics functionality.

The built-in VDC contains circuitry and memory that support either 4 color medium resolution (320 × 200 pixels) graphics, 1 color high resolution (640 × 200 pixels) graphics, or 1 color super resolution (640 × 400 pixels) graphics.

The DEB contains additional circuitry and memory that can be combined with the capabilities of the built-in VDC to produce up to 16 color combinations in either high or super resolution. You can also program the VDC and DEB separately, treating them as two separate images that are combined on one screen to produce an overlaying effect. The overlay modes let you use up to 8 colors on the DEB screen and up to 16 colors on the VDC screen.

The DEB Driver

You load the DEB device driver by entering a "DEVICE" statement in the CONFIG.SYS file (see **Chapter 2, Programming Tips**). The driver installs an Interrupt 10H "filter" during the loading process.

When you are using the DEB and are running some programs that use the DEB and some that do not, the "filter" provides video support for both kinds of programs. For programs that do not use the DEB, the filter passes control to the standard Interrupt 10H ROM BIOS routine.

The DEB driver installs a filter for Interrupt 9H. This filter resets the DEB to transparent mode whenever you warmstart the system through **CTRL/ALT/DEL**. The filter controls scrolling when you press **CTRL/NUMLOCK**.

16-Color Graphics

This feature lets you display 16 color combinations in either high resolution (640×200) or super resolution (640×400). Not only can you use the standard 16 colors, you can also combine colors to form new colors and cause pixels to blink from one color to another.

The DEB provides 5 palettes for you to use when programming in color. At any point in your program, you select one of the palettes as the "active" palette. The color combinations contained in that palette determine what colors and effects show on the screen.

Each of the first 4 palettes contains a default set of 16 color combinations, but to suit the needs of your program you can change the contents of the palette to any one of the following:

- any of the 16 standard colors with which you are already familiar from the standard applications. The standard colors are:

0 = black	8 = gray
1 = blue	9 = light blue
2 = green	10 = light green
3 = cyan	11 = light cyan
4 = red	12 = light red
5 = magenta	13 = light magenta
6 = brown	14 = yellow
7 = white	15 = high intensity white

- a mixture, or "dithering," of any 2 of the 16 standard colors
- an alternation, or blinking, between any 2 of the standard 16 colors

The last palette contains no default combinations. You program the fifth palette by loading color values into a 256-byte array. The DEB device driver uses this special palette to program the DEB's color Look-Up Table (LUT). By using the LUT you can add the capability of dithering or blinking between any four colors.

Look-Up Table (LUT)

The LUT resides in RAM on the DEB board, and is accessed through write-only hardware registers. The device driver keeps a copy of the register values in the LUT. The register values are accessible to software applications through the device driver. The LUT contains 256 values that determine the colors, blinking, and dithering that appear on the screen. Whether you need to learn about the use and layout of the LUT depends on the application you are writing.

If you use the standard palettes, you need not be concerned with the LUT. The DEB device driver automatically programs the LUT to correspond to the way you set up the palettes. If you program a custom LUT, you greatly increase the color combinations and blinking effects available to you.

Overlay Modes

The overlay modes let you use the screen to display two images at once, independently. For example, you can display a high resolution color graphics image with its own foreground and background. Then, on "top" of that image, you can display a box of text and scroll the text without affecting the graphics image.

The overlay modes use the DEB to control one image and use the standard controller board to control the other image. You can select from many combinations of graphics, text, color, and high or super resolution in designing the two images.

The overlay modes offer 5 palettes. Each of the first 4 palettes has 8 positions. These four palettes have default colors that you can change to suit your needs. You can choose 8 color combinations from any of the 16 standard colors, or blink between 2 of the standard colors. The dithering combinations of the 16-color graphics modes are not available. You can also use the last palette to custom program the LUT.

2 Programming Tips

- Presence of Hardware/Software
- Hardware/Software Compatibility
- Setup

Presence of Hardware/Software

Whenever you plan an application, it is important to use the DEB device driver to test for the presence of both the DEB and the associated driver. Test for the presence of the hardware by checking for DEB video memory. This is accomplished by writing and reading back data patterns into memory, in the range A000H:0H to B800H:0H. Test for the software device driver by issuing a function call to open the device called "DEBDRIVE," then immediately issuing a call to close "DEBDRIVE." If the open fails (carry set on return from Interrupt 21H) then the driver is not present. No functions are implemented in the driver, which is used only to detect the presence of the software.

Hardware/Software Compatibility

The driver software has been designed to fit into the structure of MS-DOS programs. The DEB hardware uses the same range of addresses as the standard video ports on any compatible machine. If your application uses a light pen, consult the DEB supplement in the *AT&T Personal Computer Technical Reference Guide*.

The DEB driver makes minor modifications to the ROM BIOS video interrupt. Mode setting and color selection offer additional functionality. Be careful when you use the following functions.

- SET MODE — uses an additional register BL
- SCROLLING — uses an additional register BH
- STATUS — returns an additional register pair ES:DI. No application should count on ES:DI not changing.

Setup

Install the DEB driver just as you would install any device driver. Be sure the CONFIG.SYS file is in the root directory. Put the line DEVICE = DEDRIVER.DEV in CONFIG.SYS. This line puts the DEB driver in the device driver chain. The driver makes patches in INT 10H and INT 9H to add the new functionality. The driver has two features:

- the INIT function, which deallocates itself after it runs
- chaining, which allows you to test for the driver's presence by issuing an open function call

3 How to Program the DEB

- Overview
- Mode Setting
- Setting Colors and Effects
- Displaying Graphics Images

Overview

There are three steps for video programming that apply whether or not you are using the DEB capability:

1. Set the hardware's mode. You also must set the active page if you are in an overlay mode and want to select the DEB screen.
2. Select the color combinations and effects you want to use.
3. Construct the graphics images you want to display.

This chapter describes each of these steps in detail. This chapter does **not** describe how to program the LUT directly (see **Chapter 5, "Programming the LUT"**).

Mode Setting

The DEB is controlled by invoking one of the DEB video modes through the Set Mode function (INT 10H, function 0H). The Set Mode function establishes the mode for both the DEB and the VDC. These modes fall into four categories: 16-color graphics, overlay, transparent, and disabled.

16-Color Graphics Modes

There are two DEB modes that provide 16-color graphics: high resolution and super resolution. Both these modes let you use 5 palettes and display up to 16 color combinations simultaneously.

Overlay Modes

When overlaying the VDC on the DEB output, you specify one of the modes for the VDC and one mode for the DEB. The VDC modes are a subset of the modes for non-DEB graphics: 80 × 25 text mode, high and super resolution modes. The DEB modes are both graphics modes: high and super resolution.

If you are using one of the four standard palettes, the VDC's output takes precedence over the output of the DEB, so that if each board writes a pixel to the same screen location, the pixel sent by the VDC is displayed. This precedence is programmed into the LUT. If you want to have the DEB take precedence over the VDC, you must change the values in the LUT. (For more information, see **Chapter 5, "Programming the LUT."**)

Transparent Mode

The non-DEB modes, modes 0-40H and mode 48H, work exactly as they work without the DEB device driver installed.

Disabled Mode

In the disabled mode, you can cause the output of the VDC, the DEB, or both to be blacked out. This allows you to draw a graphics image or to fill a screen with text and not have them displayed while you are building them. You can then have the image "pop up" by taking VDC or DEB out of the disabled mode. You can also achieve this result by using the programmable palettes and the LUT.

Setting Colors and Effects

Colors and effects are controlled by the Set Color Palette command, (INT 10H function 0BH). Use this function to set color values in one of the four palettes, to switch between palettes, or to reset palettes to their default values. You also use Set Color Palette to program the LUT directly.

Displaying Graphics Images

There are two methods for displaying graphics images using the DEB: writing dots at screen locations or directly programming the VDC and DEB memory.

To write dots (pixels) to the screen, use the Write Dot function (INT 10H, function 0CH). Write Dot requires that you specify the display page, the row and column where you want the dot to appear, and the color or pattern for the dot.

If you want to program the VDC and DEB graphics memory directly , you need to learn the details of how the LUT is structured and how LUT addresses are formed (see the section on "Programming the Bit Planes" in Chapter 5).

4 Interrupt 10H Functions

- Introduction
- Functions

Introduction

The following section describes the DEB device driver software functions. This interface is an extension of the INT 10H software function to the PC6300 ROM BIOS that controls the VDC. The ROM BIOS screen driver has 16 functions:

0H	set the display mode
2H	set the cursor position
3H	read the cursor position
5H	select the active display page
6H	scroll the active page up
7H	scroll the active page down
8H	read character/attribute at the current cursor position
9H	write character/attribute at cursor position
AH	write only the character at current cursor position
BH	change the color palette
CH	write a point on the screen

DH read a point on the screen

EH write in teletype style to the active page

FH return information about the active video state

Not all these functions are applicable to the DEB. The filter receives the Interrupt 10H function call, filters the functions that are applicable to the DEB and performs them. The functions that are not applicable to the DEB are passed on to the ROM BIOS INT 10H routine or to a previously installed filter or driver routine. The following section describes the functions which are processed by the DEB Interrupt 10H filter.

Functions

Set Mode

The function establishes the mode for both the DEB and the VDC. If you select a non-DEB related mode, control is passed to the ROM resident Set Mode function. Set Mode initializes palette 0 as the active palette.

Input

(AH)	= 0H	function number for Set Mode
(AL)	= new mode	
(BL)	= optional overlay mode	

Setting AL bit 7 = 0 puts you in either the DEB transparent mode or 16-color graphics mode:

(AL) = 0H	40 × 25 monochrome, text
(AL) = 1H	40 × 25 color, text
(AL) = 2H	80 × 25 monochrome, text
(AL) = 3H	80 × 25 color, text
(AL) = 4H	320 × 200 color
(AL) = 5H	320 × 200 monochrome
(AL) = 6H	640 × 200 color
(AL) = 40H	640 × 400 with 2-position programmable palette, defaulting to black and white
(AL) = 41H	640 × 200 16-color graphics with four palettes
(AL) = 42H	640 × 400 16-color graphics with four palettes
(AL) = 44H	Disable mode (disables both DEB and VDC output)

Setting AL bit 7 = 1 puts you in overlay mode. The following values are only used in overlay mode. AL contains the setting for the VDC; BL contains the mode setting for the DEB. In overlay modes, the active page defaults to zero.

VDC Settings

(AL) = 82H	80 × 25 monochrome, text
(AL) = 83H	80 × 25 color, text
(AL) = 86H	640 × 200 color graphics
(AL) = 0C0H	640 × 400 color graphics
(AL) = 0C4H	Disable mode. Disables only the VDC.

DEB settings

(BL) = 6H	640 × 200 graphics with four 8-position palettes.
(BL) = 40H	640 × 400 graphics with four 8-position palettes.
(BL) = 44H	Disable mode. Disables only the DEB.

Output

Contents of all registers are preserved.

Example

```
MOV AH,0      ; Select Set Mode
MOV AL,41H    ; Select 16 color graphics
INT 10H       ; Change the mode
```

Set Cursor Position

This function sets the cursor position for either the DEB, the VDC, or both.

Input

(AH) = 2H Function number for Set Cursor Position

(DH,DL) = row, column of new position

(BH) = page number
Valid page numbers for DEB modes are 0 for the VDC and 80H for the DEB in overlay mode. Row values are 0 thru 23, column values are 0 thru 79, in DEB modes.

Output

Contents of all registers are preserved.

Example

```
MOV AH,2    ; SCP function
MOV DH,ROW
MOV DL,COL
MOV BH,PAGE
INT 10H    ; Moves cursor to position defined in
             above variables.
```

Read Cursor Position

This function returns the position of the cursor for the DEB, VDC, or both.

Input

(AH) = 3H Function number for Read Cursor Position

(BH) = page number
Valid page numbers for DEB modes are 0 for VDC and 80H for the DEB in overlay mode. Row values are 0 thru 23, column values are 0 thru 79, in DEB modes.

Output

(DH,DL) = row, column of current position.

Contents of all other registers are preserved.

Example

```
MOV AH,3
MOV BH,PAGE
INT  10H
MOV ROW,DH
MOV COL,DL
```

Select Active Display	This command allows selection of the DEB display in overlay mode.
Input	(AH) = 5H Function number for Select Action Display (AL) = active page Values for the active page in DEB modes are, 0 for the VDC and 80H for the DEB.
Output	Contents of all registers are preserved.
Example	**MOV AH,5** **MOV AL,PAGE** **INT 10H**

Scroll Active Page Up

This function defines a pattern that is to be displayed on the blank lines as the screen scrolls. The pattern consists of ones and zeros. Zeros are interpreted as the background color (palette position zero). Ones are interpreted as the foreground color, which is defined in BL. Care should be taken when scrolling in DEB modes, to insure that all applications set the additional argument in BH correctly.

Input

(AH)	=	6H function number for Scroll Active Page Up
(AL)	=	number of lines to scroll
(CH,CL)	=	row, column of upper left corner to scroll
(DH,DL)	=	row, column of lower right corner to scroll
(BH)	=	pattern to be used on blank lines
(BL)	=	foreground color

The range of lines to be scrolled is 0 thru 23 (where 0 specifies clear screen).
Row values are 0 thru 23, column values are 0 thru 79, in DEB modes.
Valid foreground colors are specified by palette position 0-FH for 16-color graphics, and 0-7H for 8-color graphics.

Output

Contents of all registers are preserved.

Example

```
MOV AH,6    ;Scroll Active Page Up
MOV AL,LINES
MOV CH,UPROW
MOV CL,UPCOL
MOV DH,LOWROW
MOV DL,LOWCOL
MOV BH,0
MOV BL,FGCOLOR
INT   10H
```

Scroll Active Page Down

This function permits you to define a pattern that is to be displayed on the blank lines as the screen scrolls downward. The pattern consists of ones and zeros. Zeros are interpreted as the background color (palette position zero). Ones are interpreted as the foreground color, which is defined in BL. Care should be taken when scrolling in DEB modes, to insure that all applications set the additional argument in BH correctly.

Input

(AH)	=	7H function number for Scroll Active Page Down
(AL)	=	number of lines to scroll
(CH,CL)	=	row, column of upper left corner to scroll
(DH,DL)	=	row, column of lower right corner to scroll
(BH)	=	pattern to be used on blank lines
(BL)	=	foreground color

The range of lines to be scrolled is 0 thru 23 (where 0 specifies clear screen).
Row values are 0 thru 23, column values are 0 thru 79, in DEB modes.
Valid foreground colors are specified by palette position 0-FH for 16-color graphics, and 0-7H for 8-color graphics.

Output

Contents of all registers are preserved.

Example

```
MOV AH,7   ; Scroll Active Page Down
MOV AL,LINES
MOV CH,UPROW
MOV CL,UPCOL
MOV DH,LOWROW
MOV DL,LOWCOL
MOV BH,0
MOV BL,FGCOLOR
INT   10H
```

Read Character and Attribute at Current Cursor Position

This function returns the value of the character at the current cursor position. The value of the character's foreground color is returned in AH.

Input

(AH)	=	8 Function number for Read Character and Attribute at Current Cursor Position.
(BH)	=	Valid page numbers for DEB modes are 0 for the VDC and 80H for the DEB in overlay mode.

Output

(AL)	=	ASCII character code
(AH)	=	foreground palette position or VDC attribute

Contents of all other registers are preserved.

Example

```
MOV AH,8    ;Read CHR function
MOV BH,PAGE
INT  10H
MOV CHAR,AL ;Save CHAR/COLOR
MOV CURCOLOR,AH
```

Write Character and Attribute at Current Cursor Position

This function displays the character whose ASCII code is in register AL. The character is displayed according to the color values in BL.

Input

(AH)	= 9H	Write Character function
(AL)	= ASCII character code	
(BL)	= foreground color	
(BH)	= page	
(CX)	= count of characters to write	

If bit 7 of BL = 1, the color value is XOR'd with the current dots in that location. Valid page numbers for DEB modes are 0 for the VDC and 80H for the DEB in overlay mode.

Valid foreground colors are specified by palette position 0-FH for 16-color graphics, and 0-7H for 8-color graphics.

Output

Contents of all registers are preserved.

Example

```
MOV AH,9
MOV AL,CHAR
MOV BL,CURCOLOR
MOV BH,PAGE
MOV CX,1
INT  10H
```

Write Character Only at Current Cursor Position

In DEB modes, this function is the same as "Write Character and Attribute."

Set Color Palette

This function is used to set color values in one of the four palettes, to switch between palettes, or to reset palettes to their default values.

In the overlay modes, the Set Color Palette function works on the active page. If the active page is set to display to the VDC board, this function works the same as the standard ROM BIOS INT 10H (function 0BH).

If you specify a palette position greater than the value allowed for the mode in which you are working, the value you specify will be put in that palette's highest position. For example, if you attempted to set palette position 13 to red when working in overlay mode, which has 8-position palettes, the 8th palette position would be set to red.

Note:
The following discussion covers the use of the simple palette programming functions. You can also use "Set Color Palette" to program the LUT. (For more information, see **Chapter 5, "Programming the LUT"**).

Input

(AH)	= 0BH	Function number for Set Color Palette
(AL)	= palette function selector	
(BH)	= positional pointer	
(BL)	= color value	

For simple palette programming functions, use the following

(AL) = 0
(BH) = palette color ID

BH = FFH	switches to the palette specified in BL, without changing to the default palettes unless there is a change in palette type (e.g., change from a 16-position palette to an 8-position palette).
BH = 80H	switches to the palette specified in BL and resets the palette to its default.
BH = 0-16	sets this palette position to the color or attribute in BL.

(BL) = actual color value or code for blinking and dithering

The special settings for using a customized LUT in Set Color Palette are as follows:

Input

(AL) = non-zero (a zero here selects a standard palette)

AL bit 0 = 1 means use ES:SI to program the palette and registers BH and BL to indicate an offset and length into the LUT. ES:SI points to the LUT table (in the above example, LUT-STRING).

In this case, BH = offset into LUT and BL = length of portion of LUT to be changed. If you are loading an entire new table, set BH and BL to 0.

AL bit 1 = 1 means use BH and BL to program the LUT one location at a time.

In this case, BH = position in LUT and BL = the value to put in that position.

AL bit 2 = 1 means use the short LUT addressing mode. (Only uses the first 16 LUT entries).

The DEB driver lets you automatically load your customized LUT and use it in place of one of the standard palettes.

The steps for loading and using the customized LUT are:

1 Define the table with DB (Define Byte) statements.

2 Load the table in by using the Set Color Palette command.

3 Use the Read Dot and Write Dot commands to access the LUT (see **Chapter 4, "Interrupt 10H Functions"**).

The code for defining the table would be similar to this:

```
LUT-STRING    DB  4          ! Signifies active palette 4
              DB  1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,
              DB  1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,
              DB  1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,
              DB  1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,
              .
              .
              .
              DB  1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,
```

To load a new table of values into the LUT, where the table in your program is named LUT-STRING, you can use these statements:

```
PUSH DS          ! save the data segment address
POP ES
MOV   S1,LUT-STRING
MOV   AL,1
MOV   AH,11
XOR   BH,BX      ! Sets BH = BL = 0
INT   10
```

The defaults for each of the four palettes are:

Default Palettes

Palette Number 0

Position	Color
0	0 = black
1	2 = green
2	4 = red
3	6 = brown
4	1 = blue
5	3 = cyan
6	5 = magenta
7	7 = white
8	8 = gray
9	9 = light blue
10	10 = light green
11	11 = light cyan
12	12 = light red
13	13 = light magenta
14	14 = yellow
15	15 = high-intensity white

Palette Number 1

Position	Color
0	0 = black
1	3 = cyan
2	5 = magenta
3	7 = white
4	1 = blue
5	2 = green
6	4 = red
7	6 = brown
8	8 = gray
9	9 = light blue
10	10 = light green
11	11 = light cyan
12	12 = light red
13	13 = light magenta
14	14 = yellow
15	15 = high-intensity white

Palettes 2 and 3 are the same, and they contain the standard colors in numerical order.

Palette Number 2 and Palette Number 3

Position	Color
0	0 = black
1	1 = blue
2	2 = green
3	3 = cyan
4	4 = red
5	5 = magenta
6	6 = brown
7	7 = white
8	8 = gray
9	9 = light blue
10	10 = light green
11	11 = light cyan
12	12 = light red
13	13 = light magenta
14	14 = yellow
15	15 = high-intensity white

DITHER COMBINATIONS FOR DEB PALETTES 0-3

Color combinations 136-255 have been pre-assigned to allow you easy access to dithering effects while using the standard palettes. The following table describes the available combinations.

A ↓ / B →	black	blue	green	cyan	red	magenta	brown	white	gray	light blue	light green	light cyan	light red	light magenta	yellow
black															
blue	136														
green	137	138													
cyan	139	140	141												
red	142	143	144	145											
magenta	146	147	148	149	150										
brown	151	152	153	154	155	156									
white	157	158	159	160	161	162	163								
gray	164	165	166	167	168	169	170	171							
light blue	172	173	174	175	176	177	178	179	180						
light green	181	182	183	184	185	186	187	188	189	190					
light cyan	191	192	193	194	195	196	197	198	199	200	201				
light red	202	203	204	205	206	207	208	209	210	211	212	213			
light magenta	214	215	216	217	218	219	220	221	222	223	224	225	226		
yellow	227	228	229	230	231	232	233	234	235	236	237	238	239	240	
high-intensity white	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255

NOTE: To select a value that combines colors A and B to create a new color, find the number at the intersection of row A and column B.

BLINKING COLOR EFFECTS FOR DEB PALETTES 0-3

Color combinations 16-135 have been pre-assigned to allow you easy access to blinking effects while using the standard palettes. The following table describes the available combinations.

A ↓ \ B →	blue	green	cyan	red	magenta	brown	white	gray	light blue	light green	light cyan	light red	light magenta	yellow	high-intensity white
black	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
blue		31	32	33	34	35	36	37	38	39	40	41	42	43	44
green			45	46	47	48	49	50	51	52	53	54	55	56	57
cyan				58	59	60	61	62	63	64	65	66	67	68	69
red					70	71	72	73	74	75	76	77	78	79	80
magenta						81	82	83	84	85	86	87	88	89	90
brown							91	92	93	94	95	96	97	98	99
white								100	101	102	103	104	105	106	107
gray									108	109	110	111	112	113	114
light blue										115	116	117	118	119	120
light green											121	122	123	124	125
light cyan												126	127	128	129
light red													130	131	132
light magenta														133	134
yellow															135

NOTE: To select a value that will cause blinking between colors A and B, find the number at the intersection of row A and column B.

Write Dot

The Write function writes a pixel to the location on the screen that you specify. If the screen is in the DEB mode, Write Dot may also write a pattern.

Input

(AH) = 0CH, Function number for Write Dot
(AL) = Palette position to be written.
(BH) = display page designator (bit 7 = 1 selects the DEB)
(CX) = column number
(DX) = row number

Output

Contents of all registers are preserved.

Example

```
MOV AH,0CH
MOV AL,PALPOS
MOV BH,PAGE
MOV CX,COL
MOV DX,ROW
INT   ;Write the Dot
```

Read Dot

This function reads a dot from the screen. If the screen is in the DEB mode this function returns the value in the LUT that corresponds to this dot. (For more information, see **Chapter 5, "Programming the LUT."**)

Input

(AH)	=	0DH Function number for Read Dot
(BH)	=	display page designator (bit 7 = 1 selects the DEB)
(CX)	=	column number
(DX)	=	row number

Output

(AH)	=	VDC value or DEB palette position
		Contents of all other registers are preserved.

Example

```
MOV AH,0DH
MOV BH,PAGE
MOV CX,COL
MOV DX,ROW
INT
MOV DOTCOL,AH    ;Save the Dot
```

Input	(AH)	=	0EH, Function number for Write Teletype
	(AL)	=	character to write
	(BL)	=	foreground color (in graphics modes) If bit 7 = 1, color is XOR'd to current contents.

Output Contents of all registers are preserved.

Example

```
MOV AH,0EH
MOV AL,CHAR
MOV BL,FGCOL
INT   10H
```

Read Current Video State	This function returns the current video state. It indicates whether the DEB or VDC is active in the overlay mode and returns the number of the active palette.
Input	(AH) = 0FH Function number for Read Current Video State
Output	(BH) = display page designator (AL) = mode currently set (ES:DI) = pointer to a copy of the current LUT
Example	**MOV AH,0FH** **INT 10H**

5 Programming the LUT

- Overview
- 16-Color Graphics LUT Programming
- Overlay Modes LUT Programming
- Programming the Bit Planes

Overview

This chapter describes programming the DEB Look-Up Table (LUT). By programming the LUT yourself, you can create color patterns that are not available when you use standard palettes. You need not read this chapter if you do not want to use this extended functionality.

The hardware uses the LUT to translate the contents of video memory patterns into graphics effects. In the standard palettes, INT 10H filter programs the LUT for you and thereby provides the preassigned color combinations and effects as described in previous chapters.

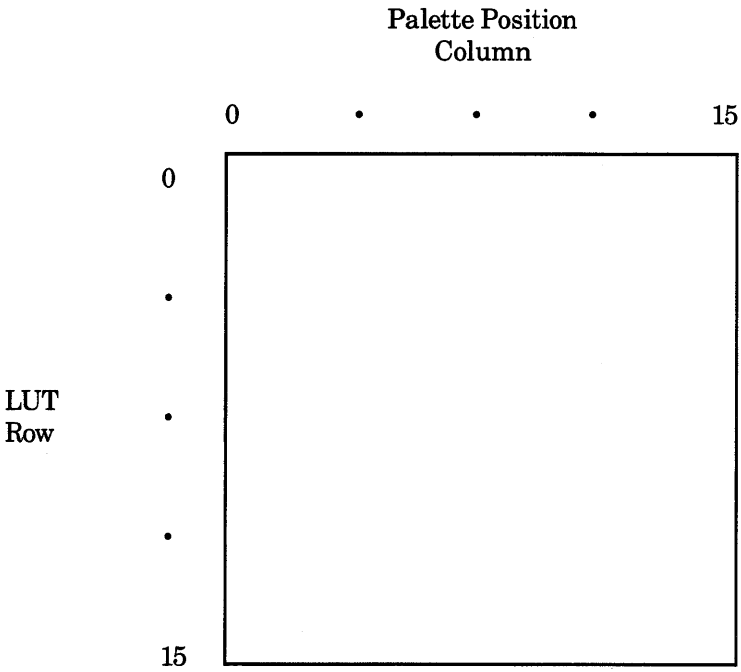
To program the LUT directly, you select Palette 4 in Set Color Palette function. Palette 4, also called the "LUT palette," has a minimum of 256 positions. Each palette position contains a value between 0 and 15. These values map into the LUT locations on the DEB. The 256 locations on the DEB collectively determine the color and special effects displayed when you specify a particular palette position. The color and special effect for each pixel on the screen are determined by:

- the palette position you specify
- the values in the LUT
- the active mode

There are some differences in the way the LUT is structured for 16-color graphics modes and overlay modes. This chapter describes LUT operation for 16-color graphics modes and overlay modes separately.

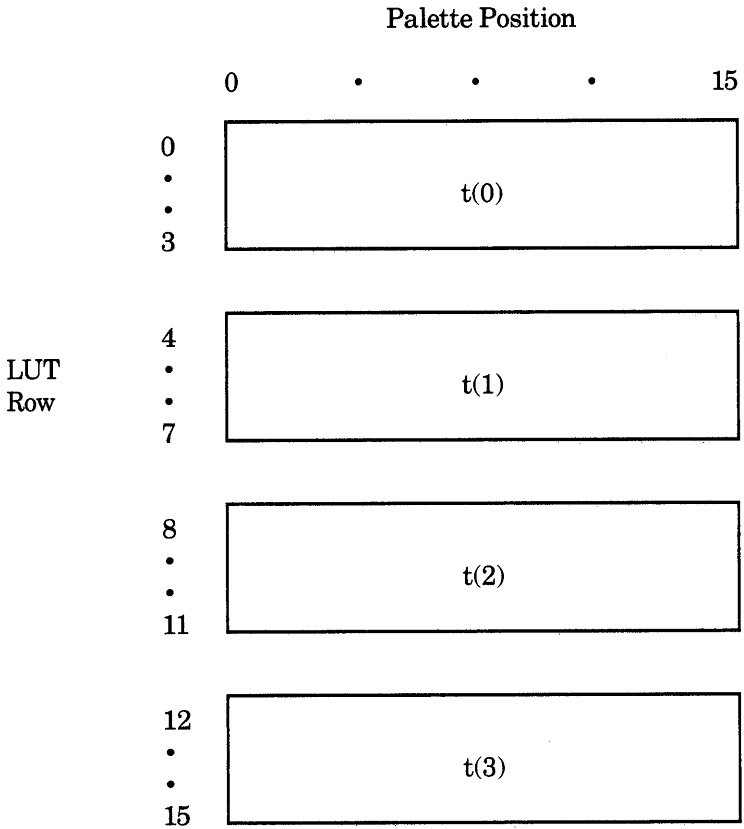
16-Color Graphics Lut Programming

In these modes, the LUT can be viewed as a two-dimensional array (16 × 16). Each location contains one of the standard 16 colors.



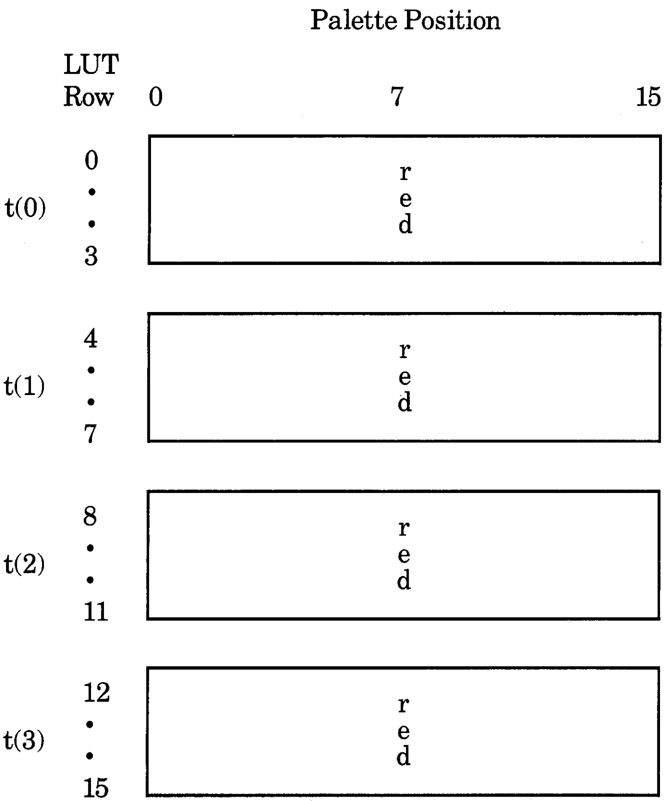
The locations in the LUT are numbered consecutively from left to right and top to bottom. Thus, location 17 corresponds to Row 1, palette position 1.

In the 16-color graphics mode, the LUT is divided into four "time states." At any one time, only one quarter of the LUT determines the display on the screen.



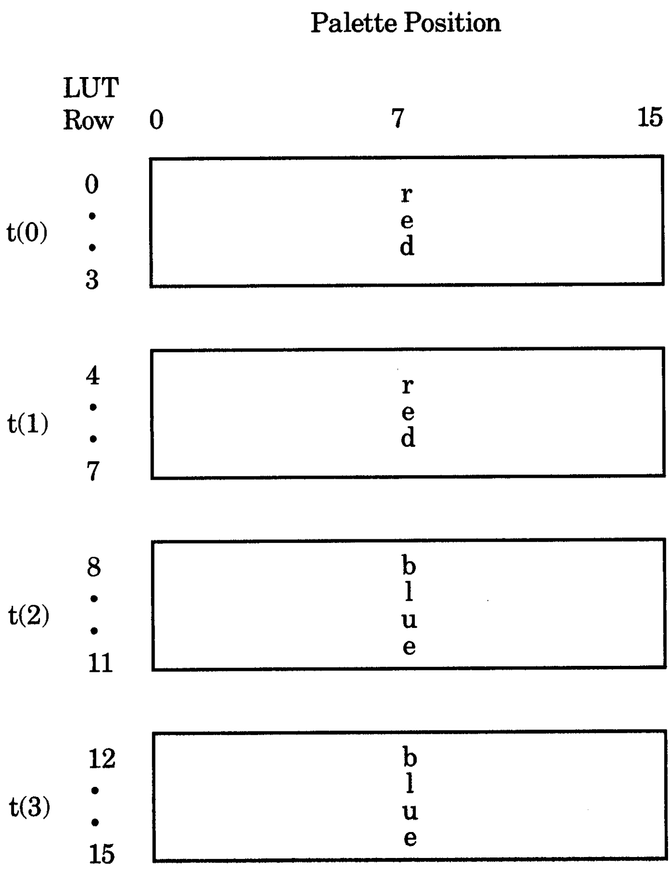
The hardware cycles through the LUT every second, so each quarter of the LUT is active for ¼ of each second. The cycling mechanism produces blinking. The following examples show the details of how you can produce several different blinking effects by setting different values in the LUT.

In this example, the Write Dot or Write Character functions specify palette position 7 and the LUT is set up as shown. Pixels are displayed as a solid red color. In the first ¼ second, the DEB displays the color in the first quarter of the LUT, which in this case is red. In the second, third, and fourth ¼ seconds, the DEB displays the color in the second, third, and fourth quarters of the LUT, respectively. In this example, the DEB keeps finding the color value for red, so what you see on the screen is a solid (non-blinking) red color.



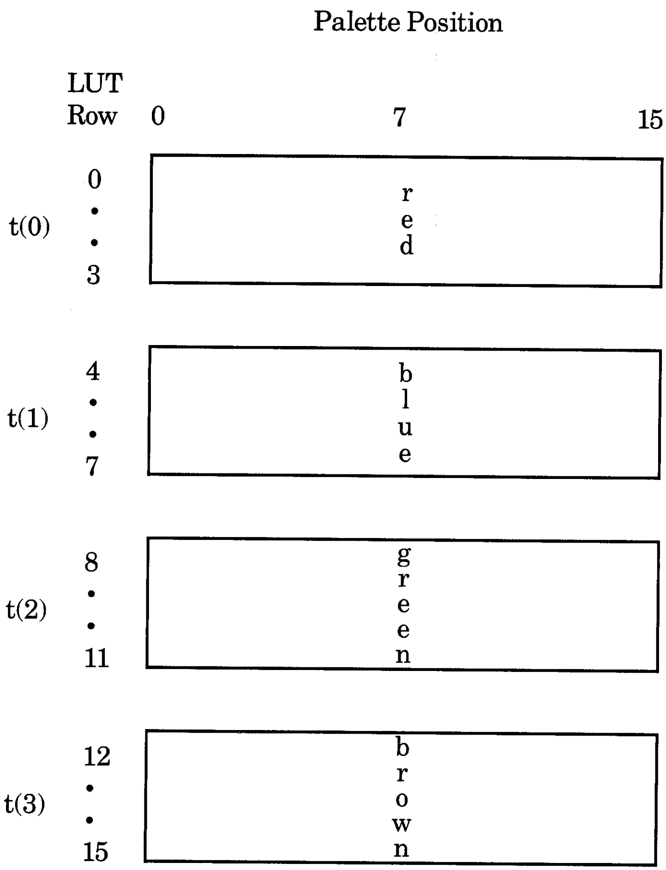
Non-Blinking Color

In this example, any item displayed on the screen with palette position 7 blinks between red and blue. For the first two ¼ seconds, the DEB picks up the color value for red from the first and second quarters of the LUT. For the second two ¼ seconds, the DEB obtains the color value of blue from the LUT. The net effect is a slow blink between red and blue.



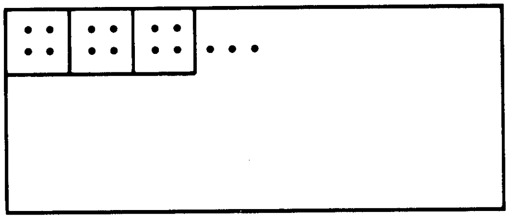
Slow Blink

In this example, any item displayed using palette position 7 blinks rapidly between red, blue, green, and brown.

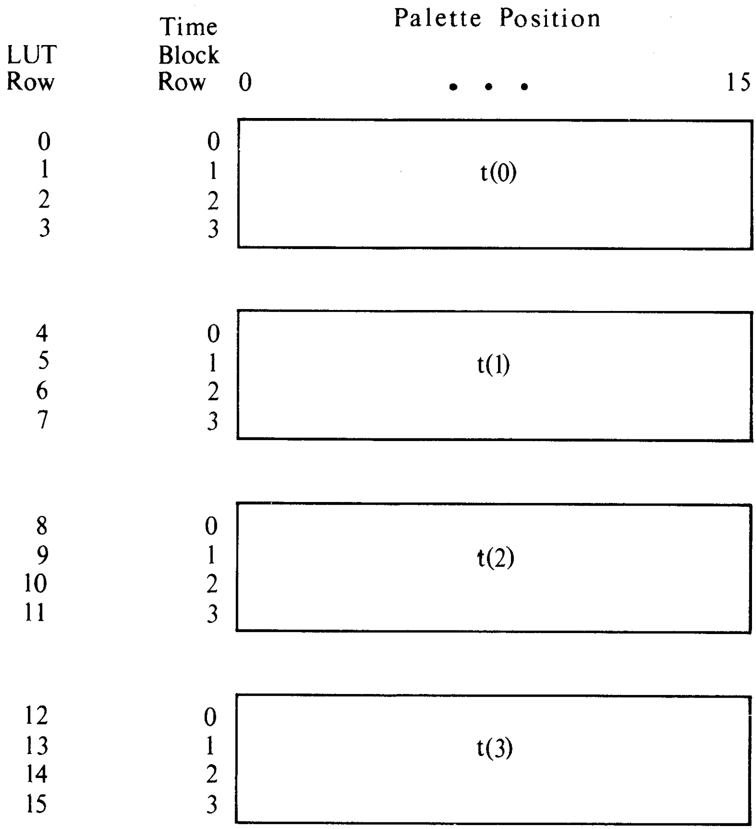


4-Color Fast Blink

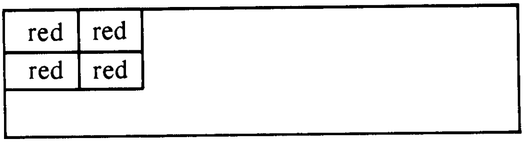
For dithering colors, the DEB uses a scheme similar to the blinking scheme. Dithering is accomplished by manipulating groups of 4 adjacent pixels. The screen is divided into blocks of 4 pixels.



Each of the 4 time states is divided into four dither states that determine the dithering effect. The rows of the time state blocks correspond to the 4-pixel blocks on the screen in the following way:



The pixels in the pixel blocks are so close together that our eyes cannot perceive them as separate. If each of the pixels in a pixel block is a different color, our eyes perceive the pixel block as one color — a combination of the color of the individual pixels. If the adjacent pixels are the same color, our eyes see just that one color.

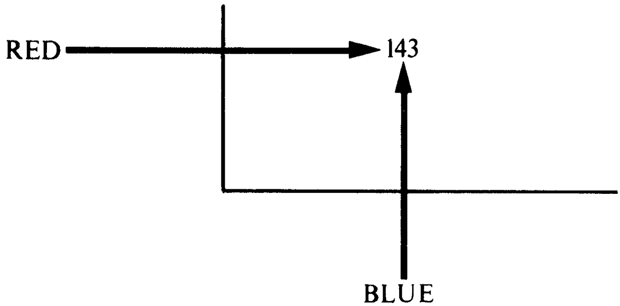


Palette Position

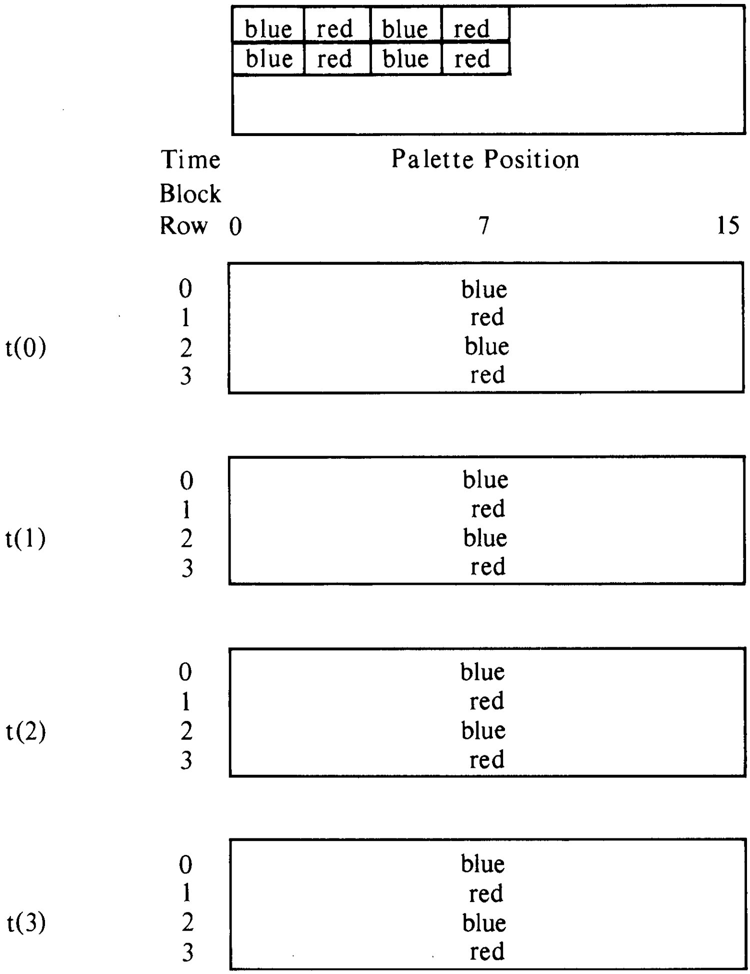
Time Block	Row	0	7	15
	0		red	
	1		red	
t(0)	2		red	
	3		red	
	0		red	
	1		red	
t(1)	2		red	
	3		red	
	0		red	
	1		red	
t(2)	2		red	
	3		red	
	0		red	
	1		red	
t(3)	2		red	
	3		red	

"Solid" Dither showing correspondence between pixel positions in a pixel block and time state rows

Remember the table of "pre-assigned" dithered colors. To combine colors, you check the table for the color number for a particular dither effect. For example, you would choose this number to produce a dither between red and blue.

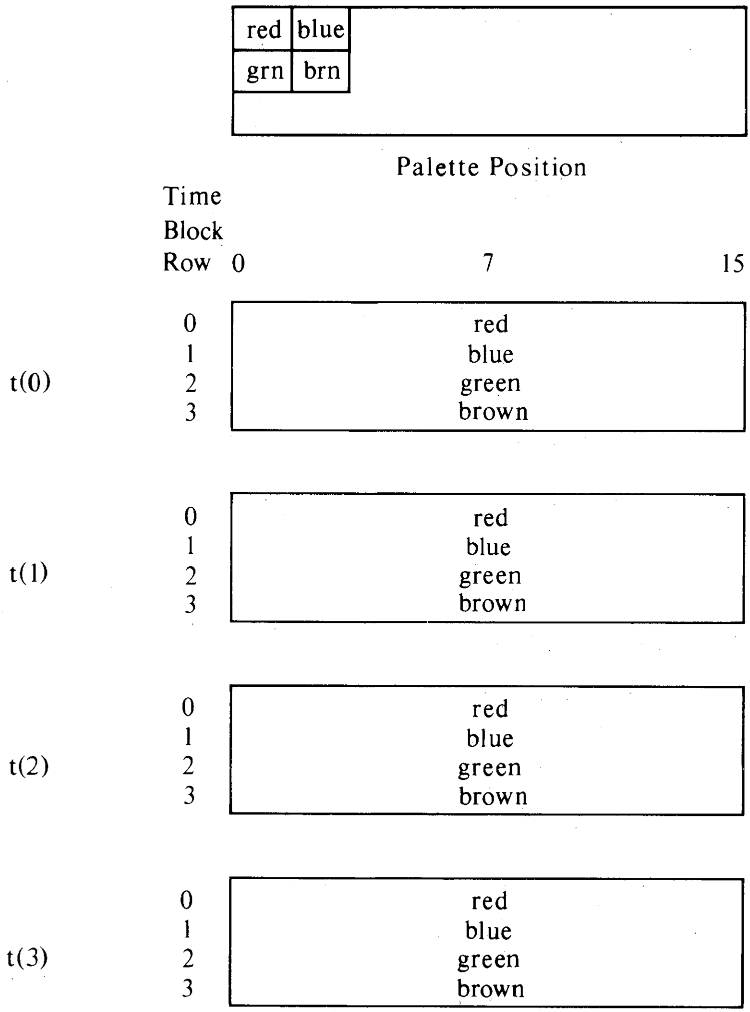


If you want to program the LUT directly to dither red and blue together, the LUT would look like this:



2-Color Dither

You can set up the LUT to dither two, three, or four colors together.



4-Color Dither

The following examples show the actual LUT values for each of the previous cases of blinking and dithering.

Palette Position

	LUT Row	0	7	15
	0		4 (red)	
	1		4	
t(0)	2		4	
	3		4	
	4		4	
	5		4	
t(1)	6		4	
	7		4	
	8		4	
	9		4	
t(2)	10		4	
	11		4	
	12		4	
	13		4	
t(3)	14		4	
	15		4	

Palette Position 7 programmed for Non-Blinking Red

Palette Position

	LUT Row	0	7	15
	0		4 (red)	
	1		4	
t(0)	2		4	
	3		4	
	4		4	
	5		4	
t(1)	6		4	
	7		4	
	8		1 (blue)	
	9		1	
t(2)	10		1	
	11		1	
	12		1	
	13		1	
t(3)	14		1	
	15		1	

Palette Position 7 programmed to blink slowly between red and blue.

Palette Position

	LUT Row	0	7	15
	0		4 (red)	
	1		4	
t(0)	2		4	
	3		4	
	4		1 (blue)	
	5		1	
t(1)	6		1	
	7		1	
	8		2 (green)	
	9		2	
t(2)	10		2	
	11		2	
	12		6 (brown)	
	13		6	
t(3)	14		6	
	15		6	

4-Color Fast Blink

	LUT Row	Palette Position 0	7	15
	0		4 (red)	
	1		4	
t(0)	2		4	
	3		4	
	4		4	
	5		4	
t(1)	6		4	
	7		4	
	8		4	
	9		4	
t(2)	10		4	
	11		4	
	12		4	
	13		4	
t(3)	14		4	
	15		4	

Solid Red Dither

	LUT Row	Palette Position 0	7	15
	0		1 (blue)	
	1		4 (red)	
t(0)	2		1 (blue)	
	3		4 (red)	
	4		1	
	5		4	
t(1)	6		1	
	7		4	
	8		1	
	9		4	
t(2)	10		1	
	11		4	
	12		1	
	13		4	
t(3)	14		1	
	15		4	

2-Color Dither: Red and Blue

Palette Position

	LUT Row	0	7	15
	0		4 (red)	
	1		2 (green)	
t(0)	2		1 (blue)	
	3		6 (brown)	
	4		4	
	5		2	
t(1)	6		1	
	7		6	
	8		4	
	9		2	
t(2)	10		1	
	11		6	
	12		4	
	13		2	
t(3)	14		1	
	15		6	

4-Color Dither Between Red, Green, Blue, and Brown

The following is an example that combines blinking and dithering:

Palette Position

	LUT Row	0	7	15
	0		1 (blue)	
	1		4 (red)	
t(0)	2		1	
	3		4	
	4		1	
	5		4	
t(1)	6		1	
	7		4	
	8		2 (green)	
	9		6 (brown)	
t(2)	10		2	
	11		6	
	12		2	
	13		6	
t(3)	14		2	
	15		6	

The following table of values can be used to program the LUT for normal 16-color graphics.

Palette Position

	LUT Row	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	0	0,	1,	2,	3,	4,	5,	6,	7,	8,	9,	10,	11,	12,	13,	14,	15,
	1	0,	1,	2,	3,	4,	5,	6,	7,	8,	9,	10,	11,	12,	13,	14,	15,
t(o)	2	0,	1,	2,	3,	4,	5,	6,	7,	8,	9,	10,	11,	12,	13,	14,	15,
	3	0,	1,	2,	3,	4,	5,	6,	7,	8,	9,	10,	11,	12,	13,	14,	15,
	4	0,	1,	2,	3,	4,	5,	6,	7,	8,	9,	10,	11,	12,	13,	14,	15,
	5	0,	1,	2,	3,	4,	5,	6,	7,	8,	9,	10,	11,	12,	13,	14,	15,
t(1)	6	0,	1,	2,	3,	4,	5,	6,	7,	8,	9,	10,	11,	12,	13,	14,	15,
	7	0,	1,	2,	3,	4,	5,	6,	7,	8,	9,	10,	11,	12,	13,	14,	15,
	8	0,	1,	2,	3,	4,	5,	6,	7,	8,	9,	10,	11,	12,	13,	14,	15,
	9	0,	1,	2,	3,	4,	5,	6,	7,	8,	9,	10,	11,	12,	13,	14,	15,
t(2)	10	0,	1,	2,	3,	4,	5,	6,	7,	8,	9,	10,	11,	12,	13,	14,	15,
	11	0,	1,	2,	3,	4,	5,	6,	7,	8,	9,	10,	11,	12,	13,	14,	15,
	12	0,	1,	2,	3,	4,	5,	6,	7,	8,	9,	10,	11,	12,	13,	14,	15,
	13	0,	1,	2,	3,	4,	5,	6,	7,	8,	9,	10,	11,	12,	13,	14,	15,
t(3)	14	0,	1,	2,	3,	4,	5,	6,	7,	8,	9,	10,	11,	12,	13,	14,	15,
	15	0,	1,	2,	3,	4,	5,	6,	7,	8,	9,	10,	11,	12,	13,	14,	15,

Non-Blinking Standard Colors

Note that palette position 7 in the first two time states has been programmed to show white and in the second two time states to show red.

Palette Position

	LUT Row	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
	0	0, 1, 2, 3, 4, 5, 6, **7,** 8, 9, 10, 11, 12, 13, 14, 15,
	1	0, 1, 2, 3, 4, 5, 6, **7,** 8, 9, 10, 11, 12, 13, 14, 15,
t(o)	2	0, 1, 2, 3, 4, 5, 6, **7,** 8, 9, 10, 11, 12, 13, 14, 15,
	3	0, 1, 2, 3, 4, 5, 6, **7,** 8, 9, 10, 11, 12, 13, 14, 15,
	4	0, 1, 2, 3, 4, 5, 6, **7,** 8, 9, 10, 11, 12, 13, 14, 15,
	5	0, 1, 2, 3, 4, 5, 6, **7,** 8, 9, 10, 11, 12, 13, 14, 15,
t(1)	6	0, 1, 2, 3, 4, 5, 6, **7,** 8, 9, 10, 11, 12, 13, 14, 15,
	7	0, 1, 2, 3, 4, 5, 6, **7,** 8, 9, 10, 11, 12, 13, 14, 15,
	8	0, 1, 2, 3, 4, 5, 6, **4,** 8, 9, 10, 11, 12, 13, 14, 15,
	9	0, 1, 2, 3, 4, 5, 6, **4,** 8, 9, 10, 11, 12, 13, 14, 15,
t(2)	10	0, 1, 2, 3, 4, 5, 6, **4,** 8, 9, 10, 11, 12, 13, 14, 15,
	11	0, 1, 2, 3, 4, 5, 6, **4,** 8, 9, 10, 11, 12, 13, 14, 15,
	12	0, 1, 2, 3, 4, 5, 6, **4,** 8, 9, 10, 11, 12, 13, 14, 15,
	13	0, 1, 2, 3, 4, 5, 6, **4,** 8, 9, 10, 11, 12, 13, 14, 15,
t(3)	14	0, 1, 2, 3, 4, 5, 6, **4,** 8, 9, 10, 11, 12, 13, 14, 15,
	15	0, 1, 2, 3, 4, 5, 6, **4,** 8, 9, 10, 11, 12, 13, 14, 15,

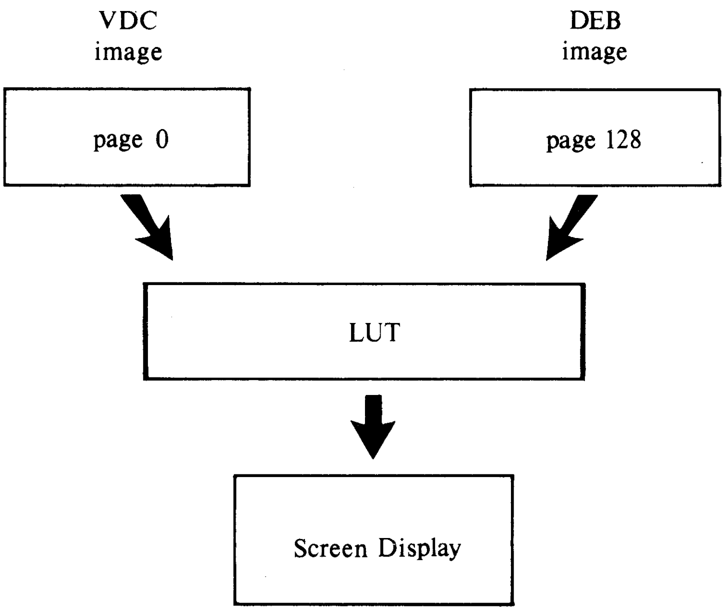
LUT for Blinking Between White and Red in Palette Position 7

Overlay Modes LUT Programming

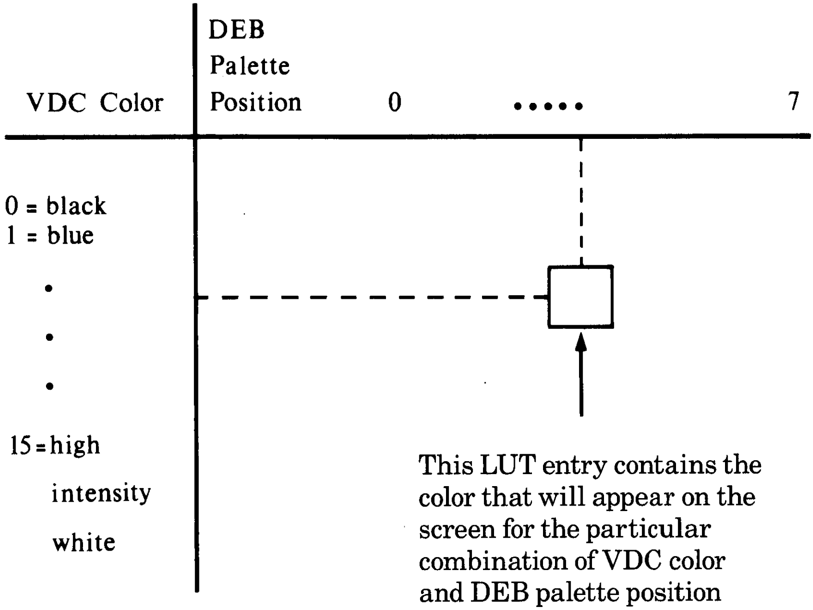
Overlay Modes LUT Programming

When the LUT is used in the overlay modes it can be viewed as a two-dimensional array with 8 columns and 32 rows. The column values are DEB palette positions. The row values are VDC color values.

In overlay modes, there are 2 separately controlled images: the VDC image and the DEB image. The 2 images are combined on the display screen. Each pixel on the screen has 2 values associated with it: the VDC color and the DEB palette position. The LUT is used to resolve contention between the 2 values associated with each pixel.



The LUT for overlay modes looks like this:

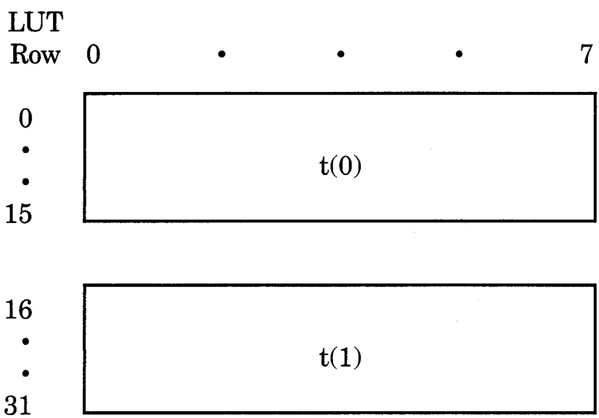


As in the 16-color graphics modes, the locations in the LUT are numbered consecutively from left to right and top to bottom. For example, location 17 corresponds to Row 2, Palette Position 0.

In the overlay modes, as in the 16-color graphics mode, the LUT is divided into time states that control blinking effects. However, in the overlay modes, the LUT is only divided into two time states. Half of the LUT determines what is being displayed at any time. The top half is used for the first ½ of each second and the bottom half is used for the second ½ of each second.

Using the overlay modes, you create blinking by making the values in the top half of the table different from the corresponding values in the bottom half of the table.

DEB Palette Position



The following example shows the LUT values for standard Palette 2 of an overlay mode. The LUT is programmed so that the DEB image is displayed only if the VDC color is 0 (black). If the VDC requests any other color, then that color is displayed no matter what the DEB requests. This has the effect of overlaying the VDC image "on top" of the DEB image.

DEB Palette Position

	VDC Color Values	0	1	2	3	4	5	6	7
	0	**0,**	**1,**	**2,**	**3,**	**4,**	**5,**	**6,**	**7,**
	1	1,	1,	1,	1,	1,	1,	1,	1,
	2	2,	2,	2,	2,	2,	2,	2,	2,
	3	3,	3,	3,	3,	3,	3,	3,	3,
t(0)	4	4,	4,	4,	4,	4,	4,	4,	4,
	5	5,	5,	5,	5,	5,	5,	5,	5,
	6	6,	6,	6,	6,	6,	6,	6,	6,
	7	7,	7,	7,	7,	7,	7,	7,	7,
	8	8,	8,	8,	8,	8,	8,	8,	8,
	9	9,	9,	9,	9,	9,	9,	9,	9,
	10	10,	10,	10,	10,	10,	10,	10,	10,
	11	11,	11,	11,	11,	11,	11,	11,	11,
	12	12,	12,	12,	12,	12,	12,	12,	12,
	13	13,	13,	13,	13,	13,	13,	13,	13,
	14	14,	14,	14,	14,	14,	14,	14,	14,
	15	15,	15,	15,	15,	15,	15,	15,	15,

DEB Palette Position

	VDC Color Values	0	1	2	3	4	5	6	7
	0	**0,**	**1,**	**2,**	**3,**	**4,**	**5,**	**6,**	**7,**
	1	1,	1,	1,	1,	1,	1,	1,	1,
	2	2,	2,	2,	2,	2,	2,	2,	2,
	3	3,	3,	3,	3,	3,	3,	3,	3,
t(1)	4	4,	4,	4,	4,	4,	4,	4,	4,
	5	5,	5,	5,	5,	5,	5,	5,	5,
	6	6,	6,	6,	6,	6,	6,	6,	6,
	7	7,	7,	7,	7,	7,	7,	7,	7,
	8	8,	8,	8,	8,	8,	8,	8,	8,
	9	9,	9,	9,	9,	9,	9,	9,	9,
	10	10,	10,	10,	10,	10,	10,	10,	10,
	11	11,	11,	11,	11,	11,	11,	11,	11,
	12	12,	12,	12,	12,	12,	12,	12,	12,
	13	13,	13,	13,	13,	13,	13,	13,	13,
	14	14,	14,	14,	14,	14,	14,	14,	14,
	15	15,	15,	15,	15,	15,	15,	15,	15,

In this example, the standard Palette 2 is modified so that position 2 is a blinking between blue (color 1) and red (color 4).

DEB Palette Position

	VDC Color Values	0	1	2	3	4	5	6	7
	0	0,	1,	1,	3,	4,	5,	6,	7,
	1	1,	1,	1,	1,	1,	1,	1,	1,
	2	2,	2,	2,	2,	2,	2,	2,	2,
	3	3,	3,	3,	3,	3,	3,	3,	3,
t(0)	4	4,	4,	4,	4,	4,	4,	4,	4,
	5	5,	5,	5,	5,	5,	5,	5,	5,
	6	6,	6,	6,	6,	6,	6,	6,	6,
	7	7,	7,	7,	7,	7,	7,	7,	7,
	8	8,	8,	8,	8,	8,	8,	8,	8,
	9	9,	9,	9,	9,	9,	9,	9,	9,
	10	10,	10,	10,	10,	10,	10,	10,	10,
	11	11,	11,	11,	11,	11,	11,	11,	11,
	12	12,	12,	12,	12,	12,	12,	12,	12,
	13	13,	13,	13,	13,	13,	13,	13,	13,
	14	14,	14,	14,	14,	14,	14,	14,	14,
	15	15,	15,	15,	15,	15,	15,	15,	15,

DEB Palette Position

	VDC Color Values	0	1	2	3	4	5	6	7
	0	0,	1,	**4,**	3,	4,	5,	6,	7,
	1	1,	1,	1,	1,	1,	1,	1,	1,
	2	2,	2,	2,	2,	2,	2,	2,	2,
	3	3,	3,	3,	3,	3,	3,	3,	3,
t(1)	4	4,	4,	4,	4,	4,	4,	4,	4,
	5	5,	5,	5,	5,	5,	5,	5,	5,
	6	6,	6,	6,	6,	6,	6,	6,	6,
	7	7,	7,	7,	7,	7,	7,	7,	7,
	8	8,	8,	8,	8,	8,	8,	8,	8,
	9	9,	9,	9,	9,	9,	9,	9,	9,
	10	10,	10,	10,	10,	10,	10,	10,	10,
	11	11,	11,	11,	11,	11,	11,	11,	11,
	12	12,	12,	12,	12,	12,	12,	12,	12,
	13	13,	13,	13,	13,	13,	13,	13,	13,
	14	14,	14,	14,	14,	14,	14,	14,	14,
	15	15,	15,	15,	15,	15,	15,	15,	15,

In this example, values in the LUT cause the DEB's output to take precedence over the VDC's output. The VDC's output is only displayed when you specify DEB palette position 0 in a graphics statement.

DEB Palette Positions

	VDC Color Values	0 1 2 3 4 5 6 7
	0	**0,** 1, 2, 3, 4, 5, 6, 7,
	1	**1,** 1, 2, 3, 4, 5, 6, 7,
	2	**2,** 1, 2, 3, 4, 5, 6, 7,
	3	**3,** 1, 2, 3, 4, 5, 6, 7,
	4	**4,** 1, 2, 3, 4, 5, 6, 7,
	5	**5,** 1, 2, 3, 4, 5, 6, 7,
	6	**6,** 1, 2, 3, 4, 5, 6, 7,
t(0)	7	**7,** 1, 2, 3, 4, 5, 6, 7,
	8	**8,** 1, 2, 3, 4, 5, 6, 7,
	9	**9,** 1, 2, 3, 4, 5, 6, 7,
	10	**10,** 1, 2, 3, 4, 5, 6, 7,
	11	**11,** 1, 2, 3, 4, 5, 6, 7,
	12	**12,** 1, 2, 3, 4, 5, 6, 7,
	13	**13,** 1, 2, 3, 4, 5, 6, 7,
	14	**14,** 1, 2, 3, 4, 5, 6, 7,
	15	**15,** 1, 2, 3, 4, 5, 6, 7,

DEB Palette Positions

	VDC Color Values	0 1 2 3 4 5 6 7
	1	**0,** 1, 2, 3, 4, 5, 6, 7,
	1	**0,** 1, 2, 3, 4, 5, 6, 7,
	2	**2,** 1, 2, 3, 4, 5, 6, 7,
	3	**3,** 1, 2, 3, 4, 5, 6, 7,
	4	**4,** 1, 2, 3, 4, 5, 6, 7,
t(1)	5	**5,** 1, 2, 3, 4, 5, 6, 7,
	6	**6,** 1, 2, 3, 4, 5, 6, 7,
	7	**7,** 1, 2, 3, 4, 5, 6, 7,
	8	**8,** 1, 2, 3, 4, 5, 6, 7,
	9	**9,** 1, 2, 3, 4, 5, 6, 7,
	10	**10,** 1, 2, 3, 4, 5, 6, 7,
	11	**11,** 1, 2, 3, 4, 5, 6, 7,
	12	**12,** 1, 2, 3, 4, 5, 6, 7,
	13	**13,** 1, 2, 3, 4, 5, 6, 7,
	14	**14,** 1, 2, 3, 4, 5, 6, 7,
	15	**15,** 1, 2, 3, 4, 5, 6, 7,

The following LUT entirely blocks out VDC output:

	VDC Color Values	DEB Palette Positions 0 1 2 3 4 5 6 7
	0	0, 1, 2, 3, 4, 5, 6, 7,
	1	0, 1, 2, 3, 4, 5, 6, 7,
	2	0, 1, 2, 3, 4, 5, 6, 7,
	3	0, 1, 2, 3, 4, 5, 6, 7,
	4	0, 1, 2, 3, 4, 5, 6, 7,
t(0)	5	0, 1, 2, 3, 4, 5, 6, 7,
	6	0, 1, 2, 3, 4, 5, 6, 7,
	7	0, 1, 2, 3, 4, 5, 6, 7,
	8	0, 1, 2, 3, 4, 5, 6, 7,
	9	0, 1, 2, 3, 4, 5, 6, 7,
	10	0, 1, 2, 3, 4, 5, 6, 7,
	11	0, 1, 2, 3, 4, 5, 6, 7,
	12	0, 1, 2, 3, 4, 5, 6, 7,
	13	0, 1, 2, 3, 4, 5, 6, 7,
	14	0, 1, 2, 3, 4, 5, 6, 7,
	15	0, 1, 2, 3, 4, 5, 6, 7,

DEB Palette Positions

	VDC Color Values	0	1	2	3	4	5	6	7
	0	0,	1,	2,	3,	4,	5,	6,	7,
	1	0,	1,	2,	3,	4,	5,	6,	7,
	2	0,	1,	2,	3,	4,	5,	6,	7,
	3	0,	1,	2,	3,	4,	5,	6,	7,
	4	0,	1,	2,	3,	4,	5,	6,	7,
t(1)	5	0,	1,	2,	3,	4,	5,	6,	7,
	6	0,	1,	2,	3,	4,	5,	6,	7,
	7	0,	1,	2,	3,	4,	5,	6,	7,
	8	0,	1,	2,	3,	4,	5,	6,	7,
	9	0,	1,	2,	3,	4,	5,	6,	7,
	10	0,	1,	2,	3,	4,	5,	6,	7,
	11	0,	1,	2,	3,	4,	5,	6,	7,
	12	0,	1,	2,	3,	4,	5,	6,	7,
	13	0,	1,	2,	3,	4,	5,	6,	7,
	14	0,	1,	2,	3,	4,	5,	6,	7,
	15	0,	1,	2,	3,	4,	5,	6,	7,

Programming the Bit Planes

Introduction

Once you have learned to program the LUT directly using the Set Color Palette command, you can make further use of the LUT's capabilities by programming the VDC and DEB video memory directly.

By directly programming the video memory of the VDC and DEB boards, you can increase the graphics display speed. The values you load into the video memory planes determine how the LUT is accessed. This section assumes that you have read and understood how to program the LUT directly.

In the 16-color graphics modes, the device driver combines the 3 bit planes of the DEB with one bit plane from the VDC to create the four bit planes necessary for 16-color graphics.

In the overlay modes, the device driver uses the 3 DEB bit planes for 8-color graphics output and uses the VDC board separately for either text or graphics output.

A0000:0	DEB Bitplane 0
A8000:0	DEB Bitplane 1
B000:0	DEB Bitplane 2
B8000:0	VDC Bitplane

Memory Map

LUT Addressing

A LUT address is an 8-bit value that points to one of the 256 locations within the LUT. The method of address formation depends on the current video mode.

For transparent and disabled modes, LUT addressing is irrelevant. In the transparent mode, VDC color values bypass the LUT processing and go directly to the monitor output. In the disabled mode, all output from the LUT is forced to the value of zero.

For the 16-color graphics and overlay modes, the LUT address is composed of bits from the DEB video bit planes, the VDC's video output, and DEB timing bits.

Timing Bits

The timing bits are called BLINK1, BLINK2, PAT1, and PAT2. BLINK1 and BLINK2 effect blinking; PAT1 and PAT2 effect patterning (dithering).

All of the timing bits are applicable in the 16-color graphics mode; only BLINK1 is part of the address formation in the overlay mode. Therefore, you have fewer options for blinking and no ability to dither in the overlay mode.

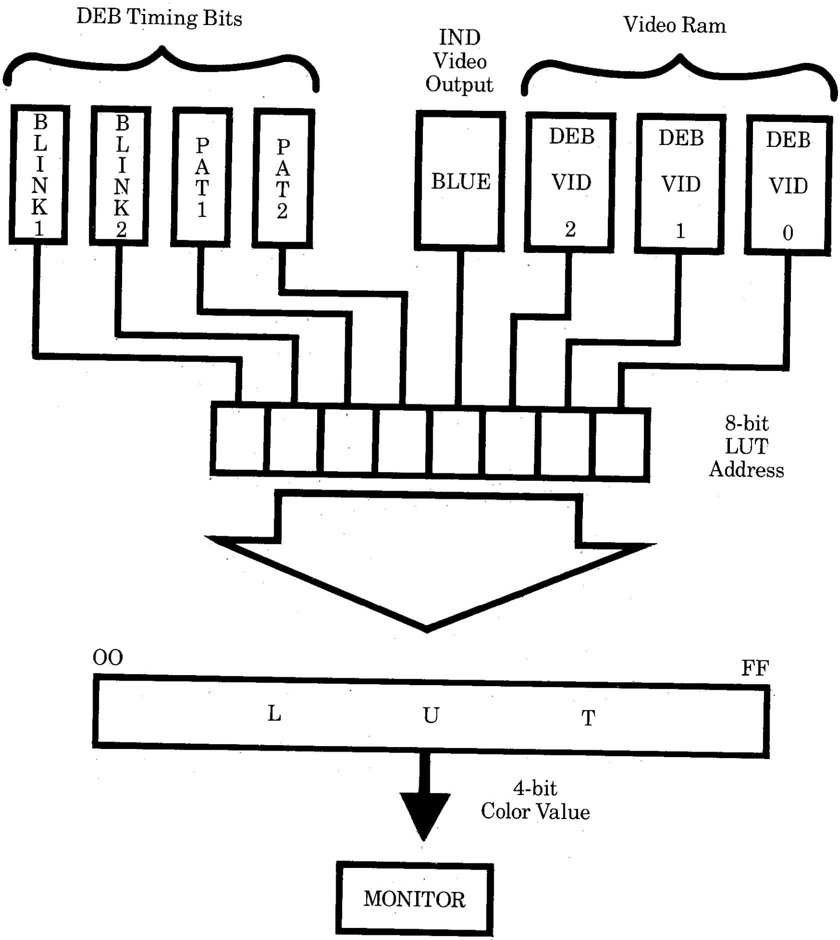
The operation of the timing bits is very fundamental to creation of special effects. The bits always cycle on and off, each at a different rate. BLINK1 cycles on and off each 1/4 second. BLINK2 cycles on and off each 1/8 second.

PAT1 and PAT2 cycle on and off so fast that the eye cannot perceive a blink (PAT1 is the fastest). A dithered color is really 2-4 separate colors that are changing so rapidly that the eye perceives them as one solid color.

PAT1 changes value at the same rate that the monitor's cathode ray moves from one pixel to the next. PAT1's effect on LUT addressing is that it switches the address by 16 LUT entries — in the previous table, between pairs of rows. PAT2 changes value at the same rate that the cathode ray changes scanlines — in the previous table, between one pair of rows and the next pair of rows.

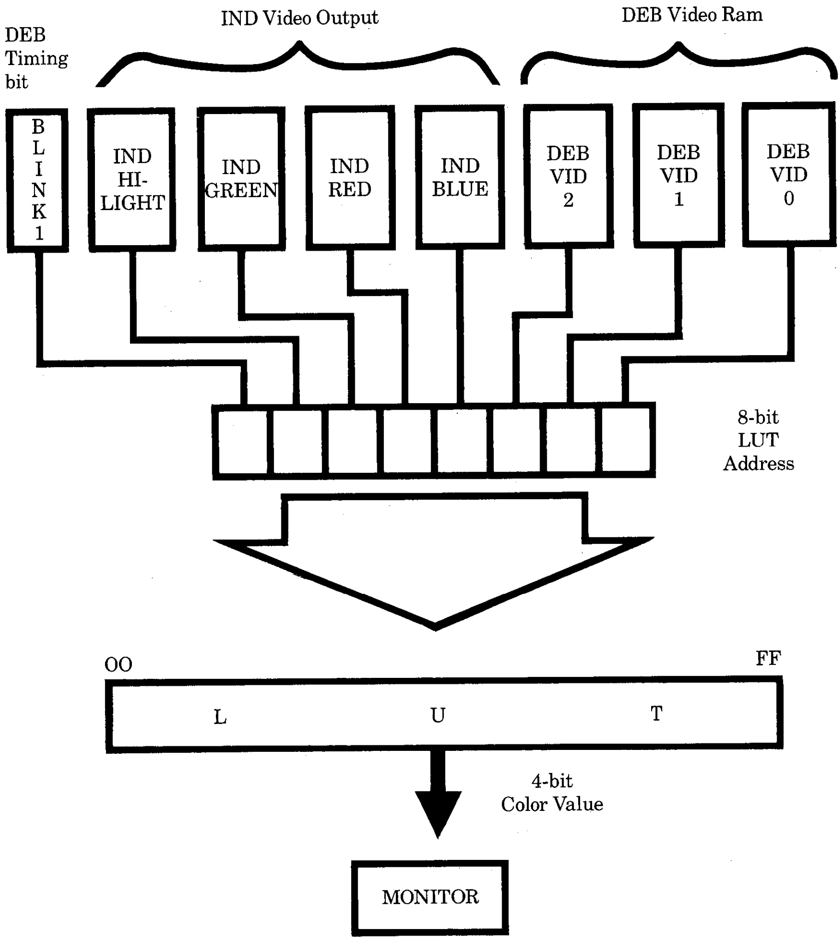
PAT2	PAT1	Portion of LUT
0	0	1st 16 entries of each quarter
0	1	2nd 16 entries of each quarter
1	0	3rd 16 entries of each quarter
1	1	4th 16 entries of each quarter

1. 16-color Graphics Mode



To output a color to the monitor, the DEB concatenates the DEB timing bits BLINK1, BLINK2, PAT1, PAT2, the BLUE output bit from the VDC, and a bit from corresponding locations on each of the three DEB bit planes.

2. Overlay Mode

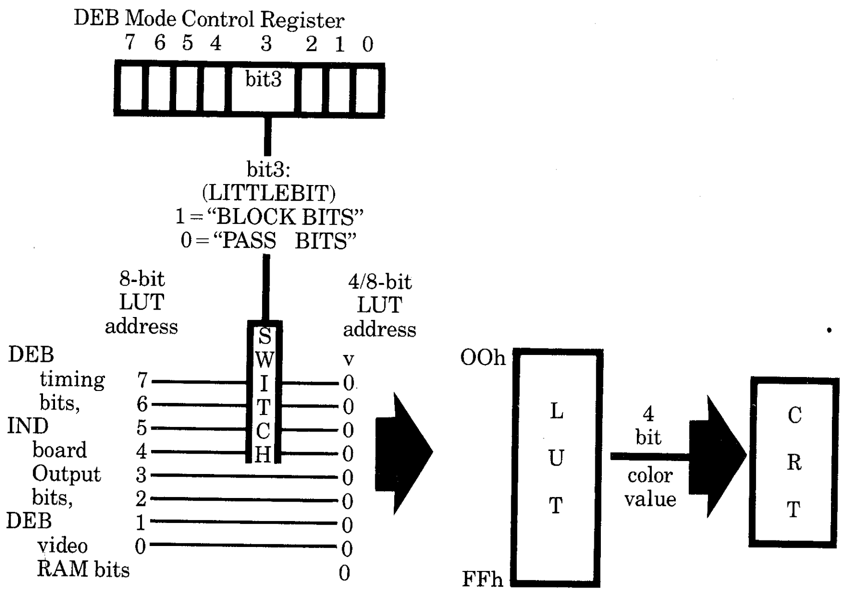


To output a color to the monitor, the DEB concatenates the following bits: BLINK1, the HILIGHT, GREEN, RED, and BLUE output bits from the VDC, and a bit from corresponding locations on each of the three DEB bit planes.

Short LUT Addresses

The DEB supports a method for you to access only the first sixteen LUT locations. This lets you use normal 16-color graphics without needing to manage all of the 256 LUT locations. You invoke this short addressing mode by a setting bit 2 in AL in the "Set Color Palette" command.

3. <u>Short LUT Addresses</u>



4. Modes, Address Formation, & DEB Mode Control Register