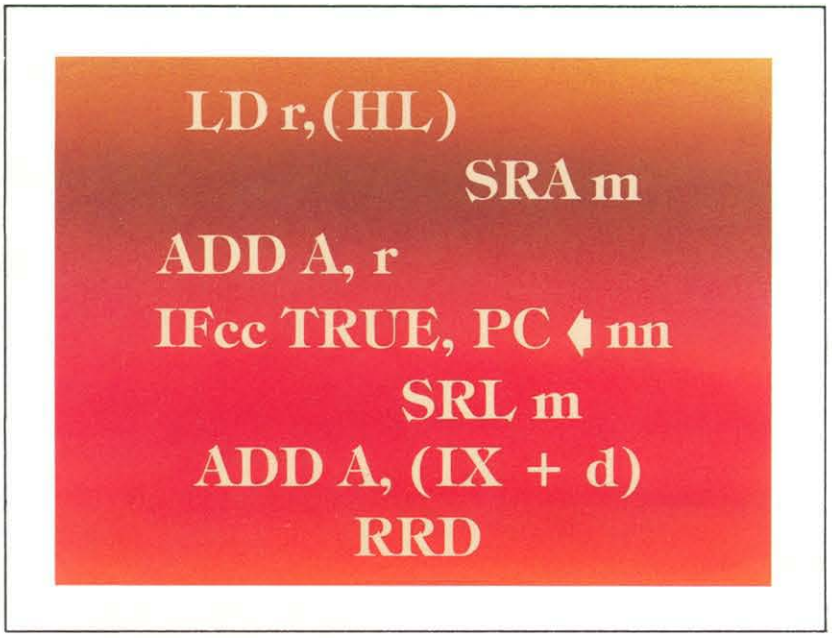
Cat. No. 26-2012

Radio Shack®

ASSEMBLY LANGUAGE DEVELOPMENT SYSTEM

TRS-80® Model III/4



Custom Manufactured in USA by RADIO SHACK, A Division of TANDY CORPORATION

TERMS AND CONDITIONS OF SALE AND LICENSE OF RADIO SHACK COMPUTER EQUIPMENT AND SOFTWARE PURCHASED FROM A RADIO SHACK COMPANY-OWNED COMPUTER CENTER, RETAIL STORE OR FROM A RADIO SHACK FRANCHISEE OR DEALER AT ITS AUTHORIZED LOCATION

LIMITED WARRANTY

I. CUSTOMER OBLIGATIONS

A. CUSTOMER assumes full responsibility that this Radio Shack computer hardware purchased (the "Equipment"), and any copies of Radio Shack software included with the Equipment or licensed separately (the "Software") meets the specifications, capacity, capabilities, versatility, and other requirements of CUSTOMER.

B. CUSTOMER assumes full responsibility for the condition and effectiveness of the operating environment in which the Equipment and Software are to function, and for its installation.

II. RADIO SHACK LIMITED WARRANTIES AND CONDITIONS OF SALE

A. For a period of ninety (90) calendar days from the date of the Radio Shack sales document received upon purchase of the Equipment, RADIO SHACK warrants to the original CUSTOMER that the Equipment and the medium upon which the Software is stored is free from manufacturing defects. THIS WARRANTY IS ONLY APPLICABLE TO PURCHASES OF RADIO SHACK EQUIPMENT BY THE ORIGINAL CUSTOMER FROM RADIO SHACK COMPANY-OWNED COMPUTER CENTERS, RETAIL STORES AND FROM RADIO SHACK FRANCHISEES AND DEALERS AT ITS AUTHORIZED LOCATION. The warranty is void if the Equipment's case or cabinet has been opened, or if the Equipment or Software has been subjected to improper or abnormal use. If a manufacturing defect is discovered during the stated warranty period, the defective Equipment must be returned to a Radio Shack Computer Center, a Radio Shack retail store, participating Radio Shack franchisee or Radio Shack dealer for repair, along with a copy of the sales document or lease agreement. The original CUSTOMER'S sole and exclusive remedy in the event of a defect is limited to the correction of the defect by repair, replacement, or refund of the purchase price, at RADIO SHACK'S election and sole expense. RADIO SHACK has no obligation to replace or repair expendable items.

B. RADIO SHACK makes no warranty as to the design, capability, capacity, or suitability for use of the Software, except as provided in this paragraph. Software is licensed on an "AS IS" basis, without warranty. The original CUSTOMER'S exclusive remedy, in the event of a Software manufacturing defect, is its repair or replacement within thirty (30) calendar days of the date of the Radio Shack sales document received upon license of the Software. The defective Software shall be returned to a Radio Shack Computer Center, a Radio Shack retail store, participating Radio Shack franchisee or Radio Shack dealer along with the sales document.

C. Except as provided herein no employee, agent, franchisee, dealer or other person is authorized to give any warranties of any nature on behalf of RADIO SHACK.

D. Except as provided herein, **RADIO SHACK MAKES NO WARRANTIES, INCLUDING WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.**

E. Some states do not allow limitations on how long an implied warranty lasts, so the above limitation(s) may not apply to CUSTOMER.

III. LIMITATION OF LIABILITY

A. EXCEPT AS PROVIDED HEREIN, RADIO SHACK SHALL HAVE NO LIABILITY OR RESPONSIBILITY TO CUSTOMER OR ANY OTHER PERSON OR ENTITY WITH RESPECT TO ANY LIABILITY, LOSS OR DAMAGE CAUSED OR ALLEGED TO BE CAUSED DIRECTLY OR INDIRECTLY BY "EQUIPMENT" OR "SOFTWARE" SOLD, LEASED, LICENSED OR FURNISHED BY RADIO SHACK, INCLUDING, BUT NOT LIMITED TO, ANY INTERRUPTION OF SERVICE, LOSS OF BUSINESS OR ANTICIPATORY PROFITS OR CONSEQUENTIAL DAMAGES RESULTING FROM THE USE OR OPERATION OF THE "EQUIPMENT" OR "SOFTWARE". IN NO EVENT SHALL RADIO SHACK BE LIABLE FOR LOSS OF PROFITS, OR ANY INDIRECT, SPECIAL, OR CONSEQUENTIAL DAMAGES ARISING OUT OF ANY BREACH OF THIS WARRANTY OR IN ANY MANNER ARISING OUT OF OR CONNECTED WITH THE SALE, LEASE, LICENSE, USE OR ANTICIPATED USE OF THE "EQUIPMENT" OR "SOFTWARE".

NOTWITHSTANDING THE ABOVE LIMITATIONS AND WARRANTIES, RADIO SHACK'S LIABILITY HEREUNDER FOR DAMAGES INCURRED BY CUSTOMER OR OTHERS SHALL NOT EXCEED THE AMOUNT PAID BY CUSTOMER FOR THE PARTICULAR "EQUIPMENT" OR "SOFTWARE" INVOLVED.

B. RADIO SHACK shall not be liable for any damages caused by delay in delivering or furnishing Equipment and or Software.

C. No action arising out of any claimed breach of this Warranty or transactions under this Warranty may be brought more than two (2) years after the cause of action has accrued or more than four (4) years after the date of the Radio Shack sales document for the Equipment or Software, whichever first occurs.

D. Some states do not allow the limitation or exclusion of incidental or consequential damages, so the above limitation(s) or exclusion(s) may not apply to CUSTOMER.

IV. RADIO SHACK SOFTWARE LICENSE

RADIO SHACK grants to CUSTOMER a non-exclusive, paid-up license to use the RADIO SHACK Software on **one** computer, subject to the following provisions:

A. Except as otherwise provided in this Software License, applicable copyright laws shall apply to the Software.

B. Title to the medium on which the Software is recorded (cassette and/or diskette) or stored (ROM) is transferred to CUSTOMER, but not title to the Software.

C. CUSTOMER may use Software on one host computer and access that Software through one or more terminals if the Software permits this function.

D. CUSTOMER shall not use, make, manufacture, or reproduce copies of Software except for use on **one** computer and as is specifically provided in this Software License. Customer is expressly prohibited from disassembling the Software.

E. CUSTOMER is permitted to make additional copies of the Software **only** for backup or archival purposes or if additional copies are required in the operation of **one** computer with the Software, but only to the extent the Software allows a backup copy to be made. However, for TRSDOS Software, CUSTOMER is permitted to make a limited number of additional copies for CUSTOMER'S own use.

F. CUSTOMER may resell or distribute unmodified copies of the Software provided CUSTOMER has purchased one copy of the Software for each one sold or distributed. The provisions of this Software License shall also be applicable to third parties receiving copies of the Software from CUSTOMER.

G. All copyright notices shall be retained on all copies of the Software.

V. APPLICABILITY OF WARRANTY

A. The terms and conditions of this Warranty are applicable as between RADIO SHACK and CUSTOMER to either a sale of the Equipment and or Software License to CUSTOMER or to a transaction whereby RADIO SHACK sells or conveys such Equipment to a third party for lease to CUSTOMER.

B. The limitations of liability and Warranty provisions herein shall inure to the benefit of RADIO SHACK, the author, owner and/or licensor of the Software and any manufacturer of the Equipment sold by RADIO SHACK.

VI. STATE LAW RIGHTS

The warranties granted herein give the **original** CUSTOMER specific legal rights, and the **original** CUSTOMER may have other rights which vary from state to state.

ALDS
Assembly Language
Development System

TRSDOS® Version 6 Operating System: Copyright 1983 Logical Systems.
All Rights Reserved. Licensed to Tandy Corporation.

ALEDIT Software: Copyright 1982, 1983 Tandy Corporation. All Rights Reserved.

ALASM Software: Copyright 1982, 1983 Tandy Corporation. All Rights Reserved.

ALBUG Software: Copyright 1982, 1983 Tandy Corporation. All Rights Reserved.

ALLINK Software: Copyright 1982, 1983 Tandy Corporation. All Rights Reserved.

ALTRAN Software: Copyright 1982, 1983 Tandy Corporation. All Rights Reserved.

TRS-80® Assembly Language Development System Manual: Copyright 1982, 1983 Tandy Corporation. All Rights Reserved.

Reproduction or use without express written permission from Tandy Corporation of any portion of this manual is prohibited. While reasonable efforts have been taken in the preparation of this manual to assure its accuracy, Tandy Corporation assumes no liability resulting from any errors or omissions in this manual, or from the use of the information obtained herein.

TRSDOS is a registered trademark of Tandy Corporation.

10 9 8 7 6 5 4 3 2 1

To Our Customers,

This Assembly Language Development System (ALDS) is a powerful tool for developing Z80 programs for the TRS-80 Models III and 4.

It contains these five systems:

ALEDIT, a Text Editor, for writing and editing source programs.

ALASM, an Assembler for converting source programs to Z80 object code. The Assembler contains more than:

- 50 powerful directives. Among many features, they allow you to build relocatable program sections, macro sections, index sections; generate a length byte for text storage; and control the assembly listing format.
- 30 arithmetic, logical and relational operators.
- 10 "extended" Z80 mnemonics, which expand into an entire group of Z80 mnemonics.

ALLINK, a Linker, for linking relocatable program sections into absolute object files.

ALBUG, a Debugger, for debugging a program in memory or altering a file on disk. ALBUG is comprised of six program files: ALBUG, ALBUG/SYS, ALBUG/OVL, ALBUGX, ALBUGRES/REL and ALBUG/RES.

ALTRAN, a File Transfer System, for transferring a file between the Models I, II, III, 4, 12 and 16.

Note: Models I, II, 12 and 16 require the Model II ALDS package.

About This Manual

This manual assumes you already know Z80 assembly language programming and have used an editor/assembler. It contains three sections:

Section I, Using ALDS, begins with a sample session which shows how to create a modular program for the Models III and 4 using all five systems. Following this session are reference chapters on each system.

Section II, ALDS Assembly Language, references the source language acceptable to the ALDS Assembler. *Chapter 7* outlines the syntax for writing source lines. The remaining chapters reference all the directives, Z80 mnemonics, and extended Z80 mnemonics available.

Section III, Error Messages, lists the error messages that may be generated by the ALDS programs.

If you are new to Z80 assembly language programming, we suggest you read:

More TRS-80 Assembly Language Programming by William Barden, Jr. (Radio Shack Catalog Number 62-2075)

Note: Before going any further, please make a backup of your ALDS diskette. See your system's owners manual for instructions on making backups.

Notation Key

The manual uses these notational conventions:

`Dot Matrix`	to represent what you will see on the screen or should type.
(KEY)	to represent a specific key you should press.
italics	to represent a value you should specify.
H	to represent a hexadecimal number. (For example, 4233H represents the hexadecimal number 4233.)
$	to represent the current value of the Assembler's location counter. (This is actually a convention of the Assembler.)
filespec	to represent a valid TRSDOS file specification. (See your TRSDOS manual for a definition of filespec.)

Section I/
Using ALDS

USAGE

Contents

Section I/ Using ALDS

Chapter 1/ Sample Session 3
Chapter 2/ The ALDS Editor 11
Chapter 3/ The ALDS Assembler 23
Chapter 4/ The ALDS Debugger 29
Chapter 5/ The ALDS Linker 43
Chapter 6/ The ALDS File Transfer System 47

Section II/ ALDS Assembly Language

Chapter 7/ Assembly Language Syntax 61
Chapter 8/ Directives 69
Chapter 9/ Z80 Mnemonics 115
Chapter 10/ Extended Z80 Mnemonics 303

Section III/ Error Messages

Error Messages 325

Appendices

Appendix A/ Undocumented Z80 Instructions 333
Appendix B/ ALDS Object Code Format 338
Appendix C/ Numeric List of Z80 Instruction Set 341
Appendix D/ Alphabetic List of Z80 Instruction Set 347
Appendix E/ Z80 Hardware 353

Tables

Table 1/ ALEDIT Command Mode Keys 13
Table 2/ ALEDIT Editor Commands 14
Table 3/ ALEDIT Insert Control functions 20
Table 4/ ALEDIT Insert Mode Special Keys 21
Table 5/ ALEDIT Line Edit Mode Subcommands 21
Table 6/ ALEDIT Line Edit Mode Special Keys 22
Table 7/ ALASM Switches 24
Table 8/ Debugger Commands 34
Table 9/ Baud Rate Change Table 48
Table 10/ Operators 64
Table 11/ Complex Expressions Allowing Relocatable or External Symbols 66

Chapter 1/ Sample Session

This manual is not a tutorial. To learn Assembly Language, see your computer dealer for information on helpful books.

This chapter is for those of you who want to try a session using the entire ALDS package. It demonstrates how to link separate program sections for the Models III and 4.

This session is for demonstration only. To find out how and why each system works the way it does, you will need to refer to specific chapters in this manual.

Note for Model 4: If at any time during this procedure you receive the message, "File Already Open," type:

`RESET` *filename*

This command closes the open file.

Creating a Source File

In this session, you need to create five source program files. To do this, use the ALDS Editor. In the TRSDOS Ready mode, type:

`ALEDIT` (ENTER)

this loads the ALDS Editor. After it displays its heading, type:

`I`

the insert command (Do not press (ENTER)). The Editor clears the screen and prints NONAME/SRC in the upper right-hand corner. You are now in the insert mode and can insert the first source program.

1. Main Program

To insert the first program, named MAIN, type the following commands (press ⊕ between columns; press (ENTER) at the end of each line.):

```
MAIN        PSECT
            PUBLIC      BEGIN
            EXTERN      PRINT,TRSDOS
BEGIN       LD          HL,MSG1
            CALL        PRINT           ;Print Line MSG1
            LD          HL,MSG2
            CALL        PRINT           ;Print Line MSG2
            JP          TRSDOS
```

```
MSG1        DEFT        'YOU WILL BE ABLE TO LINK THIS'
            DEFB        ØDH
MSG2        DEFT        'AS EITHER A MODEL III OR 4 PROGRAM'
            DEFB        ØDH
            END         BEGIN
```

When you are finished press (BREAK). This puts you in the Editor command mode. If you made mistakes, you can use the Editor commands to edit the program. They are all listed in *Chapter 2*, The ALDS Editor.

After pressing (BREAK), save this source program on disk by typing this Editor command:

`W MAIN (ENTER)`

this saves the program as a source file named MAIN/SRC. (The Editor changes the top right-hand corner display to MAIN/SRC.) Clear the edit buffer by typing:

`K (ENTER)`

the kill command and answer Y (ENTER) to the prompt. The screen will then clear.

Now repeat the same procedures for inserting and saving MOD4, MODIII, PROG4, and PROGIII. (If you have a Model 4, insert all of these programs on your Model 4 — even MODIII and PROGIII. Otherwise, insert all of these programs on your Model III.)

2. MOD4 Program

```
MOD4        PSECT       ;Model 4 Print Routines
            PUBLIC      PRINT,TRSDOS
@DSPLY      EQU         10
@EXIT       EQU         22
PRINT       INC         HL
            SVC         @DSPLY      ;Display Line
            RET
TRSDOS      LD          HL,Ø
            JP          @EXIT       ;Exit
            END
```

3. MODIII Program

```
MODIII      PSECT       ;Model III Print Routines
            PUBLIC      PRINT,TRSDOS
VDLINE      EQU         Ø21BH
JP2DOS      EQU         402DH
PRINT       INC         HL
            CALL        VDLINE      ;Display Line
            RET
TRSDOS      JP          JP2DOS      ;Exit
            END
```

4. PROG4 Program

```
PROG4       PSECT       ;Model 4 Linking Program
            EXTERN      BEGIN
START       JP          BEGIN
            LINK        'MAIN/REL'      ;Links Main Program
            LINK        'MOD4/REL'      ;Links Print Routines
            END         START
```

5. PROGIII Program

```
PROGIII     PSECT       ;Model III Linking Program
            EXTERN      BEGIN
START       JP          BEGIN
            LINK        'MAIN/REL'      ;Links Main Program
            LINK        'MODIII/REL'    ;Links Print Routines
            END         START
```

When you have finished inserting all five source files, exit the Editor by typing:

`Q` (ENTER)

which returns you to TRSDOS Ready.

Assembling a File

You should now have stored five source files:

MAIN/SRC
MOD4/SRC
MODIII/SRC
PROG4/SRC
PROGIII/SRC

To see that they are all on your diskette, check the disk directory by typing DIR (ENTER).

These files contain three types of instructions:

- Z80 mnemonics (LD, CALL, INC, and RET), which the Assembler converts into Z80 object code. *Chapter 9* describes Z80 mnemonics.
- An extended mnemonic (SVC), which the Assembler converts into a group of Z80 instructions. *Chapter 10* describes extended mnemonics.
- Directives (PSECT, EXTERN, DEFT, PUBLIC, EQU, LINK and END), which are instructions to the Assembler or the Linker. *Chapter 8* describes directives.

To assemble the source files, use the ALDS Assembler (ALASM). In the TRSDOS Ready mode, type:

`ALASM MAIN/SRC MAIN/REL` (ENTER)

The assembler processes the source file MAIN/SRC into an object file named MAIN/REL. If it displays any errors, edit or re-insert MAIN/SRC and re-assemble it. (An explanation of the Assembler error messages is in the Error Messages Section of this manual.)

You can assemble the other source files in the same way.

Note: You can omit the /SRC and /REL extensions. The Assembler knows to append them:

```
ALASM MOD4 MOD4 (ENTER)
ALASM MODIII MODIII (ENTER)
ALASM PROG4 PROG4 (ENTER)
ALASM PROGIII PROGIII (ENTER)
```

When finished, the Assembler produces these object files:

```
MAIN/REL
MOD4/REL
MODIII/REL
PROG4/REL
PROGIII/REL
```

The extension REL means that the files are relocatable. That is, they do not have absolute load and execution addresses. Because of this, they cannot be loaded and executed in their present form.

The Assembler converts them into relocatable rather than absolute files because of the PSECT directives. See *Chapter 8* for more information on the directives. See *Chapter 3* for information on operating the Assembler.

Linking a Relocatable File

Two of the relocatable files created by the Assembler are:

PROG4/REL
PROGIII/REL

which consist solely of LINK directives. They are for the ALDS Linker to process. Type:

```
ALLINK PROG4/REL PROG4 $=5200 (ENTER)
```

This causes the Linker to:

(1) process the LINK directives, LINKing MAIN/REL and MOD4/REL to PROG4/REL.

(2) assign absolute addresses beginning with 5200H to PROG4/REL.

(3) save the resulting absolute object code PROG4.

You can link PROGIII/REL in the same way. (Notice that you can optionally omit the /REL extension, since the Assembler will automatically append it.) Type:

```
ALLINK PROGIII PROGIII $=5200
```

Using the same processes as above, the Linker creates PROGIII, an absolute object file, composed of MAIN/REL and MODIII/REL.

Chapter 5, The Linker, discusses the Linker itself. *Chapter 8*, Directives, discusses the directives which control the Linker.

Executing a File

The Linker created two absolute object files:

PROG4/CMD
PROGIII/CMD

which are actually two versions of the same main program. PROG4/CMD runs on the Model 4; PROGIII/CMD is for the Model III. (Model III and 4 executable programs must have the /CMD extension.)

Assuming you created these files on the Model 4, if you wish to run PROG4/CMD on your Model 4, simply type (in the TRSDOS Ready mode):

```
PROG4 (ENTER)
```

Transferring a File

You will, of course, need to transfer the program which does not correspond with your computer to the model in which it can be used, before you can execute it. For example, if PROGIII was created on the Model 4, it would need to be transferred to the Model III. If you have a Model III and 4 and an appropriate modem or cable, you can transfer the program with the ALDS File Transfer System. It will produce a Model III or 4 disk file of PROGIII/CMD or PROG4/CMD.

To transfer PROGIII/CMD to a Model III, use the following instructions:

Connect the two systems (see *Chapter 6*, The ALDS File Transfer System for instructions).

Load the ALTRAN program on both the Model III and Model 4 by typing:

```
ALTRAN (ENTER)
```

After ALTRAN displays its menu, type:

```
9 (ENTER)
```

This puts you in the 'Mini-Terminal' mode. To test the communication of your computers, on your Model 4 type:

```
COMMUNICATION
```

This word should appear on your Model III screen as well as your Model 4 screen. Next on your Model III type:

```
TEST
```

This word should also appear on your Model III screen and your Model 4 screen. If both computer screens have "COMMUNICATION TEST" written on them, then ALTRAN is communicating in both directions. Otherwise, recheck your connection procedure (see *Chapter 6*, The ALDS File Transfer System).

Press the ⏏ key on both the Model III and 4 to return to the ALTRAN menu.

On the Model 4 type:

```
1 (ENTER)
PROGIII/CMD (ENTER)
```

and on the Model III type:

```
2 (ENTER)
PROGIII/CMD (ENTER)
```

This transfers PROGIII/CMD to the Model III diskette and names it PROGIII/CMD.

ALTRAN re-displays its menu when it has finished the transfer. Press (BREAK) or (Q) to exit the ALTRAN program and return to TRSDOS Ready. You can then execute PROGIII on the Model III in the same way PROG4 was executed on the Model 4 above. Type:

```
PROGIII (ENTER)
```

Debugging a File

You can debug any of the object files with the ALDS Debugger on the Models III and 4. On your Model III type:

```
LOAD PROGIII/CMD (ENTER)
ALBUG (ENTER)
```

You can now debug PROGIII/CMD by entering:

```
J
```

On your Model 4 type:

```
LOAD PROG4/CMD (ENTER)
ALBUG
```

You can now debug PROG4/CMD by entering:

```
J
```

Answer the corresponding prompt with the following response:

Model III:

```
J [ADR][,BP1][,BP2][,BP3][,BP4] <E>? 5200,5200 (ENTER)
```

Model 4:

```
J [ADR][,BP1][,BP2][,BP3][,BP4] <E>? 5200,5200 (ENTER)
```

You can now single step through the program by pressing (E).

For more information on ALBUG, refer to *Chapter 4*.

LANGUAGE

Chapter 2/
The ALDS Editor (ALEDIT)

The ALDS Editor allows you to enter and edit an assembly language source program. You can save this program on disk as a source file to be assembled into Z80 object code.

This section describes the use of the Editor itself. For information on how to write an assembly language source program, see Section II, "ALDS Assembly Language."

Loading the Editor

This command, typed in the TRSDOS Ready mode:

`ALEDIT` *source filespec*

loads the Editor and then loads the specified *source filespec* into the Editor. The *source filespec* is optional. For example:

`ALEDIT` (ENTER)

causes the Editor to load and display a similar heading:

```
TRS-80 Model 4 Text Editor Version v.r.p.
Copyright (c) 1982, 83 Tandy Corp.
```

(*v.r.p.* is the version, release and patch numbers.)

`ALEDIT SORTER` (ENTER)

causes the Editor to load, display the above heading, then load a source file named SORTER/SRC.

If the *source filespec* does not contain an extension, the Editor appends /SRC to it.

The Editor loads into all of the memory above TRSDOS. It reserves approximately the top 33K bytes in a Model III and the top 40K bytes in a Model 4 as an "edit buffer" for inserting your programs. However, if you have also loaded one of the High Memory TRSDOS utilities the edit buffer will be smaller.

Using the Editor

The following pages define the three modes in which you can use the Editor:

- the command mode
- the insert mode
- the line edit mode

The Command Mode

When you first load the Editor, it is in the command mode. While in this mode, you can use any of the special keys listed in *Table 1* or the commands listed in *Table 2*.

All commands except I and E return to the command mode after executing. To return to the command mode from I (insert mode) or E (line edit mode), press (BREAK) or (ENTER) respectively.

When you enter an Editor command, it creates a blank "work line" and points to the line just beneath it. To redisplay the screen after an error message and delete the work line, use the N command.

Sample Use

For an example of using the command mode, use the I command to insert this program:

```
;THIS IS THE FIRST LINE (ENTER)
;THIS IS THE SECOND (ENTER)
;AND HERE IS ANOTHER (ENTER)
;AND ANOTHER (ENTER)
(↓) END (ENTER)
```

Press (BREAK) to return to the command mode.

You can move the cursor and rearrange the lines of the program. For example type the following Editor command:

```
T
```

the cursor moves to the top of the text. Type B to move it to the bottom. Press (↑) and (↓) to move it to specific lines.

Move the cursor to the third line and type:

```
1
```

The < appears to the left of the line. This specifies the beginning of a block. Move the cursor to the fourth line and type:

```
2
```

The > appears to the left of the line. This specifies the last line in the block. Move the cursor up to the second line and type:

```
O
```

which is the O command. This copies the block between the first and second line. Move the cursor to the next to last line and type:

```
D
```

delete command (executes without pressing (ENTER)). The last line is now deleted.

To save this program on disk you can use the W command. Type (it does not matter which line the cursor is positioned at):

```
W TEST (ENTER)
```

This saves this program on disk as a file named TEST/SRC. You can exit the Editor by typing:

```
Q (ENTER)
```

the quit command.

Q will exit the Editor without writing the text to disk. If you forgot to save the text first, type ALEDIT * (ENTER) to re-enter the Editor. Your text will be retained.

Be sure you use the ALEDIT * command immediately after you exit the Editor. It will not work predictably after you run a command which modifies memory. Also, be sure you type one blank space between ALEDIT and the asterisk(*).

Table 1 / ALEDIT Command Mode Keys

Model 4 Keys	Description	Model III Keys
←	moves the cursor one position to the left.	←
↓	positions the cursor down one line (ignored if the cursor is not in the first column)	↓
↑	positions the cursor up one line (ignored if the cursor is not in the first column)	↑
(CTRL)(A)	positions the cursor to the top of the screen.	(SHIFT)(↓)(B)
(CTRL)(B)	positions the cursor to the bottom of the screen or to the first line after the last line of text.	(SHIFT)(↓)(C)
.	displays the current line sequence number. This number will change as you insert and delete lines.	.

#line(ENTER)	positions the cursor to the specified *line* sequence number and moves that line to the top of the screen.	*#line*(ENTER)
(BREAK)	cancels any command being executed and returns to the command mode.	(BREAK)
(SHIFT)(↑)	cancels the current command line if you have not yet pressed (ENTER).	(SHIFT)(↑)

Table 2/ ALEDIT Editor Commands

Description of Terms
current line the line where the cursor is currently positioned. ***del*** (stands for delimiter) One of the following characters which marks the beginning and ending of a string: ! " # $ % & ' ( ) * + , - . / : ; < = > ? ***string*** one to 37 ASCII characters on the Model 4 and one to 29 ASCII characters on the Model III. ***text*** the source program or text currently in RAM. **A** (ENTER) Re-executes the last executed command. This command only works with the Editor Commands C, F, X, L and W. **B** Moves the cursor to the bottom of the text. **C *del string1 del string2 del occurrence*** (ENTER) Changes *string1* to *string2* for the number of *occurrences* you specify. Occurrences must range from 1 to 255. The changes begin at the current line and are made only to the first occurrence on a given line. If you omit *occurrence*, only the first occurrence of string1 is changed. You may specify *occurrence* with an asterisk, in which case the change is made to the first occurrence of string1 in all the remaining lines. For example: C/TEXT/FILE/3 (ENTER) changes the first 3 occurrences of TEXT to FILE.

C?TEXT?FILE?* (ENTER)

changes all occurrences of TEXT to FILE. (Change acts on only the first occurrence within a line.) After executing the command, the cursor positions itself at the last change or, at the top of the file if changes went through the whole file.

D
Deletes the current line or block of lines. To delete a block, position the cursor at the first line in the block and type (1). Then position it at the last line and type the D command. (The block may be on several pages.) The cursor must be positioned on a line within the file.

For example:

```
                LD      A,B
(1)             ADD     A,1
                ADD     A,3
(D)             ADD     A,4
                DEC     B
```

deletes all but the following:

```
                LD      A,B
                DEC     B
```

You can cancel a block deletion after pressing (1) but before typing D. To do this, press (3).

E
Allows you to edit the current line using line edit mode subcommands. The line will appear in reverse video (Model 4 only). See the edit mode for a listing of subcommands.

F *del string del occurrence* (ENTER)
Finds the specified *occurrence* of string. If you omit *occurrence*, finds the first occurrence of string. If you omit string, the last string specified is found. Occurrences must range from 1 to 255. For example:

F/TEXT/2 (ENTER)

finds the second occurrence of TEXT.

F/TEXT/ (ENTER)

finds the next occurrence of TEXT.

F (ENTER)

finds the next occurrence of the last specified string.

F% % (ENTER)

finds the next occurrence of five blank spaces. The Editor will search for only one occurrence of the string in each line.

G (ENTER)
Deletes all text from the current line to the end. You will first be prompted with:

"Are you sure?"

Type Y (ENTER) to delete; N (ENTER) to cancel.

H (ENTER)
Prints the entire text if entered as the first command or the specified block on the printer. To print a block, move the cursor to the first line of the block and type (1). Move the cursor to the last line of the block and type (H). For example:

```
          LD      A,B
(1)       ADD     A,1
          ADD     A,3
(H)       ADD     A,4
          DEC     B
```

prints a block of ADD instructions.

You can cancel a block printing after pressing (1) but before typing H. To do this, press (3).

Press (BREAK) to terminate printing. If the printer is off-line or goes off-line during printing, some characters may be lost.

I
Enters the insert mode for inserting lines just before the current line. See "Insert Mode" for more information.

J
Displays current size of text and how much memory remains. Memory size does not include a small work area when the buffer is full, but the text size may reflect some of this work area.

K (ENTER)
Deletes ALL text. (Does not delete text from the disk file, only from the edit buffer. Before deleting your text, the Editor will ask you "Are you sure". Type Y (ENTER) to execute the command; N (ENTER) to not execute it.

L *filespec* $C (ENTER)
Loads *filespec* into the Editor. $C is optional. If specified, the Editor chains the new filespec to the end of the text currently in memory. If not specified, the new filespec overlays the current text.

For example:

L TEST (ENTER)

loads TEST/SRC into the Editor.

L TEST $C (ENTER)

chains TEST/SRC to the end of the text currently in memory.

The Editor will load fixed length record (FLR) files with a record length of one. If the file is fixed length, each line must be ended with a carriage return.

Note: When the Editor completes, the record length will be 256.

M
Moves the specified block just ahead of the current line. Use ① and ② to specify the block. The Editor displays a line count as it moves each line. For example:

```
                ADD     A,B
①               PUSH    DE
                PUSH    HL
                PUSH    IY
②               PUSH    BC
                LD      A,8
Ⓜ               ADD     A,10
```

moves the block of PUSH instructions just ahead of the last line:

```
ADD     A,B
LD      A,8
PUSH    DE
PUSH    HL
PUSH    IY
PUSH    BC
ADD     A,10
```

You can cancel the block after specifying it but before typing M. To do this, press ③.

N
Updates the display. You might want to use this after executing the J command or cancelling the G command.

O
Copies the specified block just above the current line. (Use ① and ② to specify a block as described in the M command.)

P
Moves the cursor to the next page (which is 24 lines from the top of the screen on the Model 4 and 17 lines on the Model III).

Q (ENTER)
Exits the Editor. If you forgot to save the file first, type ALEDIT * (ENTER) immediately upon exiting the Editor. The Editor will load with your text retained in memory.

R (ENTER)
Deletes the current line and enters the insert mode. Using the J command, if there is 0000 memory left in the buffer, executing the R command will delete the line but will not allow it to be replaced with new text.

T
Moves the cursor to the top of the text.

U
Moves the cursor to the previous page (which is the 24 preceding lines for Model 4 and 17 lines for Model III).

V
Scrolls current line to the top of the screen.

W *filespec $option1* ... (ENTER)
Saves all text on disk as *filespec*. *filespec* is optional; if omitted, it is the filespec you used to load the file. The Editor appends /SRC to filespec unless it already includes an extension.

The *options* are:

E	Exits the Editor after saving the file unless there is an error.
L, ML, OR LM	Saves the file with line numbers in this format: ASCII line number/dummy TAB/text.
M	Saves the file as a fixed length record (FLR) file with a LRL of 256 in this format: text/carriage return This option is the default. You can use ALEDIT to edit a "DO-file" created with the TRSDOS "BUILD" command and save this format, which can be loaded by the TRSDOS "DO" command.

For example:

W SAMPLE (ENTER)

saves all text as a file named SAMPLE/SRC.

W SAMPLE $E

saves text as SAMPLE/SRC. The Editor will exit back to TRSDOS Ready after saving the file.

Without using the L or the M options, the Editor saves the file in the format required by the ALDS Assembler:

- Each character is saved exactly as it appears on the display.
- No carriage returns or end of text code is saved.

• Each line is saved in this format: length/text/

X *del string1 del string2 del occurrence*

Same as the C command, but prompts before making the change. Occurrence must range from 1 to 255.

The Insert Mode

The I command gets you into the insert mode. Type:

```
I
```

(Do not press (ENTER).) The editor clears the screen and positions the cursor at the upper left-hand corner. You can now insert source lines into the edit buffer.

Do not use line numbers. The Assembler will consider them syntax errors.

Each source line may have up to 78 characters. After typing the line, press (ENTER) to insert it. To cancel it and return to the Editor command mode, press (BREAK). For example:

```
;THIS IS THE FIRST LINE (ENTER)
;THIS IS THE SECOND (ENTER)
;AND HERE IS ANOTHER (BREAK)
```

inserts only the first two lines in the Editor's memory; then returns to the Editor command mode.

While inserting lines, you might find it convenient to use the (→) key. This key is used as a tab key. The Editor has tabs set every eight columns.

The Editor offers certain control functions for quick insertion. To activate a control function, press the (CTRL) on the Model 4 or (SHIFT) (↓) on the Model III, at the same time you press the function key. For example, pressing these keys at the same time:

Model 4: (CTRL)(D)
Model III: (SHIFT)(↓)(D)

causes the Editor to insert a semicolon and the current date in the text and then position the cursor on the next line.

Model 4: (CTRL)(E)
Model III: (SHIFT)(↓)(E)

causes the Editor to insert ":", tab to the next tab stop, insert "EQU", and then tab again to the next tab stop.

If the line becomes full while inserting the control function, the Editor stops and awaits the next insert mode instruction.

Table 3 lists all the insert control functions.

Table 4 lists the special control keys available in the insert mode.

Note: When the edit buffer is full, it will give you a buffer full message and return to the command mode.

Table 3/ ALEDIT Insert Control Functions

Model 4 FUNCTION	INSERTS	Model III FUNCTION
CTRL D	;*current date* ENTER (i.e. ;02/25/83 ENTER)	SHIFT ↓ D
CTRL E	: → EQU →	SHIFT ↓ E
CTRL G	→ GLOBAL →	SHIFT ↓ G
CTRL L	→ INCLUDE →'	SHIFT ↓ L
CTRL N	; → ENTRY: →	SHIFT ↓ N
CTRL O	{ (open braces)	SHIFT ↓ O
CTRL P	→ PUBLIC →	SHIFT ↓ P
CTRL Q	} (closed braces)	SHIFT ↓ Q
CTRL R	; → EXIT: →	SHIFT ↓ R
CTRL S	;********** ... ENTER (semicolon followed by 64 asterisks)	SHIFT ↓ S
CTRL T	^	SHIFT ↓ T
CTRL U	; → USES: →	SHIFT ↓ U
CTRL V	[	SHIFT ↓ V
CTRL X	→ EXTRN →	SHIFT ↓ X
CTRL Y	displays the tab positions. Nothing is inserted.	SHIFT ↓ Y
CTRL Z	;———————— ... ENTER (semicolon followed by 64 dashes)	SHIFT ↓ Z
CTRL SHIFT F	~	SHIFT ↓ 6
CTRL ·	]	SHIFT ↓ −

Table 4/ ALEDIT Insert Mode Special Keys

◄	moves cursor back one space and deletes a character
ENTER	ends current line, carriage return, and goes to next line still in "I" mode. **Note:** ENTER inserts a blank line if executed by itself.
BREAK	cancels current line, and returns to CMD-mode with the cursor on the next line.
►	moves to next tab position on the line. **Note:** ►will reverse tab.

The LINE EDIT MODE

The E command enters the line edit mode for editing characters within the current line. When you enter this mode, the Editor displays the line in reverse video on the Model 4 only. You can then use any of the edit subcommands listed in *Table 5* or the special edit keys listed in *Table 6*.

For example, assume the cursor is on the following line:

```
;THIS IS THE FIRST LINE
```

To change the word FIRST to THIRD from the command mode, type:

```
E
```

(Do not press ENTER.) The Editor will display the line in reverse video (Model 4 only). You are now in the line edit mode.

Use the SPACEBAR to position the cursor at the F in FIRST and type:

```
5CTHIRD ENTER
```

This stores the change and returns to the Editor command mode.

Table 5/ ALEDIT Line Edit Mode Subcommands

COMMAND	DESCRIPTION
A	Clears all changes and re-enters the edit mode for the current line.
***n*Cstring**	Changes the next n characters to the specified *string*. If n is omitted, only one character is changed. (Press SHIFT ◄ to exit the change early.)
***n*D**	Deletes *n* characters. If *n* is omitted, one character is deleted.
E	Exits the edit mode and stores changes.
Hstring	Deletes the remaining characters, enters the insert mode and allows you to insert a *string*.

Istring	Allows you to insert material beginning at the current cursor position on the line. Pressing ◀ will delete characters from the line. The line may be up to 78 characters in length on the Model 4 and 61 characters in length on the Model III.
nKcharacter	Kills all characters preceding the *n*th occurrence of the *character*.* If n is omitted, the first occurence is used. If no match is found, the rest of the line is killed.
L	Moves cursor to beginning of line.
Q	Quits the edit mode, cancelling all changes.
nScharacter	Positions the cursor at the nth occurrence of character.* If no match is found, positions the cursor at the end of the line.
Xstring	Moves the cursor to the end of the line, enters the insert mode, and allows you to insert a *string*.

*The compare begins on the character following the current cursor position.

Table 6/ ALEDIT Line Edit Mode Special Keys

SPACEBAR	Moves cursor one position to the right.
SHIFT ◀	Returns to edit command mode from the I, X, C, or H subcommands.
▶	Moves cursor to next tab position (or the end of the line) while in the I, X, or H subcommand mode.
◀	Moves cursor one position to the left.
ENTER	Identical to the E subcommand.

Chapter 3/

The ALDS Assembler (ALASM)

The ALDS Assembler produces Z80 object code. It does this by inputting a source file — composed of Z80 instructions, assembler language directives, and data — and assembling it into Z80 code.

In this Section, we'll show how to use the Assembler. For information on the source file, see the sections on the ALDS Editor, Assembler Language Directives, and Z80 Instruction Set.

The Assembler Command

This command, typed in the TRSDOS Ready mode, loads and executes the Assembler:

ALASM *filespec1 filespec2* {*switches*}

filespec1 is the source file you want assembled. If you do not specify an extension, the Assembler assigns it the extension /SRC. *filespec1* must not be read protected. Do not specify a password.

filespec2 is optional. It stores the assembled object code. You can specify *filespec2* with an asterisk (*). If so, the Assembler assigns it *filespec1's* name (less the extension).

If the program is relocatable and *filespec2* does not have an extension, the Assembler assigns it the extension /REL. (The Assembler uses the PSECT directive, discussed in *Chapter 8*, to determine whether the program is absolute or relocatable.)

filespec2 overrides any OBJ directive you have in your program. *filespec1* and *filespec2* must be in the standard TRSDOS filespec notation.

Examples:

ALASM TEST TEST (ENTER)

assembles TEST/SRC and saves the object code as TEST if the program is absolute or TEST/REL if it's relocatable.

ALASM TEST * (ENTER)

does the same.

`ALASM TEST/PAY * (ENTER)`

assembles TEST/PAY and saves the object code as TEST or TEST/REL.

`ALASM TEST/PAY FILE/ACC (ENTER)`

assembles TEST/PAY and saves the object code as FILE/ACC.

`ALASM TEST (ENTER)`

assembles TEST/SRC. No object file is produced unless TEST/SRC contains an OBJ directive.

Switches

You may specify one or more switches to create a listing or control the assembly output. If you do not specify filespec2, you must enclose the switches in parenthesis. For example:

`ALASM TEST * L (ENTER)`

assembles TEST/SRC into TEST or TEST/REL and displays a listing (L) of the assembly.

`ALASM TEST * LXP (ENTER)`

does the same as the above and also creates a cross reference listing (X) and prints it all on the printer (P).

`ALASM TEST (L) (ENTER)`

assembles TEST/SRC and creates a listing. Since filespec2 is omitted, the parenthesis are required.

The details of all the available switches are in *Table 7:*

Table 7/ ALASM Switches

L (Listing)
Generates a complete listing on the video display *Figure 1* shows a sample assembly listing on the Model 4.

The Assembler prints a character to the left of a line number if the line is affected by one of these special conditions:

Character	Condition
-	the symbol in symbol field is never referenced
p	the symbol in symbol field is PUBLIC
g	the symbol in symbol field is GLOBAL
+	a symbol in operand field is defined in global file
x	a symbol in operand field is defined in an external file
r	some or all the object data is relocatable

X (Cross Reference)
Generates an alphabetical cross reference listing of all symbols defined in the program.

P (Printer)
Outputs the listing on the printer in addition to the video display. Use this option with the L option. You may not use this switch with the Assembler D switch, nor can you use it with the TRSDOS SPOOL command's "capture file" option (the "N" option). Be sure that the printer is on-line.

W (Wait On Errors)
Causes the Assembler to stop the listing at each assembly error. Press (ENTER) to continue the listing.

T (Truncate the Listing)
Truncates the listing output to the printer so that you can use 80 column paper.

D*drive number* (Store Listing on Disk)
Stores the listing in a disk file named *filespec1*/LST. Use this option with the L option. If the listing will not fit on the diskette, the Assembler closes the file and prompts you to change diskettes. Do so and press (ENTER). (Be sure the diskette you remove does not contain the source, object, ALASM files or important data.)

The Assembler stores the remainder as *filespec1*/LSU on the newly inserted diskette. If this diskette also becomes full, the listing goes to the next diskette as *filespec1*/LSV.

The Assembler repeats this process until it has saved the entire listing. Each time it creates a new listing file, it will increment the third character in the extension:

filespec1/LST, *filespec1*/LSU, ... *filespec1*/LSZ, *filespec1*/LSA, *filespec1*/LSB, ... *filespec1*/LSS

You may optionally omit the drive number. If you do so, the Assembler outputs the listing file to the lowest numbered write-enabled drive (usually drive 0) and continues the listing in the next drive. This is not a good method to use, since the Assembler might run out of work space before completing the listing.

Files created with the D option should be printed with the LIST command.

The D switch overrides the P switch.

G (Go)
Executes the program after assembling it. The program must be absolute and have no errors.

F (Memory image)
Causes the assembled object file to be in memory image form, rather than the TRSDOS program file format. The program must be absolute and have no errors. See the NOLOAD directive in *Chapter 8* for more information.

Examples:

ALASM SOURCE OBJTST LDX (ENTER)

assembles SOURCE/SRC into OBJTST/REL or OBJTST. Displays a listing and a cross reference of this assembly and saves these in one or more files named SOURCE/LST, SOURCE/LSU, SOURCE/LSV, etc.

ALASM TEST * G (ENTER)

assembles TEST/SRC into TEST or TEST/REL, then executes the program (unless it is relocatable or has errors).

ALASM MOD1 PROG/CMD LPW (ENTER)

assembles MOD1/SRC into PROG/CMD and generates a listing which is printed on the video display and the printer. Each time the Assembler encounters an error, it stops the listing.

ALASM XYZ/COD TST/ABC:2 LD3 (ENTER)

assembles XYZ/COD and stores it as TST/ABC on the diskette in drive 2. The Assembler generates a listing which it displays and saves as XYZ/LST on the diskette in drive 3. If the drive 3 diskette becomes full, the assembler prompts you to insert another diskette to hold XYZ/LSU, a continuation of the listing.

Note: Be sure the CLOCK is not turned on (CLOCK (OFF)) while running the Assembler.

```
Tandy Corp. ALDS ALASM copr. 1982,83 v.03.02          07/01/83
Source=TEST/SRC      Object=TEST
Pass No. 1 Complete

0000'     00001    ;THIS IS THE FIRST LINE
0000'     00002    ;AND HERE IS ANOTHER
0000'     00003    ;AND ANOTHER
0000'     00004    ;THIS IS THE SECOND
0000'     00005    ;AND HERE IS ANOTHER
0000'     00006       END

No Assembly Errors

Time=0:01
Bytes=0
Lines=6

Pass No. 2 Complete
```

Figure 1

Chapter 4/
The ALDS Debugger

The ALDS Debugger is an easy-to-use system for debugging absolute object code programs. It includes all the features found on the DEBUG utility program of your TRSDOS disk. In addition, it includes several new, powerful debugging tools.

The Model III and Model 4 Debugger are on your system diskette in the module ALBUG/CMD.

Note: This module resides in all memory above E000H (57344 decimal), therefore it cannot be used by programs which exceed this amount. This also means ALBUG should not be used on the Model III within a DO file, or on the Model 4 when certain high memory drivers are loaded.

Among many other features, the ALDS Debugger allows you to:

- set both permanent breakpoints with pass counts and temporary breakpoints (see the J and B commands in *Table 8*).
- execute one or more instructions at a time (see the I and E commands in *Table 8*).
- specify a memory address as an offset. This is useful in debugging a program which you assemble in the relocatable mode (see the O command in *Table 8*).

What you can debug with the ALDS Debugger:

You can debug any absolute program. The program must lie in memory between 5200H and DFFFH on the Model III and between 3000H and DFFFH on the Model 4.

In addition, you can use the Debugger to change the contents of disk files, using the DISK ZAP mode (see the Z command).

Loading The ALDS Debugger

To use the Model III or Model 4 Debugger you must first load the program that you wish to debug with the TRSDOS LOAD command. Refer to your TRSDOS III or 4 Disk System Owner's Manual for more information. For example type:

LOAD *filespec* (ENTER)

Next you must turn on the ALDS Debugger by typing:

ALBUG (ENTER)

The Debugger display appears on your screen and you are now in the Debugger command mode. You can use any of the commands listed in *Table 8*. In order to begin debugging or executing your program, you must change the PC register to the address of the beginning of your program by using the "R" command.

If you wish to enter the Debugger without loading one of your programs (i.e. to enter the DISK ZAP mode), from the TRSDOS Ready mode type:

ALBUG (ENTER)

The Debugger begins execution.

The Debugger Display

This is a sample Debugger display.

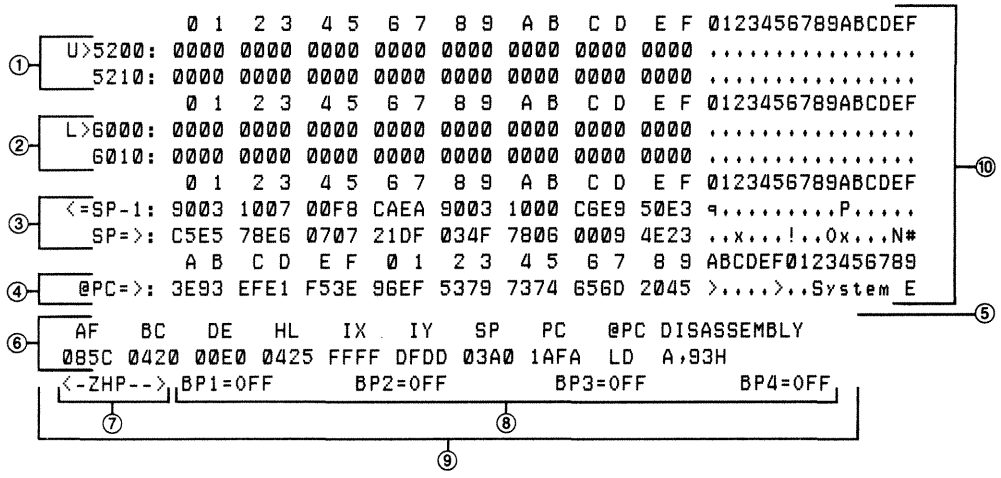


Figure 2

Refer to *Figure 2* for reference to the following explanations:

1. Upper Dump. This is a 32 byte section in the memory. U stands for Upper Dump. The 5200 signifies that the memory address of the first byte in that row is 5200H. When you load ALBUG this address is automatically set to 3000H on the Model 4 and 5200H on the Model III. To the right of the 5200 are the contents of memory locations 5200H through 520FH. To the right of the 5210 are the contents of memory locations 5210H through 521FH. Above these lines are numbers which represent the memory address of the data listed below them. For example, the byte under the 7 and in the row marked as 5200, is the memory location 5207H.
2. Lower Dump. This is another 32 byte section of memory. It is arranged exactly like the Upper Dump, except that it is originally set to 6000H on the Model III and 3000H on the Model 4.
3. The memory location pointed to by one of the register pairs (in this case SP) is displayed here along with the 15 bytes immediately following it. The label of this line is SP = >. Directly above it is a line labeled <SP-1. It contains the 16 bytes preceding the memory location pointed to by the register pair.

(The byte on the far left of the line is at the address (SP)-16 and the byte on the far right is at the address (SP)-1.)

4. The memory location pointed to by the PC is on this line, marked @PC=>. It is followed by the contents of the next 15 bytes in memory. Above this line is the memory location of the respective bytes.
5. This line (here shown blank) displays certain information such as base register addresses and math function results. When you enter a new command, it is erased.
6. These lines show the contents of the Z80 registers. At the right side of the lower line, below "@PC DISASSEMBLY", is a Z80 instruction. The address pointed to by the PC contains this instruction in machine code, and the Debugger has disassembled it into an assembly level instruction. The Debugger uses as many bytes following the PC address as necessary to make a complete instruction. This means that what is disassembled can be one to four bytes long.
7. <SZHPNC> are the condition codes set in the F register. The codes are:

S	sign
Z	zero
H	half carry
P	parity
N	BCD condition
C	carry

When a condition bit is set (i.e. when it is equal to 1) the Debugger encloses the letter within the < > characters. Otherwise it simply displays a hyphen (-). For example, <-Z---C> shows that the zero (Z) and the carry (C) bits have been set and all other bits have not.

8. This area lists the status of the permanent breakpoints. BP1 = 5200/000C translates as breakpoint 1 set at 5200H with the pass counter set at 12 decimal passes. BP2 = OFF means that breakpoint 2 is not set. (See *Table 8* for more information).
9. When you first enter the Debugger, this line gives version and copyright information. Thereafter, it displays commands and prompts used in debugging code.
10. This area displays the ASCII value of any hexadecimal data to its left. If the hex number has no printable value, a period (.) is displayed.

Entering Commands

Table 8 lists all the Debugger commands. You can execute most of them by simply pressing the appropriate letter. By pressing (BREAK), you can abort any command in the middle of execution and return to the command level.

Most commands prompt you to specify a register or data (the prompt is in area 9 of *Figure 2*). The prompts use these abbreviations:

Adr	Address
Asc	ASCII
BP1	Breakpoint 1
CHR	Character
C(lr)	Clear
DEC	Decimal
<E>	(ENTER)
Eadr	End address
H(ex)	Hexadecimal
Pas	Pass counter
Reg	Register
SAdr	Start address
Str	String

The commands usually prompt you for a certain number of parameters. If you fail to provide enough parameters, or if you use an invalid number as a parameter (e.g. hex when a decimal number is expected), you receive the message:

```
Illegal Parameter
```

and the Debugger returns to the command mode.

Specifying Registers

Certain commands require you to input a register or register pair. For example, the Debugger might prompt you with:

```
C,E,L,A(F),B(C),D(E),H(L),(I)X,(I)Y,S(P) or P(C)
```

To enter a single register, you simply press the appropriate letter. To enter a register pair, you must press the letter NOT shown in parenthesis. For example, (C) enters the C register, but (B) enters the BC register pair.

Specifying Data

As a Constant

Some commands require constants. When entering a hexadecimal constant, you must follow it with H. For example, "10" indicates the decimal number 10, while "10H" stands for the hexadecimal number 10 (the decimal number 16).

As an Address

Other commands require addresses. These must be in hexadecimal. There is no need to follow hex addresses with an H.

You can specify an address by referring to a register pair which contains that address. For example, if BC contains the number 6000H, you can enter $B

instead of the address 6000H. The register abbreviations for this type of addressing are:

AF	$A
BC	$B
DE	$D
HL	$H
IX	$X
IY	$Y
SP	$S
PC	$P

You can also specify any address as an "offset" to a base register. This is useful if you assemble the program in the relocatable mode. It allows you to use a relocatable location to specify an address. (See the O command in *Table 8*).

The ALDS Debugger is for debugging your own code. Hence, you cannot enter an address which is in the system memory (i.e., below 3000H on the Model 4 or 5200H on the Model III). In addition, the Debugger protects itself by not allowing you to interfere with the memory above E000H on the Models III and 4. If you enter an invalid address the Debugger returns to the command mode.

Breakpoints

The Debugger allows you to set "breakpoints" within your code. Breakpoints are commands causing the execution of your program to stop at a given point. There are two types of breakpoints, temporary and permanent.

You can assign temporary breakpoints with the J command (Jump to an address and execute). They apply only to this one execution of J. With them you can execute a short section of code, or determine which way control goes at a branch statement. (See the J command in *Table 7*).

With the B (Breakpoint) command, you can set permanent breakpoints. They remain in your program until you leave the Debugger or clear them. Permanent breakpoints may have a pass count associated with them.

You must be cautious when setting breakpoints. Set them only at the first byte of an instruction. If you are writing a self-modifying code where the first byte of an instruction may change during the course of running the program, be careful not to place a breakpoint at that instruction.

Another point of caution: If you return to TRSDOS Ready other than through the Q(quit) command, the breakpoints will not automatically clear. If you return to ALBUG without reloading your program, the breakpoints will still be there, although they will not be displayed in the display area. You must personally reset them by using the M(modify memory) command.

Table 8/ Debugger Commands

***n*; (semicolon)**
Advances the memory location of the Upper Dump. The default advance is 16 bytes. You can precede the semicolon with *n*, a decimal number, which changes the default to *n* bytes, until you press (BREAK), when the default returns to 16 bytes.

***n*+ (plus)**
Advances the memory location of the Lower Dump. The default advance is 16 bytes. You can precede the plus sign with *n*, a decimal number, which changes the default to *n* bytes.

***n*− (minus)**
Decrements the memory location of the Upper Dump. The default decrement is 16 bytes. You can precede the minus sign with *n*, a decimal number, which changes the default to *n* bytes.

***n*= (equal sign)**
Decrements the memory location of the Lower Dump. The default decrement is 16 bytes. You can precede the equal sign with *n*, a decimal number, which changes the default to *n*-bytes.

B
Sets or clears permanent breakpoints and their pass counters. After you press (B), a prompt appears:

1,2,3,4 or C(lr)?

You can now choose to set or alter any of the four breakpoints, or clear all four. To set breakpoint 1, for example, press (1). The Debugger prompts with:

0 <E> or [Adr][,Pas]<E>?

You can now select the address where you want the breakpoint. You must set it at the first byte of an instruction. You can not place a breakpoint on top of an existing breakpoint.

Each permanent breakpoint is associated with a pass counter. Pass counters are useful to stop execution after an instruction has been executed a given number of times. A pass count is specified by following the breakpoint address with a comma and then the pass count value.

To set the breakpoint at 6000H, with a pass of 12 type:

6000,12(ENTER)

You can clear the breakpoint by entering a value of 0:

0 (ENTER)

To clear all four of the breakpoints, press (C) in response to the first prompt. The Debugger asks you:

Are You Sure (Y/N)?

to allow you to change your command (the Debugger accepts only Y or N). The status of all four breakpoints is displayed in area 8 of *Figure 2*.

When you set each breakpoint, the Debugger saves the contents of the breakpoint address, and replaces it with an RST 18H instruction on the Model 4 which assembles into 0DFH or an RST 30H instruction on the Model III which assembles into 0F7H. Now, in typing Y to remove the breakpoints, the Debugger restores the memory addresses to their original contents.

The contents of the pass counter can be updated without respecifying the address of the breakpoint. For example, if you had previously set a permanent breakpoint at 5200H, you can update the pass count to 24 by typing:

,24 (ENTER)

in response to:

0<E> or [Adr][,Pas]<E>?

Whenever ALBUG executes a program instruction which is associated with a permanent breakpoint with a nonzero pass count, the count is decremented and execution resumes. Execution halts when a permanent breakpoint with a pass count of zero is reached. ALBUG is designed so that once execution is halted by reaching a pass count of zero, you may single step over a permanent breakpoint.

The permanent breakpoint remains in the program until it is explicitly cleared with the (B) command or until ALBUG is exited with the (Q) command. Note: if a return to DOS is executed in your program, the permanent breakpoints remain intact and ALBUG can be re-entered by typing ALBUG.

ALBUG uses RST 18H instructions on the Model 4 and RST 30H instructions on the Model III to handle all breakpoint processing. If ALBUG encounters an RST 18H instruction on the Model 4 or an RST 30H instruction on the Model III, which you placed in your program, execution will halt. To resume execution, the program counter must be reset using the (R) command.

C

Copies one section of memory to another. After you press (C), a prompt appears:

Start Adr,End Adr,To Adr <E> ?

Type the appropriate start, ending, and destination addresses. For example, type:

5800,582F,6000 (ENTER)

to copy the data contained in addresses 5800H-582FH to addresses 6000H-602FH.

D
Dumps the data contained in the address pointed to by a register pair in the Debugger display. (See area 3 in *Figure 2*). The data on either side of this address is also displayed. After you press (D), the Debugger displays:

Reg Dump B(C),D(E),H(L),(I)X,(I)Y,S(P) or P(C)?

To see the data referenced by the IX register pair, respond with:

(X)

The screen updates to display the new dump.

***n*E**
This command is identical to the (I) command with one exception: If the current instruction is a call the debugger executes the entire routine.

***n*F**
Searches for a string within a given range in the memory. After you press (F), a prompt appears:

Sadr,Eadr <E> or <E>?

After you enter a valid start and end address, the Debugger asks you:

H(ex) or A(scii)?

Depending on whether you enter (H) or (A), the Debugger then prompts you with:

Hex Str <E>?
or
Asc Str <E>?

When you enter the appropriate type string the Debugger searches through the given memory for it. If the Debugger finds a matching string, the Lower Dump is set to display this part of memory. If no match is found, the Debugger returns to the command level.

To find the next occurence of the string, you need only to press (F) from the command level and respond to the prompt with (ENTER). You can continue to search for matching strings until you reach the ending address (EAdr) or until there are no more string matches in the specified range.

To specify which occurrence of the string you want to find, precede the F command with n, a decimal number between 1 and 254. For example, to find the fifth occurrence of 1FH, start by entering 5F.

The F command will find an ASCII string of up to 24 characters or a HEX string of up to 12 digits.

You may also specify a new range for the current string. Enter the new range, abort the (F) command with the (BREAK) key at the 'H(EX) or A(SCII)? ' prompt, and press:

F (ENTER).

G

Examines a 256 byte area in memory. After you press (G), a prompt appears:

U(pper) or L(ower)?

Depending on your answer, the Debugger displays the 256 byte multiple of memory which contains the address of the Uppper or Lower Dump. For example, if the Upper Dump starts at 5207H and you press (G) and then (U), your screen changes so that it now contains a dump of memory starting with 5200H.

The <*n*;>, <*n* −>, <*n* =>, and <*n* +> commands may be used in this display mode as they were in the partial screen display mode, except that the value of n is always rounded up to a multiple of 256.

Press (BREAK) to return to the regular Debugger display.

***n*I**

The (I) and (E) commands are ALBUG's single step instructions. The (I) command executes the current instruction in your program (the instruction pointed to by the PC register.) ALBUG then increments the PC register to the next instructions address and returns to the command mode.

By preceding (I) with n, a decimal number, you can indicate the number of times it is to be repeated. For example, if you type:

10I

the (I) command is executed 10 times.

There are a couple of considerations you should be aware of when single stepping. ALBUG will not place a breakpoint in a protected area. This implies that an attempt to single step an instruction in a protected area will cause a jump to that instruction. Single stepping a call to a protected area will cause the entire call to be executed at full speed. These precautions are necessary since many of the system calls such as video and disk I/O will work properly only when executed at full speed.

J

Executes a specific section of your program. After you press (J), a prompt appears:

J [ADR][,BP1][,BP2][,BP3][,BP4] <E>?

The start address (ADR) is optional. If you omit it, the execution begins at the contents of the PC. BP1-BP4 are temporary breakpoints and are also optional. You can include any or all of them.

The first temporary breakpoint encountered causes the execution to terminate. This clears all temporary breakpoints. The execution also terminates if a permanent breakpoint with a pass of zero is encountered.

For example, suppose you want to execute the instructions between 5200H and 5221H, inclusively. After pressing (J), you would type:

5200,5221 (ENTER)

Temporary breakpoints are often useful near branch points. If you set breakpoints at the possible jump locations, you can see which way your program goes. For example, say you have a set of conditional jumps which could go to 6040H, 6080H or 60F0H. When you enter:

5800,6040,6080,60F0 (ENTER)

your program begins at 5800, and terminate after jumping. You can then examine the PC to see which breakpoint caused the execution to stop (i.e., which way the jump went).

K

Allows you to convert between decimal, hex, and ASCII characters. With this command, you can also perform addition and subtraction. After you press K, a prompt appears:

Enter value or equation ?

You can then enter a value or equation. For example, to find out the ASCII character for 32H, type:

32H (ENTER)

The displays on the Models 4 and III (in area 5 of *Figure 2*) are then:

Model 4:
HEX String = 0032 DEC String = 50 CHR String = ".2"

Model III:
HEX String = 0032 DEC String = 00050 CHR String = ".2"

To do addition or subtraction, simply type in the equation. You can mix decimal, hex, or character constants in the equation. Only single characters are allowed, and unprintable characters are output as periods (.); all characters must be preceded by a quote mark ("). For example, if you type this equation:

1124-40H + "Z (ENTER)

the Debugger displays:

Model 4:
HEX String = 047E DEC String = 1150 CHR String = ".~"

Model III:
HEX String = 047E DEC String = 01150 CHR String = ".~"

the result must lie between 0 and FFFFH, or else the number is represented modulo FFFFH. For example, -1H is represented as FFFFH, and 10001H as 1H.

L
Loads a given range of memory with a constant value. After you press (L), a prompt appears:

SAdr,EAdr,Value <E> ?

When you enter a start address, end address, and value, the area in memory is filled inclusively with the value. For example:

6000,6FFF,FFH(ENTER)

fills addresses 6000H to 6FFFH with FFH.

6000,6FFF,16(ENTER)

fills addresses 6000H to 6FFFH with 10H (the hexadecimal equivalent of decimal 16).

M
Changes values in user memory. After you press (M), a prompt appears:

Address = ?

Enter a hexidecimal address and press (ENTER). The Debugger then displays a 256 byte block of memory and puts the cursor on the specified memory location. The numbers along the left-hand side are the memory addresses for the first byte in their respective lines. You may reposition the cursor with the up, down, left, and right arrow keys when entering data. Press (ENTER) to return to the debugger display.

N
Toggles the Debugger display between the primed and unprimed register set.

O
Sets values for offset base registers. You can use these offset registers for debugging a program you assembled in the relocatable mode. When you press (O) a prompt appears:

1,2,3,4,5,6,7,8 or <E>?

If you press (ENTER) the Debugger displays the values of the base registers in area 5 of the screen (see *Figure 2*). There are eight offset base registers. They supply the "base" or start address of the program or O module.

After you set an offset address, you can specify an address as a relocatable location, followed by a colon, followed by the number of the offset register. (Your Assembler listing gives the relocatable locations of each instruction.)

For example, if an instruction in the assembly listing is at relocatable 0001A, and you linked the program using an absolute start address of 6000H, press (1) in response to the above prompt, and you will receive:

Base Adr <E>?

type:

6000 (ENTER)

This sets base register 1 to 6000H. Then, an address 1AH bytes after the beginning of 6000H can be entered as 1A:1.

P (Model III) or (CTRL) (:) (Model 4)
Prints what is currently displayed on your screen. If your printer is not ready, you must press the (BREAK) key to return to the command line.

Q
Exits the Debugger and returns to the TRSDOS Ready mode. All existing breakpoints are cleared. The Debugger is turned off.

R
Alters the contents of any of the registers. When you press (R), a prompt appears:

C,E,L,A(F),B(C),D(E),H(L),(I)X,(I)Y,S(P) or P(C)?

After you press the appropriate letter, the Debugger prompts you for a value to put in the register. For example, if you are changing the C register, a prompt appears:

(C =## or # <E>) C=?

To change the register to FFH, type:

FF (ENTER)

The screen is updated and the C register now contains FFH.

You can also change register pairs. For example, if you were changing the contents of the HL register pair to A064H, after you press (R), respond to the register prompt by pressing (H). You are then prompted with:

(HL =#### or ### or H =## or # <E>) HL=?

To complete the change, simply type:

A064 (ENTER)

If you are changing a register pair and you input only 3 digits, the Debugger assumes leading zeros. By using the N command first, you may alter the contents of the prime register set.

The stack pointer and the program counter may not be changed to point at the protected areas. Keep in mind when changing the stack pointer that ALBUG uses the stack. To be safe allow for a stack size of 256 bytes.

S
Executes a TRSDOS system command. Enter the system command after the S. For example:

S DIR (ENTER)

returns the directory of drive Ø, and then prompts you with:

<ENTER> to continue

Note: Some commands automatically jump to TRSDOS Ready if there is an error such as "File not found". If this occurs, be aware that the breakpoints are not cleared.

V
Changes the start address of the Upper or Lower Dump. When you press Ⓥ, a prompt appears:

(U)pper or (L)ower?

Depending on which you press, Ⓤ or Ⓛ, you will be prompted with either:

U Address = ?
or
L Address = ?

For example, to change the start address of the Upper Dump to 6ØØØH, respond to "U Address = ?" with:

6ØØØ ENTER

Z
Enters the DISK ZAP mode, allowing you to debug disk files. See the explanation below.

The Disk Zap Mode

The DISK ZAP mode allows you to change the contents of your fixed length record disk files. When you enter the Z command, the screen clears, and you are prompted:

```
ALDS Disk Zap
Enter Filespec ?
```

After you enter the filespec, DISK ZAP asks you for the sector or record number.

Note: DISK ZAP on the Models III and 4 only work on files that have an LRL of 256, therefore the sector number and the record number will be the same.

```
Enter Sector/Record Number (# <E> or <E>) ?
```

You can specify a sector number, or just press ENTER. If you press ENTER, the DISK ZAP displays the first disk sector containing your file (relative sector Ø).

The display for the sector is similar to what the M (Modify memory) command displays, except that the relative sector and starting byte numbers are listed along the left side in hexadecimal. For example, the number 11ØØ refers to sector 11 hex (17 decimal) and byte ØØ.

You can move from sector to sector by pressing the semicolon ; which advances the display to the next sector. The minus sign − decrements the

display to the previous sector. If you cross a file boundary (i.e. if you go to a sector not used by your file), you will return to the DISK ZAP filespec prompt.

You can modify the data in your file much like you modify memory. When you press (M), the Debugger puts the cursor onto the first byte of the sector. You can then position the cursor to the correct byte with the up, down, left, and right arrows. After you have completed your change, press (ENTER) to write the change to the disk. If you don't want the change written, press the (BREAK) key.

Technical Note: Decimal numbers in ALBUG are treated modulo 65536. For example, a number entered as 65537 will be treated by ALBUG as 1. Thus, ALBUG will not let you access any sector or record above 65535.

Disk Zap Errors

If you get an error message while using DISK ZAP, it is a TRSDOS error message. See your TRSDOS Owner's Manual for an explanation.

Leaving The DISK ZAP Mode

Pressing (BREAK) at the DISK ZAP filespec prompt returns you to the Debugger. Pressing (BREAK) from any other level of DISK ZAP returns you to the original DISK ZAP filespec prompt.

Chapter 5/

The ALDS Linker (ALLINK)

The ALDS Linker converts a relocatable object file into absolute object code.

Unlike many linkers, ALDS Linker receives its commands through directives in your program. You can use these directives to get the Linker to link in external program sections and use external symbols. The Linker directives are:

PSECT — begins a program section and determines its mode (absolute or relocatable)
PUBLIC — declares symbol definitions PUBLIC so that other program sections can use them
EXTERN — brings in external symbols
GLOBAL — creates a global symbol file
GLINK — brings in global symbols
LINK — links an external absolute or relocatable program section

For information and examples on how to write a relocatable program containing Linker directives, see *Chapter 8*.

The Linker Command

This command, typed in the TRSDOS Ready mode, loads and executes the Linker:

`ALLINK` *filespec1 filespec2* {*options*}

filespec1 is the relocatable file you want converted. If you do not specify an extension, the Linker assigns it the extension /REL.

filespec2 is optional. If specified, it stores the converted absolute object file. If not, the Linker will still processes the file so that you can test for undefined symbols, missing files, or generate a listing.

On the Models III and 4, *filespec2* must have the extension /CMD to load and execute. You can use an asterisk (*) to specify *filespec2*. If so, the Linker assigns it *filespec1's* name with the extension /CMD.

You can specify one or more of these options, separated by a blank space:

$=*nnnn* specifies the absolute hexadecimal start address of the program. If omitted the start address is 3000H (Model 4) or 5200H (Model III).

MAP prints each PSECT name, its absolute start address, and the start, end, and transfer address of the program.

SYM prints the absolute address of each PUBLIC and GLOBAL symbol, sorted alphabetically by symbol. You cannot use this option with the XREF option.

XREF prints an alphabetical cross-reference of each PUBLIC and GLOBAL symbol, its absolute address, and all addresses which reference it. This option overrides the SYM option, if both are specified.

DISK saves the listing requested by the MAP, SYM, or XREF options on disk. The resulting disk file has the same name as filespec1 with the extension /MAP.

PRT directs the listing requested by the MAP, SYM, or XREF options to the printer.

Examples:

```
ALLINK PROG/REL PROG $=7000 MAP SYM DISK (ENTER)
```

assigns absolute addresses beginning with 7000H to PROG/REL and stores the resulting file as PROG/CMD. The Linker displays a PSECT MAP and a table of absolute symbol definitions then stores this listing in a file named PROG/MAP.

```
ALLINK PROG DONE (ENTER)
```

assigns absolute addresses beginning with 3000H (Model 4) or 5200H (Model III) to PROG/REL and stores the resulting file as DONE/CMD.

```
ALLINK PROG * (ENTER)
```

assigns absolute addresses beginning with 3000H or 5200H to PROG/REL and stores the resulting file as PROG/CMD.

Technical Information

Operation

The Linker processes the file in two passes. In pass 1, the Linker:

- processes any LINK directives by linking in the specified program sections.
- assigns the file absolute addresses. It does this by offsetting the relocatable locations (assigned by the Assembler) to the absolute start address.
- processes any LINK directives by linking in the specified program sections (PSECTs). If the PSECT to be LINKed is relocatable, the Linker assigns it addresses which immediately follow the last relocatable PSECT. If it is absolute, the Linker will assign it the same addresses the Assembler assigned it.

- processes any PUBLIC or GLOBAL directives by inserting the declared symbols and their corresponding definitions in a Linker symbol table.
- processes any GLINK directives by inputting the specified global file's symbols into the Linker symbol table.

In pass 2 the Linker:

- fills in the addresses of any EXTERNal symbols, and generates error messages for all undefined symbols.
- if *filespec2* is specified, saves the resulting absolute file.
- processes any GLOBAL directives, by creating a global file.

Maximum Sizes:

The Linker links up to 200 external program sections (PSECTs).

The Linker Symbol Table holds at least 2,000 external symbols. However, if you use symbols smaller than the maximum size of 10 characters, the Symbol table can hold more.

The maximum absolute object file which the Linker creates can be as large as TRSDOS will load. See your TRSDOS manual.

Chapter 6/

ALDS File Transfer System (ALTRAN)

The ALTRAN program transfers files created under the ALDS package between any two TRS-80s (Model I, II, III, 4, 12 or 16) by either hardwire or modem. It transmits or receives object code, source code or data files. This chapter explains how files can be transferred between the Models III and 4. If you wish to transfer files on your Model I, II, 12 or 16, you will need the Model II ALTRAN package.

Since ALTRAN was developed specifically for files created with the ALDS package, we cannot guarantee that it will accurately transfer files created with other software.

Set-up

You can use two types of connections in ALTRAN: modem or hardwire.

Modem

The standard RS-232C Interface is appropriate if you plan to transfer files via a modem. You can use any TRS-80 modem provided that both ends can use the same baud rate and can communicate with each other (i.e. both can't be originate only or answer only modems).

See your Radio Shack modem operation manual for installation instructions.

Hardwire

If you plan to hardwire the Models III and 4, you will need:

Model III/4 to Model III/4	26-1408 RS-232C Cable 26-1496 Adapter Box 26-1497 12″ Extension Cable

Baud Rate

The factory sets the baud rate at 300 for all ALTRAN packages. As a general rule with most systems, the quality of transmissions is directly proportional to the

ratio of distance versus baud rate. In other words, the higher the baud rate, the shorter the distance allowed.

If you want to change the factory-set baud rate, you can use the PATCH utility. The patch for the Model III is:

```
PATCH ALTRAN/CMD (ADD=5200,FIND=55,CHG=nn) (ENTER)
```

where nn is the value in *Table 9*.

The patch for the Model 4 is:

```
PATCH ALTRAN/CMD (D00,04=nn:F00,04=55) (ENTER)
```

where *nn* is the value in *Table 9*.

Table 9/ Baud Rate Change Table

Baud Rate Desired	Model III and 4 Patch
75	11
110	22
150	44
300	55
600	66
1200	77
1800	88
2400	AA
3600	BB
4800	CC
7200	DD
9600	EE

The following table shows the recommended maximum distance (hardwired) versus baud rate for high quality transmissions. The factors that govern this table are for worse case non-modem situations.

Note: All values are approximate.

Baud Rate	Maximum Model III/4 Distance
75 – 300	500 feet
600 – 1200	50 feet
1800 – 3600	25 feet
4800 +	10 feet

Loading ALTRAN

To load ALTRAN from TRSDOS Ready, type:

```
ALTRAN (ENTER)
```

The program immediately displays the menu of operations and the settings of the RS-232C parameter list.

Figure 3 shows the menu of ALTRAN.

```
Tandy Systems Design Model 4 File Transfer Program
Copyright 1982,1983 Tandy Corp. Ver.. vv.rr.pp
300 baud, 8 data bits, no parity, 1 stop bit

1 - Transmit OBJECT file          2 - Receive OBJECT file
3 - Transmit SOURCE file          4 - Receive SOURCE file
5 - Transmit DATA file            6 - Receive DATA file

7 - Transmit via COMMAND file
8 - Receive via received COMMAND file or WILDCARD mask
9 - Enter 'Mini-Terminal' Mode
Q - Return to TRSDOS
```

Figure 3. THE ALTRAN MENU

Operations 1, 3, 5, and 7 are the transmission modes. The one you select depends on the type of file you want to transfer.

Operations 2, 4, 6 and 8 are the receiving modes. Again, the one you select depends on the type of file you'll be receiving.

You can use operation 9, 'Mini-Terminal', for terminal to terminal communications.

See COMMAND FILE for instructions on creating a command file.

The transmit WILDCARD operation is only available on Model II ALTRAN.

Operation

Once you load ALTRAN, as a final test to ensure both transmitting and receiving stations are operational, send a test message via Operation 9 - 'Mini-Terminal' mode in both directions. ALTRAN must be able to communicate in both directions to function properly.

Beginning the Transmission

1. Determine the type of file you want to transfer.

   Use operations 1 and 2 (OBJECT file) for:

   - ALDS object files (both executable and relocatable)

   Use operations 3 and 4 (SOURCE file) for:

   - ALDS source files
   - Series I Editor/Assembler source files (the file transfer system will write the file to the receiving station in ALDS source file format).

Use operations 5 and 6 (DATA file) for:

- fixed length record files (assembler global files, application program data files, assembler listing files, and non-ALDS source files such as some BASIC files.)

  Please note some non-ALDS Model III and Model 4 files, with an EOF byte which is not zero (as displayed in the directory) may not transfer properly. This is because ALTRAN will change the EOF byte to zero, thereby changing the length of the file.

Note: When transferring files from one model to another, you must consider the differences between systems. It is unlikely that the same object file can run on all models due to the difference in ROM and RAM addresses, etc. In addition, we can't guarantee successful transfer of file formats not used by ALDS, even though some files may transfer.

2. Select an operation.

   The number of the operation you choose depends on the type of file you want to transfer and whether you're the transmitting or receiving station. If you are the transmitting station and plan to send an OBJECT file, type 1 (ENTER) in response to the Which? prompt. The receiving station enters a 2 in answer to the Which? prompt. (The order in which the stations enter their operations doesn't affect the transfer, i.e. the receiving station can specify operation 2 before the transmitting station specifies operation 1.)

3. Specify a file.

   After each station selects an operation, ALTRAN prompts for a filespec with File Name? . . . . . .

   Both stations should enter the name of the file. Be sure to include the extension and drive number (if not the system drive).

   If you choose Operation 7, ALTRAN prompts the the transmitting station with File Name? (See COMMAND FILE later in this section on how to create one.)

   Using Operation 8, ALTRAN prompts the receiving station with Drive Number?. To avoid the possibility of accidently writing over a file, the receiving station should use a blank formatted diskette.

During the Transmission

When operation actually begins, the transmitting station immediately sends the first block of the file. During transmission, the display reads:

```
'Transmitting Block 1 '
```

As each block is sent, it increments the block number by one. (Depending on the baud rate and LRL, this increment may take from a fraction of a second to about a minute.) This message is not displayed if you are transferring a null file (EOF and no other information).

At the same time, the message:

```
'Receiving Block 1'
```

appears on the receiving station's video display. This indicates that the station is *ready to receive* the first block of the file, and is not necessarily receiving it. After each file is received, the block number displayed is one more than what was actually received.

This message may not come on immediately in operations 5 and 6 because the transmitting station must first send the file type and the logical record length of the file before the receiving station can be readied to receive the first block of the file.

After receiving each block, ALTRAN increments the block number by one, then stores that block to disk under the filespec named in step 3.

If the receiving station is not ready, the transmitting station keeps trying to transmit a block until it receives an acknowledgement or until the (BREAK) key is pressed.

Once transmission actually takes place, the receiving station expects a block until it receives an EOF marker or until the (BREAK) key is pressed. If the (BREAK) key is pressed during transmission of a file, the file won't be valid or useable.

On SOURCE file transfer only, prior to transmission, ALTRAN at the transmitting station checks the first line of the file for an existent line number. If there is none, it automatically adds line numbers to the entire file before sending the file.

The receiving station strips the bytes corresponding to the line number from all lines of the transferred file as it stores them.

In the 'Mini-Terminal' mode, you can transmit any character except (↑) and (BREAK) and the receiving station will output the character to the screen. However, not all of the TRS-80 models (at the receiving station) interpret the characters in the same way. One model may interpret the control characters differently and display a character other than what was transmitted. On other models, certain characters may activate features such as dual routing, reverse video, or 40-character mode. And, the Models III and 4 won't output tabs.

Ending the Transmission

After all transmissions are complete for operations 1 through 8, ALTRAN returns to the menu, unless you are sending a command file ending with operation 9.

To escape from the menu, type (Q) (ENTER). To exit the 'Mini-Terminal' mode, press (↑) on the Model III/4.

If you want to transfer another file, return to Step 2.

When an Error Occurs

If an error occurs at one station (not including 'Unknown or unuseable baud rate was patched' which automatically returns to TRSDOS Ready), ALTRAN will cease transmission, close the file, return a descriptive error message, and display the following:

Further transmission not possible
Press (ENTER) to go into Mini-Terminal mode
Press (ENTER)(↑) to return to menu
Press (BREAK) to exit to TRSDOS Ready

When an error occurs, the computer making the error will send a cancellation message ((CLEAR)(X) or 18H) which the other computer will display minus the descriptive error message.

Under certain circumstances, such as transmitting or receiving the LRL, a byte of data, or the checksum, this feature is disabled so that a legitimate 18H won't cause a cancellation and an error message won't be displayed. Therefore, if your computer remains idle for a period of time (the length depending upon your baud rate), you can assume an error has occurred. Press (BREAK) to return to TRSDOS Ready.

Note: It is always a good idea for both stations to arrange to go to 'Mini-Terminal' mode if an error occurs. Because the station not causing the error isn't always informed of an error, you should return to 'Mini-Terminal' mode if your computer locks up for an unusual length of time.

Command File

A command file is an automatic input file. This file executes a series of operations with one command. By building a command file, you will be able to transmit several files with this one command.

You must enter the Editor to create a command file. The procedure is:

1. Load the Editor
2. Enter the Insert Mode
3. Enter the filespec you are sending
4. Tab over one position and enter the operation code number used for transmitting the file (1, 3, or 5)
5. Repeat steps 3 and 4 until all files are entered.
6. If you want to invoke Mini-Terminal mode, enter it last. A dummy filespec must precede it.
7. Exit insertion mode
8. Write the command file to disk. Do NOT use the line numbers option.

Example:

At TRSDOS Ready, type:

```
ALEDIT (ENTER)
```

to enter the Screen Editor. Then type I to enter the insertion mode.

In the insertion mode, type:

```
FILE1/SRC      3 (ENTER)
FILE2/OBJ      1 (ENTER)
FILE3/DAT      5 (ENTER)
DUMMY          9 (ENTER)
```

to create the command file.

When run, this command file transmits three files in a row with one input command. It transmits the first filespec, FILE1/SRC, as a source file, the second, FILE2/OBJ, as an object file, and the third, FILE3/DAT, as a data file. The last file, DUMMY, isn't transmitted. It invokes the 'Mini-Terminal' mode.

If for some reason you don't have ALEDIT, you may download the Command File from another computer, using the SOURCE File Transfer.

Technical Information

Definitions:

ACK = Acknowledgement of receipt of correct block or inquiry and request to transmit next block. (code 06H)

NAK = Acknowledgement of receipt of incorrect block and request for retransmission. (code 15H)

WAK = Acknowledgement of receipt of correct block, but wait before transmitting next block (so the computer may write out block). (code 1BH)

EOT = End of transmission of this file. (code 04H)

ENQ = Enquire for a ready to receive. (code 05H)

ETX = End of text. (code 03H)

CAN = Cancellation (aborts current transfer) (code 18H)

Algorithms

Object Files

ALTRAN transmits and receives OBJECT files as 256 byte, fixed length record (FLR) blocks.

It uses this algorithm to transmit OBJECT files:

1 open file for read
2 read a sector into a buffer
  if end of file, send EOT, receive ACK, and return to menu
3 display xmit block number
4 send ENQ

```
5     receive ACK
6     output sector
7     output checksum
8     receive ACK or NAK or WAK
        repeat block if NAK
        if WAK, wait for ACK
9     goto 2, "read a sector"
```

It uses this algorithm to receive OBJECT files:

```
1     open file for write
2     display received block number
3     receive ENQ
        if EOT, send ACK, close file and exit
4     send ACK
5     receive sector
6     receive checksum
7     output ACK,NAK,WAK
        repeat receive if NAK
8     send WAK
9     write sector
10    send ACK
11    goto 2, "display block number"
```

Source File

ALTRAN transmits SOURCE files as fixed length records (FLR) 256 on Models III/4.

It uses the following algorithm to transmit the SOURCE file:

```
1     open file for read
2     read in a line (if MOD III/4 strip bit 7 from line numbers). If a line
        number is not present on the first byte of the line, add a line number.
        Be sure the source does not have numbers in column 1. They may be
        accidentally deleted. If end of file, send EOT and receive ACK.
3     display xmit block number
4     send ENQ
5     receive ACK
6     send line length
7     output the line
8     output the checksum
9     receive ACK, or NAK, or WAK
        repeat line if NAK
        if WAK, wait for ACK
10    goto 2, "read in a line"
```

It uses this algorithm to receive the SOURCE file.

```
1     open file for write
2     display receive block number
```

```
3     receive ENQ
        if EOT, send ACK, close file and exit
4     send ACK
5     receive line length
6     receive the line
7     receive the checksum
8     send ACK, or NAK, or WAK
        repeat receive if NAK
        send WAK
        write the line, without the line number
9     goto 2, "display block number"
```

Data File

The ALTRAN program sends DATA files as fixed length records (FLR) on the Models III/4.

It uses the following algorithm to transmit the DATA file:

```
1     open file for read
2     send file type (F) and file's LRL
3     read in one record of data
        if end of file, send EOT, receive ACK, and exit.
4     display xmit block number
5     send ENQ
6     receive ACK
7     send data record length
8     send data
9     send checksum
10    receive ACK or NAK or WAK
        repeat xmit if NAK
        if WAK, wait for ACK
11    goto 3, "read in one record"
```

It uses this algorithm to receive the DATA file:

```
1     receive file type (F) and file's LRL
2     open file for write with those parameters
3     display receive block number
4     receive ENQ
        if EOT, send ACK, close the file and exit
5     send ACK
6     receive data record length
7     receive data
8     receive checksum
9     send ACK or NAK or WAK
        repeat receive of NAK
        send WAK, write data record
```

10 send ACK
11 goto 3, "display block number"

Indirect Command File

ALTRAN uses this algorithm to transmit the COMMAND file:

1 open IND file for read
2 build a text line
  if end of file, send ETX, wait for ACK, and return to menu.
3 send ENQ
4 receive ACK
5 send file name and function
6 send checksum
7 receive ACK or NAK or WAK
  if NAK, goto send ENQ
8 display file name
9 transmit file through functions 1, 3, or 5
10 goto 2, "build a text line"

It uses this algorithm to receive the COMMAND file:

1 receive ENQ or ETX
  if ETX, send ACK and return to main menu
2 send ACK
3 receive file name and function
4 receive checksum
5 send ACK or NAK or WAK
  if NAK, goto receive ENQ or ETX
6 display file name
7 receive file through functions 2, 4, or 6
8 goto 1, "receive ENQ or ETX"

Mini-Terminal Mode

ALTRAN uses the following algorithm to transmit and receive keyboard characters:

1 scan keyboard for character
  if escape character, exit mini-terminal mode
  if character, then display and output to RS-232C
2 scan RS-232C input
  if character, then display
3 goto 1, "scan keyboard"

Building an Adapter Connection

If you want to, you have the option to build your own adapter connection instead of buying a Radio Shack Adapter Box (Catalog Number 26-1496).

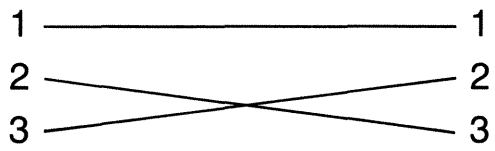
Required Materials

Model III/4	RS-232C Interface Board
	RS-232C Cable
	DB-25 Male Connector (2)

When hardwiring for Model III/4 to Model III/4, the pin connections are as shown below:

Figure 4. / Model III/4 Pin Connections

The pin connections are as shown:



Section II
ALDS Assembly Language

Chapter 7/
ALDS Assembly Language Syntax

This chapter describes how the ALDS Assembler interprets source lines. The next chapters list all the instructions available with ALDS.

An ALDS assembly language source line can contain up to four fields. They are:

- the label
- the instruction
- the operands
- the comment

The Label

The label is optional. It is a symbol which defines the location of the instruction immediately following it. For example:

```
NAME        LD        A,5
```

NAME is a symbol used as a label. The Assembler uses it to store the location of the LD A,5 instruction. For example, if LD A,5 is at location 5200H, the Assembler assigns the value 5200H to NAME and stores this in the symbol table.

The label must begin in column one (the first character in the line) or be followed by a colon. For example, this line produces a syntax error:

```
 NAME        LD        A,5
```

since the label NAME is not in column one.

However, this is acceptable:

```
 NAME:       LD        A,5
```

since NAME is followed by a colon.

Valid Symbols

A symbol can consist of up to ten of the following characters:

alpha characters
(A-Z) in either upper or lower case (the Assembler treats upper and lower case letters differently. "NAME", for example, is a different symbol than "Name").

numeric characters
(Ø-9) (the symbol cannot begin with a number).

special characters
the underscore (_)
the question mark (?)
the dollar sign ($)
the @ character

It may not contain a space character. These are examples of valid symbols:

Date? $B_? A1D2 B2345678

The following are reserved words. You cannot use them as ordinary symbols, since this conflicts with the way the Assembler notes register names, branch conditions, or the location counter value:

$	A	B	C	D	E	H	L	F	Z	P
M	I	R	V	AF	BC	DE	HL	SP	IX	IY
XH	XL	YH	YL	NC	NZ	PE	PO	NV		

Reserved words are reserved in both upper and lower case. For example, SP, sp, Sp, and sP are all reserved.)

The Instruction

The instruction is usually required. It can be either:

a Z8Ø mnemonic
(*Chapter 9*), which is an instruction to the microprocessor that the Assembler converts into a Z8Ø operation code.

an assembler directive
(*Chapter 8*), which is an instruction to the Assembler itself.

an extended Z8Ø mnemonic
(*Chapter 1Ø*), which the Assembler expands into a group of Z8Ø mnemonics.

a macro call
(*Chapter 8*), which the Assembler expands into one or more of the above types of instructions.

You can begin the instruction anywhere but in column one. If the line contains a symbol, there must be at least one space, tab, or colon between the instruction and the symbol.

For example, the Assembler interprets LDIR as an instruction in all of these lines:

```
SYMBOL  LDIR
SYMBOL      LDIR
 LDIR
            LDIR
```

However, in these two lines:

```
SYMBOLLDIR
LDIR
```

the Assembler interprets LDIR as part of the symbol field.

You can use either upper or lower case to indicate the instruction. For example, you can indicate the LDIR instruction as:

```
ldir
```

Of course, in the case of a macro call, you must be careful that you use the same case that you used when you defined the macro.

The Operands

Many instructions allow you to specify data as operands. Some instructions allow you to use a register name or a flag as an operand. Some allow you to indicate a specific value.

You must use at least one space or tab to separate the operands from the instruction. In these examples, A and 3 are operands:

```
SYMBOL LD A,3
 LD                A,3
 LD    A,3
```

However, this line produces an error:

```
SYMBOL LDA,3
```

since there is no space between the instruction and the operands.

Expressions

When specifying a certain value as an operand (such as "3" in the above example), you must use a valid assembler expression. The expression can consist of one or more terms connected by operators.

Terms

A term can be:

a number
The Assembler assumes the number is decimal (base 10) unless you use a base suffix or the RADIX directive. Changing number bases is described in the next chapter.

an ASCII character
You must enclose the character in single quotes. The Assembler will assemble it into its ASCII code.

a symbol
The Assembler fills in its value using the symbol table.

$ (the dollar sign character)
The Assembler interprets this character as the location counter's current value.

For example, each of these are valid terms:

152

which represents the decimal number 152 (unless you have used the RADIX directive described in the next chapter).

'A'

which represents the ASCII character code of decimal 65 or hexadecimal 41.

SYMBOL

which represents the value of SYMBOL.

$

which represents the current value of the Assembler's location counter.

Operators

The operators and their functions are listed on *Table 10*. If an asterisk (*) follows the function, the operator is unary (acts on one operand). Otherwise it is binary (acts on two operands).

Table 10/ Operators

OPERATOR	FUNCTION	PRIORITY
+	unary plus*	1
–	unary minus*	1
.NOT.	logical not*	1
.HIGH.or.MSB.	high order byte*	1
.LOW.or.LSB.	low order byte*	1
.BIT.	bit* (one shifted n bits to the left)	1
** or ^	exponentiation	2
*	multiplication	3
/	integer division	3
.MOD.	modulo	3
.SHR.	logical shift right	3
.SHL.	logical shift left	3
.RR.	logical rotate right	3
.RL.	logical rotate left	3
+	addition	4
–	subtraction	4
.AND.	logical and	5

.OR.	logical or	6
.XOR.	logical exclusive or	6
.ABS.	absolute value*	7
.EQ. or =	equals	7
.GT. or >	greater than	7
.GE.	greater than or equal to	7
.LT. or <	less than	7
.LE.	less than or equal to	7
.RES.	result* (ignore overflow)	7
.SGN.	sign*	7
.UGT.	unsigned greater than	7
.UGE.	unsigned greater than or equal to	7
.ULT.	unsigned less than	7
.ULE.	unsigned less than or equal to	7

Examples:

```
4321H.SHL.3
```

returns the number 4321H shifted three bits to the left.

```
4321H.SHL.1
```

returns the number 4321H shifted one bit to the left.

```
.RES.(7FFF*7FFF)
```

multiplies 7FFFH by 7FFFH and returns the result. (The RES. operator causes the Assembler to ignore the overflow error this operation would normally cause.)

```
.SGN.SYMBOL
```

returns a -1 if SYMBOL is negative, 0 if it's zero, or 1 if it's positive.

Priority of Operators

When you use multiple operators, the Assembler evaluates them using the priority number indicated. If two operators have the same priority, the Assembler evaluates them from left to right.

You can use parentheses to change the priority of operators.

Examples:

```
4+4/2
```

The division is performed first. (Division is priority 3; addition is priority 4.)

```
(4+4)/2
```

The addition is performed first.

```
4*4/2
```

The multiplication is performed first.

Note: You must use parentheses to separate two operators which are both enclosed in periods. For example:

LD HL,5.AND..ABS. −4 is illegal
LD HL,5.AND.(.ABS. −4) is valid

Using Relocatable or External Symbols in Complex Expressions

When using complex expressions, i.e., expressions using more than one term, you need to be careful about using symbols which are:

- external (defined in an external program section), or
- relocatable (defined in a relocatable program section).

Table 11 shows which types of complex expressions allow relocatable or external symbols, and the type of value which the Assembler will return. If the expression is not on this table, you cannot use a relocatable or external symbol. Under no conditions can you use relocatable and external symbols within an absolute program.

TABLE 11/ Complex Expressions Allowing Relocatable or External Symbols

Definition of Terms:

ABS is an absolute constant, symbol or expression
EXT is an external symbol or expression
REL is a relocatable symbol or expression
ALL is any of the above

COMPLEX EXPRESSION	RESULTING TYPE
EXT + ABS	EXT
ABS + EXT	EXT
EXT − ABS	EXT
REL + ABS	REL
ABS + REL	REL
REL − REL	ABS
REL − ABS	REL
ALL.EQ.ALL**	ABS
REL.GE.REL	ABS
REL.GT.REL	ABS
REL.LT.REL	ABS
REL.LE.REL	ABS
REL.UGE.REL	ABS
REL.ULE.REL	ABS
REL.UGT.REL	ABS
REL.ULT.REL	ABS
.HIGH.REL	*

.MSB.REL	*
.LOW.REL	*
.LSB.REL	*
.HIGH.EXT	*
.MSB.EXT	*
.LSB.EXT	*
.LOW.EXT	*

*these expressions cannot be used as a term in a larger expression. Also, they must be used only where an 8-bit quantity is expected.

**the terms must be of the same type (absolute, external, or relocatable) in order to be equal. Two externals are never equal, including the special case of comparing an external to itself.

Other Special Conditions Regarding Relocatable or External Expressions

These are some additional considerations you need to be aware of when using relocatable or external expressions:

- If you attempt to fit a relocatable or external value outside of the range of −256 to 255 into an 8-bit field, you will not get an error message. The Assembler will store the low order byte into this field. (Absolute values outside this range generate an error message.)
- You can use the .HIGH., .MSB., .LOW., or .LSB. operators only where an 8-bit value is expected. If you use one of these operators where a 16-bit value is expected, the Assembler will either give you an error message or unpredictable results.
- If you use the .HIGH. or .MSB. operator, the Assembler saves the entire value in the object code so it can properly compute the carry into the high order byte (which might result from adding the load address to the expression value during linking)

The Comment

The comment is an optional way to document your program. The Assembler ignores it.

To insert a comment at the end of a line, you must precede it with a semicolon. For example, all of these lines contain comments:

```
NAME      LD    A,3;This is a comment
LDIR;AND SO IS THIS
;and here is another comment
          LD    A,3    ;and another
```

The Assembler ignores every character following the semicolon. However, this line produces a syntax error:

```
NAME      LD    A,3    This is an illegal comment since there
is no semicolon preceding the comment.
```

Another way to insert a comment is by typing an asterisk (*) in column one. The Assembler ignores all lines which follow until it encounters another * in column one.

For example:

```
LD    A,3
*This begins a comment section which the Assembler will
ignore.
comment, comment
comment, comment
This is the last line in the comment section
*

ADD   B
```

the Assembler ignores all lines between LD A,3 and ADD B.

Chapter 8/
Assembler Directives

Assembler directives are commands to the Assembler oror, in a few cases, the Linker. They are not instructions to the Z-80 Microprocessor and are not a part of your executable program. Generally, you can type them in the same form as the Z80 mnemonics and insert them throughout the program.

This chapter contains two parts. Each part contains sample programs or segments of programs which are used to help explain the use of assembler directives. You will not be able to run these sample programs or program segments on your computer.

Part A is a tutorial. It describes the different types of directives — what their purpose is and how they inter-relate with each other in the program.

Part B is a reference. It contains an alphabetical listing of each directive. Each listing gives the syntax, a definition, and an example use.

Introduction to Assembler Directives

ALDS assembler directives allow you to:

- Change Number Bases
- Define Symbols
- Define Data
- Define Storage
- Initialize the Location Counter
- Manipulate the Location Counter
- Terminate or Hold the Assembly
- Use External Symbols
- Create Index Sections
- Define Macros
- Create a Conditional Section
- Control the Assembly Listing

Changing Number Bases

The Assembler recognizes number bases 2 (binary), 8 (octal), 10 (decimal) and 16 (hexadecimal). The default is base 10.

You can change the default with the RADIX instruction. For example:

```
RADIX     8
```

tells the Assembler to evaluate all subsequent numbers as base 8.

Using a base suffix identifies a base for a particular number. The base suffixes are:

H	Hexadecimal
d	Decimal
b	Binary
Q or O	Octal

For example, in this instruction:

```
LD        A,33H
```

the 33 is evaluated as a hexadecimal number, regardless of which default base you are in.

You can use upper case "d" and "b" suffixes. Be careful with this, though, since the hexadecimal base interprets "D" and "B" as numbers. For example, in base 16, "1b" is a binary 1; "1B" is hexadecimal 1B.

Defining Symbols

Defining symbols allows you to refer to data or memory addresses symbolically. This makes the program easier to read and revise.

ALDS allows you to use a symbol to label the location of any Z80 instruction and most directives. It also contains these directives which define symbols:

- EQU — equates a symbol to a constant value
- DEFL — defines a symbol to a variable value

For example:

```
NUMBER    EQU     12          ;EQUates NUMBER to 12
LOOP      LD      A,NUMBER    ;loads A with 12
            •
            •
          LD      HL,LOOP     ;loads HL with LOOP
```

This program uses NUMBER and LOOP as symbols. The first line EQUates NUMBER to 12. The next line uses NUMBER as an operand.

LOOP will define the location of LD A,NUMBER. The last line uses LOOP to specify this location.

Defining Data

Data definition directives insert data into RAM. ALDS contains these data definition directives:

- DEFM — defines string data
- DEFE — defines "encrypted data"
- DEFT — defines data and includes a length byte
- DEFB — defines a byte
- DEFW — defines a word
- DEFR — defines a Roman Numeral
- DATE — defines the current date
- TIME — defines the current time

For example:

```
          LD       HL,TABLE
          CALL     PRINT      ;PRINT TABLE ON VIDEO SCREEN
TABLE     DEFM     'THIS BEGINS A TABLE OF DATA'
          DEFB     0DH
```

DEFM inserts the ASCII codes for THIS BEGINS A TABLE OF DATA in the next 27 locations. The symbol TABLE defines the beginning of this location.

The subroutine PRINT is used as an example for a routine that displays the specified information on the screen.

Defining Storage

Defining storage reserves an area of RAM which you can use for such functions as inputting and outputting data. ALDS contains these storage definition directives:

- DEFS — reserves RAM
- FILL — sets the "fill mode" so that DEFS will fill the reserved area with zeroes
- NOFILL — ends the fill mode

For example:

```
          LD         HL,BUFFER
          LD         B,20
          CALL       KEY        ;Keyboard input into
                                ;BUFFER area
           .
           .
          FILL
BUFFER    DEFS       20         ;reserves the next 20 bytes
          NOFILL
```

FILL sets the fill mode. DEFS reserves the next 20 locations for storage and fills them with zeroes. NOFILL unsets the FILL mode.

Initializing The Location Counter

The Assembler contains a "location counter" which it uses to:

- assign locations to each executable instruction, and
- define the symbols which identify these locations

The locations it assigns are either absolute or relocatable depending on how you initialize the counter.

Initializing The Location Counter To An Absolute Location

To initialize an absolute location, you must use PSECT:

```
        START     PSECT    7000H
7000    NUM       LD       A,5        ;begin assembling at 7000H
7002              PUSH     A
7003              LD       A,6
•                 •
•                 •
                  END      NUM
```

This program section initializes the counter to an absolute 7000H. The Assembler then assigns all the instructions absolute locations, beginning with 7000H.

The Assembler saves this assembly on disk as an "absolute object file". You can load it in the TRSDOS Ready mode simply by typing the filespec followed by (ENTER). Each instruction will load into the same (or "absolute") memory location the Assembler assigned it.

Many other assemblers, such as the Series I, use ORG rather than PSECT to accomplish the same task. If you want to assemble such a program with ALDS, you need to change the first ORG to PSECT.

Initializing The Location Counter To A Relocatable Location

PSECT without an argument initializes the location counter to a relocatable 0000 (the ' signs indicates that the locations are relocatable, rather than absolute):

```
                  PSECT
0000'   NUM       LD       A,5        ;begin assembling at
                                      ;relocatable zero
0002'             PUSH     A
0003'             LD       A,6
•                  •
•                  •
                  END      NUM
```

The Assembler saves this assembly on disk as a relocatable, rather than absolute, file. You cannot load a relocatable file. You need to use the Linker to convert it into an absolute file.

For example, if the name of the assembled relocatable file is PROG/REL, this Linker command:

```
ALLINK PROG PROG $=7000 (ENTER)
```

assigns absolute locations beginning with 7000H to all the instructions in PROG/REL. It does this by adding 7000H to each relocatable location. The resulting program is saved as an absolute file named PROG/CMD.

Manipulating The Location Counter

There are several instructions which manipulate the counter within a program section. They are:

- ORG — changes the value of the counter
- LITORG — changes the value of the counter and allows room for literal operands
- SETLOC — manipulates the counter for symbols only
- RESLOC — ends the SETLOC manipulation

For example:

```
                PSECT   7000H
7000    BEGIN   LD      A,5     ;begin assembling at 7000H
7002            LD      B,2
        SECOND  ORG     8000H
8000            LD      HL,ADD  ;increment counter to 8000H
8002            PUSH    AF
                .
                .
                END     BEGIN
```

This program section initializes the counter to an absolute 7000H. The Assembler begins assigning consecutive absolute addresses until it reaches ORG, which changes the value of the counter to 8000H. The Assembler assigns 8000H to the next instruction and continues again sequentially.

Since the above program is absolute, ORG's parameter sets an absolute location of 8000H.

In the relocatable mode, ORG's parameter sets a relocatable location of 8000H. This means that when you link the program, 8000H serves as an offset to the program's absolute start address.

For example, assume you assemble the same program in the relocatable mode. The Assembler assigns it these locations:

```
                    PSECT
0000'   BEGIN       LD      A,5     ;begin assembling at
                                    ;relocatable zero
0002'               LD      B,2
        SECOND      ORG     8000H
8000'               LD      HL,ADD  ;increment counter to
                                    ;relocatable 8000H
8002'               PUSH    AF
                    •
                    •
                    END     BEGIN
```

Now assume you link the relocatable file to the absolute start address of 6000H. The Linker assigns it these addresses:

```
                    PSECT
6000    BEGIN       LD      A,5     ;begin assembling at
                                    ;relocatable zero
6002                LD      B,2
        SECOND      ORG     8000H
E000                LD      HL,ADD  ;increment to
                                    ;relocatable 8000H
E002                PUSH    AF
                     •
                     •
                    END     BEGIN
```

Notice that here, ORG 8000H offsets the absolute start address of 6000H. This causes the absolute address following ORG to be E000H (6000H + 8000H).

Assembly Termination Or Hold Instructions

ALDS contains several directives which terminate or hold the assembly. They are:

- END — ends the assembly and saves the output object file
- QUIT — quits the assembly
- NOEND — ends assembly of a non-executable "load-only" program
- STOP — temporarily halts the assembly

For example, all of the above programs contain an END directive. This tells the Assembler to end the assembly, store the assembled file, and return to TRSDOS Ready.

In most programs, you'll want to use a parameter with END to specify the transfer address (the address of the first executable instruction in the program). The Assembler then stores the transfer address so that when loaded, the program immediately begins execution.

Program Sections

All the above programs are "program sections". You can store several relocatable program sections in the same file.

For example:

```
        MAIN    PSECT
0000'   BEGIN   LD      A,3         ;begin first PSECT
•                 •
0500'           RET
        SUB1    PSECT
0000'           LD      HL,DATA     ;begin second PSECT
•                 •
0100'           RET
        SUB2    PSECT
0000'   LOOP    LD      B,10        ;begin third PSECT
•                 •
0200'           SVC     36
                END     BEGIN
```

Since each section is independent, it must declare its symbols "PUBLIC" (discussed below) for another section to use them. Otherwise, two sections may not share the same symbols. (Only the MAIN program can use BEGIN; only SUB2 can use LOOP; and DATA must be defined in SUB1.)

Notice the Assembler initializes each program section to a relocatable 0000. Now assume you link the program to an absolute start address of 7000H:

```
7000    BEGIN   LD      A,3         ;begin first PSECT
•                 •
7500            RET
7501            LD      HL,DATA     ;begin second PSECT
•                 •
7601            RET
7602    LOOP    LD      B,10        ;begin third PSECT
•                 •
7802            CALL    LIST
                END     BEGIN
```

The Linker assigns each relocatable program section an address immediately following the preceding one.

Using External Symbols

ALDS allows two or more program sections to share the same symbols. For example, you could write and test several independent subprograms — such as PAYROLL, PAYABLES, RECEIVABLES, and INVENTORY. You could then mix and match them into separate application packages.

ALDS offers two ways of doing this:

1. By linking the programs into one file
2. By creating a "global symbol file"

The first is more common. The second is for special applications such as overlays where you want to use only the symbol definitions of an external program, but not the entire program itself.

1. Combining Program Sections

For combining program sections, ALDS offers these directives:

- PUBLIC — declares symbols public
- EXTERN — declares symbols external
- LINK — appends an outside program file

These are actually directives to the Linker, as well as the Assembler.

As an example, assume you want to combine a subprogram named PAYROLL with a main program named ACCTG. You want both programs to share the same symbols. This is how you could go about it:

a. Declare the symbols you want shared.

You do this by using the PUBLIC or EXTERN directives at the beginning of your program. In the PAYROLL subprogram:

```
PAYROLL     PSECT
            PUBLIC    SUBPAY,MENU     ;SUBPAY and
                                      ;MENU are for
                                      ;PUBLIC use
                                      ;
            EXTERN    STORE1          ;STORE1 is in
                                      ;an EXTERNal
                                      ;PROGRAM
                                      ;
SUBPAY      CALL      CLS             ;defines SUBPAY
                                      ;and clears
                                      ;screen
            LD        HL,MENU
            CALL      PRINT           ;Print MENU
            LD        HL,STORE1
            CALL      PRINT           ;Print STORE1
            JP        EXIT            ;Jump to TRSDOS
MENU:       DEFM      'THIS BEGINS PAYROLL FOR'
            DEFB      0DH
                                      ;defines
                                      ;MENU
CLS:
;
; The routine to clear the screen should be placed here
;
```

```
            RET
PRINT:
;
; The routine to display a line should be placed here
;
            RET
EXIT:
;
; The routine to return to TRSDOS should be placed here
;
            JP          $
            END
```

The definitions for the symbols SUBPAY and MENU are declared PUBLIC. This means another program can use the same definitions.

The definition for STORE1 is declared EXTERNal. This means that although the existing program uses STORE1, an external program defines it.

In the ACCTG program:

```
ACCTG       PSECT
            PUBLIC      STORE1          ;STORE1 is for
                                        ;PUBLIC use
                                        ;
            EXTERN      SUBPAY,MENU     ;SUBPAY and
                                        ;MENU are in
                                        ;EXTERNal ;programs
                                        ;
MAIN        CALL        SUBPAY
STORE1      DEFM        'ABC DRUGS'
            DEFB        0DH
;
;This part of the program defines other stores
;
            LINK        'PAYROLL/REL'   ;insert
                                        ;PAYROLL/REL
                                        ;file
            END         MAIN
```

STORE1 is declared PUBLIC. This means that this program defines STORE1 and another program can use STORE1's definition.

SUBPAY and MENU are declared EXTERNal. They are used in this program but are defined in an external program (namely, PAYROLL).

If you want to try this exercise, use the ALDS Editor to insert the above two program files. Save the first as PAYROLL/SRC and the second as ACCTG/SRC.

b. Insert a directive to combine the programs

Notice LINK at the end of the ACCTG program. This tells the Linker to link the assembled code of PAYROLL at the end of ACCTG.

c. Assemble the programs

Assemble both the PAYROLL and ACCTG source program files in the normal way. In the TRSDOS Ready mode, type:

```
ALASM PAYROLL PAYROLL (ENTER)
ALASM ACCTG ACCTG (ENTER)
```

The Assembler creates two relocatable files — PAYROLL/REL and ACCTG/REL.

The Assembler marks every occurrence of the PUBLIC, EXTERN, and LINK directives, as well as every occurrence of EXTERNal symbols. However, you will need to use the Linker to complete the processing of these directives.

d. Link the programs

To link PAYROLL to ACCTG, you can use this Linker command at TRSDOS Ready:

```
ALLINK ACCTG/REL ACCTG $=5200 (ENTER)
```

The Linker processes the LINK, PUBLIC, and EXTERN directives and assigns the entire file absolute addresses beginning with 5200H. This is done in two passes. In pass 1 the Linker:

- processes the LINK directive by linking PAYROLL/REL to the end of ACCTG/REL
- assigns the entire file absolute addresses
- creates a Linker Symbol Table which contains the definitions of all the symbols declared PUBLIC.

In pass 2, the Linker:

- fills in the values of all EXTERN symbols (using the Linker Symbol Table created in pass 1)
- saves the resulting program as ACCTG, an absolute object file.

e. Executing the program

You now have an absolute file, ACCTG/CMD, which consists of both ACCTG/REL and PAYROLL/REL. To execute it, type at TRSDOS Ready:

```
ACCTG (ENTER)
```

Note: In order for this program to execute you must insert the CLS, PRINT and EXIT routines. Refer to your TRSDOS manual for information on how to execute these routines.

2. Creating A Global File

Creating a global file is useful if you want to conserve memory by "overlaying" one program on top of the other. To create and use a global file, ALDS offers these directives:

- GLOBAL — declares symbols global
- EXTERN — declares symbols external
- GLINK — tells the Linker to use a global file
- EXT — tells the Assembler to use a global file

As an example, assume you want to create a file name MAIN which consists of a number of subroutines, such as printing lines on the display.

You also want to create several accounting system files, one of which is LEDGER. Users will use only one of these accounting systems at a time. However, each accounting system uses routines from MAIN.

It is therefore necessary to have MAIN and LEDGER in memory at the same time. However, there is not enough room in memory for both programs.

The alternative is to "overlay" one program on top of the other. In this example, MAIN loads LEDGER. When loaded LEDGER overlays sections of MAIN which it will not use.

These procedures clarify how this is done:

a. Declare the symbols you want shared.

This time, you do this with GLOBAL and EXTERN directives. In the MAIN program:

```
MAIN        PSECT
            GLOBAL    PRINT
BEGIN       CALL      CLS             ;clear screen
            CALL      ROUTINE
;
;load LEDGER routine begins here
;
            LD        HL,LEDGERM
            CALL      LOADER          ;load LEDGER file
;
; PRINT routine begins here
;
PRINT       LD        B,(HL)
LOOP        INC       HL
            LD        A,(HL)
            CALL      PRINTCHR        ;print character
            DJNZ      LOOP
                                      ;print contents of
                                      ;register HL
            RET
LEDGERM     DEFM      'LEDGER'
            DEFB      0DH
ROUTINE     EQU       $
```

```
;
;This part of the program contains 18000 bytes
;of subroutines which only MAIN uses.
;Since LEDGER does not need them
;LEDGER will load into this area
;
            RET
LOADER:
;
; The routine which loads and runs a disk file
; should be placed here
;
            RET
PRINTCHR:
;
; The routine which displays characters should be
; placed here
;
            RET
CLS:
;
; The routine which clears the screen should be
; placed here
;
            RET
            END         BEGIN
```

The definition for PRINT is declared GLOBAL. When you assemble this program, the Assembler will create a global file named MAIN/GBL which contains PRINT's definition.

Notice that this program loads LEDGER. Also notice that it intends to load LEDGER on top of the ROUTINEs at the end.

This is the beginning of the LEDGER program:

```
LEDGER      PSECT
            EXTERN      PRINT
BEGIN       LD          HL,MENU
            CALL        PRINT
            JP          EXIT            ;Jump to TRSDOS
MENU        DEFT        'THIS BEGINS THE GENERAL LEDGER MENU'
            ;
            ; the rest of the very long
            ; LEDGER program goes here
            ;
            GLINK       'MAIN/GBL'
```

```
EXIT:
            ;
            ; The routine to return to TRSDOS should be
            ; placed here
            ;
            RET
            END         BEGIN
```

The definition for PRINT is declared EXTERN. Another program (MAIN) defines it.

(If you want to try this exercise, use the Editor to insert and save the first file as MAIN/SRC and the second as LEDGER/SRC.)

b. Insert a directive to search the global file

Notice the GLINK directive in the above program. This tells the Linker to look for PRINT's definition in a global file named MAIN/GBL.

c. Assemble the programs

Assemble MAIN and LEDGER in the normal way:

```
ALASM MAIN MAIN (ENTER)
ALASM LEDGER LEDGER (ENTER)
```

The Assembler creates MAIN/REL and LEDGER/REL.

d. Link the program which creates the GLOBAL file

You must link MAIN/REL before linking LEDGER/REL. This is because MAIN/GBL contains a GLOBAL symbol that must be available to link LEDGER/REL. Type:

```
ALLINK MAIN MAIN $=5200 (ENTER)
```

The Linker assigns absolute addresses to MAIN/REL beginning with 5200H and saves the resulting absolute file as MAIN.

It also processes the GLOBAL directive. This causes it to create a global file named MAIN/GBL. This file contains only a symbol table defining PRINT.

e. Link the program which uses the GLOBAL file

After creating MAIN/GBL, you can link LEDGER. Type:

```
ALLINK LEDGER LEDGER $=5300 (ENTER)
```

The Linker processes the EXTERN directive. This tells it to look for PRINT's definition in an outside file.

It then processes the GLINK directive. GLINK tells the Linker to look for PRINT's definition in a file named MAIN/GBL.

The Linker also assigns absolute addresses to LEDGER/REL beginning 5300H.

f. Executing the program

You now have two absolute program files:

MAIN and LEDGER

Type:

```
MAIN (ENTER)
```

MAIN loads beginning at address 5200H and begins executing. It then loads LEDGER beginning at address 5300H, which overlays the last portion of MAIN.

Note: In order for this program to run you must add the routines for CLS, EXIT, LOADER and PRINTCHR. Refer to your TRSDOS manual for information on how to execute these routines.

Notes And Options

ALDS offers several alternatives for linking programs:

- You can use INCLUDE rather than LINK. If you do this, you must include a source file rather than a relocatable object file. INCLUDE is a directive which the Assembler processes at assembly time. (See INCLUDE)
- You can use REF to reference only the symbol definitions of a source file only. (See REF)
- You can create indirect LINK files composed solely of LINK directives. By doing this, you can create several files containing different combinations of program sections. An example of this is PROG4 and PROGIII in *Chapter 1*.
- You can use EXT rather than GLINK to combine absolute, as well as relocatable symbols. EXT is a directive to the Assembler (whereas GLINK is a directive to the Linker)

Index Sections

ALDS contains directives which allow you to create an index section. They are:

- ISECT — begins an index section
- ENDI — ends an index section
- USING — associates an index register with an index section
- DROP — drops the index association established by USING

An index section is for EQUating symbols you want to use as offsets from an index register. For example:

```
PROG        PSECT     5000H
             •
            ISECT     1              ;begins index section 1
```

```
DATA        EQU       10H
            ENDI                        ;ends index section 1
            •
            LD        IX,4000H
            USING     1,IX              ;associates IX
                                        ;with the symbol
                                        ;in index
                                        ;section 1
            •
            LD        A,(DATA)          ;loads A indexed
                                        ;with IX, which
                                        ;will be (IX+
                                        ;DATA) or
                                        ;(4000H+10H)
            •
            DROP      1                 ;drops association
                                        ;of IX and index
                                        ;section 1
            •
            LD        A,(DATA)          ;loads A with (DATA)
                                        ;which is (10H)
```

Index section 1 (ISECT 1) equates DATA to 10H. USING associates all the symbol equations from ISECT 1 with index register IX. This means any time a symbol from ISECT 1 appears in the program, the Assembler generates an instruction to access memory with the indexed addressing mode (IX + the displacement value).

Later in the program, the Assembler encounters the symbol DATA (defined in ISECT 1.) The Assembler sets DATA as an offset to the IX register so that when you run the program, the processor will add DATA to the contents of register IX (The contents of register IX remains unchanged.)

Then the Assembler DROPs the association between IX and ISECT 1. After DROPping the association, the Assembler interprets DATA as simply DATA.

You car. temporarily clear a USING association and return to it later with:

- APUSH — saves the current USING associations in an Assembler stack
- APOP — restores the USING status saved with APUSH by "popping" it from the Assembler stack

For more information, see the individual definitions of each directive.

Macro Sections

ALDS allows you to define your own "macro" symbol as a group of Z80 instructions. Whenever the Assembler encounters this macro symbol, it expands it into its defined Z80 instructions.

For example:

```
START       PSECT     7000H
DISPLAY     MACRO     #L              ;begins macro
                                      ;section defining
                                      ;DISPLAY #L
                                      ;(#L is a dummy
                                      ;parameter)
            LD        HL,#L
            LD        B,(HL)
            INC       HL
            LD        A,(HL)
            CALL      PRINTCHR
            DJNZ      $-5
            ENDM                    ;ends macro section
;
BEGIN       DISPLAY   FIRST           ;call DISPLAY and
                                      ;pass it FIRST
;
            DISPLAY   SECOND          ;call DISPLAY and
                                      ;pass it SECOND
;
            JP        EXIT            ;Jump to TRSDOS
;
FIRST       DEFT      'THIS IS THE FIRST SENTENCE'
SECOND      DEFT      'AND THIS IS THE SECOND'
            END       BEGIN
```

The MACRO section begins with MACRO and ends with ENDM and in this example defines a MACRO named DISPLAY which displays a dummy parameter named #L.

The program then calls the DISPLAY macro and passes it the parameter FIRST. The Assembler expands this DISPLAY instruction into its macro definition, substituting FIRST for #L:

```
            LD        HL,FIRST
            LD        B,(HL)
            INC       HL
            LD        A,(HL)
            CALL      PRINTCHR
            DJNZ      $-5
```

Next, the program calls the DISPLAY macro passing it the parameter SECOND. This expands into:

```
            LD        HL,SECOND
            LD        B,(HL)
            INC       HL
            LD        A,(HL)
```

```
        CALL    PRINTCHR
        DJNZ    $-5
```

When you assemble this program, notice that the macro SECTION (not the macro CALL) is for the Assembler's memory only. It is not assembled as part of the executable program.

For more information on macros, see MACRO.

IF Sections

An "IF" section is a section of your program you only want assembled if a certain condition is true. ALDS offers these directives for conditional sections:

- IFT — assemble if operand is a true expression
- IFF — assemble if operand is a false expression
- IFZ — assemble if operand equals zero
- IFNZ — assemble if operand does not equal zero
- IFP — assemble if operand is positive
- IFM — assemble if operand is negative
- IFDEF — assemble if operand is a defined symbol
- IFUND — assemble if operand is an undefined symbol
- ELSE — assemble if IF condition is false
- ENDIF — end conditional section

For example, assume you want to create two versions of a program — a Model 4 version and a Model III:

```
START       PSECT     7000H
MOD4        EQU       0               ;defines MOD4
                                      ;(any value will do)
BEGIN       LD        B,3
;
            IFDEF     MOD4            ;assemble the following
                                      ;IF MOD4 is DEFined
                                      ;
            CALL      ABCD            ;
                                      ;
            ELSE                      ;assemble the following
                                      ;if ABCD is NOT defined
                                      ;
            JP        EXIT            ;Jump to TRSDOS
                                      ;
            ENDIF                     ;END the IF section
            END       BEGIN
```

IF the program defines the symbol MOD4, the Assembler processes CALL ABCD or ELSE it processes CALL EXIT.

The above program defines MOD4. The Assembler processes CALL ABCD, thereby producing a Model 4 version of the program. To have the Assembler return to TRSDOS, delete the MOD4 EQU Ø directive.

Assembler Listing Commands

Assembler listing commands change the way the Assembler processes the listing.

ALDS offers these listing commands:

- EJECT — ejects the printer listing to the next page
- VERSION — prints the time on the second line
- TITLE — prints a title on the third line
- HEADER — prints a heading on the fourth line
- PRINT — prints or does not print what you specify

See each directive listing for more information

Other Assembler Commands

The remaining Assembler commands are:

- ADISP — displays or prompts you for information
- NOLOAD — assembles in memory image form
- OBJ — specifies the object file name to use
- PATCH — fills the remaining bytes in a sector with FF's to create a patch area

Assembler Directives Reference

The following pages list the syntax and a brief definition of the assembler directives available with ALDS. This is a definition of the terms used in the syntax:

expression
a valid assembler expression. (See *Chapter 7.*)

absolute expression
an *expression* with an absolute (non-relocatable, non-external) value. This can include a relocatable symbol as long as the resulting value is absolute. See *Chapter 7.*

expression list
one or more *expressions*, separated by commas.

location
an *expression* designating an assembly location.

filespec
a TRSDOS file specification (see your Owner's Manual).

string
a string of ASCII characters. The entire line must be 78 characters or less.

symbol
a one to ten character name which you may reference in your program.

symbol list
one or more *symbols*, separated by commas.

ADISP

ADISP *'string^symbol'*
ADISP *'string~symbol'*

Displays or inputs certain parameters during the assembly of your program. You can specify one or both of these parameters:

(1) a *string* to be displayed
(2) a *symbol* to be displayed or input

Model 4: (CTRL)(T)
Model III: (SHIFT)(↓)(T)

inserts the ^ character which causes the Assembler to display the symbols value.

Model 4: (CTRL)(SHIFT)(F)
Model III: (SHIFT)(↓)(6)

inserts the ~ character which causes the Assembler to prompt you to input the symbol's value.

The Assembler executes ADISP during pass one only.

Example:

```
ADISP 'THE VALUE OF START IS ^START'
```

causes the Assembler to display: THE VALUE OF START IS followed by the value of the symbol START.

```
ADISP 'WHAT IS THE VALUE OF START ~START'
```

displays WHAT IS THE VALUE OF START? . . . You can then input a hexadecimal value for START.

```
ADISP 'This is my Message'
```

displays the message.

```
ADISP '^$'
```

displays the current address of the PC (program counter) register.

```
ADISP     'NEW ORIGIN ~STARTLOC'
ORG       STARTLOC
```

displays NEW ORIGIN? and prompts you to input a value for STARTLOC. The next instruction resets the location counter to the value you input. Note that ADISP 'NEW ORIGIN ~$' does *not* accomplish the same thing.

APOP

APOP PRINT
APOP USING
APOP PRINT,USING

Restores the PRINT or USING status which was saved by a previous APUSH instruction.

Example:

```
APOP USING
```

restores the USING status.

```
APOP USING,PRINT
```

restores both the USING and PRINT status.

APUSH

APUSH PRINT
APUSH USING
APUSH PRINT, USING

Pushes the current PRINT and/or USING status into an assembly stack. Use APOP to get this current status back from the stack.

You may nest APUSH only one level deep. That is, you can not use APUSH twice without an APOP in between them.

Examples:

```
APUSH USING
```

saves the USING status.

```
APUSH USING,PRINT
```

saves both the USING and PRINT status.

APUSH is useful when you want the Assembler to treat a certain section of your program differently. For example:

```
          •
MAIN      PRINT     ON
          •
          PRINT     CON
          •
          PRINT     SHORT
          •
          •
          APUSH     PRINT
          PRINT     OFF
          CALL      SUB1
          APOP      PRINT
          •
```

When the Assembler encounters APUSH PRINT, the current status of PRINT is ON, CON, SHORT (print the first 6 bytes of all source lines, including conditionals).

The Assembler PUSHes this status into an assembly stack and turns PRINT OFF. This causes it not to print any lines in SUB1.

The Assembler then POPs the PRINT ON, CON, SHORT status back from the stack, which causes it to restore the printing status.

DATE

symbol DATE

Stores the current date in memory beginning with the current address. The optional *symbol* labels this address.

The Assembler stores the date as a string in the form of Day of Week, Month Date, Year (Model 4) or MM/DD/YY (Model III).

For example, if today's date is Saturday, February 29, 1984:

```
DATE
```

stores SAT FEB 29, 1984 in Model 4 memory, or 02/29/84 in Model III memory.

DEFB

symbol DEFB *expression*
symbol DEFB *absolute expression list*
symbol DEFB *absolute repeat count% absolute expression*

Stores one or more one-byte expressions in memory beginning with the current address. The optional *symbol* labels this address. The optional *repeat* must be in the 1-255 range and will repeat a single *absolute expression* only.

```
TCONV     DEFB     NUM
```

stores NUM in the current memory address, defined as TCONV. NUM must be in the range of one byte numbers (− 256 to + 255 decimal).

If you use multiple expressions, all of them must be absolute. For example:

```
QSYM:     DEFB     7,9BH,BTABLE+3
```

stores decimal 7 at QSYM, the current memory address. Hexadecimal 9B and BTABLE + 3 are stored in the next two bytes. None of these bytes can be relocatable. BTABLE must be defined in the existing program unit.

```
DEFB 128%'*'
```

fills the next 128 bytes with the character '*'.

You can substitute BYTE or DB for DEFB.

DEFE

symbol DEFE *'string'*

Stores an "encrypted" *string* in memory beginning with the current memory address. The optional *symbol* labels this address.

Using DEFE makes it difficult for users to read the string by listing the object code. The first byte contains the unencrypted length of the string. The following bytes contains each character code XOR'd with 55H.

Example:

```
MESSAGE   DEFE    'hidden data'
```

stores 'hidden data' in the next 12 bytes and names the first byte MESSAGE. The first byte contains an 0BH (decimal 11). The next bytes contain codes for 'hidden data'.

DEFL

symbol DEFL *expression*

Defines *symbol* as *expression*. DEFL allows you to redefine a symbol in the same program. For example:

```
IMMED     DEFL     5
          ADD      A,IMMED
IMMED     DEFL     12
          ADD      A,IMMED
```

defines IMMED as 5 and adds it to the contents of register A. The next instruction defines IMMED as 12 and adds this to the contents of A.

Once you define a symbol with DEFL, you should not attempt to define it with EQU, EXTRN, or use it as a label.

DEFM

symbol DEFM *'string'*

Stores *string* in memory beginning with the current address. The optional *symbol* labels this address. For example:

```
MESSAGE     DEFM 'THIS IS THE MESSAGE'
```

stores 'THIS IS THE MESSAGE' in the next 19 bytes and names the first byte MESSAGE.

You can use these two special characters in the string:

- the tilde "~" (typed as CTRL SHIFT F on the Model 4 and SHIFT ↓ 6 on the Model III) to store a carriage return (hexadecimal 0D).
- the circumflex "^" (typed as CTRL T on the Model 4 and SHIFT ↓ T on the Model III) to toggle the high bit (80H) on and off.

For example:

```
TEXT        DEFM '^J^OHN BROWN~M STREET'
```

stores JOHN BROWN then a carriage return followed by M STREET in the next 19 bytes and flags the letter J by setting the high bit. J is stored as 0CAH, the code for J, plus 80H.

You can substitute ASCII for DEFM.

DEFR

symbol DEFR *'decimal number'*

Converts a *decimal number* into a Roman numeral *string* and stores it in memory beginning at the current address. The first byte contains the hexadecimal length of the Roman numeral string. The following bytes contain the ASCII codes for the Roman numerals.

The *decimal number* must be in the range of 1 to 65535. The optional *symbol* allows you to name the first address.

For example:

```
DEFR '1981'
```

stores MCMLXXXI in the next 9 bytes. The first byte contains 8, the length of the Roman numeral string.

DEFS

symbol DEFS *absolute expression*

Reserves *expression* bytes, beginning with the current address, for storage. The optional *symbol* names this storage area.

This Assembler will not insert anything in the reserved area unless the FILL mode is in effect (see FILL).

Example:

```
          ORG    7000H
BUF1      DEFS   100H
BUF2      DEFS   50H
BUF3      DEFS   10
START     LD     HL,BUF1
```

assigns BUF1 to location 7000H, BUF2 to 7100H, and BUF3 to 7150H. START begins execution at location 7160H, loading HL with 7000H.

You can substitute DS or BLOCK for DEFS.

DEFT

symbol DEFT *'string'*

Stores *string* in memory, beginning with the current address. The optional *symbol* labels this address. The first byte contains the length of the string. You may use the two special characters described under DEFM (the tilde and the circumflex).

For example:

```
MESSAGE    DEFT    'this is my message'
```

stores the number 12H (decimal 18) in the next byte of memory and 'this is my message' in the following 18 bytes; then assigns the name MESSAGE to the address of the first byte.

DEFW

symbol DEFW *expression*
symbol DEFW *absolute expression list*
symbol DEFW *absolute repeat count% absolute expression*

Stores one or more two-byte *expressions* in memory beginning with the current memory address. The optional *symbol* labels this address. The least significant

byte is stored first, followed by the most significant byte. The optional *repeat* must be in the 1-127 range and will repeat a single *absolute expression* only.

Examples:

```
MAXCNT      DEFW     1000
```

stores decimal number 1000 in the next two bytes and labels that location as MAXCNT. Since 1000 decimal is 03E8H, the first byte contains E8H and the second byte contains 03H.

```
            DEFW     3333,VAL
```

stores 3333 and VAL in the next four bytes. The same rules that DEFB uses for multiple expressions apply here. VAL must be defined in the existing program sections. Relocatable and external expressions may be used only if DEFW has a single, non-repeated expression.

```
DEFW  30%1000
```

fills the next 30 words with decimal 1000s, repeated 30 times.

You can substitute DW or WORD for DEFW.

DROP

DROP 1
DROP 2
DROP

Terminates the index register association, specified by USING, with ISECT 1, ISECT 2, or all the ISECTs. This allows you to change USING associations. For example:

```
DROP 1
```

The index register is no longer associated with ISECT 1.

```
DROP
```

The index register is no longer associated with any of the ISECTs.

EJECT

EJECT

During the assembly listing, causes the printer to go to the next page before listing the next instruction. The EJECT instruction will not appear in the listing.

END

END address

Ends the assembly of the source program. The optional *address* causes the Assembler to store the entry address of the program.

Examples:

```
END          7FFFH
```

ends assembly and stores address 7FFFH in the assembled file as the entry point of the program. When you load the assembled file, it will immediately begin execution at address 7FFFH.

```
END          BEGIN
```

ends assembly and stores the address defined by BEGIN as the entry address.

```
END
```

ends assembly of the program. Since no entry point is specified, the Assembler stores it as absolute zero. This is an invalid entry point for TRSDOS. Therefore, you will be able only to load this program with the LOAD command — not execute it.

ENDI

ENDI

Marks the end of an index section, initiated by ISECT.

ENDM

ENDM

Ends a macro definition, initiated by MACRO.

EQU

symbol EQU *expression*

Equates a *symbol* to an *expression*. For example:

```
START EQU 5200H
```

causes the symbol START to be equal to hexadecimal 5200.

```
POINT EQU 15+START
```

equates POINT to 5215, the sum of 15 and START.

Symbols defined by EQU may not be defined elsewhere in the program.

EXT

EXT *'filespec'*

Tells the Assembler that the absolute definitions for certain symbols in your program are contained in the specified global *file* (created by GLOBAL). Since

these symbols will have an established value at assembly time, you should not declare them EXTERNal or define them elsewhere in the program.

You can specify only one *filespec* per EXT instruction. It must have a /GBL extension. If you omit /GBL, the Assembler will automatically append it.

The EXT statement allows the programmer to have several absolute object files "talk" to each other. This requires considerable prior planning, but is useful and powerful.

Since EXT includes only the symbol definitions of the external program and not the program code, you will need to load the external program before attempting to use code in it.

For example:

```
EXT 'PROG1/GBL'
EXT 'PROG2'
```

tells the Assembler that your program contains symbols which are defined in PROG1/GBL and PROG2/GBL.

EXTERN

EXTERN *symbol list*

Declares that one or more *symbols* are not defined in the existing main program. They are defined externally in either:

- an external program section (which contains a corresponding PUBLIC instruction), or
- an external global file (which was created by a corresponding GLOBAL instruction).

For example:

```
EXTERN      LOOP1,LOOP2
```

declares that LOOP1 and LOOP2 are defined externally.

You may substitute EXTRN for EXTERN.

FILL

FILL

Causes any subsequent storage areas, initiated by DEFS, to be filled with zeros. Use NOFILL to turn it off.

For example:

```
          FILL
BUF1      DEFS        100
          NOFILL
BUF2      DEFS        200
```

BUF1 is filled with zeros. BUF2 is not filled with zeros.

You can use FILL only with DEFS instructions which reserve 255 or less bytes.

GLINK

GLINK *'filespec'*

Tells the Linker that the absolute definitions for certain symbols in your program are contained in the specified global file (created by GLOBAL). Your program must also contain an EXTERN instruction for each of the symbols referenced, to avoid undefined symbol errors.

You can specify only one *filespec* per GLINK instruction. It must have a /GBL extension. If you omit /GBL, the Linker will automatically append it.

GLINK accomplishes the same function as EXT, except it is an instruction to the Linker, rather than the Assembler. Because of this you need not have the external file written at assembly time, but you must have it loaded when you link the program.

For example:

```
GLINK 'PROG1'
GLINK 'PROG2'
```

tells the Linker that your file contains certain symbols which are defined in PROG1/GBL and PROG2/GBL.

GLINK must be the last instruction in your program before LINK, END, or another GLINK.

GLOBAL

GLOBAL *symbol list*

Declares one or more *symbols* as global and stores their values in a "global" file. Like PUBLIC, this permits another program section to use the same symbols. GLOBAL, however, goes one step further. It stores these symbols in a global file.

The global file will contain a symbol table only. It will define the absolute values of all the global symbols. If your program is absolute, the Assembler will create this global file. If your program is relocatable, the Linker creates it.

For example:

```
          PSECT       7000H
          GLOBAL      DATA
DATA      DEFM        'THIS STARTS A DATA TABLE'
```

declares that DATA is a global symbol and stores DATA's value, hexadecimal 7000, in a global file. Since this program is absolute, the Assembler will create the global file.

```
          PSECT
          GLOBAL      LOOP1,LOOP2
```

declares that LOOP1 and LOOP2 are global symbols to be stored in a global file. Since this program is relocatable, the Linker will create the global file.

The global file will have the same name as the assembled object file with the extension /GBL. You will be able to access this file with any other program, provided it has these two instructions:

(1) GLINK, which specifies that some symbols in the global file should be used, and
(2) EXTERN, which specifies which global (or external) symbol definitions should be used

or simply:

(1) EXT, which tells the Assembler to look for the definitions of some symbols in the global file

Symbols declared PUBLIC or GLOBAL must be defined on both passes, that is, not defined with REF, ASISP, or EXT. The Linker may flag these symbols as undefined.

Symbols defined with DEFL more than once should not be declared PUBLIC or GLOBAL. The Linker will flag these symbols as multiply defined.

HEADER

HEADER *'string'*

Prints the specified *string* on the fourth line of each page in the assembly listing until the Assembler encounters a new HEADER instruction. HEADER starts a new page.

For example:

```
HEADER  'Electronics'
```

causes the Assembler to print "Electronics" on the fourth line of each page in the assembly heading.

For the header string to appear on the first page, HEADER must precede all listed instructions in the program. Otherwise, it ejects to the next page before

printing the header string. TITLE, HEADER, and PRINT instructions are not listed.

You must specify a string when using HEADER. You may substitute HEADING for HEADER.

IFDEF

symbol IFDEF *symbol*

Assembles the following source lines IF the *symbol* is defined. IF NOT, the Assembler goes to the next ELSE or ENDIF directive. The optional *symbol* labels this directive.

```
IFDEF    SYMBOL
```

assembles the next lines IF the program defines SYMBOL. If not, the Assembler goes to the next matching ELSE or ENDIF. If the symbol is defined at all, it must be defined before the IFDEF.

The Assembler will not print the IF sections (instructions beginning with an IF directive and ending with ENDIF) unless PRINT CON is in effect. (See PRINT.)

All IF directives are nestable to six levels.

IFF

symbol IFF *expression*

Same as IFDEF except the *expression* must be false for the next lines to be assembled. For example:

```
IFF    5.GT.SYMBOL
```

assembles the next lines if 5 is not greater than SYMBOL.

IFM

symbol IFM *expression*

Same as IFDEF except the *expression* must be negative for the next lines to be assembled. For example:

```
IFM    SYMBOL
```

assembles the next lines if SYMBOL is a negative number.

IFNZ

symbol IFNZ *expression*

Same as IFDEF except the *expression* must not equal zero for the next lines to be assembled. For example:

```
IFNZ   SYMBOL
```

assembles the next lines if SYMBOL does not equal zero.

IFP

symbol IFP *expression*

Same as IFDEF except the *expression* must be positive for the next lines to be assembled. For example:

```
IFP    SYMBOL
```

assembles the next lines if SYMBOL is a positive number.

IFT

symbol IFT *expression*

Same as IFDEF except the *expression* must be true (that is, bit 0 must be 1) for the next lines to be assembled.

For example:

```
IFT    5.GT.SYMBOL
```

assembles the next lines IF 5 is greater than SYMBOL.

IFUND

symbol IFUND *symbol*

Same as IFT except the *symbol* must not be defined for the next lines to be assembled. For example:

```
IFUND    SYMBOL
```

assembles the next lines if the program does not define SYMBOL. If the symbol is defined at all, it must be defined before the IFDEF.

IFZ

symbol IFZ *expression*

Same as IFDEF except the *expression* must equal zero for the next lines to be assembled. For example:

```
IFZ    SYMBOL
```

assembles the next lines if SYMBOL equals zero.

INCLUDE

INCLUDE *'source filespec'*

Inserts *filespec* at the point where INCLUDE appears in the program. The Assembler will assemble the INCLUDEd file before processing the next instruction.

The optional END instruction of the INCLUDEd file tells the Assembler to continue assembling the main program. The END of the main program will terminate the assembly.

You may specify only one filename per INCLUDE. You may use as many INCLUDE instructions as you want.

For example:

```
INCLUDE 'PROG1'
```

inserts and assembles PROG1, a source file, before processing the next instruction.

```
INCLUDE 'PROG1'
INCLUDE 'PROG2'
```

inserts and assembles PROG1; then inserts and assembles PROG2; then proceeds with the next instruction.

INCLUDE is nestable to five levels. That is, file 1 can call file 2; 2 can call 3; 3 can call 4; and finally, 4 can call 5. But at no time can a called file (file 5) call a calling file (file 4). This results in an Error 37 — Open attempt for a file already open.

ISECT

ISECT *name*

Begins an "index section" of EQU instructions, terminated by ENDI. If you wish, you can name the section 1 or 2 (no other names are allowed).

Using an index section allows you to specify certain index symbols. You can then use the index symbols to offset an index register.

For example, this is an index section named ISECT 1:

```
            ISECT1
SYMBOL1     EQU  5
SMBL3       EQU  3
SMBL26      EQU  26
SYMBL       EQU  100
            ENDI
```

It specifies four index symbols. Whenever the Assembler encounters one of these index symbols enclosed in parentheses, it evaluates it as the expression:

(the contents of an index register + index symbol)

You must specify which index register to use with the USING instruction. For example:

```
LD          IY,7000H
USING       1,IY
LD          A,(SYMBOL1)
```

The Assembler evaluates this as:

```
LD          IY,7000H
USING1,     IY
LD          A,(IY+SYMBOL1)
```

You cannot use a register name or a flag condition to name an index symbol.

LINK

LINK *'filespec'*
LINK *'filespec(symbol)'*

Tells the Linker to insert *filespec*, an absolute or relocatable object file, at the point where LINK is encountered in the current program. This instruction is similar to INCLUDE, except it applies only to the Linker. It allows you to link one or more files together.

LINK must be at the end of your program section. (Only END, GLINK, or another LINK can follow it.) Each LINK instruction can specify only one filename. You can use as many LINK instructions as you want.

For example:

```
LINK        'FILE1'
LINK        'FILE2'
END         PROG
```

inserts FILE1 and then FILE2 at the end of your main program. FILE1 and FILE2 must both be assembled object files.

```
LINK        'TAX(TABLE)'
```

inserts a program section named TABLE which exists in a file named TAX at the end of your program. TAX must be an object file. TABLE is a PSECT label.

The LINK statement is nestable to five levels. That is, file 1 can call file 2, 2 can call 3, 3 can call 4, and finally, 4 can call 5. But at no time can a called file (file 5) call a calling file (file 4).

LITORG

symbol LITORG *location*

Allows you to specify where to place literals used as operands. LITORG should be used only once per assembly and placed in the same PSET as all references to the literals, and after the last reference.

If you omit the optional *location*, the Assembler stores the literals in the current location. If you include it, LITORG resets the location counter (in the same way that ORG does) and stores the literals at the newly reset location.

The optional *symbol* labels this location. The Assembler assigns the remaining instructions locations immediately following the literals.

All literal operands must be preceded by an equal sign (=) and surrounded with single quotes ('). For example:

```
            LD          HL,='INPUT THE ITEM NUMBER'
```

This instruction uses INPUT THE ITEM NUMBER as a literal operand. Here is how you could use it in a program:

```
START       PSECT       5200H
BEGIN       LD          HL,='INPUT THE ITEM NUMBER'
            LD          B,(HL)
            INC         HL
            CALL        PRTCHR
            CALL        EXIT
            LITORG
            DEFM        'THIS IS A LONG TABLE OF PROMPTS'
            DEFM        'INPUT THE ITEM NUMBER'
            DEFM        'INPUT THE PRICE'
            DEFM        'IS THERE A DISCOUNT?'
            DEFM        'INPUT THE DISCOUNT'
            END         BEGIN
```

Notice that INPUT THE ITEM NUMBER is defined by DEFM later in the program. The Assembler stores it in two locations: (1) the location where

LITORG appears in the program, and (2) the location where DEFM 'INPUT THE ITEM NUMBER' appears.

Note that if literals are used and the program ends with a LINK or GLINK, LITORG is mandatory to place the literals before the LINK or GLINK statement.

MACRO

name MACRO *dummy parameter list*

Begins a section of the program which defines a macro name. Use ENDM to end this macro definition.

The optional *dummy parameter list* allows you to pass parameters to the macro. You may use up to ten dummy parameters separated by commas. Each can be only one character and must be preceded by a # sign.

Defining a macro allows you to "call" an entire block of instructions with a single program line. This is useful when you will be using the same block many times in your program.

For example, this is a macro definition:

```
SCROLL    MACRO
          LD          A,5
          CALL        PROTECT
          ENDM
```

which defines a macro named SCROLL, that protects 5 lines from scrolling. Every time the Assembler encounters SCROLL, it "expands" SCROLL into the LD A,5 and CALL PROTECT instructions. That is, if this is your source program:

```
          •
          LD          A,3
          SCROLL
          LD          HL,DATA
          •
```

The Assembler will interpret SCROLL as a macro call and expand it into the appropriate instructions:

```
•
LD        A,3
LD        A,5
CALL      PROTECT
LD        HL,DATA
•
```

The next example defines a macro named ADNUM which acts on four dummy parameters named #0, #1, #2, and #3:

```
ADNUM     MACRO     #0,#1,#2,#3
          ADD       A,#0
          ADD       A,#1
          ADD       A,#2
          ADD       A,#3
          ENDM
```

This definition allows you to "pass" four values to ADNUM when you call it. For example:

```
ADNUM     B,10,NUMB,LST
```

calls ADNUM and passes four values to it. The Assembler expands this macro call into:

```
          ADD       A,B
          ADD       A,10
          ADD       A,NUMB
          ADD       A,LST
```

Notice that B, the first value, replaces #0, the first parameter; 10 replaces #1; NUMB replaces #2; and LST replaces #3.

When using a macro, remember that you must define it before you use it. You might want to put all the macro definitions in one file and then INCLUDE or REF them at the beginning of your main file.

We do not recommend that you use a macro name which is the same as an extended mnemonic or directive name. If you do this, the Assembler will use the definition you assigned the macro. This will of course give undesirable results.

When using dummy parameters, be sure not to insert them inside quoted strings. If you do this, the Assembler will treat them as ordinary characters.

A macro cannot call another macro.

NOEND

NOEND

Ends the assembly of a non-executable program. The Assembler marks the assembled code as load-only and will not execute the file when used as a TRSDOS command. This command is useful for creating overlays to be loaded with the CMDDOS system call.

NOFILL

NOFILL

Terminates the mode initiated by FILL.

NOLOAD

NOLOAD

Assembles the program sequentially in memory image form, rather than in the standard TRSDOS object format. You must use NOLOAD as the first line of the main source file (before comments, titles, PSECT, etc.), otherwise some TRSDOS object code load headers may be placed into the file.

You cannot use NOLOAD with these features:

- the relocatable mode
- EXTERNal, or PUBLIC symbols
- LINK or GLINK

If you want the file to contain an accurate memory image of the program, you must also avoid these instructions:

- DEFS(unless the FILL mode is on)
- ORG
- more than one PSECT

(These instructions change the value of the location counter but do not output object code. This causes the load address and location counter to differ.)

OBJ

OBJ *'filespec'*

Tells the Assembler that it should write the assembled *filespec* to disk. The Assembler will ignore this instruction if you specify an object filespec in the assembly command line.

Example:

```
OBJ  'ACCOUNTS'
```

Unless you specify an object filespec in the assembly command line, the above instruction saves the assembled object program as ACCOUNTS.

ORG

symbol ORG *location, boundary*

Resets the Assembler's location counter to the specified *location*. For example, in an absolute program:

```
ORG          6000H
```

resets the location to an absolute 6000H.

In a relocatable program:

```
ORG         6000H
```

resets the location counter to a relocatable 6000H. Assuming you link the program to an absolute start address of 5200H, the Linker determines the effective address to be B200H (the sum of 5200 and 6000.)

The second parameter allows you to reset the location counter to a *boundary* divisible by decimal 2, 4, 8, 16, 32, 64, 128, or 256. For example, if the value of the counter is currently 6005H:

```
ORG         $,4
```

resets the counter to 6008H, which is the next highest number divisible by decimal 4.

Unlike many other assemblers, ORG will not initialize the location counter. You need to use PSECT for this purpose.

ORG will not change the location counter from the relocatable to the absolute mode, or vice versa. You must assemble absolute and relocatable programs as different files.

location may not be an external symbol.

PATCH

PATCH

Fills the remaining bytes in the last sector in the assembled object file with FF's. This reserves an area for patches.

The Assembler will print a message on pass 2 giving the address and length of the patch area (if the file produces object code).

This must be the last command prior to the END directive. You cannot use it with LINK, and it is for use with absolute assemblies only.

PRINT

PRINT *command list*

Controls what is printed or not printed in the assembly listing. You may use one or more of the following commands, separated by commas or blank spaces:

ALL — print all source lines (Same as ON,MAC,CON)
ON — print all normal open code source instructions
OFF — do not print anything except error messages and diagnostics until (1) the end of the assembly or (2) a PRINT ON command

MAC — print all source lines generated in macro expansions (except those which might be overridden by other PRINT options).
NOMAC — do not print source lines generated by macro expansions. Only the macro instruction itself will appear in the listing file.
CON — print all conditional assembly source lines, whether they generate code or not.
NOCON — print only the conditional assembly source lines that generate code.
LST — output the listing, regardless of what was on the command line. The listing will be printed on the video, and if the D or P options were specified, the listing will also go to disk or to the printer. You cannot save this option with APUSH.
NOLST — do not output a listing, regardless of what was on the command line.
SHORT — print only the first 6 bytes of object code generated by each line.
LONG — print all of the object code generated, even if it requires several lines.

For example:

```
PRINT MAC,SHORT
```

prints all the macro expansions in the assembly listing. It limits printing to the first six bytes of object code for each line.

Only PRINT instructions specifying OFF, NOMAC, and NOCON will appear in the listing.

You can use comments with PRINT.

PRINT defaults to ON, MAC, NOCON, LONG.

PSECT

symbol PSECT *location*

Initializes the Assembler's location counter to a relocatable zero or to the absolute *location* you specify. The Assembler assembles all subsequent instructions sequentially throughout the program.

The optional *symbol* labels the program section and can be up to six characters. This symbol is for the Linker, and will be listed on the Linker map. The symbol will not be defined by the Assembler and cannot be used in expressions.

PSECT begins an independent, executable "program section". You can have several relocatable program sections in one program file. One program section cannot use symbols from another program section unless you declare them EXTERN and PUBLIC.

For example:

```
          PAYROLL   PSECT
0000'     BEGIN     LD        A,3
                    •
```

```
          PAYABLE   PSECT
0000'               PUSH      A
                    •
                    END
```

This program has two sections: "PAYROLL" and "PAYABLE". Both begin with a relocatable 0000. When you link this file, the Linker assigns "PAYABLE" addresses which immediately follow "PAYABLE". Since no symbols are declared PUBLIC and EXTERNal, "PAYROLL" and "PAYABLE" cannot share the same symbols.

The following instructions do not have to be part of a program section:

- comments
- index sections
- conditional assembly instructions
- macro sections
- macro instructions (which will not affect the location counter)
- EQU or DEFL (as long as they do not reference the location counter)
- assembler directives (which do not affect the location counter)

You can define *symbol* (with EQU, for example) prior to your first PSECT. This permits you to use a conditional assembly such as:

```
          IFT       RELOC
XYZ       PSECT
          ELSE
XYZ       PSECT     5200H
          ENDIF
```

which starts a relocatable PSECT if RELOC equals 1, and an absolute PSECT if RELOC equals 0. Doing this will create two PSECTs with the same name, one being zero-length. This will appear on the Linker map but it will not affect the assembly.

The PSECTs within an assembly must either be all relocatable or all absolute. Relocatable and EXTERN expressions cannot be used in absolute assemblies.

The PSECT location you specify cannot be an external value.

PUBLIC

PUBLIC *symbol list*

Declares one or more *symbols* as "public". This permits another program section to use the same symbols.

When you assemble a program with public symbols, the Assembler will mark all their definitions. Then, when you link it to an external program section, the Linker will insert these definitions in the Linker Symbol Table.

For example:

```
PUBLIC   LOOP1
```

declares LOOP1's definition to be public.

Another program can use the public symbol definitions provided it contains a corresponding EXTERN directive.

You can substitute ENTRY for PUBLIC.

Symbols declared PUBLIC or GLOBAL must be defined on both passes, that is, not defined with REF, ADISP, or EXT. The Linker may flag these symbols as undefined.

Symbols defined with DEFL more than once should not be declared PUBLIC or GLOBAL. The linker will flag these symbols as multiply defined.

QUIT

QUIT

Quits the assembly and returns to TRSDOS Ready. This Assembler only recognizes this instruction at the second pass of a listing (specified by the L assembly option). It will not save the object file.

RADIX

RADIX *expression*

Specifies *expression* as the default number base. That is, the Assembler will interpret any numbers without a base suffix in the default base.

You may use any expression with a value of 2, 8, 10, or 16. Without RADIX, the Assembler defaults to 10 (decimal).

For example:

```
RADIX    16
```

causes the Assembler to interpret all the numbers which do not have "b" or "d" suffixes as hexadecimal numbers.

Remember that the Assembler uses the current default base to evaluate your RADIX instruction. For example, if you want to change the default base of 16 to 10, use RADIX 10d or 0A, not RADIX 10. While in base 16, the Assembler would evaluate the 10 as a hexadecimal 10.

Example:

```
RADIX       10H     ;Use Hexadecimal
DEFB        1B      ; This is 1B (hex)=27 (decimal)
```

```
DEFB        1b      ; This is 1 (binary)
DEFB        25      ; This is 25 (hex)=37 (decimal)
RADIX       10      ; Radix is still hex (10 hex=
                    ; 16 decimal)
RADIX       10D     ;ERROR 10D hex=269 decimal _
                    ; too large.
RADIX       10d     ; Radix is now decimal
DEFB        1B      ; This is a 1 binary
DEFB        1b      ; This is also a 1 binary
DEFB        25      ; This is 25 (decimal)=19 (hex)
```

REF

REF *'source filename'*

Includes only the symbol definitions from the specified source file. This is useful for referencing a file of EQU directives or MACROs.

REF tells the Assembler to INCLUDE the source file during Pass 1 only. After processing the source file, the Assembler restores the location counter to its original value. Thus, the Assembler uses the referenced file's symbols, but not its assembled code.

For example:

```
REF   'TEST/SRC'
```

The Assembler will define macros and symbols contained in TEST/SRC. It will not insert the code for TEXT/SRC.

The Assembler will not report any errors in the referenced file. Also, if there is a conflict between symbols of the referenced file and the main program, the first definitions will be used with no error message. You might want to use INCLUDE instead of REF until all conflicts have been resolved.

Symbols defined in the REF file should not be declared PUBLIC or GLOBAL. The Linker may flag these symbols as undefined.

RESLOC

RESLOC *location*

Resets the location counter to the location computed as:

the value of the counter prior to executing SETLOC	+	the number of bytes of code generated by the SETLOC block

For example, assuming the value of the location counter was 6000H prior to SETLOC and there are two 3-byte instructions following SETLOC:

```
RESLOC
```

resets the location counter to 6006H.

SETLOC

SETLOC *location*

Temporarily changes the location counter's value to the absolute *location* specified. The Assembler uses this changed location for defining symbols only. It does not use the changed location for assembling the instructions.

For example:

```
7000            LD          A,3
                SETLOC      6000H
6000  POS       PUSH        AF
```

The actual PUSH AF instruction is not stored at location 6000H. Rather, it is stored at 7002H, the location which immediately follows LD A,3. However, the Assembler defines POS, the symbol which labels the location of PUSH AF, as 6000H.

SETLOC is useful anytime you are writing a routine which you want to load in one location, and then move and execute at a different location. By using SETLOC, the Assembler defines this routine's symbols as if they were already in their execution location.

For example, you might want to run a memory test from a very low memory address. You cannot load it on top of TRSDOS. However, after loading it, you can move and execute it in that location. Since TRSDOS will be overwritten, the memory test must do its own input/output.

Using SETLOC, you could write the routine this way:

```
                    PSECT     5000H
                      •
5100      MOVE      EQU       $           ;SETLOC block begins
                    SETLOC    500H
500       LOOP      LD        A,3
                      •                   ;code for memory
560                   •                   ;test
                    RESLOC
5200      LDBLOCK   EQU       $-MOVE      ;SETLOC block ends
                      •
                    LD        HL,MOVE     ;move SETLOC block
                    LD        DE,LOOP     ;to its proper loop
                    LD        BC,LDBLOCK
                    LDIR
                    JP        LOOP
                      •
```

Here, the Assembler defines LOOP as though it were at address 500H — the address the program will eventually move it to. However, it actually assembles the code for LOOP at address 5100H.

MOVE defines where the actual assembled code of the SETLOC block (ended by RESLOC) begins. LDBLOCK defines the length of the SETLOC block by subtracting MOVE from the current contents of the PC register. (The $ sign indicates the current value of PC).

LDIR then moves the SETLOC block from location 5100, defined by MOVE, to location 500. Since LOOP has already been defined as if it were at location 500, you do not have to redefine it.

Note: If your program is relocatable, SETLOC still sets an absolute location. You need to avoid using these instructions within the SETLOC block: ORG, DEFS (unless the FILL mode is in effect), PSECT, and relocatable and external expressions.

STOP

STOP

Stops the assembly listing. Press any key to continue the listing. Press (BREAK) to abort it.

TIME

symbol TIME

Stores the time in memory as a string beginning at the current address. The optional *symbol* labels this address. For example if the time is 1:45 p.m. and 55 seconds when the Assembler reaches this instruction:

```
TIME
```

it will store the string 13.45.55 (Model 4) or 13:45:55 (Model III) in the next eight bytes of memory.

TITLE

TITLE *'string'*

Prints the specified *string* on the third line of each page in the assembly listing. For example:

```
TITLE       'THIS IS THE TITLE'
```

prints THIS IS THE TITLE on the third line of every page.

If you are using both TITLE and HEADER, TITLE should precede HEADER (otherwise the TITLE will not appear until the next page).

USING

USING *index section name, index register*
USING *index register, expression*
USING *index register*

Associates an *index register* — IX or IY — with the *index sections*. For example:

```
USING       IX
```

associates IX with all the ISECTS.

You can optionally specify one (but not both) of the following:

- an *index section name* (1 or 2), as the only section to be associated with the register
- an *expression* to be loaded into the register

For example:

```
USING       1,IX
```

associates the IX register with ISECT 1 only.

```
USING       IX,DCB
```

loads IX with the value of DCB, then associates IX with all the ISECTs.

The index sections are specified with the ISECT instruction.

USING does not apply to any external program sections.

VERSION

VERSION

Prints the current time on the second line of the assembly listing heading.

* (block comment)

*

Turns on and off the *block comment* function. The asterisk must be in the first column.

When the Assembler encounters a line beginning with an asterisk, it begins interpreting the lines as comments rather than instructions. The next asterisk ends the block comment.

For example:

```
*
The following program is a . . .
•
•
•
•
*
```

Note: Be careful when using the asterisk. One asterisk out of place near the beginning of your program can cause the Assembler to treat most of your program as a comment. If a block comment is placed before a header created by the TITLE directive, the title will not appear on the first page of the assembly listing.

Chapter 9/
Z-80 Mnemonics

This section contains a description of each Z-80 mnemonic, organized as follows:

8 Bit Load Group
16 Bit Load Group
Exchange, Block Transfer and Search Group
8 Bit Arithmetic and Logical Group
General Purpose Arithmetic and CPU Control Groups
16 Bit Arithmetic Group
Rotate and Shift Group
Bit Set, Reset and Test Group
Jump Group
Call and Return Group
Input and Output Group

Please note than you can specify the PO (parity odd) and PE (parity even) conditions with NV and V. For example:

JP PO, 1000H
JP NV, 1000H

Both of these instructions tell the Assembler to branch to 1000H if there is the Parity is Odd, which means there is No Overflow.

JP PE, 1000H
JP V, 1000H

These instructions tell the Assembler to branch to 1000H if the Parity is Even, which means there is an overflow.

The Z-80 Instruction Set

Notation and Other Conventions

This section includes a detailed description of all the Z-80 assembly language instructions. The first line of each of these pages shows the assembly language opcode mnemonic followed by its operand(s). Some instructions have no operands at all. Other instructions have one or two operands. Anything which is capitalized should be copied exactly when you use the editor to write the assembly language source code. Anything shown in lowercase letters will be replaced by an appropriate register, number, or label. For example, the first instruction described in the eight-bit load group is:

LD r,r′

LD is the mnemonic for the Load instruction. If you wish to move the contents of register H into register A, the actual source code is

LD A,H

This should be read as "load register A with the contents of register H."

A detailed explanation of the operand notation is given below, but in general you should note that single lowercase letters are used for eight-bit numbers or registers and double lowercase letters are used for 16-bit numbers or registers. Also note that parentheses around a register pair indicates that the register pair is to be used as a pointer to a memory location. For example, the instruction INC HL means that 1 is to be added to the HL register pair. The instruction INC (HL) means that 1 will be added to a number in memory whose address is found in register pair HL.

Symbol	**Specifies one of the registers**
r	A, B, C, D, E, H, or L.

Symbol	**Specifies a register pair**
qq	BC, DE, HL, or AF
ss	BC, DE, HL, or SP
dd	BC, DE, HL, or SP
pp	BC, DE, IX, or SP
rr	BC, DE, IY, or SP

Symbol	**Specifies a number or symbol in the range**
n	0 to 255 (one byte)
nn	0 to 65535 (two bytes)
d	−128 to 127 (one byte)
e	−126 to 129 (one byte)

Symbol	Specifies any of the following
s	r, n, (HL), (IX + d), or (IY + d)
m	r, (HL) (IX + d), or (IY + d)
(nn)	Specifies the contents of memory location **nn**
b	Specifies an expression in the range (0,7)
cc	Specifies the state of the Flags for conditional JR, JP, CALL and RET instructions

Instruction Format Examples With Explanation

Format Example 1

LD r,(HL)

Operation: r ◊ (HL)

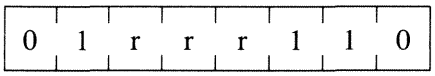
This is the shorthand description of the instruction. The arrow indicates that data is moved into register r.

When you write the assembly language code, the lowercase r will be replaced by A, B, C, D, E, H or L.

Format:

Mnemonic: LD **Operands:** r,(HL)

Object Code:



The object code for this instruction is one byte long. To figure out the object code, replace bits 3, 4 and 5 with the appropriate numbers from the table. For example:

Source Code	Object Code
LD A,(HL)	01111110
LD B,(HL)	01000110
LD C,(HL)	01001110

This instruction uses two machine (M) cycles. The first machine cycle consists of four timing (T) states and the second machine cycle consists of three T states for a total of seven T states. One T state takes approximately 250 nanoseconds for a 4MHz machine and 500 nanoseconds for a 2MHz machine. The execution time (E.T.), in microseconds, is calculated for the TRS-80. (One microsecond is 10^{-6} seconds or 1/1,000,000 of a second.)

Description:

The eight-bit contents of memory location (HL) are loaded into register r, where r identifies register A, B, C, D, E, H or L, assembled as follows in the object code:

Register		r
A	=	111
B	=	000
C	=	001
D	=	010
E	=	011
H	=	100
L	=	101

M cycles: 2 T states: 7(4,3) 4 MHz E.T.: 1.75

Condition Bits Affected: None

Example:

If register pair HL contains the number 75A1H, and memory address 75 A1H contains the byte 58H, the execution of

LD C, (HL)

will result in 58H in register C.

Format Example 2

JP cc,nn

Operation: IF cc TRUE, PC ◊ nn

The jump is made only if the condition cc is true. The arrow indicates that the number nn is moved into the program counter PC. This will cause the program to jump to address nn.

When you write the assembly language code, cc will be replaced by one of the following: NZ, Z, NC, C, PO, PE, P or M. nn will be replaced by a number from 0 to 65535 or a label.

Format:

Mnemonic: JP **Operands:** cc, nn

Object Code:

1	1	cc	cc	cc	0	1	0

n	n	n	n	n	n	n	n

n	n	n	n	n	n	n	n

Note: The first n operand in this assembled object code is the low order byte of a two-byte memory address.

The object code for this instruction is three bytes long. To figure out the object code, replace bits 3, 4 and 5 of the first byte with the appropriate number from the table. The second two bytes of the object code are the address being jumped to. For example:

Source Code	Object Code	
JP NZ, 0FF00H	11000010	C2H
	00000000	00H
	11111111	FFH
JP M, 1002H	11111010	FAH
	00000010	02H
	00010000	10H

Note that the low order, or right hand byte, of the address comes first in the object code.

Description:

If condition cc is true, the instruction loads operand nn into register pair PC (Program Counter), and the program continues with the instruction beginning at address nn. If condition cc is false, the Program Counter is incremented as usual, and the program continues with the next sequential instruction. Condition cc is programmed as one of eight status bits which correspond to condition bits in the Flag Register (register F). These eight status bits are defined in the table below which also specifies the corresponding cc bit fields in the assembled object code.

The Relevant Flag column shows the value the flag must have if the jump is to occur.

cc	Condition	Relevant Flag
• 000	NZ non zero	Z = 0
001	Z zero	Z = 1
010	NC no carry	C = 0
011	C carry	C = 1
100	PO parity odd or no overflow	P/V = 0
101	PE parity even or overflow	P/V = 1
110	P sign positive	S = 0
111	M sign negative	S = 1

M cycles: 3 T states: 10(4,3,3) 4 MHz E.T.: 2.50

Condition Bits Affected: None

Example:

If the Carry Flag (C flag in the F register) is set and the contents of address 1520 are 03H, after the execution of

JP C,1520H

the Program Counter will contain 1520H, and on the next machine cycle the CPU will fetch from address 1520H the byte 03H. In other words, program execution jumps to the instruction at 1520H.

Format Example 3

CPIR

Operation: A − (HL), HL ◊ HL + 1, BC ◊ BC − 1

The shorthand description indicates that three different things are happening:

1. BC is decremented
2. HL is incremented
3. A byte in memory is subtracted from the A register (but the results are not saved).

Format:

Mnemonic: CPIR **Operands:**

Object Code:

1	1	1	0	1	1	0	1	ED

1	0	1	1	0	0	0	1	B1

The assembly language instruction has no operands.

The object code is two bytes long.

Description:

The contents of the memory location addressed by the HL register pair is compared with the contents of the Accumulator. In case of a true compare, a condition bit is set. The HL is incremented and the Byte Counter (register pair BC) is decremented. If decrementing causes the BC to go to zero or if A = (HL), the instruction is terminated. If BC is not zero and A ≠ (HL), the program counter is decremented by 2 and the instruction is repeated. Note that if BC is set to zero before the execution, the instruction will loop through 64K bytes, if no match is found. Also, interrupts will be recognized after each data comparison.

For BC ≠ 0 and A ≠ (HL):

M cycles: 5 T states: 21(4,4,3,5,5) 4 MHz E.T.: 5.25

For BC = Ø or A = (HL):

M cycles: 4 T states: 16(4,4,3,5) 4 MHz E.T.: 4.ØØ

The total execution time of this instruction depends on how long it takes to find the byte being searched for and the length of the block being searched. If the instruction loops three times before BC = Ø or A = (HL), then there will be 58 (2x21 + 16) timing (T) states executed.

Condition Bits Affected:

S: Set if result is negative; reset otherwise
Z: Set if A = (HL); reset otherwise
H: Set if borrow from Bit 4; reset otherwise
P/V: Set if BC becomes zero; reset otherwise
N: Set
C: Not affected

Example:

If the HL register pair contains 1111H, the Accumulator contains F3H, the Byte Counter contains ØØØ7H, and memory locations have these contents:

(1111H) : 52H
(1112H) : ØØH
(1113H) : F3H

then after the execution of

CPIR

the contents of register pair HL will be 1114H, the contents of the Byte Counter will be ØØØ4H. Since BC ≠ Ø, the P/V flag is still set. This means that it did not search through the whole block before the instruction stopped. Since a match was found, the Z flag is set.

The CPIR instruction will affect five of the six condition codes.

8 Bit Load Group

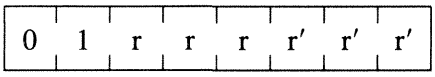
LD r,r′ LoaD

Operation: r ⇦ r′

Format:

Mnemonic: LD **Operands:** r, r′

Object Code:



Description:

The contents of any register r′ are loaded into any other register r. Note: r, r′ identifies any of the registers A, B, C, D, E, H, or L, assembled as follows in the object code:

Register		r, r′
A	=	111
B	=	000
C	=	001
D	=	010
E	=	011
H	=	100
L	=	101

M cycles: 1 T states: 4 4 MHz E.T.: 1.0

Condition Bits Affected: None

Example:

If the H register contains the number 8AH, and the E register contains 10H, the instruction

LD H,E

would result in both registers containing 10H.

LD r,n

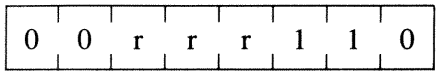
LoaD

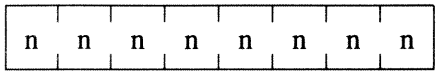
Operation: r ◊ n

Format:

Mnemonic: LD **Operands:** r, n

Object Code:





Description:

The eight-bit integer n is loaded into any register r, where r identifies register A, B, C, D, E, H or L, assembled as follows in the object code:

Register		r
A	=	111
B	=	000
C	=	001
D	=	010
E	=	011
H	=	100
L	=	101

M cycles: 2 T states: 7(4,3) 4 MHz E.T.: 1.75

Condition Bits Affected: None

Example 1:

After the execution of

LD E,A5H

the contents of register E will be A5H.

Example 2:

After the execution of

LD A,0

register A will contain zero.

LD r,(HL)

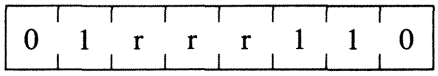
LoaD

Operation: r ◊ (HL)

Format:

Mnemonic: LD **Operands:** r, (HL)

Object Code:



Description:

The eight-bit contents of memory location (HL) are loaded into register r, where r identifies register A, B, C, D, E, H or L, assembled as follows in the object code:

Register		r
A	=	111
B	=	000
C	=	001
D	=	010
E	=	011
H	=	100
L	=	101

M cycles: 2 T states: 7(4,3) 4 MHz E.T.: 1.75

Condition Bits Affected: None

Example:

If register pair HL contains the number 75A1H, and memory address 75A1H contains the byte 58H, the execution of

LD C,(HL)

will result in 58H in register C.

LD r,(IX + d)

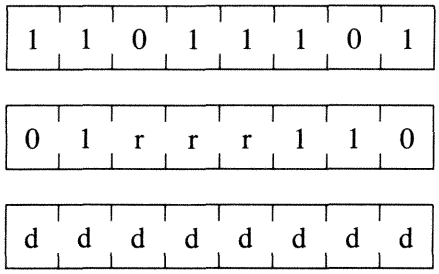
LoaD

Operation: r ◊ (IX + d)

Format:

Mnemonic: LD **Operands:** r, (IX + d)

Object Code:



Description:

The operand (IX + d) (the contents of the Index Register IX summed with a displacement integer d) is loaded into register r, where r identifies register A, B, C, D, E, H or L, assembled as follows in the object code:

Register		r
A	=	111
B	=	000
C	=	001
D	=	010
E	=	011
H	=	100
L	=	101

M cycles: 5 T states: 19(4,4,3,5,3) 4 MHz E.T.: 4.75

Condition Bits Affected: None

Example:

If the Index Register IX contains the number 25AFH, the instruction

LD B,(IX + 19H)

will cause the calculation of the sum 25AFH + 19H, which points to memory location 25C8H. If this address contains byte 39H, the instruction will result in register B also containing 39H.

A typical use of this instruction is shown below. If TABL is a location in memory this program will load the first four bytes of the table into registers A, B, C and D.

```
LD    IX, TABL      ; IX points to the table
LD    A, (IX+0)     ; Load first byte
LD    B, (IX+1)     ; Load second byte
LD    C, (IX+2)     ; Load third byte
LD    D, (IX+3)     ; Load fourth byte
```

LD r,(IY+d)

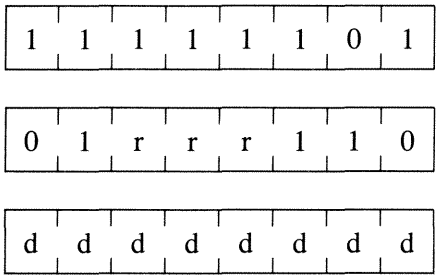
LoaD

Operation: r ◊ (IY+d)

Format:

Mnemonic: LD **Operands:** r, (IY+d)

Object Code:



Description:

The operand (IY+d) (the contents of the Index Register IY summed with a two's complement displacement integer d) is loaded into register r, where r identifies register A, B, C, D, E, H, or L, assembled as follows in the object code:

Register		r
A	=	111
B	=	000
C	=	001
D	=	010
E	=	011
H	=	100
L	=	101

M cycles: 5 T states: 19(4,4,3,5,3) 4 MHz E.T.: 4.75

Condition Bits Affected: None

Example:

If the Index Register IY contains the number 25AFH, the instruction

LD B,(IY+19H)

will cause the calculation of the sum 25AFH+19H, which points to memory location 25C8H. If this address contains byte 39H, the instruction will result in register B also containing 39H.

LD (HL),r

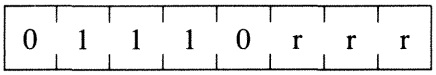
LoaD

Operation: (HL) ◊ r

Format:

Mnemonic: LD **Operands:** (HL), r

Object Code:



Description:

The contents of register r are loaded into the memory location specified by the contents of the HL register pair. The symbol r identifies register A, B, C, D, E, H or L, assembled as follows in the object code:

Register		r
A	=	111
B	=	000
C	=	001
D	=	010
E	=	011
H	=	100
L	=	101

M cycles: 2 T states: 7(4,3) 4 MHz E.T.: 1.75

Condition Bits Affected: None

Example:

If the contents of register pair HL specify memory location 2146H, and the B register contains the byte 29H, after the execution of

LD (HL),B

memory address 2146H will also contain 29H.

LD (IX + d),r

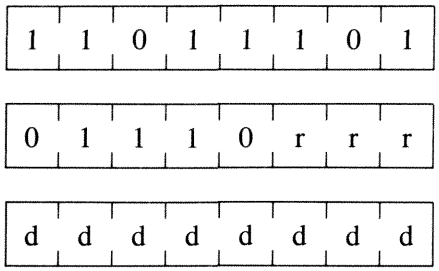
LoaD

Operation: (IX + d) ◊ r

Format:

Mnemonic: LD **Operands:** (IX + d), r

Object Code:



Description:

The contents of register r are loaded into the memory address specified by the contents of Index Register IX summed with d, a two's complement displacement integer. The symbol r identifies register A, B, C, D, E, H or L, assembled as follows in the object code:

Register		r
A	=	111
B	=	000
C	=	001
D	=	010
E	=	011
H	=	100
L	=	101

M cycles: 5 T states: 19(4,4,3,5,3) 4 MHz E.T.: 4.75

Condition Bits Affected: None

Example:

If the C register contains the byte 1CH, and the Index Register IX contains 3100H, then the instruction

LD (IX+6H), C

will perform the sum 3100H+6H and will load 1CH into memory location 3106H.

LD (IY+d),r

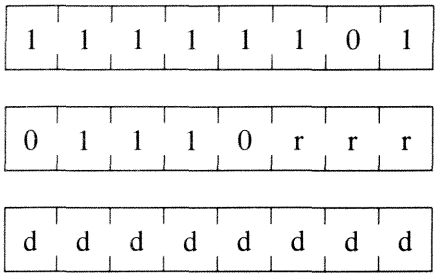
LoaD

Operation: (IY+d) ◊ r

Format:

Mnemonic: LD **Operands:** (IY+d), r

Object Code:



Description:

The contents of register r are loaded into the memory address specified by the sum of the contents of the Index Register IY and d, a two's complement displacement integer. The symbol r is specified according to the following table.

Register		r
A	=	111
B	=	000
C	=	001
D	=	010
E	=	011
H	=	100
L	=	101

M cycles: 5 T states: 19(4,4,3,5,3) 4 MHz E.T.: 4.75

Condition Bits Affected: None

Example:

If the C register contains the byte 48H, and the Index Register IY contains 2A11H, then the instruction

LD (IY + 4H),C

will perform the sum 2A11H + 4H, and will load 48H into memory location 2A15.

LD (HL),n

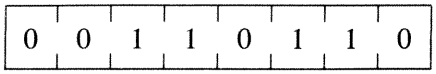
LoaD

Operation: (HL) ◊ n

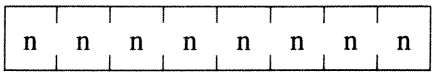
Format:

Mnemonic: LD **Operands:** (HL), n

Object Code:



36



Description:

Integer n is loaded into the memory address specified by the contents of the HL register pair.

M cycles: 3 T states: 10(4,3,3) 4 MHz E.T.: 2.50

Condition Bits Affected: None

Example:

If the HL register pair contains 4444H, the instruction

LD (HL),28H

will result in the memory location 4444H containing the byte 28H.

LD (IX + d),n

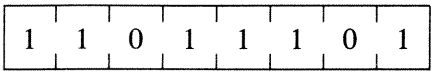
Load

Operation: (IX + d) ◊ n

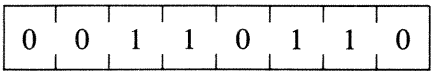
Format:

Mnemonic: LD **Operands:** (IX + d), n

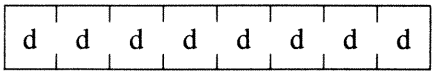
Object Code:

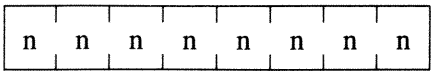


DD



36





Description:

The n operand is loaded into the memory address specified by the sum of the contents of the Index Register IX and the two's complement displacement operand d.

M cycles: 5 T states: 19(4,4,3,5,3) 4 MHz E.T.: 4.75

Condition Bits Affected: None

Example:

If the Index Register IX contains the number 219AH the instruction

LD (IX + 5H),5AH

would result in the byte 5AH in the memory address 219FH.
(219FH = 219AH + 5H.)

LD (IY + d),n

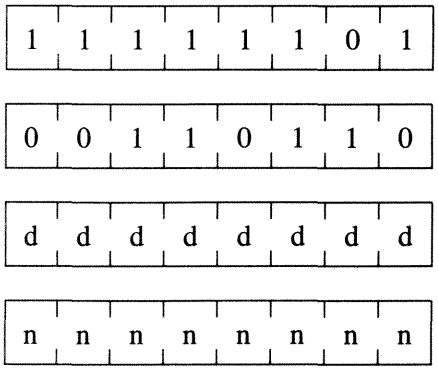
LoaD

Operation: (IY + d) ⇦ n

Format:

Mnemonic: LD **Operands:** (IY + d), n

Object Code:



Description:

Integer n is loaded into the memory location specified by the contents of the Index Register summed with a two's complement displacement integer d.

M cycles: 5 T states: 19(4,4,3,5,3) 4 MHz E.T.: 4.75

Condition Bits Affected: None

Example:

If the Index Register IY contains the number A940H, the instruction
LD (IY + 10H),97H
would result in byte 97H in memory location A950H.

LD A,(BC)

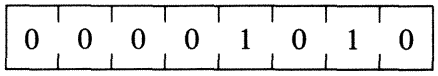
LoaD

Operation: A ◊ (BC)

Format:

Mnemonic: LD **Operands:** A, (BC)

Object Code:



0A

Description:

The contents of the memory location specified by the contents of the BC register pair are loaded into the Accumulator.

M cycles: 2 T states: 7(4,3) 4 MHz E.T.: 1.75

Condition Bits Affected: None

Example:

If the BC register pair contains the number 4747H, and memory address 4747H contains the byte 12H, then the instruction
LD A,(BC)
will result in byte 12H in register A.

LD A,(DE)

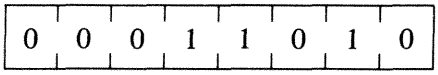
LoaD

Operation: A ◊ (DE)

Format:

Mnemonic: LD **Operands:** A, (DE)

Object Code:



1A

Description:

The contents of the memory location specified by the register pair DE are loaded into the Accumulator.

M cycles: 2 T states: 7(4,3) 4 MHz E.T.: 1.75

Condition Bits Affected: None

Example:

If the DE register pair contains the number 30A2H and memory address 30A2H contains the byte 22H, then the instruction

LD A,(DE)

will result in byte 22H in register A.

LD A,(nn)

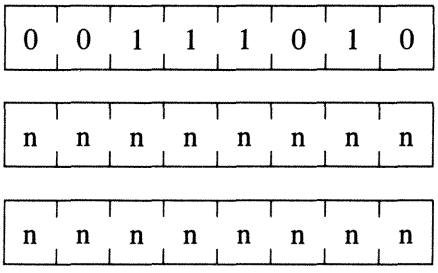
LoaD

Operation: A ◊ (nn)

Format:

Mnemonic: LD **Operands:** A, (nn)

Object Code:



3A

Description:

The contents of the memory location specified by the operands nn are loaded into the Accumulator. The first n operand is the low order byte of a two-byte memory address.

M cycles: 4 T states: 13(4,3,3,3) 4 MHz E.T.: 3.25

Condition Bits Affected: None

Example:

If the contents of memory address 8832H is byte 04H, after the instruction
LD A,(8832H)
byte 04H will be in the Accumulator.

LD (BC),A

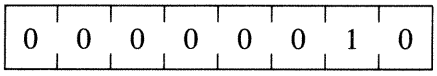
LoaD

Operation: (BC) ◊ A

Format:

Mnemonic: LD **Operands:** (BC), A

Object Code:



02

Description:

The contents of the Accumulator are loaded into the memory location specified by the contents of the register pair BC.

M cycles: 2 T states: 7(4,3) 4 MHz E.T.: 1.75

Condition Bits Affected: None

Example:

If the Accumulator contains 7AH and the BC register pair contains 1212H the instruction
LD (BC),A
will result in 7AH being in memory location 1212H.

LD (DE),A

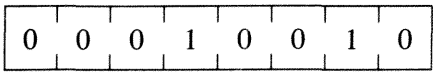
LoaD

Operation: (DE) ◊ A

Format:

Mnemonic: LD **Operands:** (DE), A

Object Code:



12

Description:

The contents of the Accumulator are loaded into the memory location specified by the DE register pair.

M cycles: 2 T states: 7(4,3) 4 MHz E.T.: 1.75

Condition Bits Affected: None

Example:

If the contents of register pair DE are 1128H, and the Accumulator contains byte A0H, the instruction

LD (DE),A

will result in A0H being in memory location 1128H.

LD (nn),A

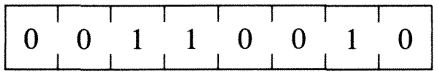
LoaD

Operation: (nn) ⇳ A

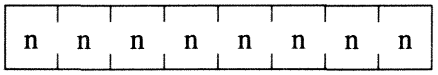
Format:

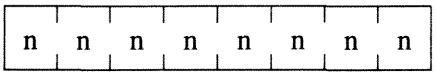
Mnemonic: LD **Operands:** (nn), A

Object Code:



32





Description:

The contents of the Accumulator are loaded into the memory address specified by the operands nn. The first n operand in the assembled object code above is the low order byte of nn.

M cycles: 4 T states: 13(4,3,3,3) 4 MHz E.T.: 3.25

Condition Bits Affected: None

Example:

If the contents of the Accumulator are byte D7H, after the execution of
LD (3141H),A
D7H will be in memory location 3141H.

LD A,I

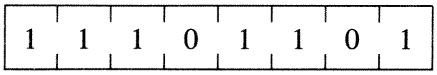
LoaD

Operation: A ⇦ I

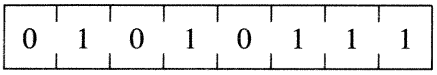
Format:

Mnemonic: LD **Operands:** A, I

Object Code:



ED



57

Description:

The contents of the Interrupt Vector Register I are loaded into the Accumulator.

M cycles: 2 T states: 9(4,5) 4 MHz E.T.: 2.25

Condition Bits Affected:

S: Set if I-Reg. is negative; reset otherwise
Z: Set if I-Reg. is zero; reset otherwise
H: Reset
P/V: Contains contents of IFF2
N: Reset
C: Not affected

Note: If an interrupt occurs during execution of this instruction, the Parity flag will contain a Ø.

Example:

If the Interrupt Vector Register contains the byte 4AH, after the execution of
LD A,I
the accumulator will also contain 4AH.

LD A,R

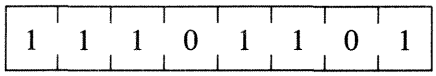
LoaD

Operation: A ⇳ R

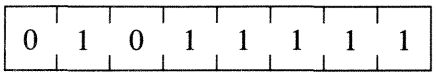
Format:

Mnemonic: LD **Operands:** A, R

Object Code:



ED



5F

Description:

The contents of Memory Refresh Register R are loaded into the Accumulator.

M cycles: 2 T states: 9(4,5) 4 MHz E.T.: 2.25

Condition Bits Affected:

S: Set if R-Reg. is negative; reset otherwise
Z: Set if R-Reg. is zero; reset otherwise
H: Reset
P/V: Contains contents of IFF2
N: Reset
C: Not affected

Example:

If the Memory Refresh Register contains the byte 4AH, after the execution of

LD A,R

the Accumulator will also contain 4AH.

LD I,A

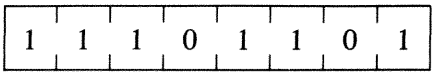
LoaD

Operation: I ⇳ A

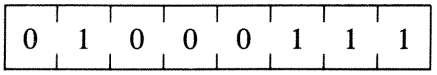
Format:

Mnemonic: LD **Operands:** I, A

Object Code:



ED



47

Description:

The contents of the Accumulator are loaded into the Interrupt Control Vector Register, I.

M cycles: 2 T states: 9(4,5) 4 MHz E.T.: 2.25

Condition Bits Affected: None

Example:

If the Accumulator contains the number 81H, after the instruction

LD I,A

the Interrupt Vector Register will also contain 81H.

LD R,A

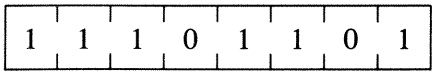
LoaD

Operation: R ◊ A

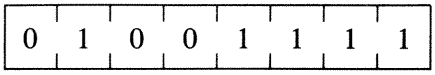
Format:

Mnemonic: LD **Operands:** R, A

Object Code:



ED



4F

Description:

The contents of the Accumulator are loaded into the Memory Refresh register R.

M cycles: 2 T states: 9(4,5) 4 MHz E.T.: 2.25

Condition Bits Affected: None

Example:

If the Accumulator contains the number B4H, after the instruction
LD R,A
the Memory Refresh Register will also contain B4H.

16 Bit Load Group

LD dd,nn

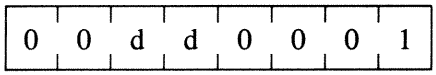
LoaD

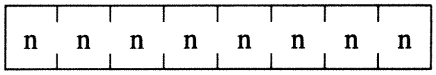
Operation: dd ◊ nn

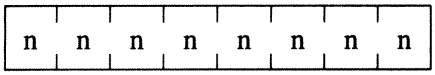
Format:

Mnemonic: LD **Operands:** dd, nn

Object Code:







Description:

The two-byte integer nn is loaded into the dd register pair, where dd defines the BC, DE, HL, or SP register pairs, assembled as follows in the object code:

Pair	dd
BC	00
DE	01
HL	10
SP	11

The first n operand in the assembled object code is the low order byte.

M cycles: 3 T states: 10(4,3,3) 4 MHz E.T.: 2.50

Condition Bits Affected: None

Example:

After the execution of

LD HL,5000H

the contents of the HL register pair will be 5000H.

After the execution of
LD BC,2501H
the BC register will contain 2501H.

LD IX,nn

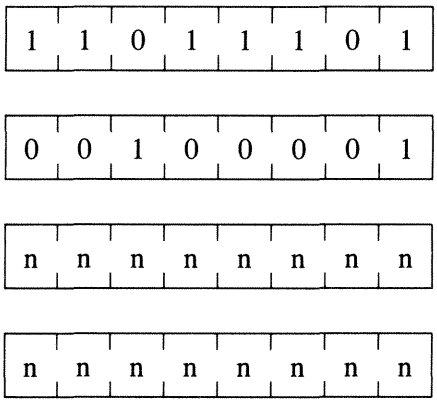
LoaD

Operation: IX ◊ nn

Format:

Mnemonic: LD **Operands:** IX, nn

Object Code:



Description:

Integer nn is loaded into the Index Register IX. The first n operand in the assembled object code above is the low order byte.

M cycles: 4 T states: 14(4,4,3,3) 4 MHz E.T.: 3.50

Condition Bits Affected: None

Example:

After the instruction
LD IX,45A2H
the Index Register will contain integer 45A2H.

LD IY,nn

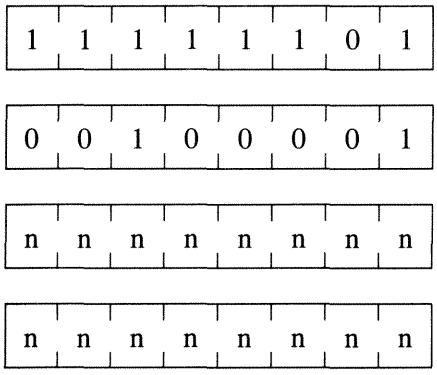
LoaD

Operation: IY ◊ nn

Format:

Mnemonic: LD **Operands:** IY, nn

Object Code:



Description:

Integer nn is loaded into the Index Register IY. The first n operand in the assembled object code above is the low order byte.

M cycles: 4 T states: 14(4,4,3,3) 4 MHz E.T.: 3.50

Condition Bits Affected: None

Example:

After the instruction:
LD IY,7733H
the Index Register IY will contain the integer 7733H.

LD HL,(nn)

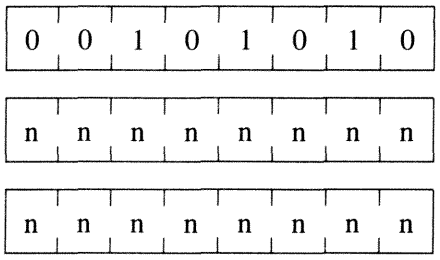
LoaD

Operation: H ◊ (nn + 1), L ◊ (nn)

Format:

Mnemonic: LD **Operands:** HL, (nn)

Object Code:



Description:

The contents of memory address nn are loaded into the low order portion of register pair HL (register L), and the contents of the next highest memory address (nn + 1) are loaded into the high order portion of HL (register H). The first n operand in the assembled object code above is the low order byte of nn.

M cycles: 5 T states: 16(4,3,3,3,3) 4 MHz E.T.: 4.00

Condition Bits Affected: None

Example:

If address 4545H contains 37H and address 4546H contains A1H, after the instruction

LD HL,(4545H)

the HL register pair will contain A137H.

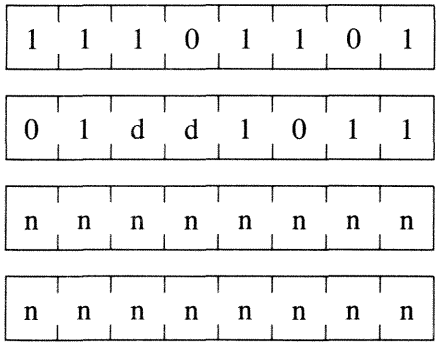
LD dd,(nn) LoaD

Operation: $dd_H \Leftarrow (nn+1)$, $dd_L \Leftarrow (nn)$

Format:

Mnemonic: LD **Operands:** dd, (nn)

Object Code:



Description:

The contents of address nn are loaded into the low order portion of register pair dd, and the contents of the next highest memory address (nn + 1) are loaded into the high order portion of dd. Register pair dd defines BC, DE, HL, or SP register pairs, assembled as follows in the object code:

Pair	dd
BC	00
DE	01
HL	10
SP	11

The first n operand in the assembled object code above is the low order byte of (nn).

M cycles: 6 T states: 20(4,4,3,3,3,3) 4 MHz E.T.: 5.00

Condition Bits Affected: None

Example 1:

If Address 2130H contains 65H and address 2131M contains 78H after the instruction

LD BC,(2130H)

the BC register pair will contain 7865H.

Example 2:

If address FFFE contains 01H and address FFFF contains 02H, then after the instruction

LD SP,(0FFFEH)

the SP will contain 0201H.

LD IX,(nn)

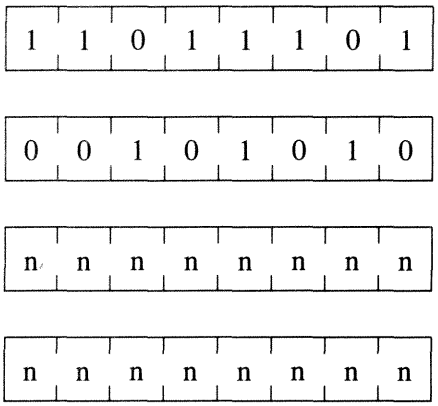
LoaD

Operation: $IX_H \Leftarrow (nn+1)$, $IX_L \Leftarrow (nn)$

Format:

Mnemonic: LD **Operands:** IX, (nn)

Object Code:



Description:

The contents of the address nn are loaded into the low order portion of Index Register IX, and the contents of the next highest memory address (nn + 1 are loaded into the high order portion of IX. The first n operand in the assembled object code above is the low order byte of nn.

M cycles: 6 T states: 20(4,4,3,3,3,3) 4 MHz E.T.: 5.00

Condition Bits Affected: None

Example:

If address 6066H contains 92H and address 6067H contains DAH, after the instruction

LD IX,(6066H)

the Index Register IX will contain DA92H.

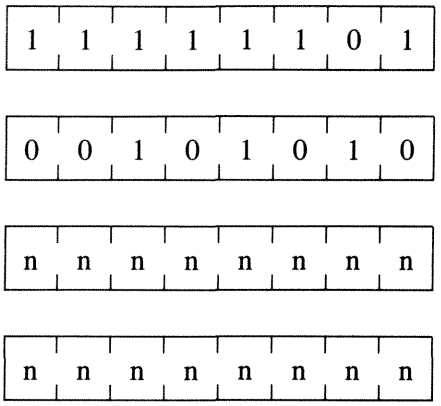
LD IY,(nn) LoaD

Operation: $IY_H \leftarrow (nn+1), IY_L \leftarrow (nn)$

Format:

Mnemonic: LD **Operands:** IY, (nn)

Object Code:



Description:

The contents of address nn are loaded into the low order portion of Index Register IY, and the contents of the next highest memory address (nn + 1) are loaded into the high order portion of IY. The first n operand in the assembled object code above is the low order byte of nn.

M cycles: 6 T states: 20(4,4,3,3,3,3) 4 MHz E.T.: 5.00

Condition Bits Affected: None

Example:

If address 6666H contains 92H and address 6667H contains DAH, after the instruction

LD IY,(6666H)

the Index Register IY will contain DA92H.

LD (nn),HL

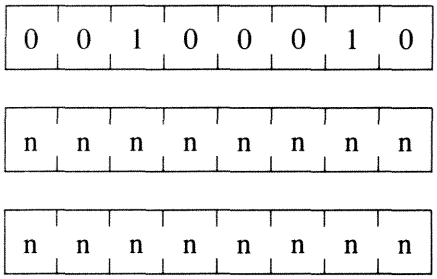
LoaD

Operation: (nn + 1) ◊ H, (nn) ◊ L

Format:

Mnemonic: LD **Operands:** (nn), HL

Object Code:



Description:

The contents of the low order portion of register pair HL (register L) are loaded into memory address nn, and the contents of the high order portion of HL (register H) are loaded into the next highest memory address (nn + 1). The first n operand in the assembled object code above is the low order byte of nn.

M cycles: 5 T states: 16(4,3,3,3,3) 4 MHz E.T.: 4.00

Condition Bits Affected: None

Example 1:

If the content of register pair HL is 483AH, after the instruction

LD (B229H),HL

address B229H will contain 3AH, and address B22AH will contain 48H.

Example 2:

If the register pair HL contains 504AH, then after the instruction

LD (PLACE),HL

the address PLACE will contain 4AH and address PLACE + 1 will contain 50H.

Note: PLACE is a label which must be defined elsewhere in the program.

LD (nn),dd

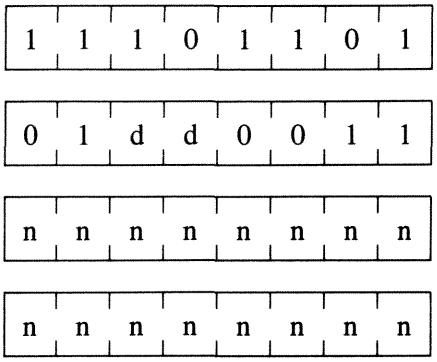
LoaD

Operation: $(nn + 1) \leftarrow dd_H$, $(nn) \leftarrow dd_L$

Format:

Mnemonic: LD **Operands:** (nn), dd

Object Code:



Description:

The low order byte of register pair dd is loaded into memory address (nn); the upper order byte is loaded into memory address (nn + 1). Register pair dd defines either BC, DE, HL, or SP, assembled as follows in the object code:

Pair	dd
BC	00
DE	01
HL	10
SP	11

The first n operand in the assembled object code is the low order byte of a two byte memory address.

M cycles: 6 T states: 20(4,4,3,3,3,3) 4 MHz E.T.: 5.00

Condition Bits Affected: None

Example:

If register pair BC contains the number 4644H, the instruction

LD (1000H),BC

will result in 44H in memory location 1000H, and 46H in memory location 1001H.

LD (nn),IX

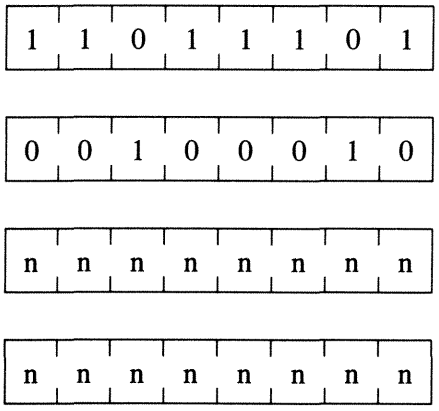
LoaD

Operation: $(nn+1) \Leftarrow IX_H$, $(nn) \Leftarrow IX_L$

Format:

Mnemonic: LD **Operands:** (nn), IX

Object Code:



Description:

The low order byte in Index Register IX is loaded into memory address nn; the upper order byte is loaded into the next highest address (nn + 1). The first n operand in the assembled object code above is the low order byte of nn.

M cycles: 6 T states: 20(4,4,3,3,3,3) 4 MHz E.T.: 5.00

Condition Bits Affected: None

Example:

If the Index Register IX contains 5A30H, after the instruction

LD (4392H),IX

memory location 4392H will contain number 30H and location 4393H will contain 5AH.

LD (nn),IY

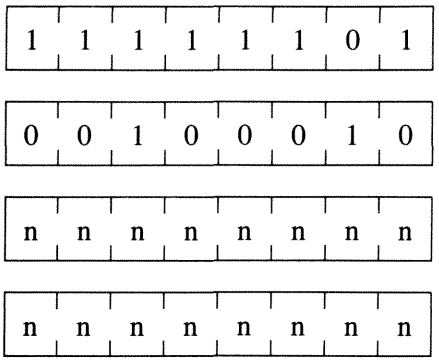
LoaD

Operation: $(nn+1) \Leftarrow IY_H$, $(nn) \Leftarrow IY_L$

Format:

Mnemonic: LD **Operands:** (nn), IY

Object Code:



Description:

The low order byte in Index Register IY is loaded into memory address nn; the upper order byte is loaded into memory location (nn + 1). The first n operand in the assembled object code above is the low order byte of nn.

M cycles: 6 T states: 20(4,4,3,3,3,3) 4 MHz E.T.: 5.00

Condition Bits Affected: None

Example:

If the Index Register IY contains 4174H after the instruction

LD 8838H,IY

memory location 8838H will contain number 74H and memory location 8839H will contain 41H.

LD SP,HL

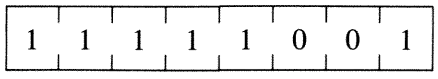
LoaD

Operation: SP ◊ HL

Format:

Mnemonic: LD **Operands:** SP, HL

Object Code:



Description:

The contents of the register pair HL are loaded into the Stack Pointer SP.

M cycles: 1 T states: 6 4 MHz E.T.: 1.50

Condition Bits Affected: None

Example:

If the register pair HL contains 442EH, after the instruction
LD SP,HL
the Stack Pointer will also contain 442EH.

LD SP,IX

LoaD

Operation: SP ◊ IX

Format:

Mnemonic: LD **Operands:** SP, IX

Object Code:

1	1	0	1	1	1	0	1	DD

1	1	1	1	1	0	0	1	F9

Description:

The two-byte contents of Index Register IX are loaded into the Stack Pointer SP.

M cycles: 2 T states: 10(4,6) 4 MHz E.T.: 2.50

Condition Bits Affected: None

Example:

If the contents of the Index Register IX are 98DAH, after the instruction
LD SP,IX
the contents of the Stack Pointer will also be 98DAH.

LD SP,IY

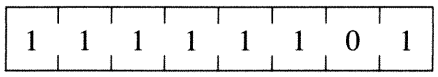
LoaD

Operation: SP ⇳ IY

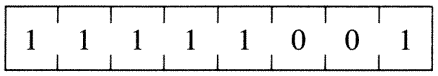
Format:

Mnemonic: LD **Operands:** SP, IY

Object Code:



FD



F9

Description:

The two byte contents of Index Register IY are loaded into the Stack Pointer SP.

M cycles: 2 T states: 10(4,6) 4 MHz E.T.: 2.50

Condition Bits Affected: None

Example:

If Index Register IY contains the integer A227H, after the instruction
LD SP,IY
the Stack Pointer will also contain A227H.

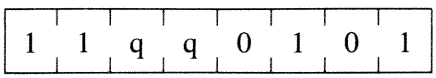
PUSH qq

Operation: $(SP-2) \Leftrightarrow qq_L$, $(SP-1) \Leftrightarrow qq_H$

Format:

Mnemonic: PUSH **Operands:** qq

Object Code:



Description:

The contents of the register pair qq are pushed into the external memory LIFO (last-in, first-out) Stack. The Stack Pointer (SP) register pair holds the 16-bit address of the current "top" of the Stack. This instruction first decrements the SP and loads the high order byte of register pair qq into the memory address now specified by the SP, then decrements the SP again and loads the low order byte of qq into the memory location corresponding to this new address in the SP. The operand qq means register pair BC, DE, HL, or AF, assembled as follows in the object code:

Pair	qq
BC	00
DE	01
HL	10
AF	11

M cycles: 3 T states: 11(5,3,3) 4 MHz E.T.: 2.75

Condition Bits Affected: None

Example:

If the AF register pair contains 2233H and the Stack Pointer contains 1007H, after the instruction

PUSH AF

memory address 1006H will contain 22H, memory address 1005H will contain 33H, and the Stack Pointer will contain 1005H. In other words the number from register pair AF is now on the top of the stack, and the stack pointer is pointing to it.

Before:

Register AF	Address	Stack
2233	1007	FF
	1008	35

Stack Pointer

1007

After: PUSH AF

Register AF	Address	Stack
2233	1005	33
	1006	22
	1007	FF
	1008	35

Stack Pointer

1005

PUSH IX

Operation: $(SP-2) \Leftarrow IX_L$, $(SP-1) \Leftarrow IX_H$

Format:

Mnemonic: PUSH **Operands:** IX

Object Code:

1	1	0	1	1	1	0	1	DD

1	1	1	0	0	1	0	1	E5

Description:

The contents of the Index Register IX are pushed into the external memory LIFO (last-in, first-out) Stack. The Stack Pointer (SP) register pair holds the 16-bit address of the current "top" of the Stack. This instruction first decrements the SP and loads the high order byte of IX into the memory address now specified by the SP, then decrements the SP again and loads the low order byte into the memory location corresponding to this new address in the SP.

M cycles: 3 T states: 15(4,5,3,3) 4 MHz E.T.: 3.75

Condition Bits Affected: None

Example:

If the Index Register IX contains 2233H and the Stack Pointer contains 1007H, after the instruction

PUSH IX

memory address 1006H will contain 22H, memory address 1005H will contain 33H, and the Stack Pointer will contain 1005H. The number from the IX register pair is now on the top of the stack.

Before:

Register IX	Address	Stack
2233	1007	FF
	1008	35

Stack Pointer

1007

After: PUSH IX

Register IX	Address	Stack
2233	1005	33
	1006	22
	1007	FF
	1008	35

Stack Pointer

1005

PUSH IY

Operation: $(SP-2) \leftarrow IY_L$, $(SP-1) \leftarrow IY_H$

Format:

Mnemonic: PUSH **Operands:** IY

Object Code:

1	1	1	1	1	1	0	1	FD

1	1	1	0	0	1	0	1	E5

Description:

The contents of the Index Register IY are pushed into the external memory LIFO (last-in, first-out) Stack. The Stack Pointer (SP) register pair holds the 16-bit address of the current "top" of the Stack. This instruction first decrements the SP and loads the high order byte of IY into the memory address now specified by the SP; then decrements the SP again and loads the low order byte into the memory location corresponding to this new address in the SP.

M cycles: 4 T states: 15(4,5,3,3) 4 MHz E.T.: 3.75

Condition Bits Affected: None

Example:

If the Index Register IY contains 2233H and the Stack Pointer contains 1007H, after the instruction

PUSH IY

memory address 1006H will contain 22H, memory address 1005H will contain 33H, and the Stack Pointer will contain 1005H. The number from register pair IY is now on the top of the stack.

Before:

Register IY	Address	Stack
2233	1007	FF
	1008	35

Stack Pointer
1007

After: PUSH IY

Register IY	Address	Stack
2233	1005	33
	1006	22
	1007	FF
	1008	35

Stack Pointer
1005

POP qq

Operation: $qq_H \Diamond (SP+1), qq_L \Diamond (SP)$

Format:

Mnemonic: POP **Operands:** qq

Object Code:

1	1	q	q	0	0	0	1

Description:

The top two bytes of the external memory LIFO (last-in, first-out) Stack are popped into register pair qq. The Stack Pointer (SP) register pair holds the 16-bit address of the current "top" of the Stack. This instruction first loads into the low order portion of qq, the byte at the memory location corresponding to the contents of SP; then SP is incremented and the contents of the corresponding adjacent memory location are loaded into the high order portion of qq and the SP is now incremented again. The operand qq defines register pair BC, DE, HL, or AF, assembled as follows in the object code:

Pair	r
BC	00
DE	01
HL	10
AF	11

M cycles: 3 T states: 10(4,3,3) 4 MHz E.T.: 2.50

Condition Bits Affected: None

Example:

If the Stack Pointer contains 1000H, memory location 1000H contains 55H, and location 1001H contains 33H, the instruction

POP HL

will result in register pair HL containing 3355H, and the Stack Pointer containing 1002H. In other words register pair HL contains the number which was on the top of the stack, and the stack pointer is pointing to the current top of the stack.

Before:

Register HL	Address	Stack
2233	1000	55
	1001	33
	1002	A4
	1003	62

Stack Pointer

1000

After: POP HL

Register HL	Address	Stack
3355	1002	A4
	1003	62

Stack Pointer

1002

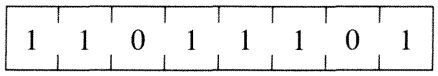
POP IX

Operation: $IX_H \leftrightarrow (SP+1)$, $IX_L \leftrightarrow (SP)$

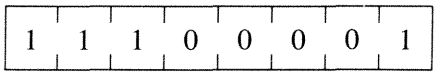
Format:

Mnemonic: POP **Operands:** IX

Object Code:



DD



E1

Description:

The top two bytes of the external memory LIFO (last-in, first-out) Stack are popped into Index Register IX. The Stack Pointer (SP) register pair holds the 16-bit address of the current "top" of the Stack. This instruction first loads into the low order portion of IX the byte at the memory location corresponding to the contents of SP; then SP is incremented and the contents of the corresponding adjacent memory location are loaded into the high order portion of IX. The SP is now incremented again.

M cycles: 4 T states: 14(4,4,3,3) 4 MHz E.T.: 3.50

Condition Bits Affected: None

Example:

If the Stack Pointer contains 1000H, memory location 1000H contains 55H, and location 1001H contains 33H, the instruction

POP IX

will result in the Index Register IX containing 3355H, and the Stack Pointer containing 1002H. Register pair IX contains the number which used to be on the top of the stack.

Before:

Register IX	Address	Stack
24F9	1000	55
	1001	33
	1002	A4
	1003	62

Stack Pointer

1000

After: POP IX

Register IX	Address	Stack
3355	1002	A4
	1003	62

Stack Pointer
1002

POP IY

Operation: $IY_H \Leftarrow (SP+1), IY_L \Leftarrow (SP)$

Format:

Mnemonic: POP **Operands:** IY

Object Code:

1	1	1	1	1	1	0	1	FD

1	1	1	0	0	0	0	1	E1

Description:

The top two bytes of the external memory LIFO (last-in, first-out) Stack are popped into Index Register IY. The Stack Pointer (SP) register pair holds the 16-bit address of the current "top" of the Stack. This instruction first loads into the low order portion of IY the byte at the memory location corresponding to the contents of SP; then SP is incremented and the contents of the corresponding adjacent memory location are loaded into the high order portion of IY. The SP is now incremented again.

M cycles: 4 T states: 14(4,4,3,3) 4 MHz E.T.: 3.5Ø

Condition Bits Affected: None

Example:

If the Stack Pointer contains 1ØØØH, memory location 1ØØØH contains 55H, and location 1ØØ1H contains 33H, the instruction

POP IY

will result in Index Register IY containing 3355H, and the Stack Pointer containing 1ØØ2H. Register pair IY contains the number which used to be on the top of the stack.

Before:

Register IY	Address	Stack
24F9	1000	55
	1001	33
	1002	A4
	1003	62

Stack Pointer

1000

After: POP IY

Register IY	Address	Stack
3355	1002	A4
	1003	62

Stack Pointer

1002

Exchange, Block Transfer and Search Group

EX DE,HL

EXchange

Operation: DE ⇔ HL

Format:

Mnemonic: EX **Operands:** DE, HL

Object Code:

1	1	1	0	1	0	1	1

EB

Description:

The two-byte contents of register pairs DE and HL are exchanged.

M cycles: 1 T states: 4 4 MHz E.T.: 1.00

Condition Bits Affected: None

Example:

If the content of register pair DE is the number 2822H, and the content of the register pair HL is number 499AH, after the instruction

EX DE,HL

the content of register pair DE will be 499AH and the content of register pair HL will be 2822H.

EX AF,AF′

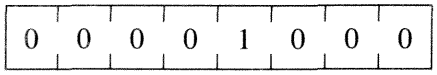
EXchange

Operation: AF ⇔ AF′

Format:

Mnemonic: EX **Operands:** AF, AF′

Object Code:



08

Description:

The two-byte contents of the register pairs AF and AF′ are exchanged. (Note: register pair AF′ consists of registers A′ and F.′)

M cycles: 1 T states: 4 4 MHz E.T.: 1.00

Condition Bits Affected: None

Example:

If the content of register pair AF is number 9900H, and the content of register pair AF′ is number 5944H, after the instruction

EX AF,AF′

the contents of AF will be 5944H, and the contents of AF′ will be 9900H.

EXX

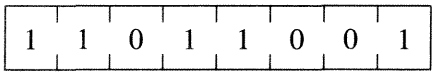
EXchange

Operation: (BC) ⇔ (BC′), (DE) ⇔ (DE′), (HL) ⇔ (HL′)

Format:

Mnemonic: EXX **Operands:**

Object Code:



D9

Description:

Each two-byte value in register pairs BC, DE, and HL is exchanged with the two-byte value in BC,′ DE,′ and HL,′ respectively.

M cycles: 1 T states: 4 4 MHz E.T.: 1.00

Condition Bits Affected: None

Example 1:

If the contents of register pairs BC, DE, and HL are the numbers 445AH, 3DA2H, and 8859H, respectively, and the contents of register pairs BC,′ DE,′ and HL′ are 0988H, 9300H, and 00E7H, respectively, after the instruction

EXX

the contents of the register pairs will be as follows: BC: 0988H; DE: 9300H; HL: 00E7H; BC′: 445AH; DE′: 3DA2H; and HL′: 8859H.

Example 2:

If the contents of the registers are as shown:

BC : 1111H
DE : 2222H
HL : 3333H
BC′ : 4444H
DE′ : 5555H
HL′ : 6666H

Then after an EXX instruction the registers will contain:

BC : 4444H
DE : 5555H
HL : 6666H
BC′ : 1111H
DE′ : 2222H
HL′ : 3333H

EX (SP), HL

EXchange

Operation: H ⇔ (SP + 1), L ⇔ (SP)

Format:

Mnemonic: EX **Operands:** (SP),HL

Object Code:

1	1	1	0	0	0	1	1

E3

Description:

The low order byte contained in register pair HL is exchanged with the contents of the memory address specified by the contents of register pair SP (Stack Pointer), and the high order byte of HL is exchanged with the next highest memory address (SP + 1).

M cycles: 5 T states: 19(4,3,4,3,5) 4 MHz E.T.: 4.75

Condition Bits Affected: None

Example:

If the HL register pair contains 7Ø12H, the SP register pair contains 8856H, the memory location 8856H contains the byte 11H, and the memory location 8857H contains the byte 22H, then the instruction

EX (SP),HL

will result in the HL register pair containing number 2211H, memory location 8856H containing the byte 12H, the memory location 8857H containing the byte 7ØH and the Stack Pointer containing 8856H.

Before:

Register HL	Address	Stack
7012	8856	11
	8857	22
	8858	

Stack Pointer

8856

After:

Register HL	Address	Stack
2211	8856	12
	8857	70
	8858	

Stack Pointer

8856

EX (SP),IX

EXchange

Operation: $IX_H \Leftrightarrow (SP+1)$, $IX_L \Leftrightarrow (SP)$

Format:

Mnemonic: EX **Operands:** (SP), IX

Object Code:

1	1	0	1	1	1	0	1	DD

1	1	1	0	0	0	1	1	E3

Description:

The low order byte in Index Register IX is exchanged with the contents of the memory address specified by the contents of register pair SP (Stack Pointer), and the high order byte of IX is exchanged with the next highest memory address (SP + 1).

Condition Bits Affected: None

Example:

If the Index Register IX contains 3988H, the SP register pair contains 0100H, the memory location 0100H contains the byte 90H, and memory location 0101H contains byte 48H, then the instruction

EX (SP),IX

will result in the IX register pair containing number 4890H, memory location 0100H containing 88H, memory location 0101H containing 39H and the Stack Pointer containing 0100H.

Before:

Register IX	Address	Stack
3988	0100	90
	0101	48

Stack Pointer

0100

After:

Register IX	Address	Stack
4890	0100	88
	0101	39

Stack Pointer

0100

EX (SP),IY

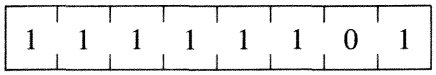
EXchange

Operation: $IY_H \Leftrightarrow (SP + 1)$, $IY_L \Leftrightarrow (SP)$

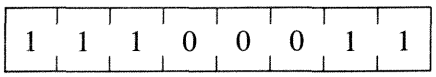
Format:

Mnemonic: EX **Operands:** (SP), IY

Object Code:



FD



E3

Description:

The low order byte in Index Register IY is exchanged with the contents of the memory address specified by the contents of register pair SP (Stack Pointer), and the high order byte of IY is exchanged with the next highest memory address (SP + 1).

M cycles: 6 T states: 23(4,4,3,4,3,5) 4 MHz E.T.: 5.75

Condition Bits Affected: None

Example:

If the Index Register IY contains 3988H, the SP register pair contains 0100H, the memory location 0100H contains the byte 90H, and memory location 0101H contains byte 48H, then the instruction

EX (SP),IY

will result in the IY register pair containing number 4890H, memory location 0100H containing 88H, memory location 0101H containing 39H, and the Stack Pointer containing 0100H.

Before:

Register IY	Address	Stack
3988	0100	90
	0101	48

Stack Pointer

0100

After:

Register IY	Address	Stack
4890	0100	88
	0101	39

Stack Pointer

0100

LDI

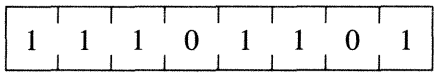
LoaD & Increment

Operation: (DE) ◊ (HL), DE ◊ DE + 1, HL ◊ HL + 1, BC ◊ BC − 1

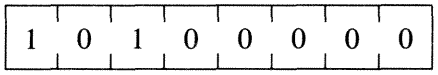
Format:

Mnemonic: LDI **Operands:**

Object Code:



ED



AØ

Description:

A byte of data is transferred from the memory location addressed by the contents of the HL register pair to the memory location addressed by the contents of the DE register pair. Then both these register pairs are incremented and the BC (Byte Counter) register pair is decremented.

M cycles: 4 T states: 16(4,4,3,5) 4 MHz E.T.: 4.ØØ

Condition Bits Affected:

S: Not affected
Z: Not affected
H: Reset
P/V: Set if BC − 1 ≠ Ø; reset otherwise
N: Reset
C: Not affected

Example 1:

If the HL register pair contains 1111H, memory location 1111H contains the byte 88H, the DE register pair contains 2222H, the memory location 2222H contains byte 66H, and the BC register pair contains 7H, then the instruction

LDI

will result in the following contents in register pairs and memory addresses:

HL : 1112H
(1111H) : 88H
DE : 2223H
(2222H) : 88H
BC : 6H

and the condition Bits will be:

		0	1	0	
S	Z	H	P/V	N	C

Example 2:

If the contents of registers and memory are as shown:

HL : 7C00H
(7C00) : FFH
DE : 3C00H
(3C00) : 00H
BC : 1H

Then after an LDI instruction the registers and memory will contain the following:

HL : 7C01H
(7C00) : FFH
DE : 3C01H
(3C00) : FFH
BC : 0H

and the condition bits will be:

		0	0	0	
S	Z	H	P/V	N	C

Example 3:

The following program will move 80 consecutive bytes from BUF1 to BUF2:

```
LD      HL, BUF1
LD      DE, BUF2
LD      BC, 80
LOOP    LDI
JP      NZ, LOOP
```

LDIR

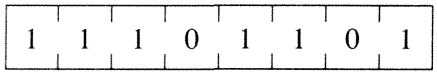
LoaD Increment & Repeat

Operation: (DE) ◊ (HL), DE ◊ DE + 1, HL ◊ HL + 1, BC ◊ BC − 1

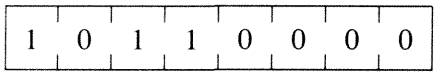
Format:

Mnemonic: LDIR **Operands:**

Object Code:



ED



B0

Description:

This two-byte instruction transfers a byte of data from the memory location addressed by the contents of the HL register pair to the memory location addressed by the DE register pair. Then both these register pairs are incremented and the BC (Byte Counter) register pair is decremented. If decrementing causes the BC to go to zero, the instruction is terminated. If BC is not zero the program counter (PC) is decremented by 2 and the instruction is repeated. Note that if BC is set to zero prior to instruction execution, the instruction will loop through 64K bytes. Also, interrupts will be recognized after each data transfer.

For BC ≠ 0:

M cycles: 5 T states: 21(4,4,3,5,5) 4 MHz E.T.: 5.25

For BC = 0:

M cycles: 4 T states: 16(4,4,3,5) 4 MHz E.T.: 4.00

Condition Bits Affected:

S: Not affected
Z: Not affected
H: Reset
P/V: Reset
N: Reset
C: Not affected

Example:

If the HL register pair contains 1111H, the DE register pair contains 2222H, the BC register pair contains 0003H, and memory locations have these contents:

(1111H)	:	88H	(2222H)	:	66H
(1112H)	:	36H	(2223H)	:	59H
(1113H)	:	A5H	(2224H)	:	C5H

then after the execution of

LDIR

the contents of register pairs and memory locations will be:

HL	:	1114H			
DE	:	2225H			
BC	:	0000H			
(1111H)	:	88H	(2222H)	:	88H
(1112H)	:	36H	(2223H)	:	36H
(1113H)	:	A5H	(2224H)	:	A5H

and the H, P/V, and N flags are all zero.

LDD

LoaD Decrement

Operation: (DE) ◊ (HL), DE ◊ DE – 1, HL ◊ HL – 1, BC ◊ BC – 1

Format:

Mnemonic: LDD **Operands:**

Object Code:

1	1	1	0	1	1	0	1	ED

1	0	1	0	1	0	0	0	A8

Description:

This two-byte instruction transfers a byte of data from the memory location addressed by the contents of the HL register pair to the memory location addressed by the contents of the DE register pair. Then both of these register pairs, including the BC (Byte Counter) register pair, are decremented.

M cycles: 4 T states: 16(4,4,3,5) 4 MHz E.T.: 4.00

Condition Bits Affected:

S:	Not affected
Z:	Not affected
H:	Reset
P/V:	Set if BC – 1 ≠ 0; reset otherwise
N:	Reset
C:	Not affected

Example 1:

If the HL register pair contains 1111H, memory location 1111H contains the byte 88H, the DE register pair contains 2222H, memory location 2222H contains byte 66H, and the BC register pair contains 7H, then the instruction

LDD

will result in the following contents in register pairs and memory addresses:

HL : 1110H
(1111H) : 88H
DE : 2221H
(2222H) : 88H
BC : 6H

and the condition bits will be:

		0	1	0	
S	Z	H	P/V	N	C

Example 2:

If the contents of registers and memory are as shown:

HL : 7CFFH
(7CFF) : 3CH
DE : 3CFFH
(3CFF) : 00H
BC : 1H

Then after a LDD instruction the registers and memory will contain the following:

HL : 7CFEH
(7CFF) : 3CH
DE : 3CFEH
(3CFF) : 3CH
BC : 0H

and the condition bits will be:

		0	0	0	
S	Z	H	P/V	N	C

LDDR

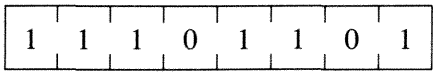
LoaD Decrement & Repeat

Operation: (DE) ◊ (HL), DE ◊ DE – 1, HL ◊ HL – 1, BC ◊ BC – 1

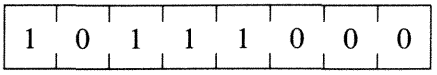
Format:

Mnemonic: LDDR **Operands:**

Object Code:



ED



B8

Description:

This two-byte instruction transfers a byte of data from the memory location addressed by the contents of the HL register pair to the memory location addressed by the contents of the DE register pair. Then both of these registers as well as the BC (Byte Counter) are decremented. If decrementing causes the BC to go to zero, the instruction is terminated. If BC is not zero, the program counter (PC) is decremented by 2 and the instruction is repeated. Note that if BC is set to zero prior to instruction execution, the instruction will loop through 64K bytes. Also, interrupts will be recognized after each data transfer.

For BC $\neq$ 0:

M cycles: 5 T states: 21(4,4,3,5,5) 4 MHz E.T.: 5.25

For BC = 0:

M cycles: 4 T states: 16(4,4,3,5) 4 MHz E.T.: 4.00

Condition Bits Affected:

S: Not affected
Z: Not affected
H: Reset
P/V: Reset
N: Reset
C: Not affected

Example:

If the HL register pair contains 1114H, the DE register pair contains 2225H, the BC register pair contains 0003H, and memory locations have these contents:

(1114H)	:	A5H	(2225H)	:	C5H
(1113H)	:	36H	(2224H)	:	59H
(1112H)	:	88H	(2223H)	:	66H

then after the execution of

LDDR

the contents of register pairs and memory locations will be:

HL	:	1111H			
DE	:	2222H			
BC	:	0000H			
(1114H)	:	A5H	(2225H)	:	A5H
(1113H)	:	36H	(2224H)	:	36H
(1112H)	:	88H	(2223H)	:	88H

and the H, P/V, and N flags are all zero.

CPI — ComPare & Increment

Operation: A − (HL), HL ◊ HL + 1, BC ◊ BC − 1

Format:

Mnemonic: CPI **Operands:**

Object Code:

1	1	1	0	1	1	0	1	ED

1	0	1	0	0	0	0	1	A1

Description:

The contents of the memory location addressed by the HL register pair is compared with the contents of the Accumulator. In case of a true compare, the Z condition bit is set. Then HL is incremented and the Byte Counter (register pair BC) is decremented.

M cycles: 4 T states: 16(4,4,3,5) 4 MHz E.T.: 4.00

Condition Bits Affected:

S:	Set if result is negative; reset otherwise
Z:	Set if A = (HL); reset otherwise
H:	Set if borrow from Bit 4; reset otherwise
P/V:	Reset if BC becomes 0; set otherwise
N:	Set
C:	Not affected

Example:

If the HL register pair contains 1111H, memory location 1111H contains 3BH, the Accumulator contains 3BH, and the Byte Counter contains 0001H, then after the execution of

CPI

the Byte Counter will contain 0000H, the HL register pair will contain 1112H, the Z flag in the F register will be set, and the P/V flag in the F register will be reset. There will be no effect on the contents of the Accumulator or address 1111H.

If the contents of memory and registers are as shown

HL : 8A00H
(8A00H) : 6DH
A : 75H
BC : 5H

Then during the execution of a CPI instruction the Arithmetic and Logic Unit will do the following subtraction:

Borrow needed here

```
          ⇦
  75H = 0111 0101
- 6DH = 0110 1101
-------------------
   8H = 0000 1000
```

After CPI is executed registers and memory will contain the following:

HL : 8A01H
(8A00H) : 6DH
A : 75H
BC : 4H

and the condition bits would be:

0	0	1	1	1	
S	Z	H	P/V	N	C
result positive	⇦					not affected
match not found		⇦				always set
borrow from bit 4			⇦			BC not zero

Example 3:

The following program is used to verify that the contents of two 80-byte buffers are identical. Each time a mismatch is found the program calls a subroutine called ERROR.

```
STRT    LD      HL, BUF1
        LD      DE, BUF2
        LD      BC, 80
LOOP    LD      A, (DE)
        CPI
        CALL    NZ, ERROR
        INC     DE
        JR      PO, LOOP
END
```

CPIR

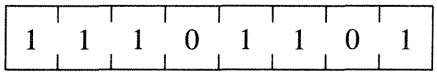
ComPare Increment & Repeat

Operation: $A - (HL), HL \Leftarrow HL + 1, BC \Leftarrow BC - 1$

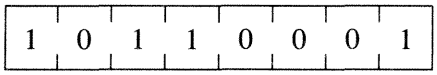
Format:

Mnemonic: CPIR **Operands:**

Object Code:



ED



B1

Description:

The contents of the memory location addressed by the HL register pair is compared with the contents of the Accumulator. In case of a true compare, the Z condition bit is set. The HL is incremented and the Byte Counter (register pair BC) is decremented. If decrementing causes the BC to go to zero or if A = (HL), the instruction is terminated. If BC is not zero and A ≠ (HL), the program counter is decremented by 2 and the instruction is repeated. Note that if BC is set to zero before the execution, the instruction will loop through 64K bytes, if no match is found. Also, interrupts will be recognized after each data comparison.

For BC ≠ 0 and A ≠ (HL):

M cycles: 5 T states: 21(4,4,3,5,5) 4 MHz E.T.: 5.25

For BC ≠ 0 or A = (HL):

M cycles: 4 T states: 16(4,4,3,5) 4 MHz E.T.: 4.00

Condition Bits Affected:

S:	Set if result is negative; reset otherwise
Z:	Set if A = (HL); reset otherwise
H:	Set if borrow from Bit 4; reset otherwise
P/V:	Reset if BC becomes 0; set otherwise
N:	Set
C:	Not affected

Example:

If the HL register pair contains 1111H, the Accumulator (Register A) contains F3H, the Byte Counter contains 0007H, and memory locations have these contents:

(1111H) : 52H
(1112H) : 00H
(1113H) : F3H

then after the execution of

CPIR

the contents of register pair HL will be 1114H, and the contents of the Byte Counter will be 0004H. Since BC ≠ 0, the P/V flag is still set. This means that it did not search through the whole block before the instruction stopped. Since a match was found, the Z flag is set.

The following program uses the CPIR instruction to count the number of nulls (00H) found in an 80-byte buffer. The count is kept in register E.

```
STRT    LD      BC, 80
        LD      HL, BUFF
        LD      A, 0
        LD      E, 0
LOOP    CPIR
        JR      NZ, FOO
        INC     E
FOO     JP      PE, LOOP
END
```

CPD

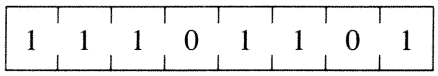
ComPare & Decrement

Operation: A – (HL), HL ◊ HL – 1, BC ◊ BC – 1

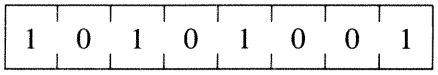
Format:

Mnemonic: CPD **Operands:**

Object Code:



ED



A9

Description:

The contents of the memory location addressed by the HL register pair is compared with the contents of the Accumulator. In case of a true compare, the Z condition bit is set. The HL and the Byte Counter (register pair BC) are decremented.

M cycles: 4 T states: 16(4,4,3,5) 4 MHz E.T.: 4.00

Condition Bits Affected:

S: Set if result is negative; reset otherwise
Z: Set if A = (HL); reset otherwise
H: Set if borrow from Bit 4; reset otherwise
P/V: Reset if BC becomes zero; set otherwise
N: Set
C: Not affected

Example:

If the HL register pair contains 1111H, memory location 1111H contains 3BH, the Accumulator contains 3BH, and the Byte Counter contains 0001H, then after the execution of

CPD

the Byte Counter will contain 0000H, the HL register pair will contain 1110H, the Z flag in the F register will be set and the P/V flag in the F register will be reset. There will be no effect on the contents of the Accumulator or address 1111H.

Since the CPD instruction decrements HL, it is used to search through memory from high to low addresses. Otherwise it is similar to the CPI instruction.

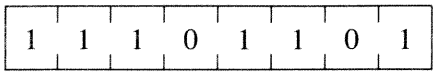
CPDR ComPare Decrement & Repeat

Operation: A – (HL), HL ◊ HL – 1, BC ◊ BC – 1

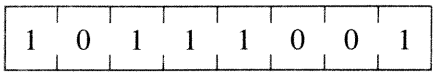
Format:

Mnemonic: CPDR **Operands:**

Object Code:



ED



B9

Description:

The contents of the memory location addressed by the HL register pair is compared with the contents of the Accumulator. In case of a true compare, the Z condition bit is set. The HL and BC (Byte Counter) register pairs are decremented. If decrementing causes the BC to go to zero or if A = (HL), the instruction is terminated. If BC is not zero and A ≠ (HL), the program counter is decremented by 2 and the instruction is repeated. Note that if BC is set to zero prior to instruction execution, the instruction will loop through 64K bytes, if no match is found. Also, interrupts will be recognized after each data comparison.

For BC ≠ 0 and A ≠ (HL):

M cycles: 5 T states: 21(4,4,3,5,5) 4 MHz E.T.: 5.25

For BC = 0 or A = (HL):

M cycles: 4 T states: 16(4,4,3,5) 4 MHz E.T.: 4.00

Condition Bits Affected:

S: Set if result is negative; reset otherwise
Z: Set if A = (HL), reset otherwise
H: Set if borrow from Bit 4; reset otherwise
P/V: Reset if BC becomes zero; set otherwise
N: Set
C: Not affected

Example:

If the HL register pair contains 1118H, the Accumulator contains F3H, the Byte Counter contains 0003H, and memory locations have these contents:

(1118H) : 52H
(1117H) : 00H
(1116H) : F3H

then after the execution of

CPDR

the contents of register pair HL will be 1115H, the contents of the Byte Counter will be 0000H, the P/V flag in the F register will be reset, and the Z flag in the F register will be set.

8 Bit Arithmetic and Logical Group

ADD A,r

Operation: A ◊ A + r

Format:

Mnemonic: ADD **Operands:** A, r

Object Code:

1	0	0	0	0	r	r	r

Description:

The contents of register r are added to the contents of the Accumulator, and the result is stored in the Accumulator. The symbol r identifies the registers A, B, C, D, E, H or L assembled as follows in the object code:

Register		r
A	=	111
B	=	000
C	=	001
D	=	010
E	=	011
H	=	100
L	=	101

M cycles: 1 T states: 4 4 MHz E.T.: 1.00

Condition Bits Affected:

S:	Set if result is negative; reset otherwise
Z:	Set if result is zero; reset otherwise
H:	Set if carry from Bit 3; reset otherwise
P/V:	Set if overflow; reset otherwise
N:	Reset
C:	Set if carry from Bit 7; reset otherwise

Example:

If the contents of the Accumulator are 44H, and the contents of register C are 11H, after the execution of

ADD A,C

the contents of the Accumulator will be 55H. See Appendix K for more details of condition bits affected.

ADD A,n

Operation: A ◊ A + n

Format:

Mnemonic: ADD **Operands:** A, n

Object Code:

1	1	0	0	0	1	1	0

C6

n	n	n	n	n	n	n	n

Description:

The integer n is added to the contents of the Accumulator and the results are stored in the Accumulator.

M cycles: 2 T states: 7(4,3) 4 MHz E.T.: 1.75

Condition Bits Affected:

S: Set if result is negative; reset otherwise
Z: Set if result is zero; reset otherwise
H: Set if carry from Bit 3; reset otherwise
P/V: Set if overflow; reset otherwise
N: Reset
C: Set if carry from Bit 7; reset otherwise

Example:

If the contents of the Accumulator are 23H, after the execution of

ADD A,33H

the contents of the Accumulator will be 56H.

ADD A,(HL)

Operation: A ◊ A + (HL)

Format:

Mnemonic: ADD **Operands:** A, (HL)

Object Code:

1	0	0	0	0	1	1	0

86

Description:

The byte at the memory address specified by the contents of the HL register pair is added to the contents of the Accumulator and the result is stored in the Accumulator.

M cycles: 2 T states: 7(4,3) 4 MHz E.T.: 1.75

Condition Bits Affected:

S: Set if result is negative; reset otherwise
Z: Set if result is zero; reset otherwise
H: Set if carry from Bit 3; reset otherwise
P/V: Set if overflow; reset otherwise
N: Reset
C: Set if carry from Bit 7; reset otherwise

Example:

If the contents of the Accumulator are A0H, and the content of the register pair HL is 2323H, and memory location 2323H contains byte 08H, after the execution of

ADD A,(HL)

the Accumulator will contain A8H.

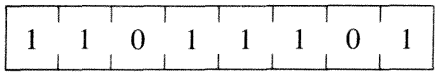
ADD A,(IX + d)

Operation: A ◊ A + (IX + d)

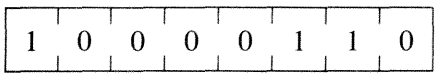
Format:

Mnemonic: ADD **Operands:** A, (IX + d)

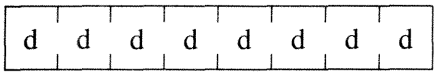
Object Code:



DD



86



Description:

The contents of the Index Register (register pair IX) is added to a two's complement displacement d to point to an address in memory. The contents of this address is then added to the contents of the Accumulator and the result is stored in the Accumulator.

M cycles: 5 T states: 19(4,4,3,5,3) 4 MHz E.T.: 4.75

Condition Bits Affected:

S: Set if result is negative; reset otherwise
Z: Set if result is zero; reset otherwise
H: Set if carry from Bit 3; reset otherwise
P/V: Set if overflow; reset otherwise
N: Reset
C: Set if carry from Bit 7; reset otherwise

Example:

If the Accumulator contents are 11H, the Index Register IX contains 1000H, and if the content of memory location 1005H is 22H, after the execution of

ADD A,(IX + 5H)

the contents of the Accumulator will be 33H.

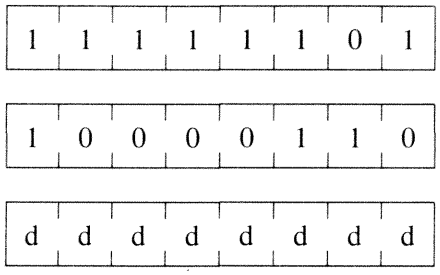
ADD A,(IY + d)

Operation: A ◊ A + (IY + d)

Format:

Mnemonic: ADD **Operands:** A, (IY + d)

Object Code:



Description:

The contents of the Index Register (register pair IY) is added to the displacement d to point to an address in memory. The contents of this address is then added to the contents of the Accumulator and the result is stored in the Accumulator.

M cycles: 5 T states: 19(4,4,3,5,3) 4 MHz E.T.: 4.75

Condition Bits Affected:

S: Set if result is negative; reset otherwise
Z: Set if result is zero; reset otherwise
H: Set if carry from Bit 3; reset otherwise
P/V: Set if overflow; reset otherwise
N: Reset
C: Set if carry from Bit 7; reset otherwise

Example:

If the Accumulator contents are 11H, the Index Register pair IY contains 1000H, and if the content of memory location 1005H is 22H, after the execution of

ADD A,(IY + 5H)

the contents of the Accumulator will be 33H.

ADC A,s

ADd with Carry

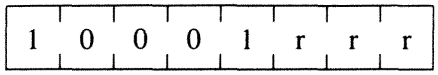
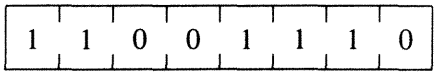
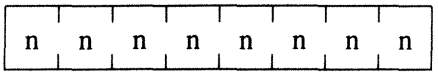
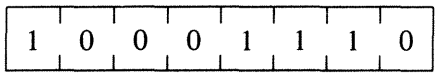
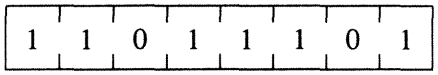
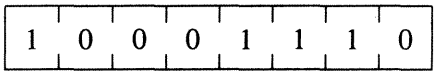
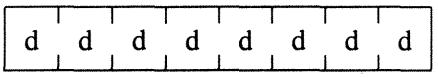
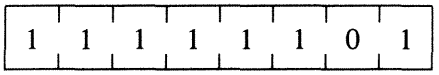
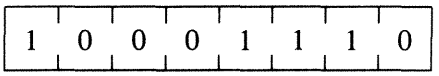
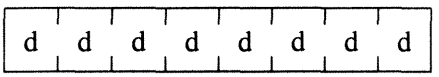
Operation: A ◊ A + s + CY

Format:

Mnemonic: ADC **Operands:** A, s

The s operand is any of r, n, (HL), (IX + d) or (IY + d) as defined for the analogous ADD instruction. These various possible opcode-operand combinations are assembled as follows in the object code:

Object Code:

ADC A, r		
ADC A, n		CE
		
ADC A, (HL)		8E
ADC A, (IX + d)		DD
		8E
		
ADC A, (IY + d)		FD
		8E
		

r identifies registers A, B, C, D, E, H, or L assembled as follows in the object code field above:

Register		**r**
A	=	111
B	=	000
C	=	001
D	=	010
E	=	011
H	=	100
L	=	101

Description:

The s operand, along with the Carry Flag ("C" in the F register) is added to the contents of the Accumulator, and the result is stored in the Accumulator.

Instruction	M Cycles	T States	4 MHz E.T. in μs
ADC A, r	1	4	1.00
ADC A, n	2	7(4,3)	1.75
ADC A, (HL)	2	7(4,3)	1.75
ADC A, (IX + d)	5	19(4,4,3,5,3)	4.75
ADC A, (IY + d)	5	19(4,4,3,5,3)	4.75

Condition Bits Affected:

S: Set if result is negative; reset otherwise
Z: Set if result is zero; reset otherwise
H: Set if carry from Bit 3; reset otherwise
P/V: Set if overflow; reset otherwise
N: Reset
C: Set if carry from Bit 7; reset otherwise

Example 1:

If the Carry Flag is set, the Accumulator contains 16H, the HL register pair contains 6666H, and address 6666H contains 10H, after the execution of

ADC A, (HL)

the Accumulator will contain 27H.

Example 2:

If the Carry Flag is set, the Accumulator contains 30H, and register C contains 05H, then after the execution of

ADC A, C

the Accumulator will contain 36H.

SUB s

SUBtract

Operation: A ← A − s

Format:

Mnemonic: SUB **Operands:** s

The s operand is any of r, n, (HL), (IX + d) or (IY + d) as defined for the analogous ADD instruction. These various possible opcode-operand combinations are assembled as follows in the object code:

Object Code:

SUB r	1 0 0 1 0 r r r	
SUB n	1 1 0 1 0 1 1 0	D6
	n n n n n n n n	
SUB (HL)	1 0 0 1 0 1 1 0	96
SUB (IX + d)	1 1 0 1 1 1 0 1	DD
	1 0 0 1 0 1 1 0	96
	d d d d d d d d	
SUB (IY + d)	1 1 1 1 1 1 0 1	FD
	1 0 0 1 0 1 1 0	96
	d d d d d d d d	

r identifies registers A, B, C, D, E, H or L assembled as follows in the object code field above:

Register		r
A	=	111
B	=	000
C	=	001
D	=	010
E	=	011
H	=	100
L	=	101

Description:

The s operand is subtracted from the contents of the Accumulator, and the result is stored in the Accumulator.

Instruction	M Cycles	T States	4 MHz E.T. in μs
SUB r	1	4	1.00
SUB n	2	7(4,3)	1.75
SUB (HL)	2	7(4,3)	1.75
SUB (IX + d)	5	19(4,4,3,5,3)	4.75
SUB (IY + d)	5	19(4,4,3,5,3)	4.75

Condition Bits Affected:

S: Set if result is negative; reset otherwise
Z: Set if result is zero; reset otherwise
H: Set if borrow from Bit 4; reset otherwise
P/V: Set if overflow; reset otherwise
N: Set
C: Set if borrow; reset otherwise

Example:

If the Accumulator contains 29H and register D contains 11H, after the execution of

SUB D

the Accumulator will contain 18H.

SBC A,s

SuBtract with borrow (Carry)

Operation: A ← A − s − CY

Format:

Mnemonic: SBC **Operands:** A, s

The s operand is any of r, n, (HL), (IX + d) or (IY + d) as defined for the analogous ADD instructions. These various possible opcode-operand combinations are assembled as follows in the object code:

Object Code:

SBC A, r

1	0	0	1	1	r	r	r

SBC A, n

1	1	0	1	1	1	1	0

DE

n	n	n	n	n	n	n	n

SBC A, (HL)	1 0 0 1 1 1 1 0	9E
SBC A, (IX + d)	1 1 0 1 1 1 0 1	DD
	1 0 0 1 1 1 1 0	9E
	d d d d d d d d	
SBC A,(IY + d)	1 1 1 1 1 1 0 1	FD
	1 0 0 1 1 1 1 0	9E
	d d d d d d d d	

r identifies registers A, B, C, D, E, H, or L assembled as follows in the object code field above:

Register		r
A	=	111
B	=	000
C	=	001
D	=	010
E	=	011
H	=	100
L	=	101

Description:

The s operand, along with the Carry Flag ("C" in the F register) is subtracted from the contents of the Accumulator, and the result is stored in the Accumulator.

Instruction	M Cycles	T States	4 MHz E.T. in μs
SBC A, r	1	4	1.00
SBC A, n	2	7(4,3)	1.75
SBC A, (HL)	2	7(4,3)	1.75
SBC A, (IX + d)	5	19(4,4,3,5,3)	4.75
SBC A, (IY + d)	5	19(4,4,3,5,3)	4.75

Condition Bits Affected:

S: Set if result is negative; reset otherwise
Z: Set if result is zero; reset otherwise
H: Set if borrow from Bit 4; reset otherwise
P/V: Set if overflow; reset otherwise
N: Set
C: Set if borrow; reset otherwise

Example 1:

If the Carry Flag is set, the Accumulator contains 16H, the HL register pair contains 3433H, and address 3433H contains 05H, after the execution of

SBC A,(HL)

the Accumulator will contain 10H.

Example 2:

If the Carry Flag is set, the Accumulator contains 21H and register B contains 0, then after the execution of

SBC A,B

the Accumulator contains 20H.

AND s

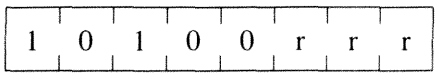
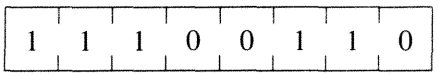
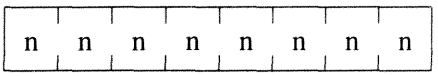
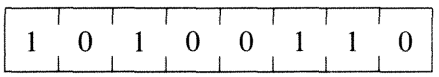
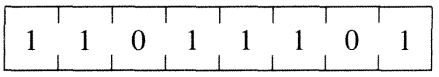
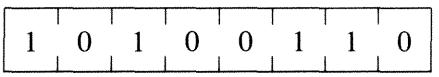
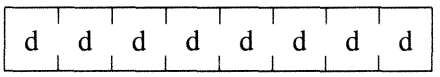
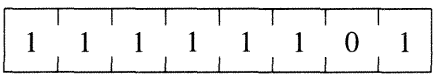
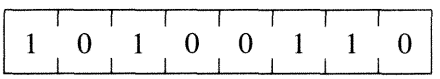
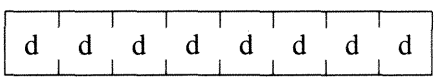
Operation: A ◊ A ⌓ s

Format:

Mnemonic: AND **Operands:** s

The s operand is any of r, n, (HL), (IX + d) or (IY + d), as defined for the analogous ADD instructions. These various possible opcode-operand combinations are assembled as follows in the object code:

Object Code:

AND r		
AND n		E6
		
AND (HL)		A6
AND (IX + d)		DD
		A6
		
AND (IY + d)		FD
		A6
		

r identifies register A, B, C, D, E, H or L assembled as follows in the object code field above:

Register		**r**
A	=	111
B	=	000
C	=	001
D	=	010
E	=	011
H	=	100
L	=	101

Description:

A logical AND operation, Bit by Bit, is performed between the byte specified by the s operand and the byte contained in the Accumulator; the result is stored in the Accumulator.

Instruction	M Cycles	T States	4 MHz E.T. in μs
AND r	1	4	1.00
AND n	2	7(4,3)	1.75
AND (HL)	2	7(4,3)	1.75
AND (IX + d)	5	19(4,4,3,5,3)	4.75
AND (IY + d)	5	19(4,4,3,5,3)	4.75

Condition Bits Affected:

S: Set if result is negative; reset otherwise
Z: Set if result is zero; reset otherwise
H: Set
P/V: Set if parity even; reset otherwise
N: Reset
C: Reset

Table of AND Values:

IF A	IF B	Then A (After)
0	0	0
0	1	0
1	0	0
1	1	1

Example:

If the B register contains 7BH (01111011) and the Accumulator contains C3H (11000011), after the execution of

AND B

the Accumulator will contain 43H (01000011).

OR s

Operation: A ◊ A ⇦ s

Format:

Mnemonic: OR **Operands:** s

The s operand is any of r, n, (HL), (IX + d), or (IY + d), as defined for the analogous ADD instructions. These various possible opcode-operand combinations are assembled as follows in the object code:

Object Code:

OR r	1 0 1 1 0 r r r	
OR n	1 1 1 1 0 1 1 0	F6
	n n n n n n n n	
OR (HL)	1 0 1 1 0 1 1 0	B6
OR (IX + d)	1 1 0 1 1 1 0 1	DD
	1 0 1 1 0 1 1 0	B6
	d d d d d d d d	
OR (IY + d)	1 1 1 1 1 1 0 1	FD
	1 0 1 1 0 1 1 0	B6
	d d d d d d d d	

r identifies register A, B, C, D, E, H or L assembled as follows in the object code field above:

Register		r
A	=	111
B	=	000
C	=	001
D	=	010
E	=	011
H	=	100
L	=	101

Description:

A logical OR operation, Bit by Bit, is performed between the byte specified by the s operand and the byte contained in the Accumulator; the result is stored in the Accumulator.

Instruction	M Cycles	T States	4 MHz E.T. in μs
OR r	1	4	1.00
OR n	2	7(4,3)	1.75
OR (HL)	2	7(4,3)	1.75
OR (IX + d)	5	19(4,4,3,5,3)	4.75
OR (IY + d)	5	19(4,4,3,5,3)	4.75

Condition Bits Affected:

S: Set if result is negative; reset otherwise
Z: Set if result is zero; reset otherwise
H: Reset
P/V: Set if parity even; reset otherwise
N: Reset
C: Reset

Table of OR Values:

IF A	IF B	Then A (After)
0	0	0
0	1	1
1	0	1
1	1	1

Example:

If the H register contains 48H (01001000) and the Accumulator contains 12H (00010010), after the execution of

OR H

the Accumulator will contain 5AH (01011010).

XOR s

eXclusive OR

Operation: A ◊ A⊕s

Format:

Mnemonic: XOR **Operands:** s

The s operand is any of r, n, (HL), (IX + d) or (IY + d), as defined for the analogous ADD instructions. These various possible opcode-operand combinations are assembled as follows in the object code:

Object Code:

XOR r	1 0 1 0 1 r r r	
XOR n	1 1 1 0 1 1 1 0	EE
	n n n n n n n n	
XOR (HL)	1 0 1 0 1 1 1 0	AE
XOR (IX + d)	1 1 0 1 1 1 0 1	DD
	1 0 1 0 1 1 1 0	AE
	d d d d d d d d	
XOR (IY + d)	1 1 1 1 1 1 0 1	FD
	1 0 1 0 1 1 1 0	AE
	d d d d d d d d	

r identifies registers A, B, C, D, E, H or L assembled as follows in the object code field above:

Register		**r**
A	=	111
B	=	000
C	=	001
D	=	010
E	=	011
H	=	100
L	=	101

Description:

A logical exclusive-OR operation, bit by bit, is performed between the byte specified by the s operand and the byte contained in the Accumulator; the result is stored in the Accumulator.

Instruction	M Cycles	T States	4 MHz E.T. in μs
XOR r	1	4	1.00
XOR n	2	7(4,3)	1.75
XOR (HL)	2	7(4,3)	1.75
XOR (IX + d)	5	19(4,4,3,5,3)	4.75
XOR (IY + d)	5	19(4,4,3,5,3)	4.75

Condition Bits Affected:

S: Set if result is negative; reset otherwise
Z: Set if result is zero; reset otherwise
H: Reset
P/V: Set if parity even; reset otherwise
N: Reset
C: Reset

Table of XOR Values:

IF A	IF B	Then A (After)
0	0	0
0	1	1
1	0	1
1	1	0

Note: in Table above that any two like numbers will result in zero.

Example 1:

If the Accumulator contains 96H (10010110), after the execution of

XOR 5DH (Note: 5DH = 01011101)

the Accumulator will contain CBH (11001011).

Example 2:

The instruction

XOR A

will zero the Accumulator.

CP s

ComPare

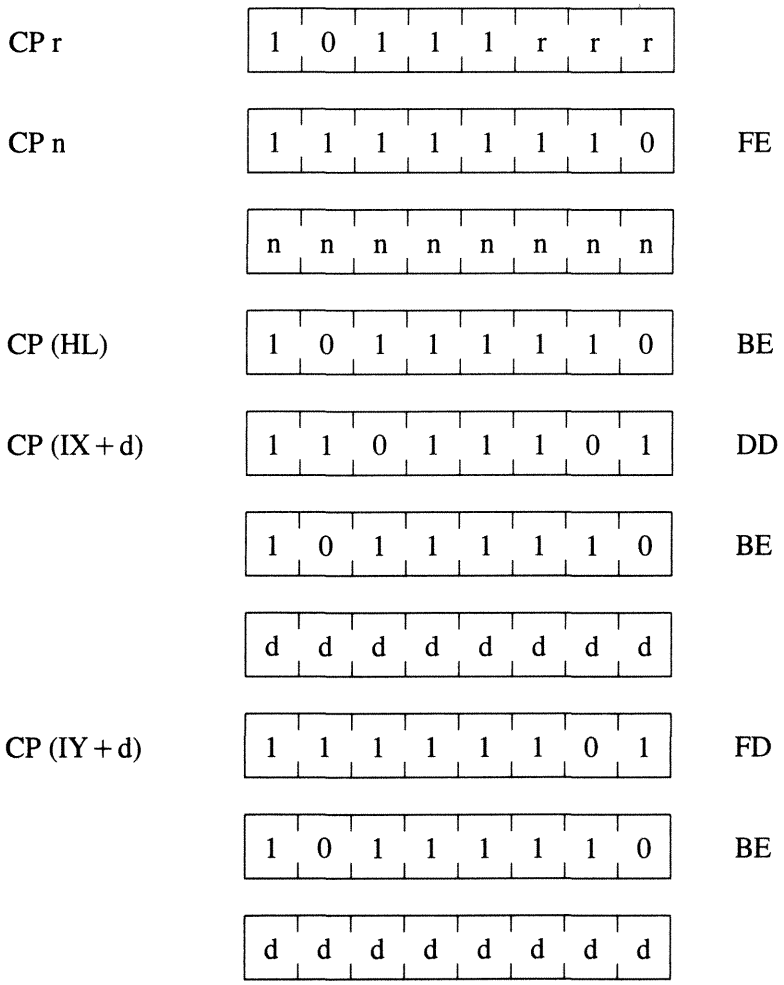
Operation: A − s

Format:

Mnemonic: CP **Operands:** s

The s operand is any of r, n, (HL), (IX + d) or (IY + d), as defined for the analogous ADD instructions. These various possible opcode-operand combinations are assembled as follows in the object code:

Object Code:



r identifies register A, B, C, D, E, H or L assembled as follows in the object code field above:

Register		r
A	=	111
B	=	000
C	=	001
D	=	010
E	=	011
H	=	100
L	=	101

Description:

The contents of the s operand are compared with the contents of the Accumulator. If there is a true compare, a flag is set.

Instruction	M Cycles	T States	4 MHz E.T. in μs
CP r	1	4	1.00
CP n	2	7(4,3)	1.75
CP (HL)	2	7(4,3)	1.75
CP (IX + d)	5	19(4,4,3,5,3)	4.75
CP (IY + d)	5	19(4,4,3,5,3)	4.75

Condition Bits Affected:

S:	Set if result is negative; reset otherwise
Z:	Set if result is zero; reset otherwise
H:	Set if borrow from Bit 4; reset otherwise
P/V:	Set if overflow; reset otherwise
N:	Set
C:	Set if borrow in Bit 7; reset otherwise

Example 1:

If the Accumulator contains 63H, the HL register pair contains 6000H and memory location 6000H contains 60H, the instruction

CP (HL)

will result in all the flags being reset except N.

Example: 2

If the Accumulator contains 65H and register C also contains 65H, then after the execution of

CP C

the Z flag will be set.

See Appendix E for more details of condition codes affected.

INC r

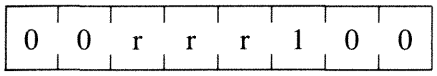
INCrement

Operation: r ◊ r + 1

Format:

Mnemonic: INC **Operands:** r

Object Code:



Description:

Register r is incremented. r identifies any of the registers A, B, C, D, E, H or L, assembled as follows in the object code.

Register		r
A	=	111
B	=	000
C	=	001
D	=	010
E	=	011
H	=	100
L	=	101

M cycles: 1 T states: 4 4 MHz E.T.: 1.00

Condition Bits Affected:

S: Set if result is negative; reset otherwise
Z: Set if result is zero; reset otherwise
H: Set if carry from Bit 3; reset otherwise
P/V: Set if r was 7FH before operation; reset otherwise
N: Reset
C: Not affected

Example:

If the contents of register D are 28H, after the execution of

INC D

the contents of register D will be 29H.

INC (HL)

INCrement

Operation: (HL) ◊ (HL) + 1

Format:

Mnemonic: INC **Operands:** (HL)

Object Code:

0	0	1	1	0	1	0	0

34

Description:

The byte contained in the address specified by the contents of the HL register pair is incremented.

M cycles: 3 T states: 11(4,4,3) 4 MHz E.T.: 2.75

Condition Bits Affected:

S: Set if result is negative; reset otherwise
Z: Set if result is zero; reset otherwise
H: Set if carry from Bit 3; reset otherwise
P/V: Set if (HL) was 7FH before operation; reset otherwise
N: Reset
C: Not Affected

Example:

If the contents of the HL register pair are 3434H, and the contents of address 3434H are 82H, after the execution of

INC (HL)

memory location 3434H will contain 83H.

INC (IX + d)

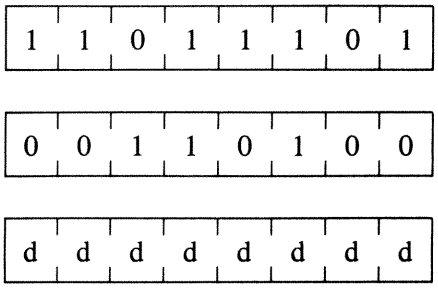
INCrement

Operation: (IX + d) ◊ (IX + d) + 1

Format:

Mnemonic: INC **Operands:** (IX + d)

Object Code:



Description:

The contents of the Index Register IX (register pair IX) are added to a two's complement displacement integer d to point to an address in memory. The contents of this address are then incremented.

M cycles: 6 T states: 23(4,4,3,5,4,3) 4 MHz E.T.: 5.75

Condition Bits Affected:

S: Set if result is negative; reset otherwise
Z: Set if result is zero; reset otherwise
H: Set if carry from Bit 3; reset otherwise
P/V: Set if (IX + d) was 7FH before operation; reset otherwise
N: Reset
C: Not affected

Example:

If the contents of the Index Register pair IX are 2020H, and the memory location 2030H contains byte 34H, after the execution of

INC (IX + 10H)

the contents of memory location 2030H will be 35H.

INC (IY + d)

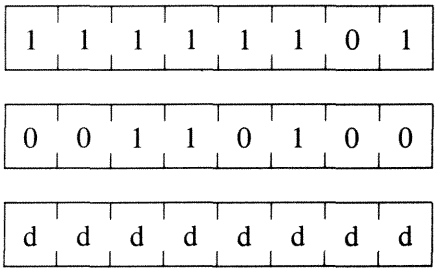
INCrement

Operation: (IY + d) ◊ (IY + d) + 1

Format:

Mnemonic: INC **Operands:** (IY + d)

Object Code:



Description:

The contents of the Index Register IY (register pair IY) are added to a two's complement displacement integer d to point to an address in memory. The contents of this address are then incremented.

M cycles: 6 T states: 23(4,4,3,5,4,3) 4 MHz E.T.: 5.75

Condition Bits Affected:

S: Set if result is negative; reset otherwise
Z: Set if result is zero; reset otherwise
H: Set if carry from Bit 3; reset otherwise
P/V: Set if (IY + d) was 7FH before operation; reset otherwise
N: Reset
C: Not Affected

Example:

If the contents of the Index Register pair IY are 2020H, and the memory location 2030H contain byte 34H, after the execution of

INC (IY + 10H)

the contents of memory location 2030H will be 35H.

DEC m

DECrement

Operation: m ◊ m − 1

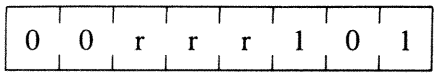
Format:

Mnemonic: DEC **Operands:** m

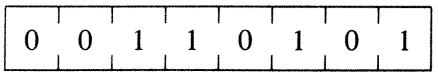
The m operand is any of r, (HL), (IX + d) or (IY + d), as defined for the analogous INC instructions. These various possible opcode-operand combinations are assembled as follows in the object code:

Object Code:

DEC r

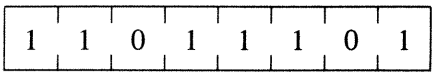


DEC (HL)

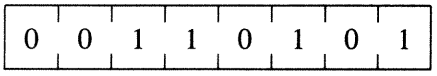


35

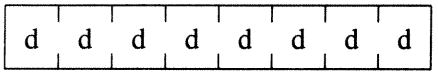
DEC (IX + d)



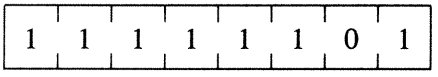
DD



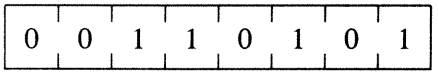
35



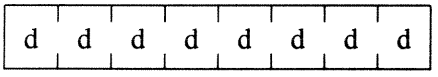
DEC (IY + d)



FD



35



r identifies register A, B, C, D, E, H or L assembled as follows in the object code field above:

Register		**r**
A	=	111
B	=	000
C	=	001
D	=	010
E	=	011
H	=	100
L	=	101

Description:

The byte specified by the m operand is decremented.

Instruction	M Cycles	T States	4 MHz E.T. in μs
DEC r	1	4	1.00
DEC (HL)	3	11(4,4,3)	2.75
DEC (IX + d)	6	23(4,4,3,5,4,3)	5.75
DEC (IY + d)	6	23(4,4,3,5,4,3)	5.75

Condition Bits Affected:

S: Set if result is negative; reset otherwise
Z: Set if result is zero; reset otherwise
H: Set if borrow from Bit 4; reset otherwise
P/V: Set if m was 80H before operation; reset otherwise
N: Set
C: Not affected

Example:

If the D register contains byte 2AH, after the execution of

DEC D

register D will contain 29H.

General Purpose Arithmetic and CPU Control Groups

DAA

Operation: Decimal-Adjust Accumulator

Format:

Mnemonic: DAA **Operands:**

Object Code:

0	0	1	0	0	1	1	1

27

Description:

This instruction modifies the results of addition or subtraction so that the results of binary arithmetic are correct for decimal numbers. The Binary Coded Decimal (BCD) code uses the 8-bit accumulator as follows: the eight bits are broken up into two groups of four bits, which represent a two-digit decimal number from 00 to 99. If numbers like this are added with the binary adder in the Z-80, answers larger than 10 may result in each decimal place. The DAA instruction will "adjust" the answer so that each decimal place has a value of 9 or less, and so that the digits have the correct decimal value, though they were added by a binary circuit. The carry and half-carry flags are used in this conversion, as is a circuit that detects digits that are 10 or bigger.

Operation	C Before DAA	HEX Value in Upper Digit (bits 7-4)	H Before DAA	HEX Value in Lower Digit (bits 3-0)	Number Added to Byte	C After DAA
	0	0-9	0	0-9	00	0
	0	0-8	0	A-F	06	0
	0	0-9	1	0-3	06	0
ADD	0	A-F	0	0-9	60	1
ADC	0	9-F	0	A-F	66	1
INC	0	A-F	1	0-3	66	1
	1	0-2	0	0-9	60	1
	1	0-2	0	A-F	66	1
	1	0-3	1	0-3	66	1
SUB	0	0-9	0	0-9	00	0
SBC	0	0-8	1	6-F	FA	0
DEC	1	7-F	0	0-9	A0	1
NEG	1	6-F	1	6-F	9A	1

M cycles: 1 T states: 4 4 MHz E.T.: 1.00

Condition Bits Affected:

S: Set if most significant bit of Acc. is 1 after operation; reset otherwise
Z: Set if Acc. is zero after operation; reset otherwise
H: See instruction
P/V: Set if Acc. is even parity after operation; reset otherwise
N: Not affected
C: See instruction

Example:

If an addition operation is performed between 15 (BCD) and 27 (BCD), simple decimal arithmetic gives this result:

```
  15
+ 27
----
  42
```

But when the binary representations are added in the Accumulator according to standard binary arithmetic,

```
  0001 0101
+ 0010 0111
-----------
  0011 1100 = 3C
```

the sum is not decimal. The DAA instruction adjusts this result so that the correct BCD representation is obtained:

```
  0011 1100
+ 0000 0110 (adding 06 from table)
-----------
  0100 0010 = 42
```

CPL — ComPLement

Operation: $A \Leftrightarrow \overline{A}$

Format:

Mnemonic: CPL **Operands:**

Object Code:

0	0	1	0	1	1	1	1	2F

Description:

Contents of the Accumulator (register A) are inverted (one's complement).

M cycles: 1 T states: 4 4 MHz E.T.: 1.00

Condition Bits Affected:

S: Not affected
Z: Not affected
H: Set
P/V: Not affected
N: Set
C: Not affected

Example:

If the contents of the Accumulator are 1011 0100, after the execution of

CPL

the Accumulator contents will be 0100 1011.

NEG

NEGate

Operation: A ◊ 0 − A

Format:

Mnemonic: NEG **Operands:**

Object Code:

1	1	1	0	1	1	0	1	ED
0	1	0	0	0	1	0	0	44

Description:

Contents of the Accumulator are negated (two's complement). This is the same as subtracting the contents of the Accumulator from zero. Note that 80H is left unchanged.

M cycles: 2 T states: 8(4,4) 4 MHz E.T.: 2.00

Condition Bits Affected:

S: Set if result is negative; reset otherwise
Z: Set if result is zero; reset otherwise
H: Set if borrow from Bit 4; reset otherwise
P/V: Set if Acc. was 80H before operation; reset otherwise
N: Set
C: Set if Acc. was not 00H before operation; reset otherwise

Example:

If the contents of the Accumulator are

1	0	0	1	1	0	0	0

after the execution of

NEG

the Accumulator contents will be

0	1	1	0	1	0	0	0

CCF

Complement Carry Flag

Operation: $CY \Leftrightarrow \overline{CY}$

Format:

Mnemonic: CCF **Operands:**

Object Code:

0	0	1	1	1	1	1	1

3F

Description:

The C flag in the F register is inverted.

M cycles: 1 T states: 4 4 MHz E.T.: 1.00

Condition Bits Affected:

S: Not affected
Z: Not affected
H: Previous carry will be copied
P/V: Not affected
N: Reset
C: Set if CY was 0 before operation; reset otherwise

SCF

Set Carry Flag

Operation: CY ◊ 1

Format:

Mnemonic: SCF **Operands:**

Object Code:

0	0	1	1	0	1	1	1	37

Description:

The C flag in the F register is set.

M cycles: 1 T states: 4 4 MHz E.T.: 1.00

Condition Bits Affected:

S: Not affected
Z: Not affected
H: Reset
P/V: Not affected
N: Reset
C: Set

NOP

No OPeration

Operation:

Format:

Mnemonic: NOP **Operands:**

Object Code:

0	0	0	0	0	0	0	0	00

Description:

CPU performs no operation during this machine cycle.

M cycles: 1 T states: 4 4 MHz E.T.: 1.00

Condition Bits Affected: None

HALT

Operation:

Format:

Mnemonic: HALT **Operands:**

Object Code:

0	1	1	1	0	1	1	0

76

Description:

The HALT instruction suspends CPU operation until a subsequent interrupt or reset is received. While in the halt state, the processor will execute NOP's to maintain memory refresh logic.

M cycles: 1 T states: 4 4 MHz E.T.: 1.00

Condition Bits Affected: None

DI

Disable Interrupts

Operation: IFF◊0

Format:

Mnemonic: DI **Operands:**

Object Code:

1	1	1	1	0	0	1	1

F3

Description:

DI disables the maskable interrupt by resetting the interrupt enable flip-flops (IFF1 and IFF2). Note that this instruction disables the maskable interrupt during its execution.

M cycles: 1 T states: 4 4 MHz E.T.: 1.00

Condition Bits Affected: None

Example:

When the CPU executes the instruction

DI

the maskable interrupt is disabled until it is subsequently re-enabled by an EI instruction. The CPU will not respond to an Interrupt Request (INT) signal.

EI

Enable Interrupts

Operation: IFF ◊ 1

Format:

Mnemonic: EI **Operands:**

Object Code:

1	1	1	1	1	0	1	1

FB

Description:

EI enables the maskable interrupt by setting the interrupt enable flip-flops (IFF1 and IFF2). Note that this instruction disables the maskable interrupt during its execution.

M cycles: 1 T states: 4 4 MHz E.T.: 1.00

Condition Bits Affected: None

Example:

When the CPU executes instruction

EI

the maskable interrupt is enabled. The CPU will now respond to an Interrupt Request (INT) signal.

IM Ø

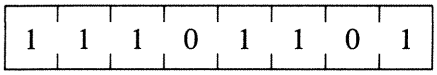
Interrupt Mode Ø

Operation:

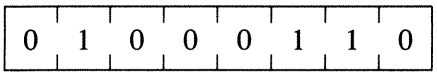
Format:

Mnemonic: IM **Operands:** Ø

Object Code:



ED



46

Description:

The IM Ø instruction sets interrupt mode Ø. In this mode the interrupting device can insert any instruction on the data bus and allow the CPU to execute it. The first byte of a multi-byte instruction is read during interrupt acknowledge cycle. Subsequent bytes are read in by a normal memory read sequence.

M cycles: 2 T states: 8(4,4) 4 MHz E.T.: 2.ØØ

Condition Bits Affected: None

IM 1

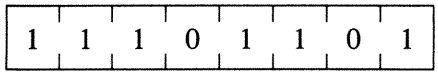
Interrupt Mode 1

Operation:

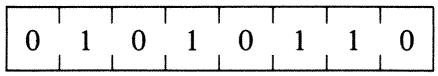
Format:

Mnemonic: IM **Operands:** 1

Object Code:



ED



56

Description:

The IM instruction sets interrupt mode 1. In this mode the processor will respond to an interrupt by executing a restart to location 0038H.

M cycles: 2 T states: 8(4,4) 4 MHz E.T.: 2.00

Condition Bits Affected: None

IM 2

Interrupt Mode 2

Operation:

Format:

Mnemonic: IM **Operands:** 2

Object Code:

1	1	1	0	1	1	0	1	ED
0	1	0	1	1	1	1	0	5E

Description:

The IM 2 instruction sets interrupt mode 2. This mode allows an indirect call to any location in memory. With this mode the CPU forms a 16-bit memory address. The upper eight bits are the contents of the Interrupt Vector Register I and the lower eight bits are supplied by the interrupting device.

M cycles: 2 T states: 8(4,4) 4 MHz E.T.: 2.00

Condition Bits Affected: None

16 Bit Arithmetic Group

ADD HL,ss

Operation: HL ◊ HL + ss

Format:

Mnemonic: ADD **Operands:** HL, ss

Object Code:

0	0	s	s	1	0	0	1

Description:

The contents of register pair ss (any of register pairs BC, DE, HL or SP) are added to the contents of register pair HL, and the result is stored in HL. Operand ss is specified as follows in the assembled object code.

Register Pair	ss
BC	00
DE	01
HL	10
SP	11

M cycles: 3 T states: 11(4,4,3) 4 MHz E.T.: 2.75

Condition Bits Affected:

S: Not affected
Z: Not affected
H: Set if carry out of Bit 11; reset otherwise
P/V: Not affected
N: Reset
C: Set if carry from Bit 15; reset otherwise

Example:

If register pair HL contains the integer 4242H and register pair DE contains 1111H, after the execution of

ADD HL, DE

the HL register pair will contain 5353H.

ADC HL,ss

ADd with Carry

Operation: HL ◊ HL + ss + CY

Format:

Mnemonic: ADC **Operands:** HL, ss

Object Code:

1	1	1	0	1	1	0	1

ED

0	1	s	s	1	0	1	0

Description:

The contents of register pair ss (any of register pairs BC, DE, HL or SP) are added with the Carry Flag (C flag in the F register) to the contents of register pair HL, and the result is stored in HL. Operand ss is specified as follows in the assembled object code.

Register Pair	
BC	00
DE	01
HL	10
SP	11

M cycles: 4 T states: 15(4,4,4,3) 4 MHz E.T.: 3.75

Condition Bits Affected:

S: Set if result is negative; reset otherwise
Z: Set if result is zero; reset otherwise
H: Set if carry out of Bit 11; reset otherwise
P/V: Set if overflow; reset otherwise
N: Reset
C: Set if carry from Bit 15; reset otherwise

Example:

If the register pair BC contains 2222H, register pair HL contains 5437H and the Carry Flag is set, after the execution of

ADC HL, BC

the contents of HL will be 765AH.

SBC HL,ss

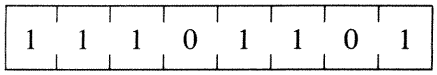
SuBtract with Carry

Operation: HL ◊ HL – ss – CY

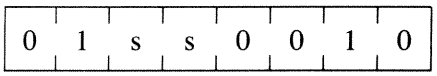
Format:

Mnemonic: SBC **Operands:** HL, ss

Object Code:



ED



Description:

The contents of the register pair ss (any of register pairs BC, DE, HL or SP) and the Carry Flag (C flag in the F register) are subtracted from the contents of register pair HL and the result is stored in HL. Operand ss is specified as follows in the assembled object code.

Register Pair	ss
BC	00
DE	01
HL	10
SP	11

M cycles: 4 T states: 15(4,4,4,3) 4 MHz E.T.: 3.75

Condition Bits Affected:

S: Set if result is negative; reset otherwise
Z: Set if result is zero; reset otherwise
H: Set if borrow from Bit 12; reset otherwise
P/V: Set if overflow; reset otherwise
N: Set
C: Set if borrow; reset otherwise

Example:

If the contents of the HL register pair are 9999H, the contents of register pair DE are 1111H, and the Carry Flag is set, after the execution of

SBC HL, DE

the contents of HL will be 8887H.

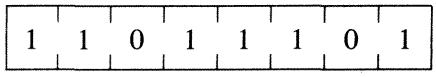
ADD IX,pp

Operation: IX ◊ IX + pp

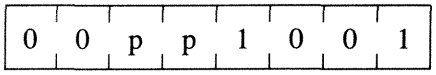
Format:

Mnemonic: ADD **Operands:** IX,pp

Object Code:



DD



Description:

The contents of register pair pp (any of register pairs BC, DE, IX or SP) are added to the contents of the Index Register IX, and the results are stored in IX. Operand pp is specified as follows in the assembled object code.

Register Pair	pp
BC	00
DE	01
IX	10
SP	11

M cycles: 4 T states: 15(4,4,4,3) 4 MHz E.T.: 3.75

Condition Bits Affected:

S: Not affected
Z: Not affected
H: Set if carry out of Bit 11; reset otherwise
P/V: Not affected
N: Reset
C: Set if carry from Bit 15; reset otherwise

Example:

If the contents of Index Register IX are 3333H and the contents of register pair BC are 5555H, after the execution of

ADD IX, BC

the contents of IX will be 8888H.

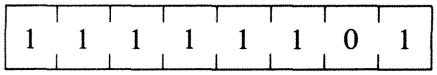
ADD IY,rr

Operation: IY ◊ IY + rr

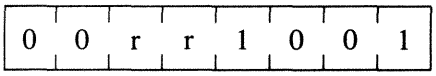
Format:

Mnemonic: ADD **Operands:** IY, rr

Object Code:



FD



Description:

The contents of register pair rr (any of register pairs BC, DE, IY or SP) are added to the contents of Index Register IY, and the result is stored in IY. Operand rr is specified as follows in the assembled object code.

Register Pair	rr
BC	00
DE	01
IY	10
SP	11

M cycles: 4 T states: 15(4,4,4,3) 4 MHz E.T.: 3.75

Condition Bits Affected:

S: Not affected
Z: Not affected
H: Set if carry out of Bit 11; reset otherwise
P/V: Not affected
N: Reset
C: Set if carry from Bit 15; reset otherwise

Example:

If the contents of Index Register IY are 333H and the contents of register pair BC are 555H, after the execution of

ADD IY, BC

the contents of IY will be 888H.

INC ss

INCrement

Operation: ss ◊ ss + 1

Format:

Mnemonic: INC **Operands:** ss

Object Code:

0	0	s	s	0	0	1	1

Description:

The contents of register pair ss (any of register pairs BC, DE, HL or SP) are incremented. Operand ss is specified as follows in the assembled object code.

Register Pair	ss
BC	00
DE	01
HL	10
SP	11

M cycles: 1 T states: 6 4 MHz E.T.: 1.50

Condition Bits Affected: None

Example:

If the register pair contains 1000H, after the execution of

INC HL

HL will contain 1001H.

INC IX

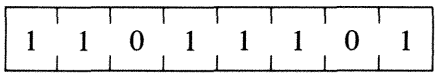
INCrement

Operation: IX ◊ IX + 1

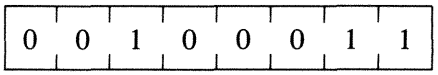
Format:

Mnemonic: INC **Operands:** IX

Object Code:



DD



23

Description:

The contents of the Index Register IX are incremented.

M cycles: 2 T states: 10(4,6) 4 MHz E.T.: 2.50

Condition Bits Affected: None

Example:

If the Index Register IX contains the integer 3300H after the execution of
INC IX
the contents of Index Register IX will be 3301H.

INC IY

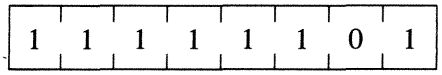
INCrement

Operation: IY ⇦ IY + 1

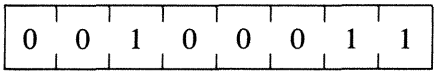
Format:

Mnemonic: INC **Operands:** IY

Object Code:



FD



23

Description:

The contents of the Index Register IY are incremented.

M cycles: 2 T states: 10(4,6) 4 MHz E.T.: 2.50

Condition Bits Affected: None

Example:

If the contents of the Index Register are 2977H, after the execution of

INC IY

the contents of Index Register IY will be 2978H.

DEC ss

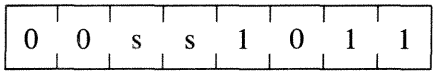
DECrement

Operation: SS ⇦ SS − 1

Format:

Mnemonic: DEC **Operands:** ss

Object Code:



Description:

The contents of register pair ss (any of the register pairs BC, DE, HL or SP) are decremented. Operand ss is specified as follows in the assembled object code.

Register Pair	ss
BC	00
DE	01
HL	10
SP	11

M cycles: 1 T states: 6 4 MHz E.T.: 1.50

Condition Bits Affected: None

Example:

If register pair HL contains 1001H, after the execution of

DEC HL

the contents of HL will be 1000H.

DEC IX

DECrement

Operation: IX ◊ IX − 1

Format:

Mnemonic: DEC **Operands:** IX

Object Code:

1	1	0	1	1	1	0	1	DD
0	0	1	0	1	0	1	1	2B

Description:

The contents of Index Register IX are decremented.

M cycles: 2 T states: 10(4,6) 4 MHz E.T.: 2.50

Condition Bits Affected: None

Example:

If the contents of Index Register IX are 2006H, after the execution of
DEC IX
the contents of Index Register IX will be 2005H.

DEC IY

DECrement

Operation: IY ◊ IY − 1

Format:

Mnemonic: DEC **Operands:** IY

Object Code:

1	1	1	1	1	1	0	1	FD
0	0	1	0	1	0	1	1	2B

Description:

The contents of the Index Register IY are decremented.

M cycles: 2 T states: 10(4,6) 4 MHz E.T.: 2.50

Condition Bits Affected: None

Example:

If the contents of the Index Register IY are 7649H, after the execution of

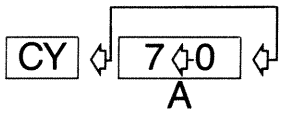
DEC IY

the contents of Index Register IY will be 7648H.

Rotate and Shift Group

RLCA — Rotate Left Circular Accumulator

Operation:



Format:

Mnemonic: RLCA **Operands:**

Object Code:

0	0	0	0	0	1	1	1

07

Description:

The contents of the Accumulator (register A) are rotated left: the content of bit 0 is moved to bit 1; the previous content of bit 1 is moved to bit 2; this pattern is continued throughout the register. The content of bit 7 is copied into the Carry Flag (C flag in register F) and also into bit 0. (Bit 0 is the least significant bit.)

M cycles: 1 T states: 4 4 MHz E.T.: 1.00

Condition Bits Affected:

S: Not affected
Z: Not affected
H: Reset
P/V: Not affected
N: Reset
C: Data from Bit 7 of Acc.

Example:

If the contents of the Accumulator are

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

after the execution of

RLCA

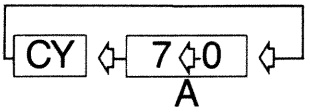
the contents of the Carry Flag and the Accumulator will be

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

RLA

Rotate Left Accumulator

Operation:



Format:

Mnemonic: RLA **Operands:**

Object Code:

0	0	0	1	0	1	1	1

17

Description:

The contents of the Accumulator (register A) are rotated left: the content of bit Ø is copied into bit 1; the previous content of bit 1 is copied into bit 2; this pattern is continued throughout the register. The content of bit 7 is copied into the Carry Flag (C flag in register F) and the previous content of the Carry Flag is copied into bit Ø. Bit Ø is the least significant bit.

M cycles: 1 T states: 4 4 MHz E.T.: 1.ØØ

Condition Bits Affected:

S: Not affected
Z: Not affected
H: Reset
P/V: Not affected
N: Reset
C: Data from Bit 7 of Acc.

Example:

If the contents of the Carry Flag and the Accumulator are

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

after the execution of

RLA

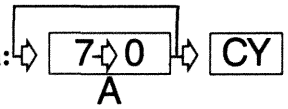
the contents of the Carry Flag and the Accumulator will be

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

RRCA

Rotate Right Circular Accumulator

Operation:



Format:

Mnemonic: RRCA **Operands:**

Object Code:

0	0	0	0	1	1	1	1	0F

Description:

The contents of the Accumulator (register A) are rotated right: the content of bit 7 is copied into bit 6; the previous content of bit 6 is copied into bit 5; this pattern is continued throughout the register. The content of bit 0 is copied into bit 7 and also into the Carry Flag (C flag in register F.) Bit 0 is the least significant bit.

M cycles: 1 T states: 4 4 MHz E.T.: 1.00

Condition Bits Affected:

S: Not affected
Z: Not affected
H: Reset
P/V: Not affected
N: Reset
C: Data from Bit 0 of Acc.

Example:

If the contents of the Accumulator are

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

After the execution of

RRCA

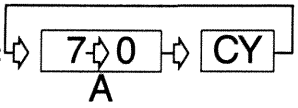
the contents of the Accumulator and the Carry Flag will be

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

RRA

Rotate Right Accumulator

Operation:



Format:

Mnemonic: RRA **Operands:**

Object Code:

0	0	0	1	1	1	1	1	1F

Description:

The contents of the Accumulator (register A) are rotated right: the content of bit 7 is copied into bit 6; the previous content of bit 6 is copied into bit 5; this pattern is continued throughout the register. The content of bit Ø is copied into the Carry Flag (C flag in register F) and the previous content of the Carry Flag is copied into bit 7. Bit Ø is the least significant bit.

M cycles: 1 T states: 4 4 MHz E.T.: 1.ØØ

Condition Bits Affected:

S: Not affected
Z: Not affected
H: Reset
P/V: Not affected
N: Reset
C: Data from Bit Ø of Acc.

Example:

If the contents of the Accumulator and the Carry Flag are

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

after the execution of

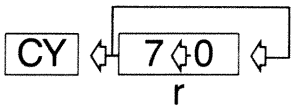
RRA

the contents of the Accumulator and the Carry Flag will be

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

RLC r

Rotate Left Circular

Operation: 

Format:

Mnemonic: RLC **Operands:** r

Object Code:

1	1	0	0	1	0	1	1

CB

0	0	0	0	0	r	r	r

Description:

The eight-bit contents of register r are rotated left: the content of bit 0 is copied into bit 1; the previous content of bit 1 is copied into bit 2; this pattern is continued throughout the register. The content of bit 7 is copied into the Carry Flag (C flag in register F) and also into bit 0. Operand r is specified as follows in the assembled object code:

Register	r
B	000
C	001
D	010
E	011
H	100
L	101
A	111

Note: Bit 0 is the least significant bit.

M cycles: 2 T states: 8(4,4) 4 MHz E.T.: 2.00

Condition Bits Affected:

S: Set if result is negative; reset otherwise
Z: Set if result is zero; reset otherwise
H: Reset
P/V: Set if parity even; reset otherwise
N: Reset
C: Data from Bit 7 of source register

Example:

If the contents of register r are

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

after the execution of

RLC r

the contents of the Carry Flag and register r will be

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

RLC (HL)

Rotate Left Circular

Operation: CY ⇦ 7 ⇦ 0 ⇦ (HL)

Format:

Mnemonic: RLC **Operands:** (HL)

Object Code:

1	1	0	0	1	0	1	1	CB
0	0	0	0	0	1	1	0	06

Description:

The contents of the memory address specified by the contents of register pair HL are rotated left: the content of bit 0 is copied into bit 1; the previous content of bit 1 is copied into bit 2; this pattern is continued throughout the byte. The content of bit 7 is copied into the Carry Flag (C flag in register F) and also into bit 0. Bit 0 is the least significant bit.

M cycles: 4 T states: 15(4,4,4,3) 4 MHz E.T.: 3.75

Condition Bits Affected:

S: Set if result is negative; reset otherwise
Z: Set if result is zero; reset otherwise
H: Reset
P/V: Set if parity even; reset otherwise
N: Reset
C: Data from Bit 7 of source register

Example:

If the contents of the HL register pair are 2828H, and the contents of memory location 2828H are

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

after the execution of

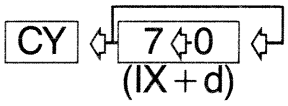
RLC (HL)

the contents of the Carry Flag and memory locations 2828H will be

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

RLC (IX + d)

Rotate Left Circular

Operation: 

Format:

Mnemonic: RLC **Operands:** (IX + d)

Object Code:

1	1	0	1	1	1	0	1	DD
1	1	0	0	1	0	1	1	CB
d	d	d	d	d	d	d	d	
0	0	0	0	0	1	1	0	06

Description:

The contents of the memory address specified by the sum of the contents of the Index Register IX and a two's complement displacement integer d, are rotated left: the contents of bit Ø is copied into bit 1; the previous content of bit 1 is copied into bit 2; this pattern is continued throughout the byte. The content of bit 7 is copied into the Carry Flag (C flag in register F) and also into bit Ø. Bit Ø is the least significant bit.

M cycles: 6 T states: 23(4,4,3,5,4,3) 4 MHz E.T.: 5.75

Condition Bits Affected:

S: Set if result is negative; reset otherwise
Z: Set if result is zero; reset otherwise
H: Reset
P/V: Set if parity even; reset otherwise
N: Reset
C: Data from Bit 7 of source register

Example:

If the contents of the Index Register IX are 1ØØØH, and the contents of memory location 1ØØ2H are

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

after the execution of

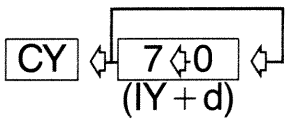
RLC (IX + 2H)

the contents of the Carry Flag and memory location 1ØØ2H will be

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

RLC (IY + d)

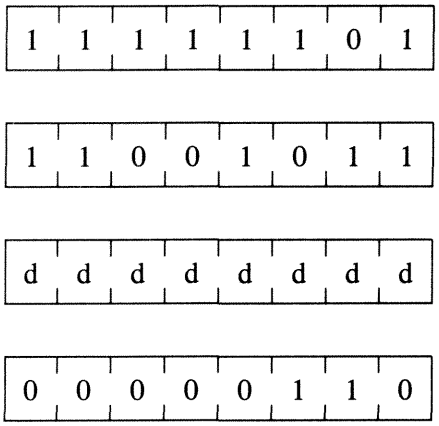
Rotate Left Circular

Operation: 

Format:

Mnemonic: RLC **Operands:** (IY + d)

Object Code:



Description:

The contents of the memory address specified by the sum of the contents of the Index Register IY and a two's complement displacement integer d are rotated left: the content of bit 0 is copied into bit 1; the previous content of bit 1 is copied into bit 2; this process is continued throughout the byte. The content of bit 7 is copied into the Carry Flag (C flag in register F) and also into bit 0. Bit 0 is the least significant bit.

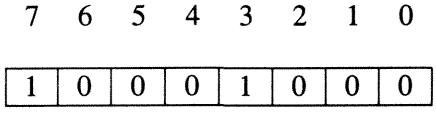
M cycles: 6 T states: 23(4,4,3,5,4,3) 4 MHz E.T.: 5.75

Condition Bits Affected:

S: Set if result is negative; reset otherwise
Z: Set if result is zero; reset otherwise
H: Reset
P/V: Set if parity even; reset otherwise
N: Reset
C: Data from Bit 7 of source register

Example:

If the contents of the Index Register IY are 1000H, and the contents of memory location 1002H are



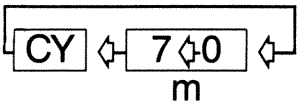
after the execution of
RLC (IY + 2H)
the contents of the Carry Flag and memory location 1002H will be

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

RL m

Rotate Left

Operation:



Format:

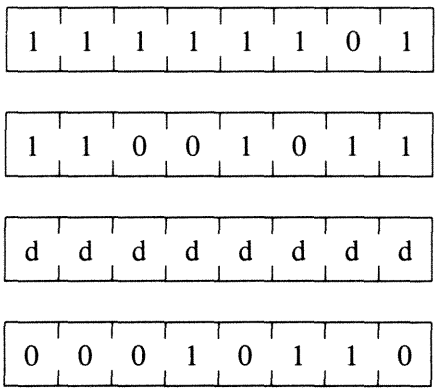
Mnemonic: RL **Operands:** m

The m operand is any of r, (HL), (IX + d) or (IY + d), as defined for the analogous RLC instructions. These various possible opcode-operand combinations are specified as follows in the assembled object code:

Object Code:

RL r	1 1 0 0 1 0 1 1	CB
	0 0 0 1 0 r r r	
RL (HL)	1 1 0 0 1 0 1 1	CB
	0 0 0 1 0 1 1 0	16
RL (IX + d)	1 1 0 1 1 1 0 1	DD
	1 1 0 0 1 0 1 1	CB
	d d d d d d d d	
	0 0 0 1 0 1 1 0	16

RL (IY + d)



r identifies register B, C, D, E, H, L or A specified as follows in the assembled object code above:

Register	r
B	000
C	001
D	010
E	011
H	100
L	101
A	111

Description:

The contents of the m operand are rotated left: the content of bit 0 is copied into bit 1; the previous content of bit 1 is copied into bit 2; this pattern is continued throughout the byte. The content of bit 7 is copied into the Carry Flag (C flag in register F) and the previous content of the Carry Flag is copied into bit 0. Bit 0 is the least significant bit.

Instruction	M Cycles	T States	4 MHz E.T. in μs
RL r	2	8(4,4)	2.00
RL (HL)	4	15(4,4,4,3)	3.75
RL (IX + d)	6	23(4,4,3,5,4,3)	5.75
RL (IY + d)	6	23(4,4,3,5,4,3)	5.75

Condition Bits Affected:

S: Set if result is negative; reset otherwise
Z: Set if result is zero; reset otherwise
H: Reset
P/V: Set if parity even; reset otherwise
N: Reset
C: Data from Bit 7 of source register

Example:

If the contents of the Carry Flag and register D are

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

after the execution of

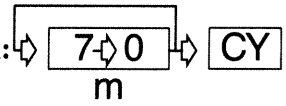
RL D

the contents of the Carry Flag and register D will be

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

RRC m

Rotate Right Circular

Operation: 

Format:

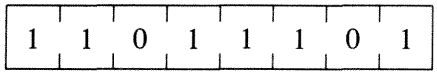
Mnemonic: RRC **Operands:** m

The m operand is any of r, (HL), (IX + d) or (IY + d), as defined for the analogous RLC instructions. These various possible opcode-operand combinations are specified as follows in the assembled object code:

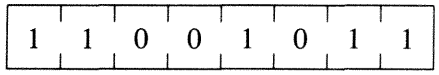
Object Code:

RRC r	1 1 0 0 1 0 1 1	CB
	0 0 0 0 1 r r r	
RRC (HL)	1 1 0 0 1 0 1 1	CB
	0 0 0 0 1 1 1 0	0E

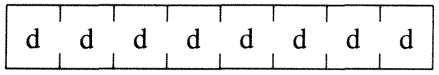
RRC (IX + d)

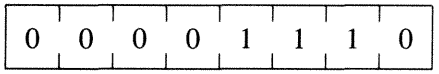


DD



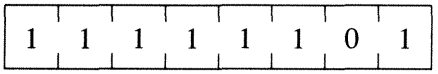
CB



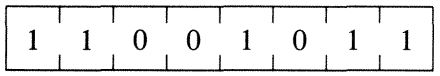


ØE

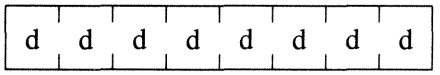
RRC (IY + d)

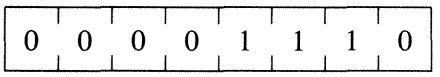


FD



CB





ØE

r identifies register B, C, D, E, H, L or A specified as follows in the assembled object code above:

Register	r
B	ØØØ
C	ØØ1
D	Ø1Ø
E	Ø11
H	1ØØ
L	1Ø1
A	111

Description:

The contents of operand m are rotated right: the content of bit 7 is copied into bit 6; the previous content of bit 6 is copied into bit 5; this pattern is continued throughout the byte. The content of bit Ø is copied into the Carry Flag (C flag in the F register) and also into bit 7. Bit Ø is the least significant bit.

Instruction	M Cycles	T States	4 MHz E.T. in μs
RRC r	2	8(4,4)	2.00
RRC (HL)	4	15(4,4,4,3)	3.75
RRC (IX + d)	6	23(4,4,3,5,4,3)	5.75
RRC (IY + d)	6	23(4,4,3,5,4,3)	5.75

Condition Bits Affected:

S: Set if result is negative; reset otherwise
Z: Set if result is zero; reset otherwise
H: Reset
P/V: Set if parity even; reset otherwise
N: Reset
C: Data from Bit 0 of source register

Example:

If the contents of register A are

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

after the execution of

RRC A

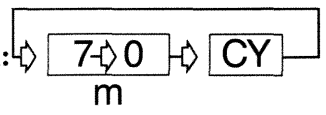
the contents of register A and the Carry Flag will be

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

RR m

Rotate Right

Operation:



Format:

Mnemonic: RR **Operands:** m

The m operand is any of r, (HL), (IX + d) or (IY + d), as defined for the analogous RLC instructions. These various possible opcode-operand combinations are specified as follows in the assembled object code:

Object Code:

RR r	1 1 0 0 1 0 1 1	CB
	0 0 0 1 1 r r r	
RR (HL)	1 1 0 0 1 0 1 1	CB
	0 0 0 1 1 1 1 0	1E
RR (IX + d)	1 1 0 1 1 1 0 1	DD
	1 1 0 0 1 0 1 1	CB
	d d d d d d d d	
	0 0 0 1 1 1 1 0	1E
RR (IY + d)	1 1 1 1 1 1 0 1	FD
	1 1 0 0 1 0 1 1	CB
	d d d d d d d d	
	0 0 0 1 1 1 1 0	1E

r identifies registers B, C, D, E, H, L or A specified as follows in the assembled object code above:

Register	r
B	000
C	001
D	010
E	011
H	100
L	101
A	111

Description:

The contents of operand m are rotated right: the contents of bit 7 is copied into bit 6; the previous content of bit 6 is copied into bit 5; this pattern is continued throughout the byte. The content of bit 0 is copied into the Carry Flag (C flag in register F) and the previous content of the Carry Flag is copied into bit 7. Bit 0 is the least significant bit.

Instruction	M Cycles	T States	4 MHz E.T. in μs
RR r	2	8(4,4)	2.00
RR (HL)	4	15(4,4,4,3)	3.75
RR (IX + d)	6	23(4,4,3,5,4,3)	5.75
RR (IY + d)	6	23(4,4,3,5,4,3)	5.75

Condition Bits Affected:

S: Set if result is negative; reset otherwise
Z: Set if result is zero; reset otherwise
H: Reset
P/V: Set if parity is even; reset otherwise
N: Reset
C: Data from Bit 0 of source register

Example:

If the contents of the HL register pair are 4343H, and the contents of memory location 4343H and the Carry Flag are

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

after the execution of

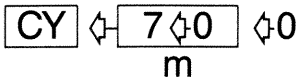
RR (HL)

the contents of location 4343H and the Carry Flag will be

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

SLA m

Shift Left Arithmetic

Operation: 

Format:

Mnemonic: SLA **Operands:** m

The m operand is any of r, (HL), (IX + d) or (IY + d), as defined for the analogous RLC instructions. These various possible opcode-operand combinations are specified as follows in the assembled object code:

Object Code:

SLA r	1 1 0 0 1 0 1 1	CB
	0 0 1 0 0 r r r	
SLA (HL)	1 1 0 0 1 0 1 1	CB
	0 0 1 0 0 1 1 0	26
SLA (IX + d)	1 1 0 1 1 1 0 1	DD
	1 1 0 0 1 0 1 1	CB
	d d d d d d d d	
	0 0 1 0 0 1 1 0	26
SLA (IY + d)	1 1 1 1 1 1 0 1	FD
	1 1 0 0 1 0 1 1	CB
	d d d d d d d d	
	0 0 1 0 0 1 1 0	26

r identifies registers B, C, D, E, H, L or A specified as follows in the assembled object code field above:

Register	r
B	000
C	001
D	010
E	011
H	100
L	101
A	111

Description:

An arithmetic shift left is performed on the contents of operand m: bit 0 is reset, the previous content of bit 0 is copied into bit 1, the previous content of bit 1 is copied into bit 2; this pattern is continued throughout; the content of bit 7 is copied into the Carry Flag (C flag in register F). Bit 0 is the least significant bit.

Instruction	M Cycles	T States	4 MHz E.T. in μs
SLA r	2	8(4,4)	2.00
SLA (HL)	4	15(4,4,4,3)	3.75
SLA (IX + d)	6	23(4,4,3,5,4,3)	5.75
SLA (IY + d)	6	23(4,4,3,5,4,3)	5.75

Condition Bits Affected:

S: Set if result is negative; reset otherwise
Z: Set if result is zero; reset otherwise
H: Reset
P/V: Set if parity is even; reset otherwise
N: Reset
C: Data from Bit 7

Example:

If the contents of register L are

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

after the execution of

SLA L

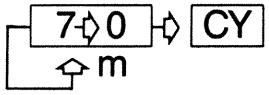
the contents of the Carry Flag and register L will be

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

SRA m

Shift Right Arithmetic

Operation:



Format:

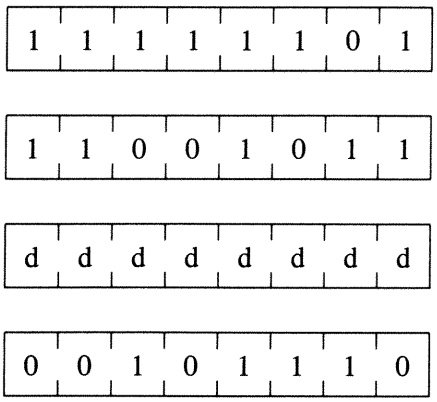
Mnemonic: SRA **Operands:** m

The m operand is any of r, (HL), (IX + d) or (IY + d), as defined for the analogous RLC instructions. These various possible opcode-operand combinations are specified as follows in the assembled object code:

Object Code:

SRA r	1 1 0 0 1 0 1 1	CB
	0 0 1 0 1 r r r	
SRA (HL)	1 1 0 0 1 0 1 1	CB
	0 0 1 0 1 1 1 0	2E
SRA (IX + d)	1 1 0 1 1 1 0 1	DD
	1 1 0 0 1 0 1 1	CB
	d d d d d d d d	
	0 0 1 0 1 1 1 0	2E

SRA (IY + d)



r means register B, C, D, E, H, L or A specified as follows in the assembled object code field above:

Register	r
B	000
C	001
D	010
E	011
H	100
L	101
A	111

An arithmetic shift right is performed on the contents of operand m: the content of bit 7 is copied into bit 6; the previous content of bit 6 is copied into bit 5; this pattern is continued throughout the byte. The content of bit 0 is copied into the Carry Flag (C flag in register F), and the previous content of bit 7 is unchanged. Bit 0 is the least significant bit.

Instruction	M Cycles	T States	4 MHz E.T. in μs
SRA r	2	8(4,4)	2.00
SRA (HL)	4	15(4,4,4,3)	3.75
SRA (IX + d)	6	23(4,4,3,5,4,3)	5.75
SRA (IY + d)	6	23(4,4,3,5,4,3)	5.75

Condition Bits Affected:

S: Set if result is negative; reset otherwise
Z: Set if result is zero; reset otherwise
H: Reset
P/V: Set if parity is even; reset otherwise
N: Reset
C: Data from Bit 0 of source register

Example:

If the contents of the Index Register IX are 1000H, and the contents of memory location 1003H are

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

after the execution of

SRA (IX + 3H)

the contents of memory location 1003H and the Carry Flag will be

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

SRL m

Shift Right Logical

Operation: 0 → 7 → 0 → CY (m)

Format:

Mnemonic: SRL **Operands:** m

The operand m is any of r, (HL), (IX + d) or (IY + d), as defined for the analogous RLC instructions. These various possible opcode-operand combinations are specified as follows in the assembled object code:

Object Code:

SRL r	1 1 0 0 1 0 1 1	CB
	0 0 1 1 1 r r r	
SRL (HL)	1 1 0 0 1 0 1 1	CB
	0 0 1 1 1 1 1 0	3E

SRL (IX + d)	1 1 0 1 1 1 0 1	DD
	1 1 0 0 1 0 1 1	CB
	d d d d d d d d	
	0 0 1 1 1 1 1 0	3E
SRL (IY + d)	1 1 1 1 1 1 0 1	FD
	1 1 0 0 1 0 1 1	CB
	d d d d d d d d	
	0 0 1 1 1 1 1 0	3E

r identifies registers B, C, D, E, H, L or A specified as follows in the assembled object code fields above:

Register	r
B	000
C	001
D	010
E	011
H	100
L	101
A	111

Description:

The contents of operand m are shifted right: the content of bit 7 is copied into bit 6; the content of bit 6 is copied into bit 5; this pattern is continued throughout the byte. The content of bit 0 is copied into the Carry Flag, and bit 7 is reset. Bit 0 is the least significant bit.

Instruction	M Cycles	T States	4 MHz E.T. in μs
SRL r	2	8(4,4,)	2.00
SRL (HL)	4	15(4,4,4,3)	3.75
SRL (IX + d)	6	23(4,4,3,5,4,3)	5.75
SRL (IY + d)	6	23(4,4,3,5,4,3)	5.75

Condition Bits Affected:

S: Set if result is negative; reset otherwise
Z: Set if result is zero; reset otherwise
H: Reset
P/V: Set if parity is even; reset otherwise
N: Reset
C: Data from Bit 0 of source register

Example:

If the contents of register B are

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

after the execution of

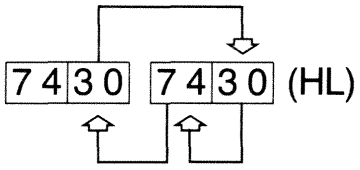
SRL B

the contents of register B and the Carry Flag will be

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

RLD

Rotate Left Decimal

Operation: 

Format:

Mnemonic: RLD **Operands:**

Object Code:

1	1	1	0	1	1	0	1	ED

0	1	1	0	1	1	1	1	6F

Description:

The contents of the low order four bits (bits 3, 2, 1 and Ø) of the memory location (HL) are copied into the high order four bits (7, 6, 5 and 4) of that same memory location; the previous contents of those high order four bits are copied into the low order four bits of the Accumulator (register A), and the previous contents of the low order four bits of the Accumulator are copied into the low order four bits of memory location (HL). The contents of the high order bits of the Accumulator are unaffected. Note: (HL) means the memory location specified by the contents of the HL register pair.

M cycles: 5 T states: 18(4,4,3,4,3) 4 MHz E.T.: 4.5Ø

Condition Bits Affected:

S: Set if Acc. is negative after operation; reset otherwise
Z: Set if Acc. is zero after operation; reset otherwise
H: Reset
P/V: Set if parity of Acc. is even after operation; reset otherwise
N: Reset
C: Not affected

Example:

If the contents of the HL register pair are 5ØØØH, and the contents of the Accumulator and memory location 5ØØØH are

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

7	6	5	4	3	2	1	0	
0	0	1	1	0	0	0	1	(5ØØØH)

after the execution of

RLD

the contents of the Accumulator and memory location 5000H will be

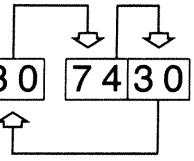
7	6	5	4	3	2	1	0	
0	1	1	1	0	0	1	1	Accumulator

7	6	5	4	3	2	1	0	
0	0	0	1	1	0	1	0	(5000H)

RRD

Rotate Right Decimal

Operation: A 7 4 3 0 7 4 3 0 (HL)



Format:

Mnemonic: RRD **Operands:**

Object Code:

1	1	1	0	1	1	0	1	ED

0	1	1	0	0	1	1	1	67

Description:

The contents of the low order four bits (bits 3, 2, 1 and 0) of memory location (HL) are copied into the low order four bits of the Accumulator (register A); the previous contents of the low order four bits of the Accumulator are copied into the high order four bits (7, 6, 5 and 4) of location (HL); and the previous contents of the high order four bits of (HL) are copied into the low order four bits of (HL). The contents of the high order bits of the Accumulator are unaffected. Note: (HL) means the memory location specified by the contents of the HL register pair.

M cycles: 5 T states: 18(4,4,3,4,3) 4 MHz E.T.: 4.50

Condition Bits Affected:

S: Set if Acc. is negative after operation; reset otherwise
Z: Set if Acc. is zero after operation; reset otherwise
H: Reset
P/V: Set if parity of Acc. is even after operation; reset otherwise
N: Reset
C: Not affected

Example:

If the contents of the HL register pair are 5000H, and the contents of the Accumulator and memory location 5000H are

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

7	6	5	4	3	2	1	0	
0	0	1	0	0	0	0	0	(5000H)

after the execution of
RRD
the contents of the Accumulator and memory location 5000H will be

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

7	6	5	4	3	2	1	0	
0	1	0	0	0	0	1	0	(5000H)

Bit Set, Reset and Test Group

BIT b, r

BIT test

Operation: $Z \Leftarrow \bar{r}_b$

Format:

Mnemonic: BIT **Operands:** b, r

Object Code:

1	1	0	0	1	0	1	1

CB

0	1	b	b	b	r	r	r

Description:

After the execution of this instruction, the Z flag in the F register will contain the complement of the indicated bit within the indicated register. Operands b and r are specified as follows in the assembled object code:

Bit Tested	b	Register	r
0	000	B	000
1	001	C	001
2	010	D	010
3	011	E	011
4	100	H	100
5	101	L	101
6	110	A	111
7	111		

M cycles: 2 T states: 8(4,4) 4 MHz E.T.: 2.00

Condition Bits Affected:

S: Unknown
Z: Set if specified Bit is 0; reset otherwise
H: Set
P/V: Unknown
N: Reset
C: Not affected

Example:

If bit 2 in register B contains Ø, after the execution of

BIT 2, B

the Z flag in the F register will contain 1, and bit 2 in register B will remain Ø. (Bit Ø in register B is the least significant bit.)

BIT b,(HL)

Bit Test

Operation: $Z \Leftarrow \overline{(HL)_b}$

Format:

Mnemonic: BIT **Operands:** b, (HL)

Object Code:

1	1	0	0	1	0	1	1

CB

0	1	b	b	b	1	1	0

Description:

This instruction tests bit b in the memory location specified by the contents of the HL register pair and sets the Z flag accordingly. Operand b is specified as follows in the assembled object code:

Bit Tested	b
Ø	ØØØ
1	ØØ1
2	Ø1Ø
3	Ø11
4	1ØØ
5	1Ø1
6	11Ø
7	111

M cycles: 3 T states: 12(4,4,4) 4 MHz E.T.: 3.ØØ

Condition Bits Affected:

S: Unknown
Z: Set if specified Bit is 0; reset otherwise
H: Set
P/V: Unknown
H: Reset
C: Not affected

Example:

If the HL register pair contains 444H, and bit 4 in the memory location 444H contains 1, after the execution of

BIT 4,(HL)

the Z flag in the F register will contain 0, and bit 4 in memory location 444H will still contain 1. (Bit 0 in memory location 444H is the least significant bit.)

BIT b,(IX + d)

Bit Test

Operation: $Z \Leftarrow \overline{(IX+d)_b}$

Format:

Mnemonic: BIT **Operands:** b, (IX + d)

Object Code:

1	1	0	1	1	1	0	1	DD
1	1	0	0	1	0	1	1	CB
d	d	d	d	d	d	d	d	
0	1	b	b	b	1	1	0	

Description:

After the execution of this instruction, the Z flag in the F register will contain the complement of the indicated bit within the contents of the memory location pointed to by the sum of the contents register pair IX (Index Register IX) and the two's complement displacement integer d. Operand b is specified as follows in the assembled object code.

Bit Tested	b
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

M cycles: 5 T states: 20(4,4,3,5,4) 4 MHz E.T.: 5.00

Condition Bits Affected:

S: Unknown
Z: Set if specified Bit is 0; reset otherwise
H: Set
P/V: Unknown
N: Reset
C: Not affected

Example:

If the contents of Index Register IX are 2000H, and bit 6 in memory location 2004H contains 1, after the execution of

BIT 6,(IX + 4H)

the Z flag in the F register will contain 0, and bit 6 in memory location 2004H will still contain 1. (Bit 0 in memory location 2004H is the least significant bit.)

BIT b,(IY + d)

BIT Test

Operation: $Z \Leftarrow \overline{(IY+d)_b}$

Format:

Mnemonic: BIT **Operands:** b, (IY + d)

Object Code:

1	1	1	1	1	1	0	1	FD
1	1	0	0	1	0	1	1	CB
d	d	d	d	d	d	d	d	
0	1	b	b	b	1	1	0	

Description:

After the execution of this instruction, the Z flag in the F register will contain the complement of the indicated bit within the contents of the memory location pointed to by the sum of the contents of register pair IY (Index Register IY) and the two's complement displacement integer d. Operand b is specified as follows in the assembled object code:

Bit Tested	b
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

M cycles: 5 T states: 20(4,4,3,5,4) 4 MHz E.T.: 5.00

Condition Bits Affected:

S: Unknown
Z: Set if specified Bit is 0; reset otherwise
H: Set
P/V: Unknown
N: Reset
C: Not affected

Example:

If the contents of Index Register are 2000H, and bit 6 in memory location 2004H contains 1, after the execution of

BIT 6,(IY + 4H)

the Z flag in the F register still contain 0, and bit 6 in memory location 2004H will still contain 1. (Bit 0 in memory location 2004H is the least significant bit.)

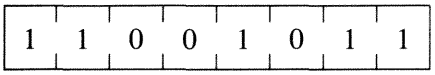
SET b,r

Operation: $r_b \Leftarrow 1$

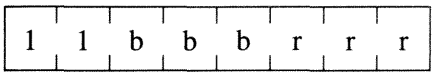
Format:

Mnemonic: SET **Operands:** b, r

Object Code:



CB



Description:

Bit b (any bit, 7 through 0) in register r (any of register B, C, D, E, H, L or A) is set. Operands b and r are specified as follows in the assembled object code:

Bit Tested	b	Register	r
0	000	B	000
1	001	C	001
2	010	D	010
3	011	E	011
4	100	H	100
5	101	L	101
6	110	A	111
7	111		

M cycles: 2 T states: 8(4,4) 4 MHz E.T.: 2.00

Condition Bits Affected: None

Example:

After the execution of

SET 4,A

bit 4 in register A will be set. (Bit 0 is the least significant bit.)

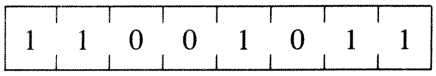
SET b,(HL)

Operation: $(HL)_b \Leftarrow 1$

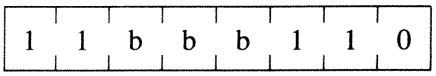
Format:

Mnemonic: SET **Operands:** b, (HL)

Object Code:



CB



Description:

Bit b (any bit, 7 through 0) in the memory location addressed by the contents of register pair HL is set. Operand b is specified as follows in the assembled object code:

Bit Tested	b
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

M cycles: 4 T states: 15(4,4,4,3) 4 MHz E.T.: 3.75

Condition Bits Affected: None

Example:

If the contents of the HL register pair are 3000H, after the execution of

SET 4,(HL)

bit 4 in memory location 3000H will be 1. (Bit 0 in memory location 3000H is the least significant bit.)

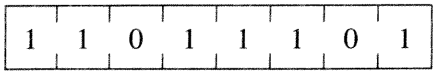
SET b,(IX + d)

Operation: $(IX + d)_b \Leftarrow 1$

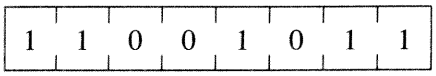
Format:

Mnemonic: SET **Operands:** b, (IX + d)

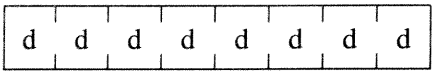
Object Code:

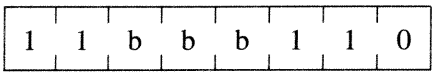


DD



CB





Description:

Bit b (any bit, 7 through 0) in the memory location addressed by the sum of the contents of the IX register pair (Index Register IX) and the two's complement integer d is set. Operand b is specified as follows in the assembled object code:

Bit Tested	b
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

M cycles: 6 T states: 23(4,4,3,5,4,3) 4 MHz E.T.: 5.75

Condition Bits Affected: None

Example:

If the contents of Index Register are 2000H, after the execution of

SET 0,(IX + 3H)

bit 0 in memory location 2003H will be 1. (Bit 0 in memory location 2003H is the least significant bit.)

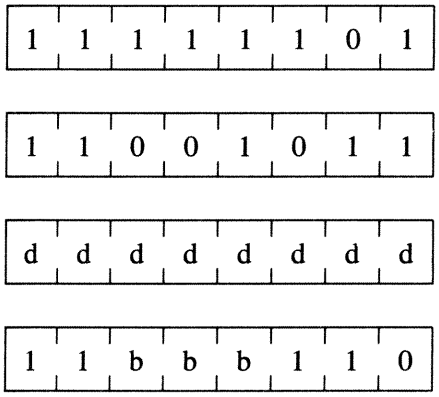
SET b,(IY+d)

Operation: $(IY+d)_b \Leftarrow 1$

Format:

Mnemonic: SET **Operands:** b, (IY + d)

Object Code:



Description:

Bit b (any bit, 7 through 0) in the memory location addressed by the sum of the contents of the IY register pair (Index Register IY) and the two's complement displacement d is set. Operand b is specified as follows in the assembled object code:

Bit Tested	b
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

M cycles: 6 T states: 23(4,4,3,5,4,3) 4 MHz E.T.: 5.75

Condition Bits Affected: None

Example:

If the contents of Index Register IY are 2000H, after the execution of

SET 0,(IY + 3H)

bit 0 in memory location 2003H will be 1. (Bit 0 in memory location 2003H is the least significant bit.)

RES b,m

RESet

Operation: $S_b \Leftarrow 0$

Format:

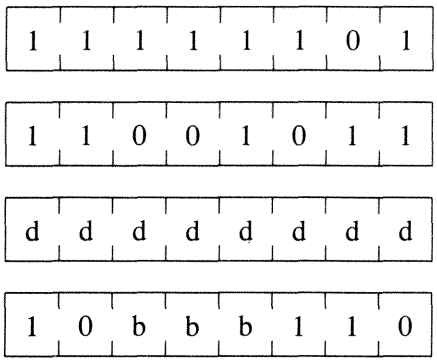
Mnemonic: RES **Operands:** b, m

Operand b is any bit (7 through 0) of the contents of the m operand, (any of r, (HL), (IX + d) or (IY + d) as defined for the analogous SET instructions. These various possible opcode-operand combinations are assembled as follows in the object code:

Object Code:

RES b, r	1 1 0 0 1 0 1 1	CB
	1 0 b b b r r r	
RES b, (HL)	1 1 0 0 1 0 1 1	CB
	1 0 b b b 1 1 0	
RES b, (IX + d)	1 1 0 1 1 1 0 1	DD
	1 1 0 0 1 0 1 1	CB
	d d d d d d d d	
	1 0 b b b 1 1 0	

RES b, (IY + d)



Bit Reset	b	Register	r
0	000	B	000
1	001	C	001
2	010	D	010
3	011	E	011
4	100	H	100
5	101	L	101
6	110	A	111
7	111		

Description:

Bit b in operand m is reset.

Instruction	M Cycles	T States	4 MHz E.T. in μs
RES r	4	8(4,4)	2.00
RES (HL)	4	15(4,4,4,3)	3.75
RES (IX + d)	6	23(4,4,3,5,4,3)	5.75
RES (IY + d)	6	23(4,4,3,5,4,3)	5.75

Condition Bits Affected: None

Example 1:

After the execution of

RES 6,D (object code CB, B2H)

bit 6 in register D will be reset. (Bit 0 in register D is the least significant bit.)

Example 2:

If HL contains 7000H and address 7000H contains FFH, after

RES 0,(HL)

address 7000H will contain FEH.

Jump Group

JP nn

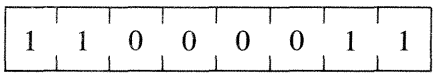
JumP

Operation: PC ◊ nn

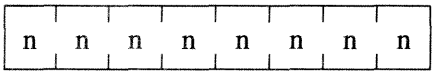
Format:

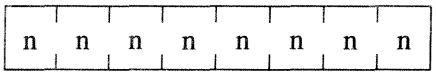
Mnemonic: JP **Operands:** nn

Object Code:



C3





Note: The first operand in this assembled object code is the low order byte of a 2-byte address.

Description:

Operand nn is loaded into register pair PC (Program Counter) and points to the address of the next program instruction to be executed.

M cycles: 3 T states: 10(4,3,3) 4 MHz E.T.: 2.50

Condition Bits Affected: None

Example:

JP 50A1H

This instruction will cause the program to jump to the instruction at 50A1H by loading the number 50A1H into the PC register.

JP cc,nn

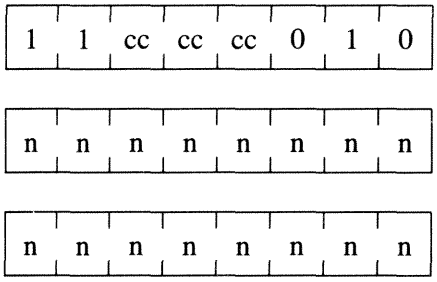
JumP

Operation: IF cc TRUE, PC ◊ nn

Format:

Mnemonic: JP **Operands:** cc, nn

Object Code:



Note: The first n operand in this assembled object code is the low order byte of a 2-byte memory address.

Description:

If condition cc is true, the instruction loads operand nn into register pair PC (Program Counter), and the program continues with the instruction beginning at address nn. If condition cc is false, the Program Counter is incremented as usual, and the program continues with the next sequential instruction. Condition cc is programmed as one of eight status bits which correspond to condition bits in the Flag Register (register F). These eight status bits are defined in the table below, which also specifies the corresponding cc bit fields in the assembled object code.

cc	Condition	Relevant Flag
000	NZ non zero	Z (=0)
001	Z zero	Z (=1)
010	NC non carry	C (=0)
011	C carry	C (=1)
100	PO parity odd	P/V (=0)
101	PE parity even	P/V (=1)
110	P sign positive	S (=0)
111	M sign negative	S (=1)

M cycles: 3 T states: 10(4,3,3) 4 MHz E.T.: 2.50

Condition Bits Affected: None

Example:

If the Carry Flag (C flag in the F register) is set and the contents of address 1520 are 03H, after the execution of

JP C,1520H

the Program Counter will contain 1520H, and on the next machine cycle the CPU will fetch from address 1520H the byte 03H.

JR e

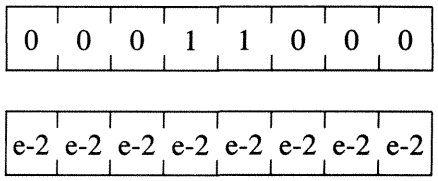
Jump Relative

Operation: PC ◊ PC + e

Format:

Mnemonic: JR **Operands:** e

Object Code:



Description:

This instruction provides for unconditional branching to other segments of a program. The value of the displacement e is added to the Program Counter (PC) and the next instruction is fetched from the location designated by the new contents of the PC. This jump as measured from the address of the instruction opcode has a range of − 126 to + 129 bytes. The assembler automatically adjusts for the twice incremented PC.

M cycles: 3 T states: 12(4,3,5) 4 MHz E.T.: 3.00

Condition Bits Affected: None

Example 1:

To jump forward five locations from address 480, the following assembly language statement is used:

JR $+5

The resulting object code and final PC value is shown below:

Location	Instruction
480	18
481	03
482	—◊PC before jump
483	—
484	—
485	◊PC after jump

Note: when using an assembler, $ + 5 used above would normally be replaced by a label.

Example 2:

This program will skip around the NOP instruction.

```
START   JR, END
        NOP
END     —
```

JR C,e

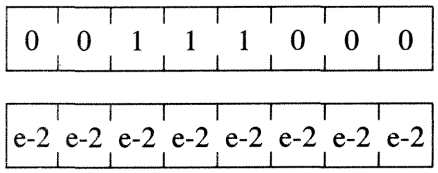
Jump Relative

Operation: If C = 0, continue
If C = 1, PC◊PC + e

Format:

Mnemonic: JR **Operands:** C, e

Object Code:



38

Description:

This instruction provides for conditional branching to other segments of a program depending on the results of a test on the Carry Flag. If the flag is equal to a '1,' the value of the displacement e is added to the Program Counter (PC) and the next instruction is fetched from the location designated by the new contents of the PC. The jump as measured from the address of the instruction opcode has a range of − 126 to + 129 bytes. The assembler automatically adjusts for the twice incremented PC.

If the flag is equal to a '0,' the next instruction to be executed is taken from the location following this instruction.

If condition is met:

M cycles: 3 T states: 12(4,3,5) 4 MHz E.T.: 3.00

If condition is not met:

M cycles: 2 T states: 7(4,3) 4 MHz E.T.: 1.75

Condition Bits Affected: None

Example:

The Carry Flag is set and it is required to jump back four locations from 480. The assembly language statement is:

JR C, $-4

The resulting object code and final PC value is shown below:

Location	Instruction
47C	◊PC after jump
47D	—
47E	—
47F	—
480	38
481	FA (two's complement – 6)
482	◊PC before jump

JR NC,e

Jump Relative

Operation: If C = 1, continue
If C = 0, PC◊PC + e

Format:

Mnemonic: JR **Operands:** NC, e

Object Code:

0	0	1	1	0	0	0	0

30

e-2	e-2	e-2	e-2	e-2	e-2	e-2	e-2

Description:

This instruction provides for conditional branching to other segments of a program depending on the results of a test on the Carry Flag. If the flag is equal to '0,' the value of the displacement e is added to the Program Counter (PC) and

the next instruction is fetched from the location designated by the new contents of the PC. The jump as measured from the address of the instruction opcode has a range of −126 to +129 bytes. The assembler automatically adjusts for the twice incremented PC.

If the flag is equal to a '1,' the next instruction to be executed is taken from the location following this instruction.

If the condition is met:

M cycles: 3 T states: 12(4,3,5) 4 MHz E.T.: 3.00

If the condition is not met:

M cycles: 7 T states: 7(4,3) 4 MHz E.T.: 1.75

Condition Bits Affected: None

Example:

The Carry Flag is reset and it is required to repeat the jump instruction. The assembly language statement is:

JR NC,$

The resulting object code and PC after the jump are shown below:

Location	Instruction
480	30 ◊ PC after jump
481	FD (two's complement − 2)
482	—◊ PC before jump

Note: this instruction would cause an infinite loop in the program.

JR Z,e

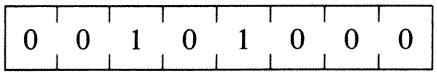
Jump Relative

Operation: Z = 0, continue
If Z = 1, PC ◊ PC + e

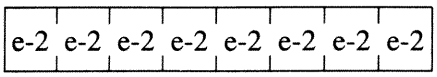
Format:

Mnemonic: JR **Operands:** Z, e

Object Code:



28



Description:

This instruction provides for conditional branching to other segments of a program depending on the results of a test on the Zero Flag. If the flag is equal to a '1,' the value of the displacement e is added to the Program Counter (PC) and the next instruction is fetched from the location designated by the new contents of the PC. The jump as measured from the address of the instruction opcode has a range of − 126 to + 129 bytes. The assembler automatically adjusts for the twice incremented PC.

If the Zero Flag is equal to a '0,' the next instruction to be executed is taken from the location following this instruction.

If the condition is met:

M cycles: 3 T states: 12(4,3,5) 4 MHz E.T.: 3.00

If the condition is not met:

M cycles: 2 T states: 7(4,3) 4 MHz E.T.: 1.75

Condition Bits Affected: None

Example:

The Zero Flag is set and it is required to jump forward five locations from address 300. The following assembly language statement is used:

JR Z, $+5

The resulting object code and final PC value is shown below:

Location	Instruction
300	28
301	03
302	— ◊ PC before jump
303	—
304	—
305	— ◊ PC after jump

JR NZ,e

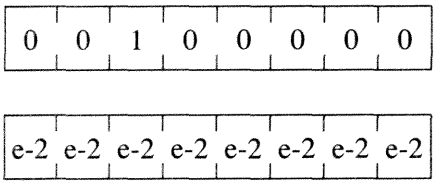
Jump Relative

Operation: If Z = 1, continue
If Z = 0, PC ◊ PC + e

Format:

Mnemonic: JR **Operands:** NZ, e

Object Code:



Description:

This instruction provides for conditional branching to other segments of a program depending on the results of a test on the Zero Flag. If the flag is equal to a '0,' the value of the displacement e is added to the Program Counter (PC) and the next instruction is fetched from the location designated by the new contents of the PC. The jump as measured from the address of the instruction opcode has a range of − 126 to + 129 bytes. The assembler automatically adjusts for the twice incremented PC.

If the Zero Flag is equal to a '1,' the next instruction to be executed is taken from the location following this instruction.

If the condition is met:

M cycles: 3 T states: 12(4,3,5) 4 MHz E.T.: 3.00

If the condition is not met:

M cycles: 2 T states: 7(4,3) 4 MHz E.T.: 1.75

Condition Bits Affected: None

Example:

The Zero Flag is reset and it is required to jump back four locations from 480. The assembly language statement is:

JR NZ, $-4

The resulting object code and final PC value is shown below:

Location	Instruction
47C	◊PC after jump
47D	—
47E	—
47F	—
480	20
481	FA (two's complement − 6)
482	—◊PC before jump

JP (HL)

Jump

Operation: PC ◊ HL

Format:

Mnemonic: JP **Operands:** (HL)

Object Code:

1	1	1	0	1	0	0	1

E9

Description:

The Program Counter (register pair PC) is loaded with the contents of the HL register pair. The next instruction is fetched from the location designated by the new contents of the PC.

M cycles: 1 T states: 4 4 MHz E.T.: 1.00

Condition Bits Affected: None

Example 1:

If the contents of the Program Counter are 1000H and the contents of the HL register pair are 4800H, after the execution of

JP (HL)

the contents of the Program Counter will be 4800H.

The program will jump to the instruction at address 4800H.

Example 2:

A typical software routine which uses JP (HL) is a jump table lookup program. Assume that n 16-bit addresses are listed in consecutive bytes of memory starting at address TBL. Also assume that the Accumulator contains a number from 0 to n-1 representing the routine to be jumped to.

```
LD      HL, TBL  ; HL points to the first byte in the table.
ADD     A, A     ; double A
LD      DE, 0
LD      E, A
ADD     HL, DE   ; if A originally contained 5, then HL now points to the 5th
                   address in the table
LD      E, (HL)
INC     HL
LD      D, (HL)  ; DE now contains the 5th address of the table
LD      HL, DE   ; HL now contains the 5th address of the table
JP      (HL)
```

JP (IX)

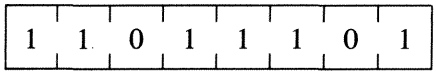
JumP

Operation: PC ◊ IX

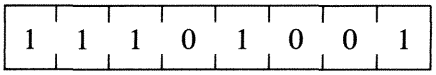
Format:

Mnemonic: JP **Operands:** (IX)

Object Code:



DD



E9

Description:

The Program Counter (register pair PC) is loaded with the contents of the IX Register Pair (Index Register IX). The next instruction is fetched from the location designated by the new contents of the PC.

M cycles: 2 T states: 8(4,4) 4 MHz E.T.: 2.00

Condition Bits Affected: None

Example:

If the contents of the Program Counter are 1000H, and the contents of the IX Register Pair are 4800H, after the execution of

JP (IX)

the contents of the Program Counter will be 4800H.

JP (IY)

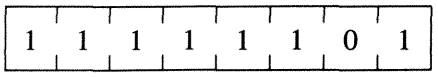
JumP

Operation: PC ◊ IY

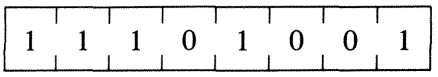
Format:

Mnemonic: JP **Operands:** (IY)

Object Code:



FD



E9

Description:

The Program Counter (register pair PC) is loaded with the contents of the IY Register Pair (Index Register IY). The next instruction is fetched from the location designated by the new contents of the PC.

M cycles: 2 T states: 8(4,4) 4 MHz E.T.: 2.00

Condition Bits Affected: None

Example:

If the contents of the Program Counter are 1000H and the contents of the IY Register Pair are 4800H, after the execution of

JP (IY)

the contents of the Program Counter will be 4800H.

DJNZ e

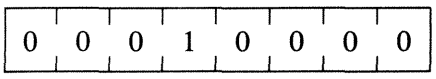
Decrement Jump Not Zero

Operation:

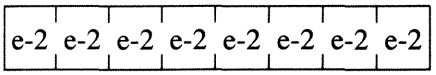
Format:

Mnemonic: DJNZ **Operands:** e

Object Code:



10



Description:

The instruction is similar to the conditional jump instructions except that a register value is used to determine branching. The B register is decremented and if a non zero value remains, the value of the displacement e is added to the Program Counter (PC). The next instruction is fetched from the location

designated by the new contents of the PC. The jump is measured from the address of the instruction opcode has a range of −126 to +129 bytes. The assembler automatically adjusts for the twice incremented PC.

If the result of decrementing leaves B with a zero value, the next instruction to be executed is taken from the location following this instruction.

If B ≠ Ø:

M cycles: 3 T states: 13(5,3,5) 4 MHz E.T.: 3.25

If B = Ø:

M cycles: 2 T states: 8(5,3) 4 MHz E.T.: 2.ØØ

Condition Bits Affected: None

Example:

A typical software routine is used to demonstrate the use of the DJNZ instruction. This routine moves a line from an input buffer (INBUF) to an output buffer (OUTBUF). It moves the bytes until it finds a carriage return, or until it has moved 80 bytes, whichever occurs first.

```
        LD      B, 8Ø       ; Set up counter
        LD      HL, Inbuf   ; Set up pointers
        LD      DE, Outbuf
LOOP:   LD      A, (HL)     ; Get next byte from
                            ; input buffer
        LD      (DE), A     ; Store in output buffer
        CP      ØDH         ; Is it a CR?
        JR      Z, DONE     ; Yes finished
        INC     HL          ; Increment pointers
        INC     DE
        DJNZ    LOOP        ; Loop back if 8Ø
                            ; bytes have not
                            ; been moved
DONE:
```

Call and Return Group

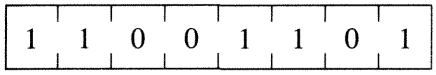
CALL nn

Operation: $(SP-1) \leftarrow PC_H$, $(SP-2) \leftarrow PC_L$, $PC \leftarrow nn$

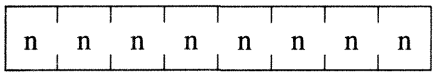
Format:

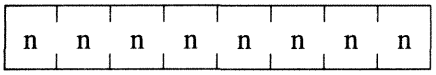
Mnemonic: CALL **Operands:** nn

Object Code:



CD





Note: The first of the two n operands in the assembled object code above is the least significant byte of a two-byte memory address.

Description:

After pushing the current contents of the Program Counter (PC) onto the top of the external memory stack, the operands nn are loaded into PC to point to the address in memory where the first opcode of a subroutine is to be fetched. (At the end of the subroutine, a RETurn instruction can be used to return to the original program flow by popping the top of the stack back into PC.) The push is accomplished by first decrementing the current contents of the Stack Pointer (register pair SP), loading the high-order byte of the PC contents into the memory address now pointed to by the SP; then decrementing SP again, and loading the low-order byte of the PC contents into the top of stack. Note: Because this is a three-byte instruction, the Program Counter will have been incremented by three before the push is executed.

M cycles: 5 T states: 17(4,3,4,3,3) 4 MHz E.T.: 4.25

Condition Bits Affected: None

Example:

If the contents of the Program Counter are 1A47H, the contents of the Stack Pointer are 3002H, and memory locations have the contents:

Location	Contents
1A47H	CDH
1A48H	35H
1A49H	21H

then if an instruction fetch sequence begins, the three-byte instruction CD3521H will be fetched to the CPU for execution. The mnemonic equivalent of this is

CALL 2135H

After the execution of this instruction, the contents of memory address 3001H will be 1AH, the contents of address 3000H will be 4AH, the contents of the Stack Pointer will be 3000H, and the contents of the Program Counter will be 2135H, pointing to the address of the first opcode of the subroutine now to be executed.

Before:

Stack Pointer	Address	Stack
3002	3002	50
	3003	1B
	3004	3C

Program Counter
1A47

After CALL 2135H:

Stack Pointer	Address	Stack
3000	3000	4A
	3001	1A
	3002	50
	3003	1B

Program Counter
2135

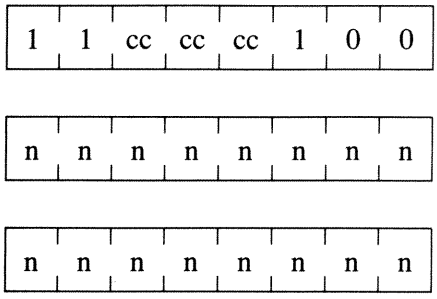
CALL cc,nn

Operation: IF cc TRUE: $(SP-1) \leftarrow PC_H$
$(SP-2) \leftarrow PC_L$, $PC \leftarrow nn$

Format:

Mnemonic: CALL **Operands:** cc, nn

Object Code:



Note: The first of the two n operands in the assembled object code above is the least significant byte of the two-byte memory address.

Description:

If condition cc is true, this instruction pushes the current contents of the Program Counter (PC) onto the top of the external memory stack, then loads the operands nn into PC to point to the address in memory where the first opcode of a subroutine is to be fetched. (At the end of the subroutine, a RETurn instruction can be used to return to the original program flow by popping the top of the stack back into PC.) If condition cc is false, the Program Counter is incremented as usual, and the program continues with the next sequential instruction. The stack push is accomplished by first decrementing the current contents of the Stack Pointer (SP), loading the high-order byte of the PC contents into the memory address now pointed to by SP, then decrementing SP again, and loading the low-order byte of the PC contents into the top of the stack. **Note:** Because this is a three-byte instruction, the Program Counter will have been incremented by three before the push is executed. Condition cc is programmed as one of eight status bits which corresponds to condition bits in the Flag Register (register F). Those eight status bits are defined in the table below, which also specifies the corresponding cc bit fields in the assembled object code:

cc	Condition	Relevant Flag
000	NZ non zero	Z (=0)
001	Z zero	Z (=1)
010	NC non carry	C (=0)
011	C carry	C (=1)
100	PO parity odd	P/V (=0)
101	PE parity even	P/V (=1)
110	P sign positive	S (=0)
111	M sign negative	S (=1)

If cc is true:
M cycles: 5 T states: 17(4,3,4,3,3) 4 MHz E.T.: 4.25
If cc is false:
M cycles: 3 T states: 10(4,3,3) 4 MHz E.T.: 2.50

Condition Bits Affected: None

Example:

If the C Flag in the F register is reset, the contents of the Program Counter are 1A47H, the contents of the Stack Pointer are 3002H, and memory locations have the contents:

Location	Contents
1A47H	D4H
1A48H	35H
1A49H	21H

then if an instruction fetch sequence begins, the three-byte instruction D43521H will be fetched to the CPU for execution. The mnemonic equivalent of this is

CALL NC, 2135H

After the execution of this instruction, the contents of memory address 3001H will be 1AH, the contents of address 3000H will be 4AH, the contents of the Stack Pointer will be 3000H, and the contents of the Program Counter will be 2135H, pointing to the address of the first opcode of the subroutine now to be executed.

RET

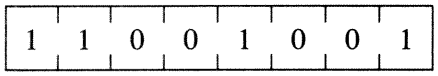
RETurn

Operation: $PC_L \Diamond (SP)$, $PC_H \Diamond (SP + 1)$

Format:

Mnemonic: RET **Operands:**

Object Code:



C9

Description:

Control is returned to the original program flow by popping the previous contents of the Program Counter (PC) off the top of the external memory stack, where they were pushed by the CALL instruction. This is accomplished by first loading the low-order byte of the PC with the contents of the memory address

pointed to by the Stack Pointer (SP), then incrementing the SP and loading the high-order byte of the PC with the contents of the memory address now pointed to by the SP. (The SP is now incremented a second time.) On the following machine cycle the CPU will fetch the next program opcode from the location in memory now pointed to by the PC.

M cycles: 3 T states: 10(4,3,3) 4 MHz E.T.: 2.50

Condition Bits Affected: None

Example:

If the contents of the Program Counter are 3535H, the contents of the Stack Pointer are 2000H, the contents of memory location 2000H are B5H, and the contents of memory location 2001H are 18H, then after the execution of

RET

the contents of the Stack Pointer will be 2002H and the contents of the Program Counter will be 18B5H, pointing to the address of the next program opcode to be fetched.

Before:

Program Counter	Address	Stack
3535	2000	B5
	2001	18
	2002	2E
	2003	30

Stack Pointer
2000

After RET:

Program Counter	Address	Stack
18B5	2002	2E
	2003	30

Stack Pointer
2002

RET cc

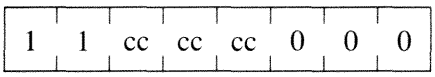
RETurn

Operation: IF cc TRUE: $PC_L \leftarrow (SP)$, $PC_H \leftarrow (SP + 1)$

Format:

Mnemonic: RET **Operands:** cc

Object Code:



Description:

If condition cc is true, control is returned to the original program flow by popping the previous contents of the Program Counter (PC) off the top of the external memory stack, where they were pushed by the CALL instruction. This is accomplished by first loading the low-order byte of the PC with the contents of the memory address pointed to by the Stack Pointer (SP), then incrementing the SP, and loading the high-order byte of the PC with the contents of the memory address now pointed to by the SP. (The SP is now incremented a second time.) On the following machine cycle the CPU will fetch the next program opcode from the location in memory now pointed to by the PC. If condition cc is false, the PC is simply incremented as usual, and the program continues with the next sequential instruction. Condition cc is programmed as one of eight status bits which correspond to condition bits in the Flag Register F). These eight status bits are defined in the table below, which also specifies the corresponding cc bit fields in the assembled object code.

cc	Condition	Relevant Flag
000	NZ non zero	Z (=0)
001	Z zero	Z (=1)
010	NC non carry	C (=0)
011	C carry	C (=1)
100	PO parity odd	P/V (=0)
101	PE parity even	P/V (=1)
110	P sign positive	S (=0)
111	M sign negative	S (=1)

If cc is true:

M cycles: 3 T states: 11(5,3,3) 4 MHz E.T.: 2.75

If cc is false:

M cycles: 1 T states: 5 4 MHz E.T.: 1.25

Condition Bits Affected: None

Example:

If the S flag in the F register is set, the contents of the Program Counter are 3535H, the contents of the Stack Pointer are 2000H, the contents of memory location 2000H are B5H, and the contents of memory location 2001H are 18H, then after the execution of

RET M

the contents of the Stack Pointer will be 2002H and the contents of the Program Counter will be 18B5H, pointing to the address of the next program opcode to be fetched.

RETI

Operation: Return from interrupt

Format:

Mnemonic: RETI **Operands:**

Object Code:

1	1	1	0	1	1	0	1	ED

0	1	0	0	1	1	0	1	4D

Description:

This instruction is used at the end of an interrupt service routine to:

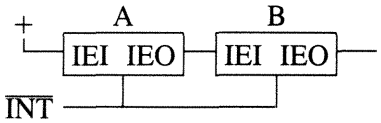
1. Restore the contents of the Program Counter (PC) (analogous to the RET instruction).
2. To signal an I/O device that the interrupt routine has been completed. The RETI instruction facilitates the nesting of interrupts, allowing higher priority devices to suspend service of lower priority service routines.

M cycles: 4 T states: 14(4,4,3,3) 4 MHz E.T.: 3.50

Condition Bits Affected: None

Example:

Given: Two interrupting devices, A and B, connected in a daisy chain configuration with A having a higher priority than B.



B generates an interrupt and is acknowledged. (The interrupt enable out, IEO, of B goes low, blocking any lower priority devices from interrupting while B is being serviced). Then A generates an interrupt, suspending service of B. (The

IEO of A goes 'low' indicating that a higher priority device is being serviced.) The A routine is completed and a RETI is issued resetting the IEO of A, allowing the B routine to continue. A second RETI is issued on completion of the B routine and the IEO of B is reset (high), allowing lower priority devices interrupt access.

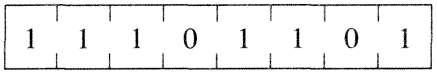
RETN

Operation: Return from non maskable interrupt

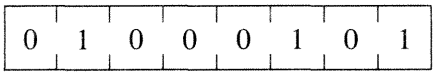
Format:

Mnemonic: RETN **Operands:**

Object Code:



ED



45

Description:

Used at the end of a service routine for a non maskable interrupt, this instruction executes an unconditional return which functions identically to the RET instruction. That is, the previously stored contents of the Program Counter (PC) are popped off the top of the external memory stack; the low-order byte of PC is loaded with the contents of the memory location pointed to by the Stack Pointer (SP), SP is incremented, the high-order byte of PC is loaded with the contents of the memory location now pointed to by SP, and SP is incremented again. Control is now returned to the original program flow: on the following machine cycle the CPU will fetch the next opcode from the location in memory now pointed to by the PC. Also the state of IFF2 is copied back into IFF1 to the state it had prior to the acceptance of the NMI.

M cycles: 4 T states: 14(4,4,3,3) 4 MHz E.T.: 3.50

Condition Bits Affected: None

Example:

If the contents of the Stack Pointer are 1000H and the contents of the Program Counter are 1A45H when a non maskable interrupt (NMI) signal is received, the CPU will ignore the next instruction and will instead restart to memory address 0066H. That is, the current Program Counter contents of 1A45H will be pushed onto the external stack address of 0FFFH and 0FFEH, high order byte first, and

0066H will be loaded onto the Program Counter. That address begins an interrupt service routine which ends with RETN instruction. Upon the execution of RETN, the former Program Counter contents are popped off the external memory stack, low-order first, resulting in a Stack Pointer contents again of 1000H. The program flow continues where it left off with an opcode fetch to address 1A45H.

RST p

ReSTart

Operation: $(SP-1) \Leftarrow PC_H$, $(SP-2) \Leftarrow PC_L$, $PC_H \Leftarrow O$, $PC_L \Leftarrow P$

Format:

Mnemonic: RST **Operands:** P

Object Code:

1	1	t	t	t	1	1	1

Description:

The current Program Counter (PC) contents are pushed onto the external memory stack, and the page zero memory location given by operand p is loaded into the PC. Program execution then begins with the opcode in the address now pointed to by PC. The push is performed by first decrementing the contents of the Stack Pointer (SP), loading the high-order byte of PC into the memory address now pointed to by SP, decrementing SP again, and loading the low-order byte of PC into the address now pointed to by SP. The ReSTart instruction allows for a Call to a subroutine at one of eight addresses as shown in the table below. The operand p is assembled into the object code using the t column of the table. **Note:** Since all addresses are in page zero of memory, the high order byte of PC is loaded with 00H. The number selected from the "p" column of the table is loaded into the low-order byte of PC.

At the end of the subroutine a RETurn instruction can be used to return to the original program by popping the top of the stack back into PC.

P	t
00H	000
08H	001
10H	010
18H	011
20H	100
28H	101
30H	110
38H	111

M cycles: 3 T states: 11(5,3,3) 4 MHz E.T.: 2.75

Example:

If the contents of the Program Counter are 15B3H, after the execution of

RST 18H (Object code 11011111)

the PC will contain 0018H, as the address of the next opcode to be fetched, and the top number on the stack will be 15B3H.

Input and Output Group

IN A,(n)

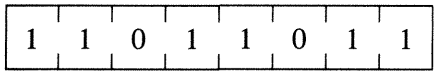
INput

Operation: A ⇦ (n)

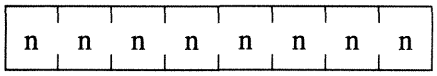
Format:

Mnemonic: IN **Operands:** A, (n)

Object Code:



DB



Description:

The number of the input port is n. Data is input to register A. The operand n is placed on the bottom half (A0 through A7) of the address bus to select the I/O device at one of 256 possible ports. The contents of the Accumulator also appear on the top half (A8 through A15) of the address bus at this time. Then one byte from the selected port is placed on the data bus and written into the Accumulator (register A) in the CPU.

M cycles: 3 T states: 11(4,3,4) 4 MHz E.T.: 2.75

Condition Bits Affected: None

Example:

If the contents of the Accumulator are 23H and the byte 7BH is available at the peripheral device mapped to I/O port address 01H, then after the execution of

```
IN  A,(01H)
```

the Accumulator will contain 7BH.

IN r,(C)

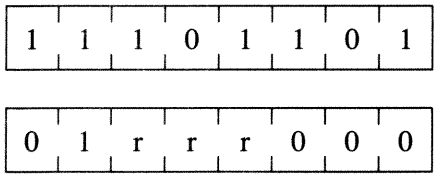
INput

Operation: r ◊ (C)

Format:

Mnemonic: IN **Operands:** r, (C)

Object Code:



Description:

Register C contains the number of the input port. Data is input to register r. The contents of register C are placed on the bottom half (A0 through A7) of the address bus to select the I/O device at one of 256 possible ports. The contents of Register B are placed on the top half (A8 through A15) of the address bus at this time. Then one byte from the selected port is placed on the data bus and written into register r in the CPU. Register r identifies any of the CPU registers shown in the following table, which also shows the corresponding three-bit "r" field for each. The flags will be affected, checking the input data.

Register	r
B	000
C	001
D	010
E	011
H	100
L	101
A	111

M cycles: 3 T states: 12(4,4,4) 4 MHz E.T.: 3.00

Condition Bits Affected:

S: Set if input data is negative; reset otherwise
Z: Set if input data is zero; reset otherwise
H: Reset
P/V: Set if parity is even; reset otherwise
N: Reset
C: Not affected

Example:

If the contents of register C are 07H, the contents of register B are 10H, and the byte 7BH is available at the peripheral device mapped to I/O port address 07H, then after the execution of

IN D,(C)

register D will contain 7BH

A typical use of the IN r, (C) instruction is for polled I/O. The following program continually polls or inputs data from port FF until a non-zero number appears. The program then reads in data from port FE. In this application, port FF is used as a data ready signal for port FE.

```
        LD      C, 0FFH       ; C points at port FF
LOOP    IN      B, (C)        ; input port FF to register B
        JR      Z, LOOP       ; continue polling until not zero
        IN      A, (0FEH)     ; input port FE to register A
```

INI

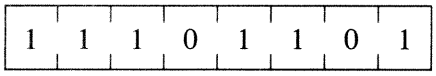
INput & Increment

Operation: (HL) ◊ (C), B ◊ B − 1, HL ◊ HL + 1

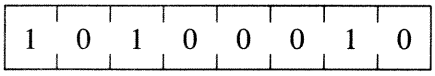
Format:

Mnemonic: INI **Operands:**

Object Code:



ED



A2

Description:

Register C contains the number of the input port. Data input is placed in memory at the address pointed at by HL. The contents of register C are placed on the bottom half (A0 through A7) of the address bus to select the I/O device at one of 256 possible ports. Register B may be used as a byte counter, and its contents are placed on the top half (A8 through A15) of the address bus at this time. Then one byte from the selected port is placed on the data bus and written to the CPU. The contents of the HL register pair are then placed on the address bus and the input byte is written into the corresponding location of memory. Finally the byte counter is decremented and register pair HL is incremented.

M cycles: 4 T states: 16(4,5,3,4) 4 MHz E.T.: 4.00

Condition Bits Affected:

S: Unknown
Z: Set if B − 1 = Ø; reset otherwise
H: Unknown
P/V: Unknown
N: Set
C: Not affected

Example:

If the contents of register C are Ø7H, the contents of register B are 1ØH, the contents of the HL register pair are 1ØØØH, and the byte 7BH is available at the peripheral device mapped to I/O port address Ø7H, then after the execution of

INI

memory location 1ØØØH will contain 7BH, the HL register pair will contain 1ØØ1H, and register B will contain ØFH.

The following program will input data from input ports 1 through 80 and place the data into a buffer in memory.

```
        LD      B, 80
        LD      C,0
        LD      HL, BUFF
LOOP    INC     C
        INI
        JP      NZ, LOOP
```

INIR

INput Increment & Repeat

Operation: (HL) ◊ (C), B ◊ B − 1, HL ◊ HL + 1

Format:

Mnemonic: INIR **Operands:**

Object Code:

1	1	1	0	1	1	0	1	ED

1	0	1	1	0	0	1	0	B2

Description:

Register C contains the number of the input port. The data input is placed in memory at the address pointed at by the HL register pair. The contents of register C are placed on the bottom half (A0 through A7) of the address bus to select the I/O device at one of 256 possible ports. Register B is used as a byte counter, and its contents are placed on the top half (A8 through A15) of the address bus at this time. Then one byte from the selected port is placed on the data bus and written to the CPU. The contents of the HL register pair are placed on the address bus and the input byte is written into the corresponding location of memory. Then register pair HL is incremented, the byte counter is decremented. If decrementing causes B to go to zero, the instruction is terminated. If B is not zero, the PC is decremented by two and the instruction repeated. Note that if B is set to zero prior to instruction execution, 256 bytes of data will be input. Also interrupts will be recognized after each data transfer.

If B $\neq$ 0:

M cycles: 5 T states: 21(4,5,3,4,5) 4 MHz E.T.: 5.25

If B = 0:

M cycles: 4 T states: 16(4,5,3,4) 4 MHz E.T.: 4.00

Condition Bits Affected:

S: Unknown
Z: Set
H: Unknown
P/V: Unknown
N: Set
C: Not affected

Example:

If the contents of register C are 07H, the contents of register B are 03H, the contents of the HL register pair are 1000H, and the following sequence of bytes are available at the peripheral device mapped to I/O port of address 07H:

51H
A9H
03H

then after the execution of

INIR

the HL register pair will contain 1003H, register B will contain zero, and memory locations will have contents as follows:

Location	Contents
1000H	51H
1001H	A9H
1002H	03H

Here is a program to input 80 bytes from I/O port number FF and put them into an 80-byte buffer starting at address BUFF.

```
LD      HL, BUFF    ; HL points at first byte of buffer
LD      B, 80       ; load byte counter
LD      C, OFFH     ; port FF
IN IR               ; input 80 bytes
```

Note: this assumes that the input port can be synchronized with the input instructions.

IND

INput & Decrement

Operation: (HL) ◊ (C), B ◊ B − 1, HL ◊ HL − 1

Format:

Mnemonic: IND **Operands:**

Object Code:

1	1	1	0	1	1	0	1	ED

1	0	1	0	1	0	1	0	AA

Description:

The contents of register C are placed on the bottom half (A0 through A7) of the address bus to select the I/O device at one of 256 possible ports. Register B may be used as a byte counter, and its contents are placed on the top half (A8 through A15) of the address bus at this time. Then one byte from the selected port is placed on the data bus and written to the CPU. The contents of the HL register pair are placed on the address bus and the input byte is written into the corresponding location of memory. Finally the byte counter and register pair HL are decremented.

M cycles: 4 T states: 16(4,5,3,4) 4 MHz E.T.: 4.00

Condition Bits Affected:

S: Unknown
Z: Set if B − 1 = 0; reset otherwise
H: Unknown
P/V: Unknown
N: Set
C: Not affected

Example:

If the contents of register C are 07H, the contents of register B are 10H, the contents of the HL register pair are 1000H, and the byte 7BH is available at the peripheral device mapped to I/O port address 07H, then after the execution of

IND

memory location 1000H will contain 7BH, the HL register pair will contain 0FFFH, and register B will contain 0FH.

INDR

INput Decrement & Repeat

Operation: (HL) ◊ (C), B ◊ B − 1, HL ◊ HL − 1

Format:

Mnemonic: INDR **Operands:**

Object Code:

1	1	1	0	1	1	0	1	ED

1	0	1	1	1	0	1	0	BA

Description:

The contents of register C are placed on the bottom half (A0 through A7) of the address bus to select the I/O device at one of 256 possible ports. Register B is used as a byte counter, and its contents are placed on the top half (A8 through A15) of the address bus at this time. Then one byte from the selected port is placed on the data bus and written to the CPU. The contents of the HL register pair are placed on the address bus and the input byte is written into the corresponding location of memory. Then HL and the byte counter are decremented. If decrementing causes B to go to zero, the instruction is terminated. If B is not zero, the PC is decremented by two and the instruction repeated. Note that if B is set to zero prior to instruction execution, 256 bytes of data will be input. Also interrupts will be recognized after each data transfer.

If B ≠ 0:

M cycles: 5 T states: 21(4,5,3,4,5) 4 MHz E.T.: 5.25

If B = 0:

M cycles: 4 T states: 16(4,5,3,4) 4 MHz E.T.: 4.00

Condition Bits Affected:

S: Unknown
Z: Set
H: Unknown
P/V: Unknown
N: Set
C: Not affected

Example:

If the contents of register C are 07H, the contents of register B are 03H, the contents of the HL register pair are 1000H, and the following sequence of bytes are available at the peripheral device mapped to I/O port address 07H:

51H
A9H
03H

then after the execution of

INDR

the HL register pair will contain 0FFDH, register B will contain zero, and memory locations will have contents as follows:

Location	Contents
0FFEH	03H
0FFFH	A9H
1000H	51H

OUT (n),A

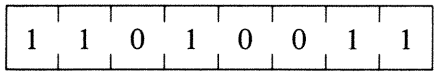
OUTput

Operation: (n) ⇦ A

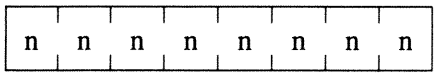
Format:

Mnemonic: OUT **Operands:** (n), A

Object Code:



D3



Description:

The operand n is placed on the bottom half (A0 through A7) of the address bus to select the I/O device at one of 256 possible ports. The contents of the Accumulator (register A) also appear on the top half (A8 through A15) of the address bus at this time. Then the byte contained in the Accumulator is placed on the data bus and written into the selected peripheral device.

M cycles: 3 T states: 11(4,3,4) 4 MHz E.T.: 2.75

Condition Bits Affected: None

Example:

If the contents of the Accumulator are 23H, then after the execution of

OUT 01H,A

the byte 23H will have been written to the peripheral device mapped to I/O port address 01H.

OUT (C),r

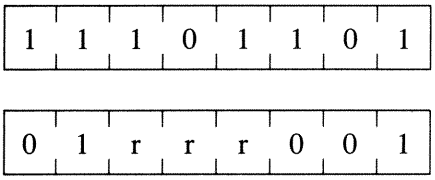
OUTput

Operation: (C) ◊ r

Format:

Mnemonic: OUT **Operands:** (C), r

Object Code:



Description:

The contents of register C are placed on the bottom half (A0 through A7) of the address bus to select the I/O device at one of 256 possible ports. The contents of Register B are placed on the top half (A8 through A15) of the address bus at this time. Then the byte contained in register r is placed on the data bus and written into the selected peripheral device. Register r identifies any of the CPU registers shown in the following table, which also shows the corresponding three-bit "r" field for each which appears in the assembled object code:

Register	r
B	000
C	001
D	010
E	011
H	100
L	101
A	111

M cycles: 3 T states: 12(4,4,4) 4 MHz E.T.: 3.00

Condition Bits Affected: None

Example:

If the contents of register C are 01H and the contents of register D are 5AH, after the execution of

OUT (C),D

the byte 5AH will have been written to the peripheral device mapped to I/O port address 01H.

OUTI

OUTput & Increment

Operation: (C) ◊ (HL), B ◊ B − 1, HL ◊ HL + 1

Format:

Mnemonic: OUTI **Operands:**

Object Code:

1	1	1	0	1	1	0	1	ED

1	0	1	0	0	0	1	1	A3

Description:

The contents of the HL register pair are placed on the address bus to select a location in memory. The byte contained in this memory location is temporarily stored in the CPU. Then, after the byte counter (B) is decremented, the contents of register C are placed on the bottom half (A0 through A7) of the address bus to select the I/O device at one of 256 possible ports. Register B may be used as a byte counter, and its decremented value is placed on the top half (A8 through

A15) of the address bus. The byte to be output is placed on the data bus and written into selected peripheral device. Finally the register pair HL is incremented.

M cycles: 4 T states: 16(4,5,3,4) 4 MHz E.T.: 4.00

Condition Bits Affected:

S: Unknown
Z: Set if B − 1 = 0; reset otherwise
H: Unknown
P/V: Unknown
N: Set
C: Not affected

Example:

If the contents of register C are 07H, the contents of register B are 10H, the contents of the HL register pair are 1000H, and the contents of memory address 1000H are 59H, then after the execution of

OUTI

register B will contain 0FH, the HL register pair will contain 1001H, and the byte 59H will have been written to the peripheral device mapped to I/O port address 07H.

OTIR

OuTput Increment & Repeat

Operation: (C) ◊ (HL), B ◊ B − 1, HL ◊ HL + 1

Format:

Mnemonic: OTIR **Operands:**

Object Code:

1	1	1	0	1	1	0	1	ED

1	0	1	1	0	0	1	1	B3

Description:

The contents of the HL register pair are placed on the address bus to select a location in memory. The byte contained in this memory location is temporarily stored in the CPU. Then, after the byte counter (B) is decremented, the contents of register C are placed on the bottom half (A0 through A7) of the address bus

to select the I/O device at one of 256 possible ports. Register B may be used as a byte counter, and its decremented value is placed on the top half (A8 through A15) of the address bus at this time. Next the byte to be output is placed on the data bus and written into the selected peripheral device. Then register pair HL is incremented. If the decremented B register is not zero, the Program Counter (PC) is decremented by two and the instruction is repeated. If B has gone to zero, the instruction is terminated. Note that if B is set to zero prior to instruction execution, the instruction will output 256 bytes of data. Also, interrupts will be recognized after each data transfer.

If B ≠ Ø:

M cycles: 5 T states: 21(4,5,3,4,5) 4 MHz E.T.: 5.25

If B = Ø:

M cycles: 4 T states: 16(4,5,3,4) 4 MHz E.T.: 4.ØØ

Condition Bits Affected:

S: Unknown
Z: Set
H: Unknown
P/V: Unknown
N: Set
C: Not affected

Example:

If the contents of register C are Ø7H, the contents of register B are Ø3H, the contents of the HL register pair are 1ØØØH, and memory locations have the following contents:

Location	Contents
1ØØØH	51H
1ØØ1H	A9H
1ØØ2H	Ø3H

then after the execution of

OTIR

the HL register pair will contain 1ØØ3H, register B will contain zero, and a group of bytes will have been written to the peripheral device mapped to I/O port address Ø7H in the following sequence:

51H
A9H
Ø3H

OUTD

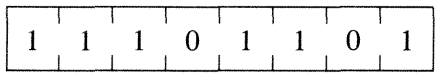
OUTput & Decrement

Operation: (C) ◊ (HL), B ◊ B − 1, HL ◊ HL − 1

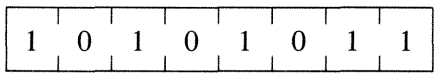
Format:

Mnemonic: OUTD **Operands:**

Object Code:



ED



AB

Description:

The contents of the HL register pair are placed on the address bus to select a location in memory. The byte contained in this memory location is temporarily stored in the CPU. Then, after the byte counter (B) is decremented, the contents of register C are placed on the bottom half (A0 through A7) of the address bus to select the I/O device at one of 256 possible ports. Register B may be used as a byte counter, and its decremented value is placed on the top half (A8 through A15) of the address bus at this time. Next the byte to be output is placed on the data bus and written into the selected peripheral device. Finally the register pair HL is incremented.

M cycles: 4 T states: 16(4,5,3,4) 4 MHz E.T.: 4.00

Condition Bits Affected:

S: Unknown
Z: Set if B − 1 = 0; reset otherwise
H: Unknown
P/V: Unknown
N: Set
C: Not affected

Example:

If the contents of register C are 07H, the contents of register B are 10H, the contents of the HL register pair are 1000H, and the contents of memory location 1000H are 59H, after the execution of

OUTD

register B will contain 0FH, the HL register pair will contain 0FFFH, and the byte 59H will have been written to the peripheral device mapped to I/O port address 07H.

OTDR OUTput Decrement & Repeat

Operation: (C) ◊ (HL), B ◊ B – 1, HL ◊ HL – 1

Format:

Mnemonic: OTDR **Operands:**

Object Code:

1	1	1	0	1	1	0	1	ED

1	0	1	1	1	0	1	1	BB

Description:

The contents of the HL register pair are placed on the address bus to select a location in memory. The byte contained in this memory location is temporarily stored in the CPU. Then, after the byte counter (B) is decremented, the contents of register C are placed on the bottom half (A0 through A7) of the address bus to select the I/O device at one of 256 possible ports. Register B may be used as a byte counter, and its decremented value is placed on the top half (A8 through A15) of the address bus at this time. Next the byte to be output is placed on the data bus and written into the selected peripheral device. Then register pair HL is decremented and if the decremented B register is not zero, the Program Counter (PC) is decremented by 2 and the instruction is repeated. If B has gone to zero, the instruction is terminated. Note that if B is set to zero prior to instruction execution, the instruction will output 256 byte of data. Also, interrupts will be recognized after each data transfer.

If B ≠ 0:

M cycles: 5 T states: 21(4,5,3,4,5) 4 MHz E.T.: 5.25

If B = 0:

M cycles: 4 T states: 16(4,5,3,4) 4 MHz E.T.: 4.00

Condition Bits Affected:

S: Unknown
Z: Set
H: Unknown
P/V: Unknown
N: Set
C: Not affected

Example:

If the contents of register C are 07H, the contents of register B are 03H, the contents of the HL register pair are 1000H, and memory locations have the following contents:

Location	Contents
0FFEH	51H
0FFFH	A9H
1000H	03H

then after the execution of

OTDR

the HL register pair will contain 0FFDH, register B will contain zero, and a group of bytes will have been written to the peripheral device mapped to I/O port address 07H in the following sequence:

03H
A9H
51H

Chapter 10
Extended Z80 Instructions

The ALDS Assembler contains a number of extended Z80 instructions. You can use them the same way you use other Z80 instructions.

An extended instruction is actually an internally defined macro. When you assemble the instruction, the Assembler expands it into a group of Z80 instructions. A description of macros is in *Chapter 8*.

Notations

In addition to the notations described in *Chapter 9*, this chapter uses:

xx	a register pair
yy	a register pair
[ ]	optional value

Format Of Each Instruction

This chapter uses the same format for the instructions as *Chapter 9*, with the following exceptions:

- many of the instruction formats show different combinations of operands. These combinations are listed under "Operands"
- following the description of each instruction is a breakdown of how the instruction expands when assembled
- the operation is not shown
- the object code is not shown

CPR *operand* — ComPare double Register

Mnemonic: CPR **Operands:** *xx* (where *xx* = BC, DE, HL, or SP)

Description:

Compares the contents of the operand to the contents of HL. If they compare, the Z bit is set.

Example:

If register pair BC contains an A0H and HL contains an A0H.

```
CPR    BC
```

sets the Z bit.

Expansion: CPR *xx*

```
PUSH   HL
OR     A
SBC    HL,xx
POP    HL
```

CMPD *operand1,operand2,[length]*

CoMPare with Decrement

Mnemonic: CMPD

Operands:		
	nn1,nn2,n	length is *n*.
	nn1,nn2	length is contents of BC.
	nn1,nn2,(nn3)	length is contents of *nn3*.
	nn1,(nn2)	length is last byte of the string beginning at *operand2*.

Description:

Compares the string beginning at *operand1* and ending at (*operand1 - length*) with the string beginning at *operand2*, and ending at (*operand2 - length*). The Z bit is set according to the result of the comparison. Zero length strings are equal.

If a mismatch occurs, HL and DE will contain the addresses preceeding that mismatch.

Example:

If memory location 4000-4006 contains the string1 "develop" and location 5000-5006 contains the string2 "envelop", the operation

```
CMPD    4006H,5006H,7
```

starts the comparison of the two strings with the last byte, in this case the 'p'. A mismatch occurs at the second letter. Because of this mismatch, the address of the preceding 'n' is now in register HL and the address of the preceding 'e' in register DE.

Exit Conditions:

All registers modified

Expansion: CMPD *nn1,nn2,n*

```
        LD      DE,nn1
        LD      HL,nn2
        LD      BC,n
X2:     LD      A,B
        OR      C
        JR      Z,X1
        LD      A,(DE)
        CP      (HL)
        JR      NZ,X1
        LDD
        JR      X2
X1:
```

Expansion: CMPD *nn1,nn2*

```
        LD      DE,nn1
        LD      HL,nn2
X2:     LD      A,B
        OR      C
        JR      Z,X1
        LD      A,(DE)
        CP      (HL)
        JR      NZ,X1
        LDD
        JR      X2
X1:
```

Expansion: CMPD *nn1,nn2,(nn3)*

```
        LD      DE,nn1
        LD      HL,nn2
        LD      A(nn3)
        LD      C,A
        LD      B,0
X2:     LD      A,B
        OR      C
        JR      Z,X1
        LD      A(DE)
        CP      (HL)
        JR      NZ,X1
        LDD
        JR      X2
X1:
```

Expansion: CMPD *nn1*,(*nn2*)

```
        LD      DE,nn1
        LD      HL,nn2
        LD      C,(HL)
        LD      B,0
        INC     HL
X2:     LD      A,B
        OR      C
        JR      Z,X1
        LD      A(DE)
        CP      (HL)
        JR      NZ,X1
        LDD
        JR      X2
X1:
```

Note: The symbols used in the expansion are shown for clarity and are not actually defined for use by other statements.

CMPI *operand1,operand2,length*

CoMPare with Increment

Mnemonic: CMPI **Operands:** *nn1,nn2,n* length is specified.
nn1,nn2 length in BC.
nn1,nn2 (nn3) length is contents of *nn3*.
nn1,(nn2) length is first byte of *nn2*.

Description:

Compares the string beginning at *operand1* with the string beginning at *operand2* for the given *length*. Depending on the operands, *length* can be specified as a constant, the contents of an address, or the contents of the BC register. If a match does not occur, HL and DE will contain the addresses following that mismatch. The Z bit is set according to the result of the comparison. Zero length strings are equal.

Example:

If memory location 4000-4006 contains the string1 "develop" and location 5000-5006 contained the string2 "envelop":

```
CMPI    4000H,5000H,7
```

starts the comparison of the two strings beginning with the first byte (in this case, the 'd' in string1 and the 'e' in string2). A mismatch occurs at the first letter. The address of 'd' is now in register DE and the address of 'e' is now in register HL where the comparison failed.

Exit Conditions:

All registers modified

Expansion: CMPI *nn1,nn2,n*

```
        LD      DE,nn1
        LD      HL,nn2
        LD      BC,n
X2:     LD      A,B
        OR      C
        JR      Z,X1
        LD      A(DE)
        CP      (HL)
        JR      NZ,X1
        LDI
        JR      X2
X1:
```

Expansion: CMPI *nn1,nn2*

```
        LD      DE,nn1
        LD      HL,nn2
X2:     LD      A,B
        OR      C
        JR      Z,X1
        LD      A,(DE)
        CP      (HL)
        JR      NZ,X1
        LDI
        JR      X2
X1:
```

Expansion: CMPI *nn1,nn2,(nn3)*

```
        LD      DE,nn1
        LD      HL,nn2
        LD      A,(nn3)
        LD      C,A
        LD      B,0
X2:     LD      A,B
        OR      C
        JR      Z,X1
        LD      A,(DE)
        CP      (HL)
        JR      NZ,X1
        LDI
        JR      X2
X1:
```

Expansion: CMPI *nn1*,(*nn2*)

```
        LD      DE,nn1
        LD      HL,nn2
        LD      C,(HL)
        LD      B,Ø
        INC     HL
X2:     LD      A,B
        OR      C
        JR      Z,X1
        LD      A,(DE)
        CP      (HL)
        JR      NZ,X1
        LDI
        JR      X2
X1:
```

Note: The labels used in the expansion are shown for clarity and are not actually defined for use by other statements.

TZ *operand*

Test register for Zero

Mnemonic: TZ **Operands:** *xx* (where *xx* = BC, DE, HL, IX, or IY)

Description:

Compares the contents of *xx* to zero. If true, the Z bit is set.

Example:

If the contents of BC contains a ØØH then the operation

```
TZ   BC
```

sets the Z bit. Any other value (i.e. AØH) sets the NZ bit.

Note: TZ IX and TZ IY are instructions which are not documented by ZILOG. Although they should assemble properly, Radio Shack does not guarantee that they will work on all processors. You should test them in your own environment to ensure their validity.

Expansion: TZ *xx*

```
        LD      A,high order byte of xx
        OR      low order byte of xx
```

EX *operand*

EXchange double register with (SP)

Mnemonic: EX **Operands:** (SP),*xx* where *xx* = AF, BC, or DE

Description:

Exchanges the low order byte contained in *xx* with the contents of the memory address specified by the contents of the stack pointer (SP). The high order byte of *xx* is exchanged with the next highest memory address (SP + 1).

Example:

If the contents of the register pair BC is 3978H and the stack pointer (SP) and its next byte (SP + 1) contains 2357H:

```
EX    (SP),BC
```

causes the register pair BC to contain 2357H and the top address of the stack to contain 4978H.

Expansion: EX (SP),*xx* where *xx* = AF or BC

```
EX    (SP),HL
PUSH  xx
PUSH  HL
POP   xx
POP   HL
EX    (SP),HL
```

Expansion: EX (SP),DE

```
EX    DE,HL
EX    (SP),HL
EX    DE,HL
```

EX *operand1,operand2*

EXchange double register

Mnemonic: EX **Operands:** *xx,yy* where *xx* and *yy* are any register pairs listed under "Expansion" below.

Description:

Exchanges the two-byte contents of *xx* with the contents of *yy*.

Example:

The contents of BC is 6789H and the contents of DE is 1234H. After the execution of:

```
EX    BC,DE
```

the values are exchanged so that BC contains 1234H and DE contains 6789H.

Expansion: EX AF,BC
EX AF,DE
EX BC,DE

```
PUSH   1st Operand
PUSH   2nd Operand
POP    1st Operand
POP    2nd Operand
```

Expansion: EX*xx,yy* (*xx* = AF, BC or DE
yy = IX or IY)

```
PUSH   1st Operand
EX     (SP),2nd Operand
POP    1st Operand
```

Expansion: EX HL,*xx* (*xx* = IX or IY)
EX IX,IY
EX *xx*,HL, (*xx* = AF or BC)

```
PUSH   1st Operand
EX     (SP),2nd Operand
POP    1st Operand
```

Expansion: EX (SP), *xx* (*xx* = AF, BC)

```
EX     (SP), HL
PUSH   2nd Operand
PUSH   HL
POP    2nd Operand
EX     (SP), HL
```

Expansion: EX (SP), DE

```
EX     (SP), HL
EX     DE, HL
EX     (SP), HL
```

LD *operand1,operand2* LoaD

Mnemonic: LD **Operands:** *xx,yy*
(xx),yy
xx,(yy)
(xx),(yy)

Description:

Loads the first operand with the second operand. The numbers shown in the tables (1-14) represent the coded expansions for the pair of operands. Details of each expansion follow the tables (i.e. BC,AF refer to expansion description #1).

Example:

The operation:

```
LD    HL,DE
```

copies the contents of DE to HL.

First Operand	Second Operand							
	BC	DE	HL	(BC)	(DE)	(HL)	(IX+DD)	(IY+DD)
(BC)	4	4	6	—	9	9	9	9
(DE)	4	4	7	9	—	9	9	9
(HL)	5	5	8	9	9	9	9	9
(IX+DD)	5	5	5	9	9	9	—	9
(IY+DD)	5	5	5	9	9	9	9	—

First Operand	Second Operand						
	AF	BC	DE	HL	IX	IY	A
AF	1	1	1	1	1	1	—
BC	1	3	3	3	1	1	2
DE	1	3	3	3	1	1	2
HL	1	3	3	3	1	1	2
IX	1	1	1	1	1	1	2
IY	1	1	1	1	1	1	2

First Operand	Second Operand				
	(BC)	(DE)	(HL)	(IX+DD)	(IY+DD)
BC	11	12	10	10	10
DE	12	11	10	10	10
HL	13	13	14	10	10

(—) indicates operand pairs not applicable

(1) Expansion: LD *xx,yy* where *xx* and *yy* are any of the following operand pairs:

```
AF,AF ; AF,BC ; AF,DE ; AF,HL ; AF,IX
  ; AF,IY
BC,AF ; BC,IX ; BC,IY
DE,AF ; DE,IX ; DE,IY
HL,AF ; HL,IX ; HL,IY
IX,AF ; IX,BC ; IX,DE ; IX,HL ; IX,IX
  ; IX,IY
IY,AF ; IY,BC ; IY,DE ; IY,HL ; IY,IX
  ; IY,IY
```

```
PUSH   2nd Operand
POP    1st Operand
```

(2) Expansion: LD *xx,yy* where *xx* and *yy* are any of the following operand pairs:

```
BC,A ; DE,A ; HL,A ; IX,A ; IY,A
```

```
LD     Low order byte of register pair,A (accumulator)
LD     High order byte of register pair,0
```

Note: LD IX,A and LD IY,A are instructions which are not documented by ZILOG. Although they should assemble properly, Radio Shack does not guarantee that they will work on all processors. You should test them in your own environment to ensure their validity.

(3) Expansion: LD *xx,yy* where *xx* and *yy* are any of the following operand pairs:

```
BC,BC ; BC,DE ; BC,HL
DE,BC ; DE,DE ; DE,HL
HL,BC ; HL,DE ; HL,HL
```

```
LD     High order byte 1st Operand, High order byte 2nd
       Operand
LD     Low order byte 1st Operand, Low order byte 2nd
       Operand
```

(4) Expansion: LD xx,yy where *xx* and *yy* are any of the following operand pairs:

```
(BC),BC ; (BC),DE
(DE),BC ; (DE),DE
```

```
PUSH   1st Operand
EX     (SP),HL
LD     (HL),Low order byte 2nd Operand
INC    HL
```

```
LD    (HL),High order byte 2nd Operand
EX    (SP),HL
POP   1st Operand
```

Side Effect: First operand register is incremented by 1.

(5) Expansion: LD *xx,yy* where *xx* and *yy* are any of the following operand pairs:

```
(HL),BC ; (HL),DE
(IX+DD),BC ; (IX+DD),DE ; (IX+DD),HL
(IY+DD),BC ; (IY+DD),DE ; (IY+DD),HL
```

```
LD    (1st Operand),Low order byte 2nd Operand
INC   Register of 1st operand
LD    (1st Operand),High order byte 2nd Operand
```

Side Effect: first operand register is incremented by 1.

(6) Expansion: LD (BC),HL

```
PUSH  AF
LD    A,L
LD    (BC),A
INC   BC
LD    A,H
LD    (BC),A
POP   AF
```

Side Effect: Register BC is incremented by 1.

(7) Expansion: LD (DE),HL

```
PUSH  AF
LD    A,L
LD    (DE),A
INC   DE
LD    A,H
LD    (DE),A
POP   AF
```

Side Effect: Register DE is incremented by 1.

(8) Expansion: LD (HL),HL

```
PUSH  AF
LD    A,H
LD    (HL),L
INC   HL
LD    (HL),A
POP   AF
```

Side Effect: Register HL is incremented by 1.

(9) Expansion: LD *xx,yy* where *xx* and *yy* are any of the following operand pairs:

```
(BC),(DE) ; (BC),(HL) ; (BC),(IX+DD)
  ; (BC),(IY+DD)
(DE),(BC) ; (DE),(HL) ; (DE),(IX+DD)
  ; (BC),(IY+DD)
(HL),(BC) ; (HL),(DE) ; (HL),(IX+DD)
  ; (HL),(IY+DD)
(IX+DD),(BC) ; (IX+DD),(DE) ;
  (IX+DD),(HL);(IX+DD),(IY+DD)
(IY+DD),(BC) ; (IY+DD),(DE) ;
  (IY+DD),(HL);(IY+DD),(IX+DD)
```

```
LD    A,(2nd Operand)
LD    (1st Operand),A
```

Side Effect: Register A is changed.

(10) Expansion: LD *xx,yy* where *xx* and *yy* are any of the following operand pairs:

```
BC,(HL) ; BC,(IX+DD) ; BC,(IY+DD)
DE,(HL) ; DE,(IX+DD) ; DE,(IY+DD)
HL,(IX+DD) ; HL,(IY+DD)
```

```
LD    Low order byte 1st Operand,(2nd Operand)
INC   Contents of 2nd Operand, register
LD    High order byte 1st Operand,(2nd Operand)
```

Side Effect: 2nd operand Register is incremented (HL,IX or IY)

(11) Expansion: LD *xx,(yy)* where *xx* and (*yy*) are either of the following operand pairs:

```
BC,(BC) ; DE,(DE)
```

```
PUSH  Contents of 2nd Operand
EX    (SP),HL
LD    Low order byte of 1st Operand,(HL)
INC   HL
LD    High order byte of 1st Operand,(HL)
POP   HL
```

(12) Expansion: LD *xx,(yy)* where *xx* and (*yy*) are either of the following operand pairs:

```
BC,(DE) ; DE,(BC)
```

```
PUSH  Contents of 2nd Operand
EX    (SP),HL
LD    Low order byte of 1st Operand,(HL)
```

```
INC    HL
LD     High order byte of 1st Operand,(HL)
EX     (SP),HL
POP    Contents of 2nd Operand
```

Side Effect: 2nd operand register is incremented by 1.

(13) Expansion: LD *xx,(yy)* where *xx* and (*yy*) are either of the following operand pairs:

```
HL,(BC) ; HL,(DE)
```

```
PUSH   AF
LD     A,(2nd Operand)
LD     L,A
INC    Contents of 2nd Operand
LD     A,(2nd Operand)
LD     H,A
POP    AF
```

Side Effect: 2nd operand Register is incremented by 1.

(14) Expansion: LD HL,(HL)

```
PUSH   AF
LD     A,(HL)
INC    HL
LD     H,(HL)
LD     L,A
POP    AF
```

MOVD *operand1,operand2,length*

MOVe with Decrement

Mnemonic: MOVD

Operands:		
	nn1,nn2,n	length is specified.
	nn1,nn2	length is in BC.
	nn1,nn2,(nn3)	length is contents of *nn3* (byte).
	nn1,(nn2)	length is first byte of *nn2*.

Description:

Moves a string of a given length (implied in the operand) from the address of *operand2* to the address of *operand1*. MOVD starts at the end of the string and moves backward starting at the address of *operand2*.

You can specify the length as a constant, the contents of an address, or the contents of the BC register.

Example:

If the address *4000* contained the string "develop":

```
MOVD 5000H,4000H,7
```

moves "develop" from address 3FFA-*4000* to 4FFA-*5000* starting with the end of the string, (i.e. 'p') which would be located at address *5000*H .

Expansion: MOVD *nn1,nn2,n*

```
        LD      DE,nn1
        LD      HL,nn2
        LD      BC,n
        LDDR
```

Expansion: MOVD *nn1,nn2*

```
        LD      DE,nn1
        LD      HL,nn2
        LD      A,B
        OR      C
        JR      Z,X1
        LDDR
X1:
```

Expansion: *MOVD nn1,nn2,(nn3)*

```
        LD      DE,nn1
        LD      HL,nn2
        LD      A,(nn3)
        LD      C,A
        LD      B,0
        OR      A
        JR      Z,X1
        LDDR
X1:
```

Expansion: MOVD *nn1,(nn2)*

```
        LD      DE,nn1
        LD      HL,nn2
        LD      C,(HL)
        LD      B,0
        INC     HL
        LD      A,B
        OR      C
        JR      Z,X1
        LDDR
X1:
```

MOVI *operand1,operand2,length*

MOVe with Increment

Mnemonic: MOVI **Operands:** *nn1,nn2,n* length is specified.
nn1,nn2 length is in BC.
nn1,nn2,(nn3) length is contents of *nn3*.
nn1,(nn2) length is first byte of *nn2*.

Description:

Moves a string of the given length from the address of *operand2* to the address of *operand1*. MOVI starts at the beginning of the string and moves forward.

You can specify the length as a constant, the contents of a memory address, or the contents of the BC register.

Example:

If location 4001H contains the string "develop", the instruction:

```
MOVI 5000H,4000H,7
```

moves "develop" from address 4001H to 5000H starting with d, the first letter.

Expansion: MOVI *nn1,nn2,n*

```
        LD      DEnn1
        LD      HLnn2
        LD      BC,n
        LDIR
```

Expansion: MOVI *nn1,nn2*

```
        LD      DE,nn1
        LD      HL,nn2
        LD      A,B
        OR      C
        JR      Z,X1
        LDIR
X1:
```

Expansion: MOVI *nn1,nn2,(nn3)*

```
        LD      DE,nn1
        LD      HL,nn2
        LD      A,nn3
        LD      C,A
        LD      B,0
        OR      A
        JR      Z,X1
        LDIR
X1:
```

Expansion: MOVI *nn1,(nn2)*

```
        LD      DE,nn1
        LD      HL,nn2
        LD      C,(HL)
        LD      B,0
        INC     HL
        LD      A,B
        OR      C
        JR      Z,X1
        LDIR
X1:
```

POP

Mnemonic: POP **Operands:** none

Description:

Increments the stack pointer one full word.

Example:

If the stack pointer contains the byte 39H on top and 45H in the next location

```
POP
```

increments the stack pointer past these two bytes to the next point.

Expansion:

```
        INC     SP
        INC     SP
```

RSTR *operand*

ReSToRe

Mnemonic: RSTR **Operands:** *n* where *n* =

none	restores HL,DE BC
4	restores HL,DE BC and AF
I	restores HL,DE BC,AF,IX,IY
P	restores HL,DE BC,AF,IX,IY,HL' DE',BC'
A	restores HL,DE BC,AF,IX,IY,HL' DE,BC',AF'

Description:

Restores the registers specified by the *operand* after a SAVE (see extended instruction). This is often used after a return from a subroutine.

Example:

If registers HL, DE, BC are saved (See SAVE),

```
RSTR
```

restores them to their original values.

Expansion: RSTR

```
POP   HL
POP   DE
POP   BC
```

Expansion: RSTR 4

```
POP   HL
POP   DE
POP   BC
POP   AF
```

Expansion: RSTR I

```
POP   HL
POP   DE
POP   BC
POP   AF
POP   IY
POP   IX
```

Expansion: RSTR P

```
POP   HL
POP   DE
POP   BC
POP   AF
POP   IY
POP   IX
EXX
POP   HL
POP   DE
POP   BC
EXX
```

Expansion: RSTR A

```
POP   HL
POP   DE
POP   BC
POP   AF
POP   IY
POP   IX
```

```
EXX
POP   HL
POP   DE
POP   BC
EXX
EX    AF,AF'
POP   AF
EX    AF,AF'
```

SAVE *operand*

Mnemonic: SAVE **Operands:** *n* where *n* =

none	saves HL,DE,BC
4	saves HL,DE,BC AF
I	saves HL,DE,BC, AF,IX,IY
P	saves HL,DE,BC AF,IX,IY,HL' DE',BC'
A	saves HL,DE,BC, AF,IX,IY,HL', DE',BC',AF'

Description:

Copies the contents of the registers specified by thes38*operand*. This is useful before executing a subroutine. The registers are restored with RSTR (see extended instruction).

Example:

```
SAVE
```

saves the contents of registers HL, DE, BC, to free them for use, then executes a SAVE.

Expansion: SAVE

```
PUSH  BC
PUSH  DE
PUSH  HL
```

Expansion: SAVE 4

```
PUSH  AF
PUSH  BC
PUSH  DE
PUSH  HL
```

Expansion: SAVE I

```
PUSH   IX
PUSH   IY
PUSH   AF
PUSH   BC
PUSH   DE
PUSH   HL
```

Expansion: SAVE P

```
EXX
PUSH   BC
PUSH   DE
PUSH   HL
EXX
PUSH   IX
PUSH   IY
PUSH   AF
PUSH   BC
PUSH   DE
PUSH   HL
```

Expansion: SAVE A

```
EX     AF,AF'
PUSH   AF
EX     AF,AF'
EXX
PUSH   BC
PUSH   DE
PUSH   HL
EXX
PUSH   IX
PUSH   IY
PUSH   AF
PUSH   BC
PUSH   DE
PUSH   HL
```

SVC *operand*

SuperVisory Call

Mnemonic: SVC **Operands:** *n*

Description:

Performs the supervisory call specified by *n*.

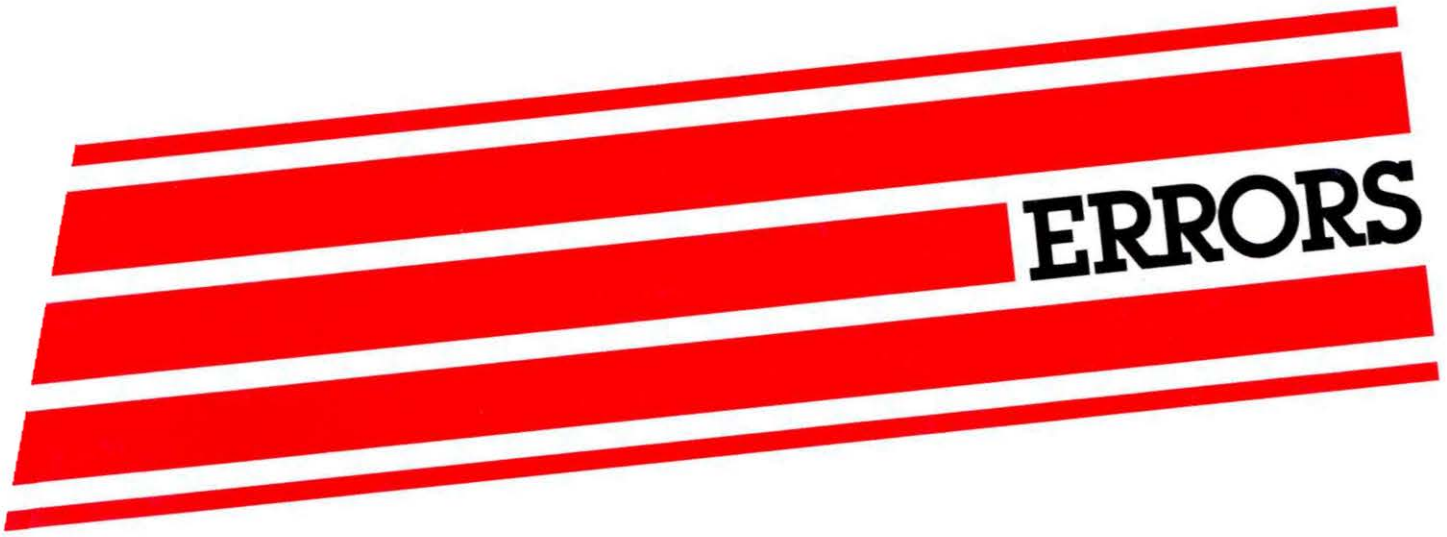
Expansion:

Model 4:

```
LD      A,n
RST     28H
```

Model III:

```
PUSH    BC
PUSH    DE
PUSH    HL
CALL    n
POP     HL
POP     DE
POP     BC
```


ERRORS

Section III
Error Messages

Error Messages

Editor Error Messages

Bad File Format

The file is not a type ALEDIT can load, either fixed LRL 1 or Variable, and with record length not greater than 256 bytes.

Bad Filename Format

The filename is too long or incorrectly formatted on a load or a write command.

Bad Parameters

The ASCII line number converted to hexadecimal is greater than 65535 decimal (for line number request).

The change string is zero or the length of the line to be changed is zero (for Change command).

Buffer Full

There is no more room in the edit buffer. Program returns from any mode back to the command mode. Note that the edit buffer is about 4K smaller if DO, HOST, COMM, SPOOL, DEBUG or ALBUG are on.

Line Length Too Long, Truncating Line

You are loading a file that has lines longer than 78 characters.

Line Number Too Large

The line number is larger than the last line number in the file.

The editor does not recognize your command. Re-type it.

No Text

The edit buffer is empty, the only commands which are effective are:

K, L, Y, I, Q, J, S

Occurrence Too Large

In the Find and Change commands the occurrence is greater than 255.

Search ARG Too Long

The string you want to search for is longer than 37 characters.

Syntax Error

The command is improperly specified.

Total Line Length Too Long

The new line created by a Change command is greater than the acceptable Line Length.

If the Editor returns an error code, it is a TRSDOS error message. You can identify it, by simply typing in the error number. For example, at TRSDOS READY type:

```
ERROR 19 (ENTER)
```

or at the Editor command mode, type:

```
S ERROR 19 (ENTER)
```

and your computer answers you with the correct identification:

```
INVALID FILE NAME
```

You can do this any time your computer identifies an error which you are not aware of.

Hit Any Key To Continue

If there is an error in the load or write routines, the Editor waits for the user to read the entire error message.

Assembler Error Codes

Code	Meaning
A	Arithmetic Overflow — result of a multiplication is outside the range of −65536 – +65535
B	Balance Error of Brackets
C	Condition Error ELSE outside an IF . . . ENDIF pair Unterminated IF ENDIF without matching IF Macro defined after a macro was expanded
D	Macro Definition Error ENDM outside a macro definition Macro not terminated when END statement was reached. Parameter substitution (i.e. "#9") specified in the body of the macro for a parameter not listed in the heading. Macro body too long.
E	Missing END statement Missing ENDM statement
F	Include files nested too deeply
I	Illegal character Control character in source file.
L	Maximum Line Length Exceeded. The limit is 254 characters a line
M	Multiple Definition of a Symbol This includes defining a symbol and declaring it EXTRN
O	Stack Overflow — expression too complicated
P	Phase Error — Symbol appears or changes value after Pass 1. This is often caused by using symbols in the operand field of EQU, DEFS, or ORG before those symbols are defined.
R	Range Error in Relative Addressing. Use a JP instead of JR, or rearrange code.

Code	Meaning
S	Syntax Error Illegal operation code Too few, too many, or the wrong type of operands Use of an external symbol or relocatable expression where it is not allowed Use of an instruction generating object code within an ISECT Use of an instruction before a PSECT Instruction illegal after a LINK directive
T	Mixing of absolute and relocatable PSECTs
U	Undefined Symbol
V	Illegal Value Value too large to fit in a single byte (−256 − +255 permitted) Illegal combination of relocatable or external symbols
W	Reserved word used as a symbol. Do not use a register name or branch condition as a symbol

Linker Error Messages

Symbol Table Overflow

There are too many external symbols to fit in memory. Reduce the number of symbols declared public or global by assembling several modules together, or using shorter names.

Multiply Defined Entry Symbol

The indicated symbol has been defined more than once (and declared public and/or global). The two or more definitions may be in the same object file (the assembler will output an 'M' error) or in different files. Note that using the same name for a public or global symbol in one file and for a local symbol (not declared PUBLIC, GLOBAL or EXTRN) in another file is permitted.

Address Different from Pass 1

The indicated symbol changed values between Pass 1 and Pass 2. Normally this error is preceeded by a "Multiply Defined Entry Symbol" message and the cause is the same. This error may also be caused by changing disks in the middle of a link, inserting a disk with a different version of the same object file in a lower drive number during the link, or linking corrupted object files.

The two addresses are the values from Pass 1 and Pass 2 respectively. These values and the PSECT map may be used to locate the modules containing the definitions, assuming that the value falls within the code area of the module.

Undefined External Symbol

The indicated symbol is declared EXTRN in at least one module and is never defined and declared PUBLIC or GLOBAL in any module included in the link. This is usually caused by failing to declare a label PUBLIC, omitting files that should have been included in the link, or linking incomplete programs to test just the implemented parts. In the last case, if the instructions referring to the undefined symbol are never used, the error may be ignored.

Missing External Transfer Address

The main program ends with NOEND, or the object file has been corrupted. The main program should terminate with END and a transfer address.

Illegal Addressing

The load address being computed by the linker wraps around from FFFFH to 0000H. Reduce the size of your program or use a lower load address.

Invalid Parameter

The LINKs are nested too deeply; an illegal character was specified in a filename on the command line, LINK, or GLINK instruction, the source filename is missing, or errors were found in the $ = XXXX parameter.

Linker TRSDOS Errors

File Not Found

Object file not found.

Note: Default extension is /REL.

Attempt to Use a Non-Program File As a Program

The file used is incomplete or in NOLOAD format, or is not an object file.

Open Attempt For a File Already Open

Another file, directly or indirectly, attempted to include itself with a LINK directive.

Note: Default extension is /REL. Also, other errors may include: disk read/write errors, password protection, illegal disk change, disk full etc.

APPENDICES

Appendix A / Undocumented Z80 Instructions

Note: These instructions are not documented by ZILOG. Radio Shack does not guarantee that they will work on all processors. You should test them in your own environment to ensure their validity.

Shift/Load Instructions

Note: These instructions are not documented by ZILOG. Radio Shack does not guarantee that they will work on all processors. You should test them in your own environment to ensure their validity.

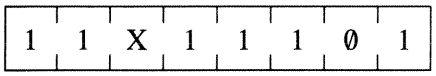
In the following list, the undocumented instructions on the left perform the same function as the corresponding instructions on the right, except that the memory location data is shifted or rotated and stored in both the register and the memory location.

RLCLD	r,m	RLC	m
RLLD	r,m	RL	m
RRCLD	r,m	RRC	m
RRLLD	r,m	RRL	m
SLALD	r,m	SLA	m
SLOLD	r,m	SLO	m
SRALD	r,m	SRA	m
SRLLD	r,m	SRL	m

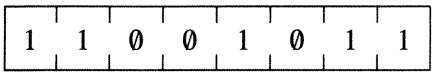
r is one of the following registers: A,B,C,D,E,H, or L
m is one of the following: (IX + d) or (IY + d)

The operation of the condition code bits and instruction timing is believed to be the same as for the corresponding shift or rotate instruction.

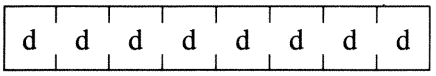
Object Code:

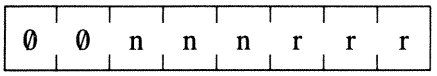


DD for (IX + d)
FD for (IY + d)



CB





n =	RLCLD	0	r =	111	A
	RLLD	2		000	B
	RRCLD	1		001	C
	RRLLP	3		010	D
	SLALD	4		011	E
	SLOLD	6		100	H
	SRALD	5		101	L
	SRLLD	7			

Bit Set/Load And Bit Reset/Load Instructions

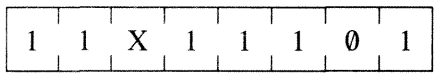
Note: These instructions are not documented by ZILOG. Radio Shack does not guarantee that they will work on all processors. You should test them in your own environment to ensure their validity.

In the following list, the undocumented instructions on the left perform the same function as the corresponding instructions on the right except that the resulting data after the bit operation is loaded in both the memory location and the register.

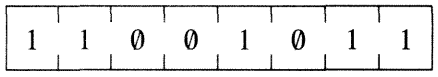
RESLD	r,n,m	RES	n,m
SETLD	r,n,m	SET	n,m

r is one of the following registers: A,B,C,D,E,H or L
n is a bit number with value between 0 and 7, inclusive
m is either (IX + d) or (IY + d)

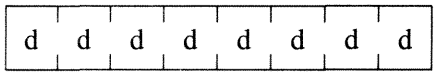
Object Code:

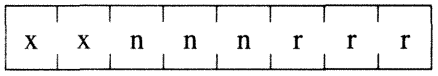


DD for (IX + d)
FD for (IY + d)



CB





x = 10 RESLD
11 SETLD

n = bit number

r = 111 A
000 B
001 C
010 D
011 E
100 H
101 L

Index Register Half Instructions

Note: These instructions are not documented by ZILOG. Radio Shack does not guarantee that they will work on all processors. You should test them in your own environment to ensure their validity.

The upper and lower bytes of the index registers IX and IY may be manipulated individually. To use these instructions, the following register names are used:

XH	High Byte of IX
XL	Low Byte of IX
YH	High Byte of IY
YL	Low Byte of IY

The object code generated has a prefix byte of DD or FD (for the halves of the IX or IY register) and otherwise is the same as the corresponding instructions with the H or L register used in place of the high or low byte of an index register.

The XH, XL, YH and YL registers may be used in the following instructions:

ADC	A,XH	LD	r,XH
ADD	A,XH	LD	XH,r
AND	XH	LD	XH,n
CP	XH	OR	XH
DEC	XH	SBC	A,XH
INC	XH	SUB	XH
		XOR	XH

r = A, B, C, D, or E

Appendix B / ALDS Object Code Format

Each record is a variable number of bytes, packed consecutively in an LRL 256 file. Records may span sector boundaries. The file is terminated by a record with an 02 or 03 header. For further information, see the Model III or Model 4 Owner's Manual.

Object Code:

HEADER (1)	LENGTH (1)		
Ø1	n+2	Load address (2)	Data bytes (n)

Absolute Entry*

Ø2	Ø2	Absolute Entry Point (2)

Relocatable Entry*

Ø2	Ø3	Relocatable Entry Offset (2)

Load — only*

Ø3	Ø2	Ø Ø Ø Ø (2)

External entry*

Ø3	ØD	FLAGS Ø1ØØØØ11	Object (2)	External Name (1Ø)

Relocatable Object Data

Ø4	Ø3	FLAGS ØØØØ1xxx	Object (2)

External Object Data

Ø4	ØD	FLAGS Ø1ØØ11xx	Object (2)	External Name (1Ø)

*One of These Records Terminates Each Object File

Public Label w/Object

04	0F	FLAGS 100x1xxx	Public Label Offset (2)
		Object (2)	Public Label Name (10)

Public Label w/o Object

04	0F	FLAGS 100x0011	Public Label Offset (2)
		0 0 0 0 (2)	Public Label Name (10)

Public Label w/External

04	19	FLAGS 110x11xx	Public Label Offset (2)
	Object (2)	Public Label Offset (2)	External Name (10)

LINK

09	n+1	FLAGS 00100000	File Name (n)

GLINK

09	n+1	FLAGS 00110000	File Name (n)

Numbers given under flags are in binary. X = varies depending on particular situation.

FLAGS for 03, 04, 07 Records

7	0 = No public name is present (bit 4 = 0) 1 = Public name is present
6	0 = External name is not present 1 = External name is present (bits 3, 2, = 1, 1)
5	0 Reserved

4	Ø = Address of public label is relocatable or not present, or this is an absolute file 1 = Address of public label in a relocatable file is absolute. (bit 7 = 1)
3	Ø = No object present (bits, 1, Ø = Ø, 1, 1) 1 = Object code is present
2	Ø = Object is absolute or not present 1 = Object is relocatable (bit 3 = 1)
1 Ø	ØØ = Illegal combination Ø1 = Use only MSB of result (bit 3 = 1) 1Ø = Use only LSB of result (bit 3 = 1) 11 = Use both LSB and MSB of result (if bit 3 = 1) or object not present (if bit 3 = Ø)

If object is absolute (bit 2 = Ø) Result = object
If object is relocatable (bit 2 = 1)
Result = object + PSECTS origin (if bit 6 = Ø)
or Result = object + External name value (if bit 6 = 1)

FLAGS for 05/06 Records

7	1 = File contains relocatable object
6	1 = file contains externals
5	Ø = Reserved
4	1 = File contains public records
3	1 = File contains a link or glink file name
2	Ø Reserved
1	Ø Reserved
Ø	Ø Reserved

Appendix C / Numeric List of Instruction Set

Following is a listing of object codes in numerical order in column two followed by the nmemonic or source statement in column four.

LOC	OBJ CODE	STMT	SOURCE STATEMENT	LOC	OBJ CODE	STMT	SOURCE STATEMENT
0000	00	1	NOP	004F	3620	55	LD (HL),N
0001	018405	2	LD BC,NN	0051	37	56	SCF
0004	02	3	LD (BC),A	0052	382E	57	JR C,DIS
0005	03	4	INC BC	0054	39	58	ADD HL,SP
0006	04	5	INC B	0055	3A8405	59	LD A,(NN)
0007	05	6	DEC B	0058	3B	60	DEC SP
0008	0620	7	LD B,N	0059	3C	61	INC A
000A	07	8	RLCA	005A	3D	62	DEC A
000B	08	9	EX AF,AF'	005B	3E20	63	LD A,N
000C	09	10	ADD HL,BC	005D	3F	64	CCF
000D	0A	11	LD A,(BC)	005E	40	65	LD B,B
000E	0B	12	DEC BC	005F	41	66	LD B,C
000F	0C	13	INC C	0060	42	67	LD B,D
0010	0D	14	DEC C	0061	43	68	LD B,E
0011	0E20	15	LD C,N	0062	44	69	LD B,H
0013	0F	16	RRCA	0063	45	70	LD B,L
0014	102E	17	DJNZ DIS	0064	46	71	LD B,(HL)
0016	118405	18	LD DE,NN	0065	47	72	LD B,A
0019	12	19	LD (DE),A	0066	48	73	LD C,B
001A	13	20	INC DE	0067	49	74	LD C,C
001B	14	21	INC D	0068	4A	75	LD C,D
001C	15	22	DEC D	0069	4B	76	LD C,E
001D	1620	23	LD D,N	006A	4C	77	LD C,H
001F	17	24	RLA	006B	4D	78	LD C,L
0020	182E	25	JR DIS	006C	4E	79	LD C,(HL)
0022	19	26	ADD HL,DE	006D	4F	80	LD C,A
0023	1A	27	LD A,(DE)	006E	50	81	LD D,B
0024	1B	28	DEC DE	006F	51	82	LD D,C
0025	1C	29	INC E	0070	52	83	LD D,D
0026	1D	30	DEC E	0071	53	84	LD D,E
0027	1E20	31	LD E,N	0072	54	85	LD D,H
0029	1F	32	RRA	0073	55	86	LD D,L
002A	202E	33	JR NZ,DIS	0074	56	87	LD D,(HL)
002C	218405	34	LD HL,NN	0075	57	88	LD D,A
002F	228405	35	LD (NN),HL	0076	58	89	LD E,B
0032	23	36	INC HL	0077	59	90	LD E,C
0033	24	37	INC H	0078	5A	91	LD E,D
0034	25	38	DEC H	0079	5B	92	LD E,E
0035	2620	39	LD H,N	007A	5C	93	LD E,H
0037	27	40	DAA	007B	5D	94	LD E,L
0038	282E	41	JR Z,DIS	007C	5E	95	LD E,(HL)
003A	29	42	ADD HL,HL	007D	5F	96	LD E,A
003B	2A8405	43	LD HL,(NN)	007E	60	97	LD H,B
003E	2B	44	DEC HL	007F	61	98	LD H,C
003F	2C	45	INC L	0080	62	99	LD H,D
0040	2D	46	DEC L	0081	63	100	LD H,E
0041	2E20	47	LD L,N	0082	64	101	LD H,H
0043	2F	48	CPL	0083	65	102	LD H,L
0044	302E	49	JR NC,DIS	0084	66	103	LD H,(HL)
0046	318405	50	LD SP,NN	0085	67	104	LD H,A
0049	328405	51	LD (NN),A	0086	68	105	LD L,B
004C	33	52	INC SP	0087	69	106	LD L,C
004D	34	53	INC (HL)	0088	6A	107	LD L,D
004E	35	54	DEC (HL)	0089	6B	108	LD L,E

LOC	OBJ CODE	STMT	SOURCE STATEMENT
008A	6C	109	LD L,H
008B	6D	110	LD L,L
008C	6E	111	LD L,(HL)
008D	6F	112	LD L,A
008E	70	113	LD (HL),B
008F	71	114	LD (HL),C
0090	72	115	LD (HL),D
0091	73	116	LD (HL),E
0092	74	117	LD (HL),H
0093	75	118	LD (HL),L
0094	76	119	HALT
0095	77	120	LD (HL),A
0096	78	121	LD A,B
0097	79	122	LD A,C
0098	7A	123	LD A,D
0099	7B	124	LD A,E
009A	7C	125	LD A,H
009B	7D	126	LD A,L
009C	7E	127	LD A,(HL)
009D	7F	128	LD A,A
009E	80	129	ADD A,B
009F	81	130	ADD A,C
00A0	82	131	ADD A,D
00A1	83	132	ADD A,E
00A2	84	133	ADD A,H
00A3	85	134	ADD A,L
00A4	86	135	ADD A,(HL)
00A5	87	136	ADD A,A
00A6	88	137	ADC A,B
00A7	89	138	ADC A,C
00A8	8A	139	ADC A,D
00A9	8B	140	ADC A,E
00AA	8C	141	ADC A,H
00AB	8D	142	ADC A,L
00AC	8E	143	ADC A,(HL)
00AD	8F	144	ADC A,A
00AE	90	145	SUB B
00AF	91	146	SUB C
00B0	92	147	SUB D
00B1	93	148	SUB E
00B2	94	149	SUB H
00B3	95	150	SUB L
00B4	96	151	SUB (HL)
00B5	97	152	SUB A
00B6	98	153	SBC A,B
00B7	99	154	SBC A,C
00B8	9A	155	SBC A,D
00B9	9B	156	SBC A,E
00BA	9C	157	SBC A,H
00BB	9D	158	SBC A,L
00BC	9E	159	SBC A,(HL)
00BD	9F	160	SBC A,A
00BE	A0	161	AND B
00BF	A1	162	AND C
00C0	A2	163	AND D
00C1	A3	164	AND E
00C2	A4	165	AND H
00C3	A5	166	AND L
00C4	A6	167	AND (HL)
00C5	A7	168	AND A
00C6	A8	169	XOR B
00C7	A9	170	XOR C
00C8	AA	171	XOR D
00C9	AB	172	XOR E
00CA	AC	173	XOR H
00CB	AD	174	XOR L
00CC	AE	175	XOR (HL)
00CD	AF	176	XOR A
00CE	B0	177	OR B
00CF	B1	178	OR C
00D0	B2	179	OR D
00D1	B3	180	OR E
0002	B4	181	OR H
00D3	B5	182	OR L
00D4	B6	183	OR (HL)
00D5	B7	184	OR A
00D6	B8	185	CP B
00D7	B9	186	CP C
00D8	BA	187	CP D
00D9	BB	188	CP E
00DA	BC	189	CP H
00DB	BD	190	CP L
00DC	BE	191	CP (HL)
00DD	BF	192	CP A
00DE	C0	193	RET NZ
00DF	C1	194	POP BC
00E0	C28405	195	JP NZ, NN
00E3	C38405	196	JP NN
00E6	C48405	197	CALL NZ,NN
00E9	C5	198	PUSH BC
00EA	C620	199	ADD A,N
00EC	C7	200	RST 0
00ED	C8	201	RET Z
00EE	C9	202	RET
00EF	CA8405	203	JP Z,NN
00F2	CC8405	204	CALL Z,NN
00F5	CD8405	205	CALL NN
00F8	CE20	206	ADC A,N
00FA	CF	207	RST 8
00FB	D0	208	RET NC
00FC	D1	209	POP DE
00FD	D28405	210	JP NC,NN
0100	D320	211	OUT N,A
0102	D48405	212	CALL NC,NN
0105	D5	213	PUSH DE
0106	D620	214	SUB N
0108	D7	215	RST 10H
0109	D8	216	RET C
010A	D9	217	EXX
010B	DA8405	218	JP C,NN
010E	DB20	219	IN A,N
0110	DC8405	220	CALL C,NN
0113	DE20	221	SBC A,N
0115	DF	222	RST 18H
0116	E0	223	RET PO
0117	E1	224	POP HL
0118	E28405	225	JP PO,NN
011B	E3	226	EX (SP),HL
011C	E48405	227	CALL PO,NN
011F	E5	228	PUSH HL
0120	E620	229	AND N
0122	E7	230	RST 20H
0123	E8	231	RET PE
0124	E9	232	JP (HL)
0125	EA8405	233	JP PE,NN
0128	EB	234	EX DE,HL

LOC	OBJ CODE	STMT	SOURCE STATEMENT
0129	EC8405	235	CALL PE,NN
012C	EE20	236	XOR N
012E	EF	237	RST 28H
012F	F0	238	RET P
0130	F1	239	POP AF
0131	F28405	240	JP P,NN
0134	F3	241	DI
0135	F48405	242	CALL P,NN
0138	F5	243	PUSH AF
0139	F620	244	OR N
013B	F7	245	RST 30H
013C	F8	246	RET M
013D	F9	247	LD SP,HL
013E	FA8405	248	JP M,NN
0141	FB	249	EI
0142	FC8405	250	CALL M,NN
0145	FE20	251	CP N
0147	FF	252	RST 38H
0148	CB00	253	RLC B
014A	CB01	254	RLC C
014C	CB02	255	RLC D
014E	CB03	256	RLC E
0150	CB04	257	RLC H
0152	CB05	258	RLC L
0154	CB06	259	RLC (HL)
0156	CB07	260	RLC A
0158	CB08	261	RRC B
015A	CB09	262	RRC C
015C	CB0A	263	RRC D
015E	CB0B	264	RRC E
0160	CB0C	265	RRC H
0162	CB0D	266	RRC L
0164	CB0E	267	RRC (HL)
0166	CB0F	268	RRC A
0168	CB10	269	RL B
016A	CB11	270	RL C
016C	CB12	271	RL D
016E	CB13	272	RL E
0170	CB14	273	RL H
0172	CB15	274	RL L
0174	CB16	275	RL (HL)
0176	CB17	276	RL A
0178	CB18	277	RR B
017A	CB19	278	RR C
017C	CB1A	279	RR D
017E	CB1B	280	RR E
0180	CB1C	281	RR H
0182	CB1D	282	RR L
0184	CB1E	283	RR (HL)
0186	CB1F	284	RR A
0188	CB20	285	SLA B
018A	CB21	286	SLA C
018C	CB22	287	SLA D
018E	CB23	288	SLA E
0190	CB24	289	SLA H
0192	CB25	290	SLA L
0194	CB26	291	SLA (HL)
0196	CB27	292	SLA A
0198	CB28	293	SRA B
019A	CB29	294	SRA C
019C	CB2A	295	SRA D
019E	CB2B	296	SRA E
01A0	CB2C	297	SRA H
01A2	CB2D	298	SRA L
01A4	CB2E	299	SRA (HL)
01A6	CB2F	300	SRA A
01A8	CB38	301	SRL B
01AA	CB39	302	SRL C
01AC	CB3A	303	SRL D
01AE	CB3B	304	SRL E
01B0	CB3C	305	SRL H
01B2	CB3D	306	SRL L
01B4	CB3E	307	SRL (HL)
01B6	CB3F	308	SRL A
01B8	CB40	309	BIT 0,B
01BA	CB41	310	BIT 0,C
01BC	CB42	311	BIT 0,D
01BE	CB43	312	BIT 0,E
01C0	CB44	313	BIT 0,H
01C2	CB45	314	BIT 0,L
01C4	CB46	315	BIT 0,(HL)
01C6	CB47	316	BIT 0,A
01C8	CB48	317	BIT 1,B
01CA	CB49	318	BIT 1,C
01CC	CB4A	319	BIT 1,D
01CE	CB4B	320	BIT 1,E
01D0	CB4C	321	BIT 1,H
01D2	CB4D	322	BIT 1,L
01D4	CB4E	323	BIT 1,(HL)
01D6	CB4F	324	BIT 1,A
01D8	CB50	325	BIT 2,B
01DA	CB51	326	BIT 2,C
01DC	CB52	327	BIT 2,D
01DE	CB53	328	BIT 2,E
01E0	CB54	329	BIT 2,H
01E2	CB55	330	BIT 2,L
01E4	CB56	331	BIT 2,(HL)
01E6	CB57	332	BIT 2,A
01E8	CB58	333	BIT 3,B
01EA	CB59	334	BIT 3,C
01EC	CB5A	335	BIT 3,D
01EE	CB5B	336	BIT 3,E
01F0	CB5C	337	BIT 3,H
01F2	CB5D	338	BIT 3,L
01F4	CB5E	339	BIT 3,(HL)
01F6	CB5F	340	BIT 3,A
01F8	CB60	341	BIT 4,B
01FA	CB61	342	BIT 4,C
01FC	CB62	343	BIT 4,D
01FE	CB63	344	BIT 4,E
0200	CB64	345	BIT 4,H
0202	CB65	346	BIT 4,L
0204	CB66	347	BIT 4,(HL)
0206	CB67	348	BIT 4,A
0208	CB68	349	BIT 5,B
020A	CB69	350	BIT 5,C
020C	CB6A	351	BIT 5,D
020E	CB6B	352	BIT 5,E
0210	CB6C	353	BIT 5,H
0212	CB6D	354	BIT 5,L
0214	CB6E	355	BIT 5,(HL)
0216	CB6F	356	BIT 5,A
0218	CB70	357	BIT 6,B
021A	CB71	358	BIT 6,C
021C	CB72	359	BIT 6,D
021E	CB73	360	BIT 6,E

LOC	OBJ CODE	STMT	SOURCE STATEMENT
0220	CB74	361	BIT 6,H
0222	CB75	362	BIT 6,L
0224	CB76	363	BIT 6,(HL)
0226	CB77	364	BIT 6,A
0228	CB78	365	BIT 7,B
022A	CB79	366	BIT 7,C
022C	CB7A	367	BIT 7,D
022E	CB7B	368	BIT 7,E
0230	CB7C	369	BIT 7,H
0232	CB7D	370	BIT 7,L
0234	CB7E	371	BIT 7,(HL)
0236	CB7F	372	BIT 7,A
0238	CB80	373	RES 0,B
023A	CB81	374	RES 0,C
023C	CB82	375	RES 0,D
023E	CB83	376	RES 0,E
0240	CB84	377	RES 0,H
0242	CB85	378	RES 0,L
0244	CB86	379	RES 0,(HL)
0246	CB87	380	RES 0,A
0248	CB88	381	RES 1,B
024A	CB89	382	RES 1,C
024C	CB8A	383	RES 1,D
024E	CB8B	384	RES 1,E
0250	CB8C	385	RES 1,H
0252	CB8D	386	RES 1,L
0254	CB8E	387	RES 1,(HL)
0256	CB8F	388	RES 1,A
0258	CB90	389	RES 2,B
025A	CB91	390	RES 2,C
025C	CB92	391	RES 2,D
025E	CB93	392	RES 2,E
0260	CB94	393	RES 2,H
0262	CB95	394	RES 2,L
0264	CB96	395	RES 2,(HL)
0266	CB97	396	RES 2,A
0268	CB98	397	RES 3,B
026A	CB99	398	RES 3,C
026C	CB9A	399	RES 3,D
026E	CB9B	400	RES 3,E
0270	CB9C	401	RES 3,H
0272	CB9D	402	RES 3,L
0274	CB9E	403	RES 3,(HL)
0276	CB9F	404	RES 3,A
0278	CBA0	405	RES 4,B
027A	CBA1	406	RES 4,C
027C	CBA2	407	RES 4,D
027E	CBA3	408	RES 4,E
0280	CBA4	409	RES 4,H
0282	CBA5	410	RES 4,L
0284	CBA6	411	RES 4,(HL)
0286	CBA7	412	RES 4,A
0288	CBA8	413	RES 5,B
028A	CBA9	414	RES 5,C
028C	CBAA	415	RES 5,D
028E	CBAB	416	RES 5,E
0290	CBAC	417	RES 5,H
0292	CBAD	418	RES 5,L
0294	CBAE	419	RES 5,(HL)
0296	CBAF	420	RES 5,A
0298	CBB0	421	RES 6,B
029A	CBB1	422	RES 6,C
029C	CBB2	423	RES 6,D
029E	CBB3	424	RES 6,E
02A0	CBB4	425	RES 6,H
02A2	CBB5	426	RES 6,L
02A4	CBB6	427	RES 6,(HL)
02A6	CBB7	428	RES 6,A
02A8	CBB8	429	RES 7,B
02AA	CBB9	430	RES 7,C
02AC	CBBA	431	RES 7,D
02AE	CBBB	432	RES 7,E
0280	CBBC	433	RES 7,H
0282	CBBD	434	RES 7,L
0284	CBBE	435	RES 7,(HL)
0286	CBBF	436	RES 7,A
0288	CBC0	437	SET 0,B
02BA	CBC1	438	SET 0,C
02BC	CBC2	439	SET 0,D
02BE	CBC3	440	SET 0,E
02C0	CBC4	441	SET 0,H
02C2	CBC5	442	SET 0,L
02C4	CBC6	443	SET 0,(HL)
02C6	CBC7	444	SET 0,A
02C8	CBC8	445	SET 1,B
02CA	CBC9	446	SET 1,C
02CC	CBCA	447	SET 1,D
02CE	CBCB	448	SET 1,E
02D0	CBCC	449	SET 1,H
02D2	CBCD	450	SET 1,L
02D4	CBCE	451	SET 1,(HL)
02D6	CBCF	452	SET 1,A
02D8	CBD0	453	SET 2,B
02DA	CBD1	454	SET 2,C
02DC	CBD2	455	SET 2,D
02DE	CBD3	456	SET 2,E
02E0	CBD4	457	SET 2,H
02E2	CBD5	458	SET 2,L
02E4	CBD6	459	SET 2,(HL)
02E6	CBD7	460	SET 2,A
02E8	CBD8	461	SET 3,B
02EA	CBD9	462	SET 3,C
02EC	CBDA	463	SET 3,D
02EE	CBDB	464	SET 3,E
02F0	CBDC	465	SET 3,H
02F2	CBDD	466	SET 3,L
02F4	CBDE	467	SET 3,(HL)
02F6	CBDF	468	SET 3,A
02F8	CBE0	469	SET 4,B
02FA	CBE1	470	SET 4,C
02FC	CBE2	471	SET 4,D
02FE	CBE3	472	SET 4,E
0300	CBE4	473	SET 4,H
0302	CBE5	474	SET 4,L
0304	CBE6	475	SET 4,(HL)
0306	CBE7	476	SET 4,A
0308	CBE8	477	SET 5,B
030A	CBE9	478	SET 5,C
030C	CBEA	479	SET 5,D
030E	CBEB	480	SET 5,E
0310	CBEC	481	SET 5,H
0312	CBED	482	SET 5,L
0314	CBEE	483	SET 5,(HL)
0316	CBEF	484	SET 5,A
0318	CBF0	485	SET 6,B
031A	CBF1	486	SET 6,C

LOC	OBJ CODE	STMT	SOURCE STATEMENT
031C	CBF2	487	SET 6,D
031E	CBF3	488	SET 6,E
0320	CBF4	489	SET 6,H
0322	CBF5	490	SET 6,L
0324	CBF6	491	SET 6,(HL)
0326	CBF7	492	SET 6,A
0328	CBF8	493	SET 7,B
032A	CBF9	494	SET 7,C
032C	CBFA	495	SET 7,D
032E	CBFB	496	SET 7,E
0330	CBFC	497	SET 7,H
0332	CBFD	498	SET 7,L
0334	CBFE	499	SET 7,(HL)
0336	CBFF	500	SET 7,A
0338	DD09	501	ADD IX,BC
033A	DD19	502	ADD IX,DE
033C	DD218405	503	LD IX,NN
0340	DD228405	504	LD (NN),IX
0344	DD23	505	INC IX
0346	DD29	506	ADD IX,IX
0348	DD2A8405	507	LD IX,(NN)
034C	DD2B	508	DEC IX
034E	DD3405	509	INC (IX + IND)
0351	DD3505	510	DEC (IX + IND)
0354	DD360520	511	LD (IX + IND),N
0358	DD39	512	ADD IX,SP
035A	DD4605	513	LD B,(IX + IND)
035D	DD4E05	514	LD C,(IX + IND)
0360	DD5605	515	LD D,(IX + IND)
0363	DD5E05	516	LD E,(IX + IND)
0366	DD6605	517	LD H,(IX + IND)
0369	DD6E05	518	LD L,(IX + IND)
036C	DD7005	519	LD (IX + IND),B
036F	DD7105	520	LD (IX + IND),C
0372	DD7205	521	LD (IX + IND),D
0375	DD7305	522	LD (IX + IND),E
0378	DD7405	523	LD (IX + IND),H
037B	DD7505	524	LD (IX + IND),L
037E	DD7705	525	LD (IX + IND),A
0381	DD7E05	526	LD A,(IX + IND)
0384	DD8605	527	ADD A,(IX + IND)
0387	DD8E05	528	ADC A,(IX + IND)
038A	DD9605	529	SUB (IX + IND)
038D	DD9E05	530	SBC A,(IX + IND)
0390	DDA605	531	AND (IX + IND)
0393	DDAE05	532	XOR (IX + IND)
0396	DDB605	533	OR (IX + IND)
0399	DDBE05	534	CP (IX + IND)
039C	DDE1	535	POP IX
039E	DDE3	536	EX (SP),IX
03A0	DDE5	537	PUSH IX
03A2	DDE9	538	JP (IX)
03A4	DDF9	539	LD SP,IX
03A6	DDCB0506	540	RLC (IX + IND)
03AA	DDCB050E	541	RRC (IX + IND)
03AE	DDCB0516	542	RL (IX + IND)
03B2	DDCB051E	543	RR (IX + IND)
03B6	DDCB0526	544	SLA (IX + IND)
03BA	DDCB052E	545	SRA (IX + IND)
03BE	DDCB053E	546	SRL (IX + IND)
03C2	DDCB0546	547	BIT 0,(IX + IND)
03C6	DDCB054E	548	BIT 1,(IX + IND)
03CA	DDCB0556	549	BIT 2,(IX + IND)
03CE	DDCB055E	550	BIT 3,(IX + IND)
03D2	DDCB0566	551	BIT 4,(IX + IND)
03D6	DDCB056E	552	BIT 5,(IX + IND)
03DA	DDCB0576	553	BIT 6,(IX + IND)
03DE	DDCB057E	554	BIT 7,(IX + IND)
03E2	DDCB0586	555	RES 0,(IX + IND)
03E6	DDCB058E	556	RES 1,(IX + IND)
03EA	DDCB0596	557	RES 2,(IX + IND)
03EE	DDCB059E	558	RES 3,(IX + IND)
03F2	DDCB05A6	559	RES 4,(IX + IND)
03F6	DDCB05AE	560	RES 5,(IX + IND)
03FA	DDCB05B6	561	RES 6,(IX + IND)
03FE	DDCB05BE	562	RES 7,(IX + IND)
0402	DDCB05C6	563	SET 0,(IX + IND)
0406	DDCB05CE	564	SET 1,(IX + IND)
040A	DDCB05D6	565	SET 2,(IX + IND)
040E	DDCB05DE	566	SET 3,(IX + IND)
0412	DDCB05E6	567	SET 4,(IX + IND)
0416	DDCB05EE	568	SET 5,(IX + IND)
041A	DDCB05F6	569	SET 6,(IX + IND)
041E	DDCB05FE	570	SET 7,(IX + IND)
0422	ED40	571	IN B,(C)
0424	ED41	572	OUT (C),B
0426	ED42	573	SBC HL,BC
0428	ED438405	574	LD (NN),BC
042C	ED44	575	NEG
042E	ED45	576	RETN
0430	ED46	577	IM 0
0432	ED47	578	LD I,A
0434	ED48	579	IN C,(C)
0436	ED49	580	OUT (C),C
0438	ED4A	581	ADC HL,BC
043A	ED4B8405	582	LD BC,(NN)
043E	ED4D	583	RETI
	ED4F		LD R,A
	ED5F		LD A,R
0440	ED50	584	IN D,(C)
0442	ED51	585	OUT (C),D
0444	ED52	586	SBC HL,DE
0446	ED538405	587	LD (NN),DE
044A	ED56	588	IM I
044C	ED57	589	LD A,I
044E	ED58	590	IN E,(C)
0450	ED59	591	OUT (C),E
0452	ED5A	592	ADC HL,DE
0454	ED5B8405	593	LD DE,(NN)
045A	ED60	595	IN H,(C)
045C	ED61	596	OUT (C),H
045E	ED62	597	SBC HL,HL
0460	ED67	598	RRD
0462	ED68	599	IN L,(C)
0464	ED69	600	OUT (C),L
0466	ED6A	601	ADC HL,HL
0468	ED6F	602	RLD
046A	ED72	603	SBC HL,SP
046C	ED738405	604	LD (NN),SP
0470	ED78	605	IN A,(C)
0472	ED79	606	OUT (C),A
0474	ED7A	607	ADC HL,SP
0476	ED7B8405	608	LD SP,(NN)
047A	EDA0	609	LDI
047C	EDA1	610	CPI
047E	EDA2	611	INI

LOC	OBJ CODE	STMT	SOURCE STATEMENT
0480	EDA3	612	OUTI
0482	EDA8	613	LDD
0484	EDA9	614	CPD
0486	EDAA	615	IND
0488	EDAB	616	OUTD
048A	EDB0	617	LDIR
048C	EDB1	618	CPIR
048E	EDB2	619	INIR
0490	EDB3	620	OTIR
0492	EDB8	621	LDDR
0494	EDB9	622	CPDR
0496	EDBA	623	INDR
0498	EDBB	624	OTDR
049A	FD09	625	ADD IY,BC
049C	FD19	626	ADD IY,DE
049E	FD218405	627	LD IY,NN
04A2	FD228405	628	LD (NN),IY
04A6	FD23	629	INC IY
04A8	FD29	630	ADD IY,IY
04AA	FD2A8405	631	LD IY,(NN)
04AE	FD2B	632	DEC IY
04B0	FD3405	633	INC (IY + IND)
04B3	FD3505	634	DEC (IY + IND)
04B6	FD360520	635	LD (IY + IND),N
04BA	FD39	636	ADD IY,SP
04BC	FD4605	637	LD B,(IY + IND)
04BF	FD4E05	638	LD C,(IY + IND)
04C2	FD5605	639	LD D,(IY + IND)
04C5	FD5E05	640	LD E,(IY + IND)
04C8	FD6605	641	LD H,(IY + IND)
04CB	FD6E05	642	LD L,(IY + IND)
04CE	FD7005	643	LD (IY + IND),B
04D1	FD7105	644	LD (IY + IND),C
04D4	FD7205	645	LD(IY + IND),D
04D7	FD7305	646	LD (IY + IND),E
04DA	FD7405	647	LD (IY + IND),H
04DD	FD7505	648	LD (IY + IND),L
04E0	FD7705	649	LD (IY + IND),A
04E3	FD7E05	650	LD A,(IY + IND)
04E6	FD8605	651	ADD A,(IY + IND)
04E9	FD8E05	652	ADC A,(IY + IND)
04EC	FD9605	653	SUB (IY + IND)
04EF	FD9E05	654	SBC A,(IY + IND)
04F2	FDA605	655	AND (IY + IND)
04F5	FDAE05	656	XOR (IY + IND)
04F8	FDB605	657	OR (IY + IND)
04FB	FDBE05	658	CP (IY + IND)
04FE	FDE1	659	POP IY
0500	FDE3	660	EX (SP),IY
0502	FDE5	661	PUSH IY
0504	FDE9	662	JP (IY)
0506	FDF9	663	LD SP,IY
0508	FDCB0506	664	RLC (IY + IND)
050C	FDCB050E	665	RRC (IY + IND)
0510	FDCB0516	666	RL (IY + IND)
0514	FDCB051E	667	RR (IY + IND)
0518	FDCB0526	668	SLA (IY + IND)
051C	FDCB052E	669	SRA (IY + IND)
0520	FDCB053E	670	SRL (IY + IND)
0524	FDCB0546	671	BIT 0,(IY + IND)
0528	FDCB054E	672	BIT 1,(IY + IND)
052C	FDCB0556	673	BIT 2,(IY + IND)
0530	FDCB055E	674	BIT 3,(IY + IND)
0534	FDCB0566	675	BIT 4,(IY + IND)
0538	FDCB056E	676	BIT 5,(IY + IND)
053C	FDCB0576	677	BIT 6,(IY + IND)
0540	FDCB057E	678	BIT 7,(IY + IND)
0544	FDCB0586	679	RES 0,(IY + IND)
0548	FDCB058E	680	RES 1,(IY + IND)
054C	FDCB0596	681	RES 2,(IY + IND)
0550	FDCB059E	682	RES 3,(IY + IND)
0554	FDCB05A6	683	RES 4,(IY + IND)
0558	FDCB05AE	684	RES 5,(IY + IND)
055C	FDCB05B6	685	RES 6,(IY + IND)
0560	FDCB05BE	686	RES 7,(IY + IND)
0564	FDCB05C6	687	SET 0,(IY + IND)
0568	FDCB05CE	688	SET 1,(IY + IND)
056C	FDCB05D6	689	SET 2,(IY + IND)
0570	FDCB05DE	690	SET 3,(IY + IND)
0574	FDCB05E6	691	SET 4,(IY + IND)
0578	FDCB05EE	692	SET 5,(IY + IND)
057C	FDCB05F6	693	SET 6,(IY + IND)
0580	FDCB05FE	694	SET 7,(IY + IND)
0584		695 NN	DEFS 2
		696 IND	EQU 5
		697 M	EQU 10H
		698 N	EQU 20H
		699 DIS	EQU 30H
		700	END

Appendix D / Alphabetic List of Instruction Set

Following is an alphabetical listing of the nmemonic or source statement in column four. The object code is shown in column two.

LOC	OBJ CODE	STMT	SOURCE STATEMENT		LOC	OBJ CODE	STMT	SOURCE STATEMENT	
0000	8E	1	ADC	A,(HL)	005E	CB43	57	BIT	0,E
0001	DD8E05	2	ADC	A,(IX + IND)	0060	CB44	58	BIT	0,H
0004	FD8E05	3	ADC	A,(IY + IND)	0062	CB45	59	BIT	0,L
0007	8F	4	ADC	A,A	0064	CB4E	60	BIT	1,(HL)
0008	88	5	ADC	A,B	0066	DDCB054E	61	BIT	1,(IX + IND)
0009	89	6	ADC	A,C	006A	FDCB054E	62	BIT	1,(IY + IND)
000A	8A	7	ADC	A,D	006E	CB4F	63	BIT	1,A
000B	8B	8	ADC	A,E	0070	CB48	64	BIT	1,B
000C	8C	9	ADC	A,H	0072	CB49	65	BIT	1,C
000D	8D	10	ADC	A,L	0074	CB4A	66	BIT	1.D
000E	CE20	11	ADC	A,N	0076	CB4B	67	BIT	1,E
0010	ED4A	12	ADC	HL,BC	0078	CB4C	68	BIT	1,H
0012	ED5A	13	ADC	HL,DE	007A	CB4D	69	BIT	1,L
0014	ED6A	14	ADC	HL,HL	007C	CB56	70	BIT	2,(HL)
0016	ED7A	15	ADC	HL,SP	007E	DDCB0556	71	BIT	2,(IX + IND)
0018	86	16	ADD	A,(HL)	0082	FDCB0556	72	BIT	2,(IY + IND)
0019	DD8605	17	ADD	A,(IX + IND)	0086	CB57	73	BIT	2,A
001C	FD8605	18	ADD	A,(IY + IND)	0088	CB50	74	BIT	2,B
001F	87	19	ADD	A,A	008A	CB51	75	BIT	2,C
0020	80	20	ADD	A,B	008C	CB52	76	BIT	2,D
0021	81	21	ADD	A,C	008E	CB53	77	BIT	2,E
0022	82	22	ADD	A,D	0090	CB54	78	BIT	2,H
0023	83	23	ADD	A,E	0092	CB55	79	BIT	2,L
0024	84	24	ADD	A,H	0094	CB5E	80	BIT	3,(HL)
0025	85	25	ADD	A,L	0096	DDCB055E	81	BIT	3,(IX + IND)
0026	C620	26	ADD	A,N	009A	FDCB055E	82	BIT	3,(IY + IND)
0028	09	27	ADD	HL,BC	009E	CB5F	83	BIT	3,A
0029	19	28	ADD	HL,DE	00A0	CB58	84	BIT	3,B
002A	29	29	ADD	HL,HL	00A2	CB59	85	BIT	3,C
002B	39	30	ADD	HL,SP	00A4	CB5A	86	BIT	3,D
002C	DD09	31	ADD	IX,BC	00A6	CB5B	87	BIT	3,E
002E	DD19	32	ADD	IX,DE	00A8	CB5C	88	BIT	3,H
0030	DD29	33	ADD	IX,IX	00AA	CB5D	89	BIT	3,L
0032	DD39	34	ADD	IX,SP	00AC	CB66	90	BIT	4,(HL)
0034	FD09	35	ADD	IY,BC	00AE	DDCB0566	91	BIT	4,(IX + IND)
0036	FD19	36	ADD	IY,DE	00B2	FDCB0566	92	BIT	4,(IY + IND)
0038	FD29	37	ADD	IY,IY	00B6	CB67	93	BIT	4,A
003A	FD39	38	ADD	IY,SP	00B8	CB60	94	BIT	4,B
003C	A6	39	AND	(HL)	00BA	CB61	95	BIT	4,C
003D	DDA605	40	AND	(IX + IND)	00BC	CB62	96	BIT	4,D
0040	FDA605	41	AND	(IY + IND)	00BE	CB63	97	BIT	4,E
0043	A7	42	AND	A	00C0	CB64	98	BIT	4,H
0044	A0	43	AND	B	00C2	CB65	99	BIT	4,L
0045	A1	44	AND	C	00C4	CB6E	100	BIT	5,(HL)
0046	A2	45	AND	D	00C6	DDCB056E	101	BIT	5,(IX + IND)
0047	A3	46	AND	E	00CA	FDCB056E	102	BIT	5,(IY + IND)
0048	A4	47	AND	H	00CE	CB6F	103	BIT	5,A
0049	A5	48	AND	L	00D0	CB68	104	BIT	5,B
004A	E620	49	AND	N	00D2	CB69	105	BIT	5,C
004C	CB46	50	BIT	0,(HL)	00D4	CB6A	106	BIT	5,D
004E	DDCB0546	51	BIT	0,(IX + IND)	00D6	CB6B	107	BIT	5,E
0052	FDBC0546	52	BIT	0,(IY + IND)	00D8	CB6C	108	BIT	5,H
0056	CB47	53	BIT	0,A	00DA	CB6D	109	BIT	5,L
0058	CB40	54	BIT	0,B	00DC	CB76	110	BIT	6,(HL)
005A	CB41	55	BIT	0,C	00DE	DDCB0576	111	BIT	6,(IX + IND)
005C	CB42	56	BIT	0,D	00E2	FDCB0576	112	BIT	6,(IY + IND)

LOC	OBJ CODE	STMT	SOURCE STATEMENT		LOC	OBJ CODE	STMT	SOURCE STATEMENT	
00E6	CB77	113	BIT	6,A	015C	E3	176	EX	(SP),HL
00E8	CB70	114	BIT	6,B	015D	DDE3	177	EX	(SP),IX
00EA	CB71	115	BIT	6,C	015F	FDE3	178	EX	(SP),IY
00EC	CB72	116	BIT	6,D	0161	08	179	EX	AF,AF'
00EE	CB73	117	BIT	6,E	0162	EB	180	EX	DE,HL
00F0	CB74	118	BIT	6,H	0163	D9	181	EXX	
00F2	CB75	119	BIT	6,L	0164	76	182	HALT	
00F4	CB7E	120	BIT	7,(HL)	0165	ED46	183	IM	0
00F6	DDCB057E	121	BIT	7,(IX + IND)	0167	ED56	184	IM	1
00FA	FDCB057E	122	BIT	7,(IY + IND)	0169	ED5E	185	IM	2
00FE	CB7F	123	BIT	7,A	016B	ED78	186	IN	A,(C)
0100	CB78	124	BIT	7,B	016D	DB20	187	IN	A,(N)
0102	CB79	125	BIT	7,C	016F	ED40	188	IN	B,(C)
0104	CB7A	126	BIT	7,D	0171	ED48	189	IN	C,(C)
0106	CB7B	127	BIT	7,E	0173	ED50	190	IN	D,(C)
0108	CB7C	128	BIT	7,H	0175	ED58	191	IN	E,(C)
010A	CB7D	129	BIT	7,L	0177	ED60	192	IN	H,(C)
010C	DC8405	130	CALL	C,NN	0179	ED68	193	IN	L,(C)
010F	FC8405	131	CALL	M,NN	017B	34	194	INC	(HL)
0112	D48405	132	CALL	NC,NN	017C	DD3405	195	INC	(IX + IND)
0115	CD8405	133	CALL	NN	017F	FD3405	196	INC	(IY + IND)
0118	C48405	134	CALL	NZ,NN	0182	3C	197	INC	A
011B	F48405	135	CALL	P,NN	0183	04	198	INC	B
011E	EC8405	136	CALL	PE,NN	0184	03	199	INC	BC
0121	E48405	137	CALL	PO,NN	0185	0C	200	INC	C
0124	CC8405	138	CALL	Z,NN	0186	14	201	INC	D
0127	3F	139	CCF		0187	13	202	INC	DE
0128	BE	140	CP	(HL)	0188	1C	203	INC	E
0129	DDBE05	141	CP	(IX + IND)	0189	24	204	INC	H
012C	FDBE05	142	CP	(IY + IND)	018A	23	205	INC	HL
012F	BF	143	CP	A	018B	DD23	206	INC	IX
0130	B8	144	CP	B	018D	FD23	207	INC	IY
0131	B9	145	CP	C	018F	2C	208	INC	L
0132	BA	146	CP	D	0190	33	209	INC	SP
0133	BB	147	CP	E	0191	EDAA	210	IND	
0134	BC	148	CP	H	0193	EDBA	211	INDR	
0135	BD	149	CP	L	0195	EDA2	212	INI	
0136	FE20	150	CP	N	0197	EDB2	213	INIR	
0138	EDA9	151	CPD		0199	E9	214	JP	(HL)
013A	EDB9	152	CPDR		019A	DDE9	215	JP	(IX)
013C	EDA1	153	CPI		019C	FDE9	216	JP	(IY)
013E	EDB1	154	CPIR		019E	DA8405	217	JP	C,NN
0140	2F	155	CPL		01A1	FA8405	218	JP	M,NN
0141	27	156	DAA		01A4	D28405	219	JP	NC,NN
0142	35	157	DEC	(HL)	01A7	C38405	220	JP	NN
0143	DD3505	158	DEC	(IX + IND)	01AA	C28405	221	JP	NZ,NN
0146	FD3505	159	DEC	(IY + IND)	01AD	F28405	222	JP	P,NN
0149	3D	160	DEC	A	01B0	EA8405	223	JP	PE,NN
014A	05	161	DEC	B	01B3	E28405	224	JP	PO,NN
014B	0B	162	DEC	BC	01B6	CA8405	225	JP	Z,NN
014C	0D	163	DEC	C	01B9	382E	226	JR	C,DIS
014D	15	164	DEC	D	01BB	182E	227	JR	DIS
014E	1B	165	DEC	DE	01BD	302E	228	JR	NC,DIS
014F	1D	166	DEC	E	01BF	202E	229	JR	NZ,DIS
0150	25	167	DEC	H	01C1	282E	230	JR	Z,DIS
0151	2B	168	DEC	HL	01C3	02	231	LD	(BC),A
0152	DD2B	169	DEC	IX	01C4	12	232	LD	(DE),A
0154	FD2B	170	DEC	IY	01C5	77	233	LD	(HL),A
0156	2D	171	DEC	L	01C6	70	234	LD	(HL),B
0157	3B	172	DEC	SP	01C7	71	235	LD	(HL),C
0158	F3	173	DI		01C8	72	236	LD	(HL),D
0159	102E	174	DJNZ	DIS	01C9	73	237	LD	(HL),E
015B	FB	175	EI		01CA	74	238	LD	(HL),H

LOC	OBJ CODE	STMT	SOURCE STATEMENT		LOC	OBJ CODE	STMT	SOURCE STATEMENT	
01CB	75	239	LD	(HL),L	0255	4D	301	LD	C,L
01CC	3620	240	LD	(HL),N	0256	0E20	302	LD	C,N
01CE	DD7705	241	LD	(IX + IND),A	0258	56	303	LD	D,(HL)
01D1	DD7005	242	LD	(IX + IND),B	0259	DD5605	304	LD	D,(IX + IND)
01D4	DD7105	243	LD	(IX + IND),C	025C	FD5605	305	LD	D,(IY + IND)
01D7	DD7205	244	LD	(IX + IND),D	025F	57	306	LD	D,A
01DA	DD7305	245	LD	(IX + IND),E	0260	50	307	LD	D,B
01DD	DD7405	246	LD	(IX + IND),H	0261	51	308	LD	D,C
01E0	DD7505	247	LD	(IX + IND),L	0262	52	309	LD	D,D
01E3	DD360520	248	LD	(IX + IND),N	0263	53	310	LD	D,E
01E7	FD7705	249	LD	(IY + IND),A	0264	54	311	LD	D,H
01EA	FD7005	250	LD	(IY + IND),B	0265	55	312	LD	D,L
01ED	FD7105	251	LD	(IY + IND),C	0266	1620	313	LD	D,N
01F0	FD7205	252	LD	(IY + IND),D	0268	ED5B8405	314	LD	DE,(NN)
01F3	FD7305	253	LD	(IY + IND),E	026C	118405	315	LD	DE,NN
01F6	FD7405	254	LD	(IY + IND),H	026F	5E	316	LD	E,(HL)
01F9	FD7505	255	LD	(IY + IND),L	0270	DD5E05	317	LD	E,(IX + IND)
01FC	FD360520	256	LD	(IY + IND),N	0273	FD5E05	318	LD	E,(IY + IND)
0200	328405	257	LD	(NN),A	0276	5F	319	LD	E,A
0203	ED438405	258	LD	(NN),BC	0277	58	320	LD	E,B
0207	ED538405	259	LD	(NN),DE	0278	59	321	LD	E,C
020B	228405	260	LD	(NN),HL	0279	5A	322	LD	E,D
020E	DD228405	261	LD	(NN),IX	027A	5B	323	LD	E,E
0202	FD228405	262	LD	(NN),IY	027B	5C	324	LD	E,H
0216	ED738405	263	LD	(NN),SP	027C	5D	325	LD	E,L
021A	0A	264	LD	A,(BC)	027D	1E20	326	LD	E,N
021B	1A	265	LD	A,(DE)	027F	66	327	LD	H,(HL)
021C	7E	266	LD	A,(HL)	0280	DD6605	328	LD	H,(IX + IND)
021D	DD7E05	267	LD	A,(IX + IND)	0283	FD6605	329	LD	H,(IY + IND)
0220	FD7E05	268	LD	A,(IY + IND)	0286	67	330	LD	H,A
0223	3A8405	269	LD	A,(NN)	0287	60	331	LD	H,B
0226	7F	270	LD	A,A	0288	61	332	LD	H,C
0227	78	271	LD	A,B	0289	62	333	LD	H,D
0228	79	272	LD	A,C	028A	63	334	LD	H,E
0229	7A	273	LD	A,D	028B	64	335	LD	H,H
022A	7B	274	LD	A,E	028C	65	336	LD	H,L
022B	7C	275	LD	A,H	028D	2620	337	LD	H,N
022C	ED57	276	LD	A,I	028F	2A8405	338	LD	HL,(NN)
022E	7D	277	LD	A,L	0292	218405	339	LD	HL,NN
022F	3E20	278	LD	A,N	0295	ED47	340	LD	I,A
	ED5F	278.1	LD	A,R	0297	DD2A8405	341	LD	IX,(NN)
0231	46	279	LD	B,(HL)	029B	DD218405	342	LD	IX,NN
0232	DD4605	280	LD	B,(IX + IND)	029F	FD2A8405	343	LD	IY,(NN)
0235	FD4605	281	LD	B,(IY + IND)	02A3	FD218405	344	LD	IY,NN
0238	47	282	LD	B,A	02A7	6E	345	LD	L,(HL)
0239	40	283	LD	B,B	02A8	DD6E05	346	LD	L,(IX + IND)
023A	41	284	LD	B,C	02AB	FD6E05	347	LD	L,(IY + IND)
023B	42	285	LD	B,D	02AE	6F	348	LD	L,A
023C	43	286	LD	B,E	02AF	68	349	LD	L,B
023D	44	287	LD	B,H	02B0	69	350	LD	L,C
023E	45	288	LD	B,L	02B1	6A	351	LD	L,D
023F	0620	289	LD	B,N	02B2	6B	352	LD	L,E
0241	ED4B8405	290	LD	BC,(NN)	02B3	6C	353	LD	L,H
0245	018405	291	LD	BC,NN	02B4	6D	354	LD	L,L
0248	4E	292	LD	C,(HL)	02B5	2E20	355	LD	L,N
0249	DD4E05	293	LD	C,(IX + IND)		ED4F		LD	R,A
024C	FD4E05	294	LD	C,(IY + IND)	02B7	ED7B8405	356	LD	SP,(NN)
024F	4F	295	LD	C,A	02BB	F9	357	LD	SP,HL
0250	48	296	LD	C,B	02BC	DDF9	358	LD	SP,IX
0251	49	297	LD	C,C	02BE	FDF9	359	LD	SP,IY
0252	4A	298	LD	C,D	02C0	318405	360	LD	SP,NN
0253	4B	299	LD	C,E	02C3	EDA8	361	LDD	
0254	4C	300	LD	C,H	02C5	EDB8	362	LDDR	

LOC	OBJ CODE	STMT	SOURCE STATEMENT		LOC	OBJ CODE	STMT	SOURCE STATEMENT	
02C7	EDA0	363	LDI		0342	CB90	426	RES	2,B
02C9	EDB0	364	LDIR		0344	CB91	427	RES	2,C
02CB	ED44	365	NEG		0346	CB92	428	RES	2,D
02CD	00	366	NOP		0348	CB93	429	RES	2,E
02CE	B6	367	OR	(HL)	034A	CB94	430	RES	2,H
02CF	DDB605	368	OR	(IX + IND)	034C	CB95	431	RES	2,L
02D2	FDB605	369	OR	(IY + IND)	034E	CB9E	432	RES	3,(HL)
02D5	B7	370	OR	A	0350	DDCB059E	433	RES	3,(IX + IND)
02D6	B0	371	OR	B	0354	FDCB059E	434	RES	3,(IY + IND)
02D7	B1	372	OR	C	0358	CB9F	435	RES	3,A
02D8	B2	373	OR	D	035A	CB98	436	RES	3,B
02D9	B3	374	OR	E	035C	CB99	437	RES	3,C
02DA	B4	375	OR	H	035E	CB9A	438	RES	3,D
02DB	B5	376	OR	L	0360	CB9B	439	RES	3,E
02DC	F620	377	OR	N	0362	CB9C	440	RES	3,H
02DE	ED8B	378	OTDR		0364	CB9D	441	RES	3,L
02E0	EDB3	379	OTIR		0366	CBA6	442	RES	4,(HL)
02E2	ED79	380	OUT	(C),A	0368	DDCB05A6	443	RES	4,(IX + IND)
02E4	ED41	381	OUT	(C),B	036C	FDCB05A6	444	RES	4,(IY + IND)
02E6	ED49	382	OUT	(C),C	0370	CBA7	445	RES	4,A
02E8	ED51	383	OUT	(C),D	0372	CBA0	446	RES	4,B
02EA	ED59	384	OUT	(C),E	0374	CBA1	447	RES	4,C
02EC	ED61	385	OUT	(C),H	0376	CBA2	448	RES	4,D
02EE	ED69	386	OUT	(C),L	0378	CBA3	449	RES	4,E
02F0	D320	387	OUT	N,A	037A	CBA4	450	RES	4,H
02F2	EDAB	388	OUTD		037C	CBA5	451	RES	4,L
02F4	EDA3	389	OUTI		037E	CBAE	452	RES	5,(HL)
02F6	F1	390	POP	AF	0380	DDCB05AE	453	RES	5,(IX + IND)
02F7	C1	391	POP	BC	0384	FDCB05AE	454	RES	5,(IY + IND)
02F8	D1	392	POP	DE	0388	CBAF	455	RES	5,A
02F9	E1	393	POP	HL	038A	CBA8	456	RES	5,B
02FA	DDE1	394	POP	IX	038C	CBA9	457	RES	5,C
02FC	FDE1	395	POP	IY	038E	CBAA	458	RES	5,D
02FE	F5	396	PUSH	AF	0390	CBAB	459	RES	5,E
02FF	C5	397	PUSH	BC	0392	CBAC	460	RES	5,H
0300	D5	398	PUSH	DE	0394	CBAD	461	RES	5,L
0301	E5	399	PUSH	HL	0396	CBB6	462	RES	6,(HL)
0302	DDE5	400	PUSH	IX	0398	DDCB05B6	463	RES	6,(IX + IND)
0304	FDE5	401	PUSH	IY	039C	FDCB05B6	464	RES	6,(IY + IND)
0306	CB86	402	RES	0,(HL)	03A0	CBB7	465	RES	6,A
0308	DDCB0586	403	RES	0,(IX + IND)	03A2	CBB0	466	RES	6,B
030C	FDCB0586	404	RES	0,(IY + IND)	03A4	CBB1	467	RES	6,C
0310	CB87	405	RES	0,A	03A6	CBB2	468	RES	6,D
0312	CB80	406	RES	0,B	03A8	CBB3	469	RES	6,E
0314	CB81	407	RES	0,C	03AA	CBB4	470	RES	6,H
0316	CB82	408	RES	0,D	03AC	CBB5	471	RES	6,L
0318	CB83	409	RES	0,E	03AE	CBBE	472	RES	7,(HL)
031A	CB84	410	RES	0,H	03B0	DDCB05BE	473	RES	7,(IX + IND)
031C	CB85	411	RES	0,L	03B4	FDCB05BE	474	RES	7,(IY + IND)
031E	CB8E	412	RES	1,(HL)	03B8	CBBF	475	RES	7,A
0320	DDCB058E	413	RES	1,(IX + IND)	03BA	CBB8	476	RES	7,B
0324	FDCB058E	414	RES	1,(IY + IND)	03BC	CBB9	477	RES	7,C
0328	CB8F	415	RES	1,A	03BE	CBBA	478	RES	7,D
032A	CB88	416	RES	1,B	03C0	CBBB	479	RES	7,E
032C	CB89	417	RES	1,C	03C2	CBBC	480	RES	7,H
032E	CB8A	418	RES	1,D	03C4	CBBD	481	RES	7,L
0330	CB8B	419	RES	1,E	03C6	C9	482	RET	
0332	CB8C	420	RES	1,H	03C7	D8	483	RET	C
0334	CB8D	421	RES	1,L	03C8	F8	484	RET	M
0336	CB96	422	RES	2,(HL)	03C9	D0	485	RET	NC
0338	DDCB0596	423	RES	2,(IX + IND)	03CA	C0	486	RET	NZ
033C	FDCB0596	424	RES	2,(IY + IND)	03CB	F0	487	RET	P
0340	CB97	425	RES	2,A	03CC	E8	488	RET	PE

LOC	OBJ CODE	STMT	SOURCE STATEMENT		LOC	OBJ CODE	STMT	SOURCE STATEMENT	
03CD	E0	489	RET	PO	044C	99	552	SBC	A,C
03CE	C8	490	RET	Z	044D	9A	553	SBC	A,D
03CF	ED4D	491	RETI		044E	9B	554	SBC	A,E
03D1	ED45	492	RETN		044F	9C	555	SBC	A,H
03D3	CB16	493	RL	(HL)	0450	9D	556	SBC	A,L
03D5	DDCB0516	494	RL	(IX + IND)	0451	DE20	557	SBC	A,N
03D9	FDCB0516	495	RL	(IY + IND)	0453	ED42	558	SBC	HL,BC
03DD	CB17	496	RL	A	0455	ED52	559	SBC	HL,DE
03DF	CB10	497	RL	B	0457	ED62	560	SBC	HL,HL
03E1	CB11	498	RL	C	0459	ED72	561	SBC	HL,SP
03E3	CB12	499	RL	D	045B	37	562	SCF	
03E5	C813	500	RL	E	045C	CBC6	563	SET	0,(HL)
03E7	CB14	501	RL	H	045E	DDCB05C6	564	SET	0,(IX + IND)
03E9	CB15	502	RL	L	0462	FDCB05C6	565	SET	0,(IY + IND)
03EB	17	503	RLA		0466	CBC7	566	SET	0,A
03EC	CB06	504	RLC	(HL)	0468	CBC0	567	SET	0,B
03EE	DDCB0506	505	RLC	(IX + IND)	046A	CBC1	568	SET	0,C
03F2	FDCB0506	506	RLC	(IY + IND)	046C	CBC2	569	SET	0,D
03F6	CB07	507	RLC	A	046E	CBC3	570	SET	0,E
03F8	CB00	508	RLC	B	0470	CBC4	571	SET	0,H
03FA	CB01	509	RLC	C	0472	CBC5	572	SET	0,L
03FC	CB02	510	RLC	D	0474	CBCE	573	SET	1,(HL)
03FE	CB03	511	RLC	E	0476	DDCB05CE	574	SET	1,(IX + IND)
0400	CB04	512	RLC	H	047A	FDCB05CE	575	SET	1,(IY + IND)
0402	CB05	513	RLC	L	047E	CBCF	576	SET	1,A
0404	07	514	RLCA		0480	CBC8	577	SET	1,B
0405	ED6F	515	RLD		0482	CBC9	578	SET	1,C
0407	CB1E	516	RR	(HL)	0484	CBCA	579	SET	1,D
0409	DDCB051E	517	RR	(IX + IND)	0486	CBCB	580	SET	1,E
040D	FDCB051E	518	RR	(IY + IND)	0488	CBCC	581	SET	1,H
0411	CB1F	519	RR	A	048A	CBCD	582	SET	1,L
0413	CB18	520	RR	B	048C	CBD6	583	SET	2,(HL)
0415	CB19	521	RR	C	048E	DDCB05D6	584	SET	2,(IX + IND)
0417	CB1A	522	RR	D	0492	FDCB05D6	585	SET	2,(IY + IND)
0419	CB1B	523	RR	E	0496	CBD7	586	SET	2,A
041B	CB1C	524	RR	H	0498	CBD0	587	SET	2,B
041D	CB1D	525	RR	L	049A	CBD1	588	SET	2,C
041F	1F	526	RRA		049C	CBD2	589	SET	2,D
0420	CB0E	527	RRC	(HL)	049E	CBD3	590	SET	2,E
0422	DDCB050E	528	RRC	(IX + IND)	04A0	CBD4	591	SET	2,H
0426	FDCB050E	529	RRC	(IY + IND)	04A2	CBD5	592	SET	2,L
042A	CB0F	530	RRC	A	04A4	CBD8	593	SET	3,B
042C	CB08	531	RRC	B	04A6	CBDE	594	SET	3,(HL)
042E	CB09	532	RRC	C	04A8	DDCB05DE	595	SET	3,(IX + IND)
0430	CB0A	533	RRC	D	04AC	FDCB05DE	596	SET	3,(IY + IND)
0432	CB0B	534	RRC	E	04B0	CBDF	597	SET	3,A
0434	CB0C	535	RRC	H	04B2	CBD9	598	SET	3,C
0436	CB0D	536	RRC	L	04B4	CBDA	599	SET	3,D
0438	0F	537	RRCA		04B6	CBDB	600	SET	3,E
0439	ED67	538	RRD		04B8	CBDC	601	SET	3,H
043B	C7	539	RST	0	04BA	CBDD	602	SET	3,L
043C	D7	540	RST	10H	04BC	CBE6	603	SET	4,(HL)
043D	DF	541	RST	18H	04BE	DDCB05E6	604	SET	4,(IX + IND)
043E	E7	542	RST	20H	04C2	FDCB05E6	605	SET	4,(IY + IND)
043F	EF	543	RST	28H	04C6	CBE7	606	SET	4,A
0440	F7	544	RST	30H	04C8	CBE0	607	SET	4,B
0441	FF	545	RST	38H	04CA	CBE1	608	SET	4,C
0442	CF	546	RST	08H	04CC	CBE2	609	SET	4,D
0443	9E	547	SBC	A,(HL)	04CE	CBE3	610	SET	4,E
0444	DD9E05	548	SBC	A,(IX + IND)	04D0	CBE4	611	SET	4,H
0447	FD9E05	549	SBC	A,(IY + IND)	04D2	CBE5	612	SET	4,L
044A	9F	550	SBC	A,A	04D4	CBEE	613	SET	5,(HL)
044B	98	551	SBC	A,B	04D6	DDCB05EE	614	SET	5,(IX + IND)

LOC	OBJ CODE	STMT	SOURCE STATEMENT		LOC	OBJ CODE	STMT	SOURCE STATEMENT	
04DA	FDCB05EE	615	SET	5,(IY + IND)	0542	CB29	658	SRA	C
04DE	CBEF	616	SET	5,A	0544	CB2A	659	SRA	D
04E0	CBE8	617	SET	5,B	0546	CB2B	660	SRA	E
04E2	CBE9	618	SET	5,C	0548	CB2C	661	SRA	H
04E4	CBEA	619	SET	5,D	054A	CB2D	662	SRA	L
04E6	CBEB	620	SET	5,E	054C	CB3E	663	SRL	(HL)
04E8	CBEC	621	SET	5,H	054E	DDCB053E	664	SRL	(IX + IND)
04EA	CBED	622	SET	5,L	0552	FDCB053E	665	SRL	(IY + IND)
04EC	CBF6	623	SET	6,(HL)	0556	CB3F	666	SRL	A
04EE	DDCB05F6	624	SET	6,(IX + IND)	0558	CB38	667	SRL	B
04F2	FDCB05F6	625	SET	6,(IY + IND)	055A	CB39	668	SRL	C
04F6	CBF7	626	SET	6,A	055C	CB3A	669	SRL	D
04F8	CBF0	627	SET	6,B	055E	CB3B	670	SRL	E
04FA	CBF1	628	SET	6,C	0560	CB3C	671	SRL	H
04FC	CBF2	629	SET	6,D	0562	CB3D	672	SRL	L
04FE	CBF3	630	SET	6,E	0564	96	673	SUB	(HL)
0500	CBF4	631	SET	6,H	0565	DD9605	674	SUB	(IX + IND)
0502	CBF5	632	SET	6,L	0568	FD9605	675	SUB	(IY + IND)
0504	CBFE	633	SET	7,(HL)	056B	97	676	SUB	A
0506	DDCB05FE	634	SET	7,(IX + IND)	056C	90	677	SUB	B
050A	FDCB05FE	635	SET	7,(IY + IND)	056D	91	678	SUB	C
050E	CBFF	636	SET	7,A	056E	92	679	SUB	D
0510	CBF8	637	SET	7,B	056F	93	680	SUB	E
0512	CBF9	638	SET	7,C	0570	94	681	SUB	H
0514	CBFA	639	SET	7,D	0571	95	682	SUB	L
0516	CBFB	640	SET	7,E	0572	D620	683	SUB	N
0518	CBFC	641	SET	7,H	0574	AE	684	XOR	(HL)
051A	CBFD	642	SET	7,L	0575	DDAE05	685	XOR	(IX + IND)
051C	CB26	643	SLA	(HL)	0578	FDAE05	686	XOR	(IY + IND)
051E	DDCB0526	644	SLA	(IX + IND)	057B	AF	687	XOR	A
0522	FDCB0526	645	SLA	(IY + IND)	057C	A8	688	XOR	B
0526	CB27	646	SLA	A	057D	A9	689	XOR	C
0528	CB20	647	SLA	B	057E	AA	690	XOR	D
052A	CB21	648	SLA	C	057F	AB	691	XOR	E
052C	CB22	649	SLA	D	0580	AC	692	XOR	H
052E	CB23	650	SLA	E	0581	AD	693	XOR	L
0530	CB24	651	SLA	H	0582	EE20	694	XOR	N
0532	CB25	652	SLA	L	0584		695 NN	DEFS	2
0534	CB2E	653	SRA	(HL)			696 IND	EQU	5
0536	DDCB052E	654	SRA	(IX + IND)			697 M	EQU	10H
053A	FDCB052E	655	SRA	(IY + IND)			698 N	EQU	20H
053E	CB2F	656	SRA	A			699 DIS	EQU	30H
0540	CB28	657	SRA	B			700	END	

Appendix E / Z-80 CPU Register and Architecture

This section gives information about the actual Z80 chip including the Central Processing Unit (CPU) Register configuration.

Z-80 CPU Architecture

A block diagram of the internal architecture of the Z-80 CPU is shown in **Figure 2**. The diagram shows all of the major elements in the CPU and it should be referred to throughout the following description.

CPU Registers

The Z-80 CPU contains 208 bits of R/W memory that are accessible to the programmer. **Figure 3** illustrates how this memory is configured into eighteen 8-bit registers and four 16-bit registers. All Z-80 registers are implemented using static RAM. The registers include two sets of six general purpose registers that may be used individually as 8-bit registers or in pairs of 16-bit registers. There are also two sets of accumulator and flag registers.

Special Purpose Registers

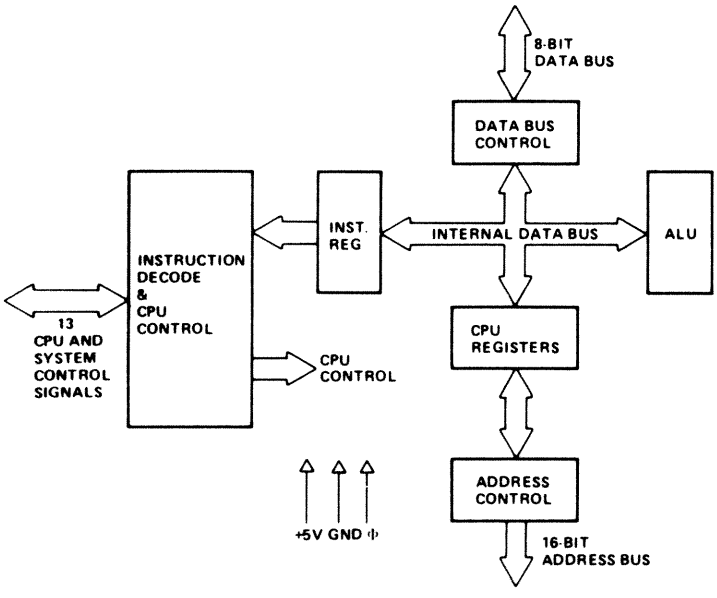


Figure 2, Z-80 CPU Block Diagram.

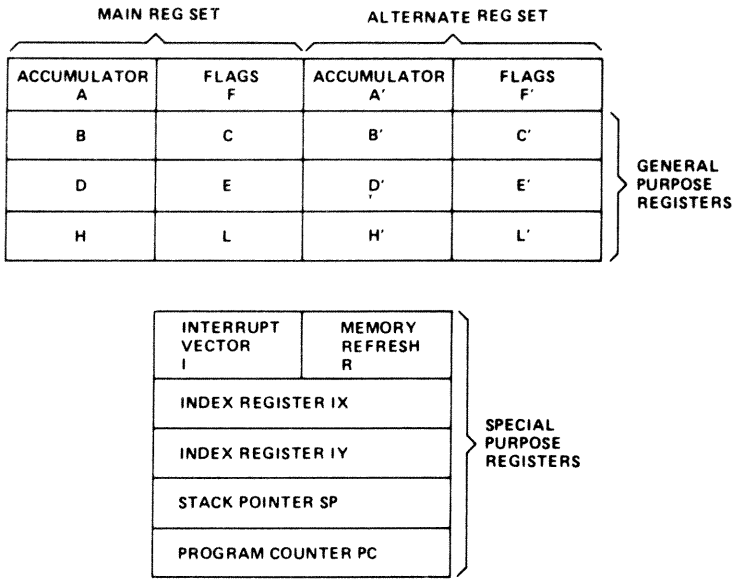


Figure 3, Z-80 CPU Register Configuration.

1. **Program Counter (PC).** The program counter holds the 16-bit address of the current instruction being fetched from memory. The PC is automatically incremented after its contents have been transferred to the address lines. When a program jump occurs the new value is automatically placed in the PC, overriding the incrementer.

2. **Stack Pointer (SP).** The stack pointer holds the 16-bit address of the current top of a stack located anywhere in external system RAM memory. The external stack memory is organized as a last-in first-out (LIFO) file.

   Data can be pushed onto the stack from specific CPU registers or popped off of the stack into specific CPU registers through the execution of PUSH and POP instructions. The data popped from the stack is always the last data pushed onto it. The stack allows simple implementation of multiple level interrupts, unlimited subroutine nesting and simplification of many types of data manipulation.

3. **Two Index Registers (IX & IY).** The two independent index registers hold a 16-bit base address that is used in indexed addressing modes. In this mode, an index register is used as a base to point to a region in memory from which data is to be stored or retrieved. An additional byte is included in indexed instructions to specify a displacement from this base. This displacement is specified as a two's complement signed integer. This mode of addressing greatly simplifies many types of programs, especially where tables of data are used.

4. **Interrupt Page Address Register (I).** The Z-80 CPU can be operated in a mode where an indirect call to any memory location can be achieved in response to an interrupt. The I Register is used for this purpose to store the high order 8-bits of the indirect address while the interrupting device provides the lower 8-bits of the address. This feature allows interrupt routines to be dynamically located anywhere in memory with absolute minimal access time to the routine.

5. **Memory Refresh Register (R).** The Z-80 CPU contains a memory refresh counter to enable dynamic memories to be used with the same ease as static memories. Seven bits of this 8 bit register are automatically incremented after each instruction fetch. The eighth bit will remain as programmed as the result of an LD R, A instruction. The data in the refresh counter is sent out on the lower portion of the address bus along with a refresh control signal while the CPU is decoding and executing the fetched instruction. This mode of refresh is totally transparent to the programmer and does not slow down the CPU operation. The programmer can load the R register for testing purposes, but this register is normally not used by the programmer. During refresh, the contents of the I register are placed on the upper 8 bits of the address bus.

Accumulator and Flag Registers

The CPU includes two independent 8-bit accumulators and associated 8-bit flag registers. The accumulator holds the results of 8-bit arithmetic or logical operations while the flag register indicates specific conditions for 8 or 16-bit operations, such as indicating whether or not the result of an operation is equal to zero. The programmer selects the accumulator and flag pair that he wishes to work with a single exchange instruction so that he may easily work with either pair.

General Purpose Registers

There are two matched sets of general purpose registers, each set containing six 8-bit registers that may be used individually as 8-bit registers or as 16-bit register pairs by the programmer. One set is called BC, DE and HL while the complementary set is called BC', DE' and HL'. At any one time the programmer can select either set of registers to work with through a single exchange command for the entire set. In systems where fast interrupt response is required, one set of general purpose registers and an accumulator/flag register may be reserved for handling this very fast routine. Only a simple exchange command need be executed to go between the routines. This greatly reduces interrupt service time by eliminating the requirement for saving and retrieving register contents in the external stack during interrupt or subroutine processing. These general purpose registers are used for a wide range of applications by the programmer. They also simplify programming, especially in ROM based systems where little external read/write memory is available.

Arithmetic & Logic Unit (ALU)

The 8-bit arithmetic and logical instructions of the CPU are executed in the ALU. Internally the ALU communicates with the registers and the external data bus on the internal data bus. The type of functions performed by the ALU include:

Add	Left or right shifts or rotates (arithmetic and logical)
Subtract	Increment
Logical AND	Decrement
Logical OR	Set bit
Logical Exclusive OR	Reset bit
Compare	Test Bit

Instruction Register and CPU Control

As each instruction is fetched from memory, it is placed in the instruction register and decoded. The control sections performs this function and then generates and supplies all of the control signals necessary to read or write data from or to the registers, control the ALU and provide all required external control signals.

Z-80 CPU Pin Description

The Z-80 CPU is packaged in an industry standard 40 pin Dual In-Line Package. The I/O pins are shown in **Figure 4** and the function of each is described below.

A_0-A_{15} (Address Bus)	Tri-state output, active high. A_0-A_{15} constitute a 16-bit address bus. The address bus provides the address for memory (up to 64K bytes) data exchanges and for I/O device data exchanges. I/O addressing uses the 8 lower address bits to allow the user to directly select up to 256 input or 256 output ports. A_0 is the least significant address bit. During refresh time, the lower 7 bits contain a valid refresh address.
D_0-D_7 (Data Bus)	Tri-state input/output, active high. D_0-D_7 constitute an 8-bit bidirectional data bus. The data bus is used for data exchanges with memory and I/O devices.
$\overline{M_1}$ (Machine Cycle one)	Output, active low. $\overline{M_1}$ indicates that the current machine cycle is the OP code fetch cycle of an instruction execution. Note that during execution of 2-byte op-codes, $\overline{M_1}$ is generated as each op-code byte is fetched. These two byte op-codes always begin with CBH, DDH, EDH or FDH. $\overline{M_1}$ also occurs with $\overline{IORQ}$ to indicate an interrupt acknowledge cycle.

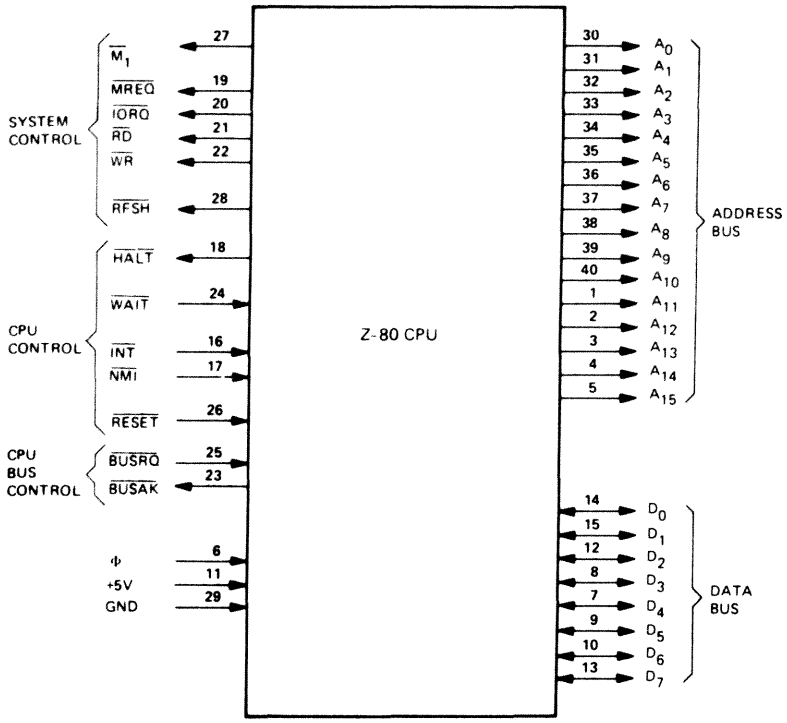


Figure 4, Z-80 Pin Configuration.

$\overline{\text{MREQ}}$ (Memory Request)	Tri-state output, active low. The memory request signal indicates that the address bus holds a valid address for a memory read or memory write operation.
$\overline{\text{IORQ}}$ (Input/Output Request)	Tri-state output, active low. The $\overline{\text{IORQ}}$ signal indicates that the lower half of the address bus holds a valid I/O address for a I/O read or write operation. An $\overline{\text{IORQ}}$ signal is also generated with an $\overline{M_1}$ signal when an interrupt is being acknowledged to indicate that an interrupt response vector can be placed on the data bus. Interrupt Acknowledge operations occur during $\overline{M_1}$ time while I/O operations never occur during $\overline{M_1}$ time.
$\overline{\text{RD}}$ (Memory Read)	Tri-state output, active low. $\overline{\text{RD}}$ indicates that the CPU wants to read data from memory or an I/O device. The addressed I/O device or memory should use this signal to gate data onto the CPU data bus.
$\overline{\text{WR}}$ (Memory Write)	Tri-state output, active low. $\overline{\text{WR}}$ indicates that the CPU data bus holds valid data to be stored in the addressed memory or I/O device.

$\overline{\text{RFSH}}$ (Refresh)	Output, active low. $\overline{\text{RFSH}}$ indicates that the lower 7 bits of the address bus contain a refresh address for dynamic memories and the current $\overline{\text{MREQ}}$ signal should be used to do a refresh read to all dynamic memories.
$\overline{\text{HALT}}$ (Halt state)	Output, active low. $\overline{\text{HALT}}$ indicates that the CPU has executed a HALT software instruction and is awaiting either a non maskable or a maskable interrupt (with the mask enabled) before operation can resume. While halted, the CPU executes NOP's to maintain memory refresh activity.
$\overline{\text{WAIT}}$ (Wait)	Input, active low. $\overline{\text{WAIT}}$ indicates to the Z-80 CPU that the addressed memory or I/O devices are not ready for a data transfer. The CPU continues to enter wait states for as long as this signal is active. This signal allows memory or I/O devices of any speed to be synchronized to the CPU.
$\overline{\text{INT}}$ (Interrupt Request)	Input, active low. The Interrupt Request signal is generated by I/O devices. A request will be honored at the end of the current instruction if the internal software controlled interrupt enable flip-flop (IFF) is enabled and if the $\overline{\text{BUSRQ}}$ signal is not active. When the CPU accepts the interrupt, an acknowledge signal ($\overline{\text{IORQ}}$ during M_1 time) is sent out at the beginning of the next instruction cycle.
$\overline{\text{NMI}}$ (Non Maskable Interrupt)	Input, negative edge triggered. The non maskable interrupt request line has a higher priority than $\overline{\text{INT}}$ and is always recognized at the end of the current instruction, independent of the status of the interrupt enable flip-flop. $\overline{\text{NMI}}$ automatically forces the Z-80 CPU to restart to location 0066_H. The program counter is automatically saved in the external stack so that the user can return to the program that was interrupted. Note that continuous WAIT cycles can prevent the current instruction from ending, and that a $\overline{\text{BUSRQ}}$ will override a $\overline{\text{NMI}}$.
$\overline{\text{RESET}}$	Input, active low. $\overline{\text{RESET}}$ forces the program counter to zero and initializes the CPU. The CPU initialization includes: 1) Disable the interrupt enable flip-flop 2) Set Register I = 00_H 3) Set Register R = 00_H 4) Set Interrupt Mode 0 During reset time, the address bus and data bus go to a high impedance state and all control output signals go to the inactive state.

$\overline{\text{BUSRQ}}$ (Bus Request)	Input, active low. The bus request signal is used to request the CPU address bus, data bus and tri-state output control signals to go to a high impedance state so that other devices can control these buses. When $\overline{\text{BUSRQ}}$ is activated, the CPU will set these buses to a high impedance state as soon as the current CPU machine cycle is terminated.
$\overline{\text{BUSAK}}$ (Bus Acknowledge)	Output, active low. Bus acknowledge is used to indicate to the requesting device that the CPU address bus, data bus and tri-state control bus signals have been set to their high impedance state and the external device can now control these signals.
Φ	Single phase TTL level clock which requires only a 330 ohm pull-up resistor to +5 volts to meet all clock requirements.

Z-80 CUP Instruction Set

The Z-80 CPU can execute 158 different instruction types including all 78 of the 8080A CPU. The instructions can be broken down into the following major groups:

- Load and Exchange
- Block Transfer and Search
- Arithmetic and Logical
- Rotate and Shift
- Bit Manipulation (set, reset, test)
- Jump, Call and Return
- Input/Output
- Basic CPU Control

INDEX

Subject	Page
*	113
8 bit load group	123-140
LD r,r'	123
LD r,n	124
LD r,(HL)	125
LD r,(IX+D)	125-126
LD r,(IY+d)	127
LD (HL),r	128
LD (IX+d),r	128-129
LD (IY+d),r	129-130
LD (HL),n	130-131
LD (IX+d),n	131
LD (IY+d),n	132
LD A,(BC)	133
LD A,(DE)	133-134
LD A,(nn)	134-135
LD (BC),A	135
LD (DE),A	135-136
LD (nn),A	136-137
LD A,I	137
LD A,R	138
LD I,A	138-139
LD R,A	139-140
8 bit arithmetic and logical	181-205
ADD A,(HL)	183
ADD A,(IX+d)	183-184
ADD A,(IY+d)	184-185
ADD A,n	182
ADD A,r	181-182
ADD A,S	185-187
AND s	191-193
CP s	198-199
DEC m	203-205
INC (HL)	201
INC (IX+d)	201-202
INC (IY+d)	202-203
INC r	200
OR s	193-195
SUB s	187-189
SBC A,s	189-191
XOR s	195-197

Subject	Page
16 bit load group	141-161
LD dd,nn	141-142
LD IX,nn	142
LD IY,nn	143
LD HL,(nn)	143-144
LD dd,(nn)	144-145
LD IX,(nn)	145-146
LD IY,(nn)	146-147
LD (nn),HL	147-148
LD (nn),dd	148-149
LD (nn),IX	149-150
LD (nn),IY	150-151
LD SP,HL	151-152
LD SP,IX	152
LD SP,IY	153
16 bit arithmetic group	217-226
ADD HL,ss	217
ADC HL,ss	218
ADD IX,pp	220
ADD IY,rr	221
DEC ss	224
DEC IX	225
DEC IY	225-226
INC ss	222
INC IX	222-223
INC IY	223-224
SBC HL,ss	219
Absolute Assembly	72
ADC A,XH	337
ADC HL,ss	218
ADD A,(HL)	183
ADD A,(IX+d)	183-184
ADD A,(IY+d)	184-185
ADD A,n	182
ADD A,r	181-182
ADD A,S	185-187
ADD A,XH	337
ADD HL,ss	217
ADD IX,pp	220
ADD IY,rr	221
Address Different from Pass 1	327
ALASM (see Assembler)	
ALBUG (see Debugger)	

Subject	Page
ALEDIT (see Editor)	
ALLINK (see Linker)	
Altran (See File Transfer)	
And s	191-193
And XH	337
APOP	88
APUSH	88
Arithmetic Operators	64-66
ASCII (see DEFM)	
Assembler	
Command	23
Description	23-27
Directives	69-113
Errors	327
Expressions	63-64
Labels	61-62
Object Code Format	338-340
Operands	63-67
Operators	64-67
Switches	24-26
Symbols	61-62
Assembler Listing	
Description	23-26
EJECT	93
HEADER	97-98
PRINT	106-107
QUIT	109
STOP	112
TITLE	113
USING	113
VERSION	113
Attempt to Use a Non-Program File as a Program	331
Bad File Format	325
Bad Filename Format	325
Bad Parameters	325
BLOCK (see DEFS)	
block comment	113-114
BIT b,(HL)	254-255
BIT b,(IX+d)	255-256
BIT b,(IY+d)	256-257
BIT b,r	253-254
Bit,set,reset, and test group	253-263
BIT b,(HL)	254-255
BIT b,(IX+d)	255-256
BIT b,(IY+d)	256-257

Subject	Page
BIT b,r	253-254
RES b,m	262-263
SET b,r	257-258
SET b,(HL)	258-259
SET b,(IX+d)	259-260
SET b,(IY+d)	261-262
Buffer Full	325
BYTE (see DEFB)	
CALL cc,nn	278-280
CALL nn	277-278
Call and return group	277-286
CALL cc,nn	278-280
CALL nn	277-278
RET	280-281
RET cc	281-283
RETI	283-284
RETN	284-285
RST p	285-286
CCF	210
CMPD operand1,operand2, [length]	304-306
CMPI operand1,operand2, length	306-308
comment	67-68,113-114
Conditional Sections (see If Sections)	
CPD	178-179
CPDR	179-180
CPI	175-177
CPIR	177-178
CPL	208-209
CP s	198-199
CP XH	337
CPR operand	303-304
DAA	207-208
Data	
Defining	70
DEFB	89-90
DEFE	90
DEFM	91
DEFR	91
DEFT	92
DEFW	92-93
DATE	89
DB (see DEFB)	

Subject	Page
Debugger	
Description	29-42
Loading	29-31
Display	30-31
Registers	32
Data	32-33
Breakpoints	33,34-36
Disk Zap	41-42
DEC IX	225
DEC IY	225-226
DEC m	203-205
DEC ss	224
DEC XH	337
DEFB	89-90
DEFE	90
DEFL	90
DEFM	91
DEFR	91
DEFS	92
DEFT	92
DEFW	92-93
DS (see DEFS)	
DW (see DEFW)	
DI	212-213
Directives	69-113
Introduction	69-86
Reference	86-114
Disk Zap	40-42
DJNZ e	275-276
DROP	93
Editor	
Description	11-13
Errors	325
Loading	11
Insert Mode	19-21
Control	20
Special Keys	21
Line Edit Mode	21-22
Subcommands	21
Special Keys	22
Command Mode	12-18
Special Keys	13
Commands	14-18
Compatibility with other Editors	18
EI	213
EJECT	93

Subject	Page
END	93-94
ENDI	94
ENDM	94
ENTRY (see PUBLIC)	
EQU	94
ERROR 24	331
ERROR 34	331
ERROR 37	331
Error Messages	323-331
EX AF,AF'	163-164
Exchange,Search, and Transfer	163-180
CPI	175-177
CPIR	177-178
CPD	178-179
CPDR	179-180
EX DE,HL	163
EX AF,AF'	163-164
EXX	164-165
EX (SP),HL	165-166
EX (SP),IX	166-167
EX (SP),IY	167-168
LDI	169-170
LDIR	170-172
LDD	172-173
LDDR	173-175
EX DE,HL	163
EX operand	309-310
Expressions	63-64
EX (SP),HL	165-166
EX (SP),IX	166-167
EX (SP),IY	167-168
EXT	94-95
Extended Z80 Mnemonics	303-321
CPR operand	303-304
CMPD operand1, operand2, [length]	304-306
CMPI operand1, operand2, length	306-308
TZ operand	308
EX operand	309-310
LD double register	310-315
MOVD operand1, operand2, length	315-316
MOVI operand1, operand2, length	317-318
POP	318

Subject	Page
RSTR operand	318-320
SAVE operand	320-321
SVC	321-322
EXTERN	95
External Symbols	66
EXTERN	95
EXT	94-95
GLINK	96
GLOBAL	96-97
LINK	101-102
PUBLIC	108-109
EXX	164-165
FILL	95-96
GLINK	96
GLOBAL	96-97
File Transfer	
Set-Up	47
Loading	48-49
Errors	52
Command File	52-53
Connector	57
Technical	53-56
Object Files	53
File Not Found	331
Hit Any Key to Continue	326
General purpose arithmetic and	
CPU control groups	207-215
CCF	210
CPL	206-207
DAA	207-208
DI	212-213
EI	213
HALT	212
IM0	214
IM1	214
IM2	215
NEG	209-210
NOP	211-212
SCF	211
Global File	78-82
HALT	212
HEADER	97-98
If Section	85-86
IFDEF	98
IFF	98
IFM	98

Subject	Page
IFNZ	99
IFP	99
IFT	99
IFUND	99
IFZ	100
IFUND	99
IFZ	100
IFDEF	98
IFF	98
IFM	98
IFNZ	99
IFP	99
IFT	99
Illegal Addressing	329
IM0	214
IM1	214-215
IM2	215
IN A,(n)	287
INC (HL)	201
INC IX	222-223
INC IY	223-224
INC (IX+d)	201-202
INC (IY+d)	202-203
INCLUDE	100
INC r	200
INC ss	222
INC XH	337
IND	292-293
Index Sections	82-83
ISECT	100-101
APOP	88
APUSH	88-89
DROP	93
INDR	293-294
INI	289-290
INIR	290-292
Initializing Location Counter	72
Input and output group	287-301
IN A,(n)	287
IND	292-293
INDR	293-294
INI	289-290
INIR	290-292
IN r,(C)	288-289
OUT (C),r	295-296
OUT (n),A	294-295

Subject	Page
OUTD	299
OUTI	296-297
OTIR	297-298
IN r,(C)	288-289
Invalid Parameter	330
ISECT	100-101
JP cc,nn	266-267
JP (HL)	273
JP (IX)	274
JP (IY)	274-275
JP nn	265
JR C,e	268-269
JR e	267-268
JR NC,e	269
JR NZ,e	272
JR Z,e	270
Jump group	265-276
DJNZ e	275-276
JP cc,nn	266-267
JP (HL)	273
JP (IX)	274
JP (IY)	274-275
JP nn	265
JR C,e	268-269
JR e	267-268
JR NC,e	269
JR NZ,e	272
JR Z,e	270
Labels	61-62
LD A,(BC)	133
LD A,(DE)	133-134
LD A,I	137
LD A,(nn)	134-135
LD A,R	138
LD (BC),A	135
LD dd,nn	141-142
LD dd,(nn)	144-145
LD (DE),A	135-136
LD (HL),n	130-131
LD HL,(nn)	143-144
LD (HL),r	128
LD I,A	137
LD (IX+d),n	131-132
LD (IX+d),r	128-129
LD IX,nn	142
LD IX,(nn)	145-146

Subject	Page
LD (IY+d),n	132
LD (IY+d),r	129-130
LD IY,nn	143
LD IY,(nn)	146-147
LD (nn),A	136-137
LD R,A	139-140
LD r,n	124
LD r,r'	123
LD r,(HL)	125
LD r,(IX+D)	125-126
LD r,(IY+d)	127
LD (nn),dd	148-149
LD (nn),HL	147-148
LD (nn),IX	149-150
LD (nn),IY	150-151
LD SP,HL	151-152
LD SP,IX	152
LD SP,IY	153
LDD	172-173
LD double register	311-315
LDDR	173-175
LDI	169-170
LDIR	170-172
LD r,XH	337
LD XH,r	337
LD XH,n	337
Line Length Too Long, Truncating line	325
Line Number Too Large	325
LINK	101-102
Linker	
Command	43-44
Technical	44-45
Errors	329-330
LITORG	102-103
Location Counter	72-74
MACRO	103-104
Macro Editor Assembler Compatibility	18
Macro Sections	83-85
ENDM	94
MACRO	103-104
Missing External Transfer Address	329
MOVD operand1,operand2, length	315-316
MOVI operand1,operand2, length	317-318
Multiply Defined Entry Symbol	329

Subject	Page
NEG	209-210
NOEND	104
NOFILL	104
NOLOAD	105
NOP	211-212
No Text	325
Number Bases	69-70
OBJ	105
Occurrence Too Large	325
Open Attempt For a File Already Open.	331
Operands	63-67
Operators	64-67
ORG	105-106
OR s	193-195
OR XH,n	337
OTIR	297-298
OUT (C),r	295-296
OUT (n),A	294-295
OUTD	299
OUTI	296-297
overflow	115
parity odd	115
parity even	115
PATCH	106
POP	318
POP IX	158-160
POP IY	160-161
POP qq	157-158
PUSH IX	155-156
PUSH IY	156-157
PUSH qq	153-154
PRINT	106-107
Program Section	75
PSECT	107-108
Pseudo Ops (see Directives)	
PUBLIC	108-109
QUIT	109
RADIX	70,109-110*
REF	110
Relocatable	72-73,75
Operators	66
RES b,m	262-263
RESLD r,n,m	336
RESLOC	110-111
RES n,m	336
RET	280-281

Subject	Page
RET cc	281-283
RETI	283-284
RETN	284-285
RL m	236-238
RL m	335
RLA	228
RLCLD r,m	335
RLC m	335
RLC r	231-232
RLC (HL)	232-233
RLC (IX+d)	233-234
RLC (IY+d)	234-236
RLD	249-251
RLLD r,m	335
Rotate and shift group	225
RL m	236-238
RLA	228
RLC r	231-232
RLC (HL)	232-233
RLC (IX+d)	233-234
RLC (IY+d)	234-236
RLD	249-251
RR m	240-242
RRA	230
RRC m	238-240
RRCA	229
RRD	251-252
SLA m	242-244
SRA m	245-247
SRL m	247-249
RR m	240-242
RRA	230
RRC m	238-240
RRCA	229
RRCLD r,m	335
RRD	251-252
RRLLD r,m	335
RST p	385-286
RSTR operand	318-320
Sample Session	3-9
SAVE operand	320-321
SBC A,s	189-191
SBC HL,ss	219
SBC A,XH	337
SCF	211
Search Arg Too Long	326

Subject	Page
SET b,r	257-258
SET b,(HL)	258-259
SET b,(IX+d)	259-260
SET b,(IY+d)	261-262
SETLOC	111-112
SET n,m	335
SETLD r,n,m	335
SLA m	242-244
SLOLD r,m	335
SRA m	245-247
SRA m	335
SRALD r,m	335
SRL m	247-249
SRLLD r,m	335
SUB s	187-189
TZ operand	308
XOR s	195-197
SLA m	335
SLALD r,m	335
SLO m	335
SRL m	335
STOP	112
Storage	
Defining	71
DEFS	92
FILL	95-96
NOFILL	104
SUB XH	337
Symbols	
Defining	70
External	66,75-82
Syntax	61
Symbol Table Overflow	329
Syntax Error	326
SVC	321
TITLE	112-113
TIME	112
Total Line Length Too Long	326
Undefined External Symbol	329
Undocumented Z80 Instructions	333-337
USING	113
VERSION	113-114
WORD (see DEFW)	
XOR XH	337
Z80	
alphabetic	347-352
extended	303-322
hardware	353-359
mnemonics	115-301
notations	117-118
numeric list	341-346
undocumented	333-337

RADIO SHACK, A DIVISION OF TANDY CORPORATION

U.S.A.: FORT WORTH, TEXAS 76102
CANADA: BARRIE, ONTARIO L4M 4W5

TANDY CORPORATION

AUSTRALIA	BELGIUM	U. K.
91 KURRAJONG ROAD MOUNT DRUITT, N.S.W. 2770	PARC INDUSTRIEL DE NANINNE 5140 NANINNE	BILSTON ROAD WEDNESBURY WEST MIDLANDS WS10 7JN

11/83 MP

Printed in U.S.A.