

Dear PDS User:

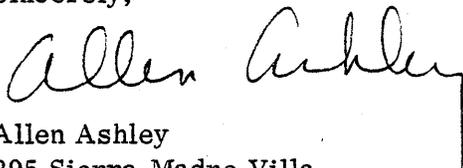
I regret being unable to include a personal note. However, there are a few points whould could not be covered in the documentation.

First, I want you to be happy with the PDS software package. If you have any difficulty, however slight, with either the documentation or the programs, please contact me. I prefer to interact by telephone, but as time allows I will correspond by mail.

Should program errors arise they will be repaired for free. I ask only that you return your original diskette with cardboard backing and a return manila envelope with sufficient return postage.

Many of the best features of PDS were suggested by users, and your comments and suggestions on the documentation or the programs are welcome. Let's keep in touch.

Sincerely,



Allen Ashley
395 Sierra Madre Villa
Pasadena, CA 91107

(213) 793-5748

P. S. LONGLABL is a version of MAKRO which allows up to 10-character labels. LONGLINK is the corresponding linkage editor.

Scrolling program output: The two PDS assemblers and the G command of EDIT allow the output to be scrolled. Pressing the space bar will freeze the display; any other key will resume scroll. This feature relies upon the non-standard Control/C detect routine in the DOS. The programs call the Control/C routine and expect the key pressed, if any, to return in the accumulator. If a blank is returned, the programs call character-in to wait for another key to be pressed before resuming operation.

PDS

ASSEMBLY LANGUAGE DEVELOPMENT SYSTEM

FOR

NORTH STAR MINIDISK

OR

NORTH STAR HORIZON

INCLUDING:

RELOCATING MACRO ASSEMBLER
INTERACTIVE ASSEMBLER/EDITOR
STRING-ORIENTED TEXT EDITOR
TRACE DEBUG/DISASSEMBLER
LINKAGE EDITOR/LOADER
RELOCATING LOADER

FEATURING:

FULL Z80 CAPABILITY
OPERATIONAL ON Z80 OR 8080
INTEL MNEMONICS
AUTOMATIC FILE HANDLING

READY TO RUN ON DISKETTE
COMPLETE DOCUMENTATION
FULL USER SUPPORT



\$99

Copyright 1978
A.M. Ashley
395 Sierra Madre Villa
Pasadena, CA 91107

(213) 793-5748

HDS

HYBRID DEVELOPMENT SYSTEM -- 340

The HDS Hybrid Development System is available for all North Star systems. A hybrid program is one in which portions are performed by assembly language and portions are performed by BASIC. Such programs may be attractive because:

1. Critical program segments may be coded in assembly language to achieve higher speed.
2. Proprietary program segments may better be protected when coded in assembly language.
3. Hybrid programs offer nearly the same execution speed as assembly code while retaining the ease of BASIC program development.
4. Certain operations are much more easily performed at the assembler level.
5. Hybrid programs can use internal BASIC routines for ease of program development.

HDS includes an interactive assembler/editor located at 40H to be co-resident with BASIC, together with modifications to North Star BASIC which facilitate communication between BASIC and assembly routines. The modifications to BASIC give access to the addresses of BASIC variables and extend the CALL function of BASIC to allow an unlimited parameter list. Access to the address of a BASIC variable is gained by enclosing the variable in square brackets. Thus A1 refers to the value of variable A1 while [A1] refers to the location of A1. Now assembly routines can use BASIC variables or strings and return results back to BASIC.

A roadmap to BASIC is included containing a list of BASIC utility entry points and their calling sequence. Examples are provided to:

1. Load an assembly language routine from BASIC using the sequence: PS = "FILE": Z9 = CALL (ADDR, LOCN, [PS])
2. Find the total of a BASIC array A(N) as: Z9 = CALL (ADDR, [A(1)], [S], N)
3. Find the minimum in a BASIC array as: Z9 = CALL (ADDR, [B], [A(1)], N)

HDS requires at least 24K memory starting at 0H. Modifications are available for standard (8-digit) North Star BASIC Release 4.0 and 5.0. Modifications for other versions of North Star BASIC are available by arrangement. As always, full user support is provided by mail or phone. Dealer discounts are available.

SOURCE MODULES

To facilitate the development of assembly language application programs, and to encourage the use and sale of PDS, a number of assembly language program modules are available. These source modules are provided to facilitate your development efforts, and no restriction is imposed on their use. Interface requirements are clearly documented.

<u>MODULE</u>	<u>FUNCTION</u>	<u>REQUIREMENTS</u>	<u>PRICE</u>
ALPHSORT	High speed alphabetic sort	None	\$ 20
NUMRSORT	High speed numeric sort	None	20
FPPACK	BCD floating point arithmetic	None	20
FOURIER	Fast Fourier transform	FPPACK	20
MINV	Matrix inversion	FPPACK	20
MATPRD	Matrix product	FPPACK	10
RATPOL	Rational function and utilities	FPPACK	15
SQRT	Square root	FPPACK	5
TRIGS	Sine, Cosine, TAN, ATAN	FPPACK, RATPOL	20
LOGEXP	Exponential, logarithm, Y ^x	FPPACK, RATPOL	20
FPIOP	Floating point I/O	None	15
FORMAT	Formatted floating point output	None	10
NFILES	North Star disk handler	None	15

ENTIRE PACKAGE: \$100

A LA CARTE: ADD \$5 PER ORDER FOR DISK

REGENT

Disk Disassembler (\$25): Generates a source file on disk from object program stored in memory. NOT for the casual or novice programmer. (NORTH STAR ONLY.)

EZ-80

Assembly Language Tutorial (\$25): FOR the novice programmer. Teaches Z-80 instruction set and operations by executing assembly language commands individually. Registers and flags are displayed for each instruction executed. (NORTH STAR ONLY.)

All programs are available from your local computer store or directly from Allen Ashley, 395 Sierra Madre Villa, Pasadena, CA 91107 (213)793-5748

CONTENTS

1. PDS OVERVIEW AND INTERFACE PROCEDURES

Interfacing PDS.....	1-5
Bringing up PDS.....	1-6
Relocating Loaders KWIK and KWIKABS.....	1-7
Relocatable DEBUG.....	1-8
EDIT Disk Files.....	1-9
ASMB Memory Files.....	1-9
MAKRO Execution.....	1-10
Special Note to Z80 Owners.....	1-11
Sample ASMB Operation.....	1-12
Memory Size.....	1-13

2. ASMB EDITOR/ASSEMBLER

Introduction.....	2-2
ASMB Organization.....	2-3
Executive Commands.....	2-4
Command List	
Command Format	
Editor.....	2-7
Automatic Line Numbering	
Assembler Operation.....	2-8
Source Line Format	
Assembler Constants	
Register Mnemonics.....	2-9
Assembly Language.....	2-10
8 Bit Load	
Accumulator Load/Store	
8 Bit Immediate	
16 Bit Load/Store	
Exchange, Block Transfer, and Search	
8 Bit Arithmetic and Logical	
General Purpose Arithmetic and CPU Control	
16 Bit Arithmetic Group	
Rotate and Shift Group	
Bit Manipulation	
Input/Output Group	
Jump Group	
Call and Return Group	
Pseudo Operations.....	2-18
Assembler Errors/Diagnostics.....	2-20
Existing Source Files.....	2-20

3. MAKRO ASSEMBLER

Introduction.....	3-2
Makro Input/Output.....	3-3
Source Line Format	
Assembler Operation.....	3-4
Special Operands	
Assembler Constants	
Register Mnemonics.....	3-5
Assembly Language.....	3-6
Pseudo Operations.....	3-14
Relocation Pseudo Operations	
Assembler Errors/Diagnostics.....	3-19
MAKRO Conditional Assembly.....	3-20
MAKRO Macro Capability.....	3-23
Introduction to Macros	
Macro Processing	
MAKRO Idiosyncracies	
Procedural and Syntactical Rules	
Using Macros.....	3-28
Repetition Control.....	3-30
MAKRO Block Structured Assembly.....	3-32
Assembly Time Input.....	3-32
Communication Between Macros.....	3-33
Relocation.....	3-34
Assembly Time Source - LINK	
Object Time - Relocatable Code	
Loader Directives	
Object File Format	
Source Code Restrictions	
Symbol Table	
PDS Relocating Loaders.....	3-41
MAKRO Expression Evaluation.....	3-42
INTEL Source Compatability.....	3-44
Sample Linkage Operation.....	3-45

4. EDIT TEXT EDITOR

Introduction.....	4-2
EDIT Organization.....	4-3
EDIT Executive.....	4-4
Command Format	
Nesting Commands	
Special Characters	
Text Pointers	
Executive Commands.....	4-6
Command List	
Special Character Commands	
Command Strings and Block Commands.....	4-11
Command String Syntax.....	4-12

4. EDIT TEXT EDITOR (Cont.)

Error Messages.....	4-13
Sample EDIT Operation.....	4-14
Sample Block Operations.....	4-15
Conditional Command Execution.....	4-16
Text Rearrangement.....	4-19
Use EDIT to Save Typing.....	4-20
Memory Organization.....	4-21
Display.....	4-22

5. DEBUG PROGRAM DEVELOPMENT AID

Introduction.....	5-2
DEBUG Organization.....	5-3
Executive Commands.....	5-4
Single Step Executive	
Using DEBUG.....	5-11
Suggestions	



HIGHLIGHTS CERTAIN TEXTUAL ITEMS WHICH MAY CAUSE
DIFFICULTY IF OVERLOOKED.

PDS

PROGRAM DEVELOPMENT SYSTEM

PDS is an exceptionally powerful assembly language development system for 8080 or Z80 microcomputers with at least one disk drive. PDS includes a unified assembler/editor, a macro assembler with a relocating linking loader, a string-oriented text editor, and a trace debugger/disassembler.

The assemblers favor the INTEL instruction mnemonics, treating the Z80 superset as a logical and syntactical extension. The debug module features breakpoint or single-step execution of programs, with trace display of all register contents, flag status, a memory window, and the mnemonics of the instruction just executed and the next instruction to be executed.

The power of PDS derives from the interactive environment afforded by the assembler/editor and the debug package. Program modules can be modified, assembled and checked in seconds under the tight control of trace execution.

While the many features of PDS will satisfy the demands of the most sophisticated programmer, PDS affords an exceptional educational environment for beginning assembly language programmers. The interactive combination of the ASMB editor/assembler and the DEBUG trace program allow the user to witness operation of his program first hand.

To facilitate development of applications programs with PDS, source modules are available for floating point arithmetic, floating point input/output, trigonometric functions, numerical and alphabetic sorting, matrix inversion, fast Fourier transform, and a full function expression evaluator.

For further information, please contact:

Allen Ashley
395 Sierra Madre Villa
Pasadena, CA 91107
(213) 793-5748

COMPARE!

	<u>PDS</u>	<u>TDL</u>	<u>CROMEMCO</u>	<u>PRO TECH ALS-8</u>	<u>CP/M</u>	<u>INTEL</u>
Macro	X	X	X			X
Relocating	X	X	X			
Trace Debug	X			X	X	
Interactive Assembler	X			X		
Z-80 Assembly	X	X	X			
8080 Operational	X			X	X	X
INTEL Mnemonics	X	X		X	X	X
String Editor	X	X	X		X	X
Linkage Edit	X		X			
Disassembler	X			X		

NO OTHER READILY AVAILABLE
PROGRAM DEVELOPMENT SYSTEM
OFFERS AS MANY FEATURES AS

P D S

\$99

P D S

The components of PDS are structured to provide the most complete, well-rounded program development system available for microcomputer use.

PDS includes:

ASMB	Assembler/Editor
MAKRO	Macro Assembler
EDIT	Text Editor
DEBUG	Debug Monitor/Disassembler
LINKED	Linkage Editor
KWIK	Relocating Loader

MAKRO and ASMB assemble the complete instruction set of the Z-80 and feature mnemonics which are a logical and syntactical extension of the widely familiar 8080 assembly language.

Each of the components of PDS is written in the 8080 instruction subset and the entire system is thus operational on either Z-80 or 8080 machines.

PDS is an ideal program development system for those owning a Z-80 machine or those 8080 owners anticipating a future expansion to the more powerful Z-80 processor.

APPROXIMATE MEMORY REQUIREMENTS

<u>PROGRAM</u>	<u>DECIMAL</u>
ASMB	6K
MAKRO	7.5K
EDIT	2K
DEBUG	3.5K (RAM at 0)

Minimum operating system: 16K RAM and one disk drive. DEBUG, LINKED and KWIK are furnished in relocatable form to satisfy the requirements of individual systems.

The sizes of disk files for relocatable modules do not reflect the memory required for execution of those modules. Such files, containing relocation and loading information in addition to program data, greatly exceed the memory space required for execution. As an example, the relocatable disk file DEBUG occupies some 55 sectors of the disk, but less than 4K of memory when loaded.

ASMB: An editor/assembler combination for the rapid development of small to medium size assembly language programs, ASMB includes all the features necessary for the creation, modification and disk storage of assembly language source files for Z80 or 8080 computers. ASMB is a very fast assembler which, together with the co-resident editor, is structured for a very rapid assemble/execute/modify cycle. The instruction set of ASMB is designed to be a logical and syntactical extension of the widely familiar INTEL instruction set for the 8080. Users already familiar with 8080 assembly language will readily acquire the extended instruction set of the Z80 processor.

MAKRO: An extraordinary assembler featuring full macro and conditional assembly capability, MAKRO incorporates the power of a relocating assembler and a linkage editor/loader. Program modules developed with the ASMB assembler can be collected into a source library for the MAKRO assembler. The considerably enhanced power of the MAKRO/EDIT combination, together with the overall reduced memory requirements of MAKRO, make the two assemblers perfect companions.

EDIT: A very powerful text editor featuring a full spectrum of text manipulation operations including string search, substitution, insertion, deletion, and block move or delete. An elaborate command interpreter allows the definition of command string macros. Segments of an input text file can be drawn from disk into memory, modified, and written back to an output disk file. Large, heavily-commented source files which exceed available memory can be developed and modified easily with the EDIT text editor.

DEBUG: An incomparable software development tool featuring single-step execution of Z80 or 8080 programs with complete display of all register contents, flag status, and trace display (in mnemonic form) of the instruction just executed and the next instruction to be executed. The single-step breakpoint can be located anywhere in the user's program.

DEBUG, together with the fast ASMB editor/assembler combination, provides an interactive environment for the development of assembly language programs. There is no more powerful development system: program modules can be assembled, checked, and modified in seconds. Programs operating under the trace mode of DEBUG are held tightly under control -- errors can be caught before they blow the program. The degree of program intimacy afforded by DEBUG greatly exceeds that of BASIC.

DEBUG includes a disassembler for translating 8080 or Z80 object code into the MAKRO/ASMB instruction mnemonics. DEBUG also includes string search and change, memory display in ASCII or hexadecimal, memory fill by byte or block, and block move or compare functions. DEBUG uses RST3 and requires RAM at low memory.

LINKED: Linkage editor, linking loader. LINKED searches library files of previously assembled modules to include those necessary to complete the assembly. Commonly used routines need only be developed once.

KWIK: Relocating loader creates an executable memory image for programs not requiring a linkage edit.

INTERFACING PDS TO NORTH STAR DOS

The components of PDS utilize the standard entry points to the North Star Disk Operating System:

DOS + 0DH	Character out	
DOS + 10H	Character in	
DOS + 16H	Control/C	
DOS + 28H	Warm start entry	2A28

File names communicated to PDS are terminated by a carriage return. The file name may be suffixed by an optional unit number. The unit number, if present, must be separated from the file name by a comma. File names not suffixed by a unit number default to drive 1.

Components of PDS which generate disk output request an output file name. The output file must be found in the directory. PDS will examine the size of the output file. A zero-length output file is treated as a new file and PDS will update the directory entry to reflect the completed disk operations.

If a required file is not found in the directory, PDS issues a '?' prompt and awaits re-entry of the file name. PDS will automatically size the output file if the user creates (under the DOS) an output file of length 0 before entering the program. As an example:

```
CR OFILE 0
GO MAKRO
```

Respond to the output file query with OFILE. PDS will update the directory entry.

It is generally not possible for PDS to predict the required output file size before disk operations commence. If the user elects to direct disk output to an existing file, he must ensure that the file size is sufficient to contain the output. PDS will cease disk operations with a 'NO ROOM' message when the existing output file is full.

BRINGING UP PDS

1. Write protect the PDS diskette before attempting to use it.
2. Make a working copy of the PDS diskette using the RD and WR commands of the DOS.
3. Store the original PDS diskette as a master backup copy.
4. Read the entire PDS documentation.
5. Several components of PDS are furnished in relocatable form to be placed at a convenient location in memory. The general procedure for making a working copy of these modules is:
 - a. Execute the relocating loader KWIKABS (see next page).
 - b. Identify the module to be loaded and the load address.
 - c. At completion of relocation, create a disk file and save the memory image of the relocated module. Set the file type = 1.
 - d. The relocatable module may be deleted to save disk space. The original version is always available on the master back-up diskette.
6. Practice using each of the components of PDS.
7. Suggestions and comments on the PDS documentation or programs are welcome.
8. PDS diskettes furnished for double density disk systems are written in single density and must be converted to double density before use. Consult the North Star documentation for instructions on effecting this conversion.

RELOCATING LOADERS KWIK AND KWIKABS

The KWIK loader is furnished in relocatable form on disk file KWIK and in absolute form on disk file KWIKABS. Entry to the absolute module is at DOS + A00H.* These two forms are furnished to allow the user to bootstrap the loader to any convenient memory location. The bootstrap procedure utilizes KWIKABS to relocate KWIK to the desired execution address. The procedure is as follows:

```
GO KWIKABS           Enter
INPUT FILE           File query
KWIK
LOAD ADDRESS
xxyy                 Desired RAM location
```

At completion, KWIKABS returns control to the warm start entry. The user should then save the memory image just created:

```
CR UKWIK 4
TY UKWIK 1 xxyy
SF UKWIK xxyy        xxyy is the previously defined RAM location.
```

The KWIK loader is subsequently accessed by GO UKWIK. (See MAKRO for discussion of KWIK.) The KWIK loader supports an optional offset address. Response to the load address query may take one of two forms: hexad or hexad,offset. The offset value is added to the execution address to determine the memory load address. Thus, code to be executed at E000H, with an offset of 3000H, is placed into memory at E000 + 3000 = 1000H.

LINKAGE EDITOR

The linkage editor is furnished in relocatable form as disk file LINKED. Either KWIKABS or the previously generated UKWIK loader can be used to generate an executable module of LINKED. The procedure is as follows:

```
GO KWIKABS
INPUT           File query
LINKED
LOAD ADDRESS
xxyy           Desired RAM address
```

At completion:

```
CR ULINK 6
SF ULINK xxyy
TY ULINK 1 xxyy
```

The linkage editor is then accessed by

```
GO ULINK
```

Library files are expected to reside on the drive containing the object file and may contain names of no more than five characters (10 for LINGLINK). If the file is not found a "?" prompt is issued, allowing the file name and drive to be re-entered. The North Star version of LINKED does not generate an object disk file. A RAM area after LINKED must be reserved for loader tables.

*000H in double density version.

RELOCATABLE DEBUG

DEBUG is furnished in relocatable form to be positioned at a convenient memory location. The relocation may be performed with KWIKABS or the user-developed loader UKWIK. Relocation of DEBUG is performed via the following sequence:

GO UKWIK	
INPUT FILE	File query
DEBUG	
LOAD ADDRESS	
xyyy	Desired RAM address

At completion:

```
CR UDEBUG 16
TY UDEBUG 1 xyyy
SF UDEBUG xyyy
```

Subsequent access to DEBUG is made via

```
GO UDEBUG
```

EDIT DISK FILES

EDIT relies upon the NORTH STAR disk operating system for the creation of disk space, the transfer of file contents to and from memory, and the console character input/output operations.

Upon initial entry, EDIT requests the name of the input text file -- the file to be modified. To create a new file, the user should respond to the INPUT query with the @. EDIT is thus cautioned to ignore any commands to read from disk. At any time the user may open a new input disk file (closing any existing input file).

Text material is transferred to memory in blocks of one sector (256 characters). The user may transfer as many sectors to memory as available space will allow. EDIT will not allow memory overflow. At termination, EDIT transfers to the output file any information still residing in the input file. The user may truncate the input file, however, by opening a new input file and responding to the INPUT query with @.

The output file is the repository for the processed textual material. Text is transferred to the output file in one-sector blocks. The name of the output file is given to EDIT in response to the OUTPUT query. If the file name is not found in the file directory, EDIT issues a '?' prompt. A new output file may be defined by re-entering EDIT at start + 2DH.

ASMB MEMORY FILES

The ASMB editor/assembler resides in memory immediately after the DOS. In the standard configuration, the memory region from 20000H up to 50000H* is reserved for the DOS, ASMB, and assembler tables. Neither source nor object files can be located within this region without damage to the programs.

* 53000H in double density version.

MAKRO EXECUTION

MAKRO requests a pass option before the assembly. The pass parameter nnn controls generation of the OBJECT file and assembly listing. The three least significant bits independently control assembler options.

Bit 0 controls the extent of the assembly. If Bit 0 = 0, the assembler skips pass 2, and neither an object file nor pass 2 diagnostics are available. This option is used to make a quick check of the source file.

Bit 1 controls the assembly listing. If Bit 1 = 0, only assembly diagnostics are generated.

Bit 2 controls the generation of the object file. If Bit 2 = 0, no object file is created.

Bit 3 controls the output device.

Assembly is normally performed with one of the pass options:

- 1 or A: No object file, pass 1 and 2 diagnostics only.
- 5 or E: Object file, pass 1 and 2 diagnostics only.
- 7 or G: Object file, full assembly listing.
- ? or 0: Object file, full listing to output device 1.

NOTE: A dummy output file must be defined even for cases in which no object code is to be written to disk.

Pressing Control-C when entering file names to MAKRO returns control to the DOS.

SPECIAL NOTE TO Z-80 OWNERS

The entire PDS package was written to be fully operational on machines using either the 8080 or Z-80 processor. As a result, one byte must be changed in DEBUG to display the additional Z-80 registers.

After generating an executable image of DEBUG at memory location xxyy (as discussed previously) the user must modify one program byte to display the Z-80 index registers.

Change memory location:	xxyy + 229H
From:	06
To:	08

DEBUG can be used to effect this change. After completing the relocation, but before saving the relocated file, perform the necessary modification.

SAMPLE ASMB OPERATION

```

>GO ASMB
ASMB DEVELOPMENT SYSTEM
F /TEST/5300
TEST 5300 5300
0010 LABEL: INX H
   DAD B
   ORA A
   END

```

Create memory file

> typed after line number, but not echoed

Auto line mode

< typed after carriage return

Print formatted listing

```

P
0010 LABEL INX H
0011      DAD B
0012      ORA A
0013      END

```

A F000

F000 23

F001 09

F002 87

F003

SYMBOL TABLE

LABEL F000

W

FILE

SAVE WRITTEN

B>LI

DOS	4	10	0
MAKRO	14	32	1 2A00
EDIT	46	11	1 2A00
END	168	0	0
SAVE	170	1	0
ASMB	57	25	1 2A00
DEBUG	82	55	0
KWIKABS	137	3	1 2A00
KWIK	140	15	0
LINKED	155	13	0

Assemble file

```

0010 LABEL ,INX H
0011      DAD B
0012      ORA A
0013      END

```

Assembly listing

Write source to disk

Disk operation completed

Source file

MEMORY SIZE

MAKRO and EDIT search memory to determine the highest available contiguous RAM address. In systems for which this is undesirable, the user may patch these programs to set a limit on the available memory.

MAKRO

MAKRO searches for memory top in a loop near the entry point. The code is:

```
2A49      MVI A,0AAH
          MTLP: INR H
              MOV M,A
              CMP M
2A4E      JZ MTLP
              DCX H
2A52      SHLD MTOP
```

The 3 bytes at 2A4E should be changed to

```
21 xx yy  (LXI H,MTOP)
```

where xx yy is the byte-reversed RAM limit.

EDIT

EDIT calls a subroutine to determine available memory. The call is:

```
2A13     LXI H,1B1BH
          SHLD THERE
2A19     CALL MEMTOP (2BCC)
```

The loop at MEMTOP is:

```
2BCC     LXI H,TEXT
2BD1     MVI A,0AAH
2BD1     MTLP: INR H
              MOV M,A
              CMP M
2BD4     JZ MTLP
              DCX H
2BDC     SHLD MTOP
```

The 3 bytes at 2BD4 should be changed to

```
21 xx yy
```

as done in the MAKRO patch.

NOTE: Entry point to double density version is 2D00H.

Addresses for the memory size patches to MAKRO and EDIT are given for the standard DOS at 2000H. DEBUG should be used to disassemble the code at the given locations before making any changes. Minor program modifications may alter the loop positions slightly.

The corrected versions of MAKRO and EDIT should be saved on disk.

SCROLLING PROGRAM OUTPUT

The two PDS assemblers and the G command of EDIT allow the output to be scrolled. Pressing the space bar will freeze the display; any other key will resume scroll.

This feature relies upon the non-standard Control-C detect routine in the DOS. The programs call the Control-C routine and expect the key pressed, if any, to return in the accumulator. If a blank is returned, the programs call character-in to wait for another key to be pressed before resuming operation.

A S M B

A disk-based assembler/editor
for the development of small to
medium size assembly language
programs.

The combination ASMB/DEBUG provides
an interactive environment for
assembly language program development.

Copyright 1978

Allen Ashley
395 Sierra Madre Villa
Pasadena, CA 91107

(213) 793-5748

INTRODUCTION

ASMB is a powerful disk-based editor/assembler system for program development on a Z80 microcomputer. Structurally and operationally similar to the program development packages SP-1 and ESP-1, ASMB offers more extensive editing and assembling features while extending the instruction assembly to the entire Z80 instruction set.

ASMB includes all the features necessary for the creation, modification and storage of assembly language programs. Departing from the cumbersome ZILOG assembly language, ASMB features instructions mnemonics similar to the more widely familiar INTEL set. Indeed, mnemonics for the 8080 subset of the Z80 instruction set are identical to the standard INTEL format. Users familiar with INTEL assembly language will appreciate the treatment of the Z80 instruction superset as a logical and syntactical extension of the INTEL instructions.

The ASMB program development system is an ideal companion to the more powerful MAKRO assembler. Small program modules are more easily and rapidly developed with the unified assembler/editor than the two-stage process of MAKRO/EDIT. The fully tested program modules can be converted to MAKRO source form by a single EDIT command. These source modules can then be saved as a source library for MAKRO.

ASMB is itself written entirely in the 8080 instruction subset, and is therefore operational on either 8080 or Z80 machines. ASMB can thus serve as a two-way cross assembler, assembling 8080 source programs on a Z80 machine, or Z80 object programs on an 8080 machine. The versatility and power of ASMB make it an ideal program development system for either those presently owning a Z80 machine or those anticipating a future expansion of their present 8080 machine to the more powerful Z80 processor.

An example of ASMB use is given in Section 1.

ASMB ORGANIZATION

The ASMB program development system consists of a combination text editor, assembler, and system executive for the creation and modification of Z80 assembly language programs.

The system executive is responsible for handling all input/output operations, invoking the editor or assembler, and dealing with the disposition of source and object files in central memory.

The text editor is responsible for the creation and modification of source programs within the memory file area. The text editor is line-oriented in that editing consists of entering or deleting source lines identified by ascending line numbers. The editor features automatic line numbering, line renumbering, moderately free-form source input, well-formatted source output, and a unique mini-editor for the modification of source code lines.

The assembler performs a two-pass translation of source to object code. The assembler includes the powerful feature of conditional assembly. Instruction mnemonics are logically and syntactically identical to the INTEL assembly language. The assembler is file-oriented with up to six source files simultaneously residing in memory. Optional symbol communication between files enables a moderate block structure development.

The concept and structure of ASMB were strongly influenced by Software Package #1. Assembly language source programs are maintained in source files under control of the system executive. Source files are created and deleted by commands to the system executive. Source code is entered into the source files under control of the editor, and the assembler can be directed to translate the source file to object code anywhere in memory.

EXECUTIVE COMMANDS

COMMAND FORMAT

Executive commands consist of a single letter identifier, together with an optional modifier character, and one or two hexadecimal parameters. The command character(s) must be separated from any numerical parameters by a single blank. Numerical parameters are likewise separated by a blank.

In the following, hexadecimal parameters are indicated by the sequence nnnn or mmmm while an optional character modifier is indicated by a lower-case c. Unless otherwise noted, the modifier c is a device control character (0-7) which will be present in the accumulator for all subsequent console I/O.

All command lines are terminated by a carriage return.

COMMAND LIST



F /NAME/ (Generic command; specific examples below)	Generic file control command. The file control command enables the user to create or destroy source files. Each source file is identified by a file NAME of up to five characters. The file name must be delimited by slashes. The opening slash must be separated by a blank from the command characters. The hexadecimal parameter nnnn and the modifier character are optional. There is no relation between memory file NAME and any disk file.
F /NAME/nnnn	Opens a source file NAME, starting at memory location nnnn, making NAME the active file. Any previously active files are maintained.
F /OTHER/	Recall previously active file, OTHER, making it the currently active file. Note the hexadecimal parameter is absent.
F /ERASE/0	Delete file named ERASE, freeing memory space for a new source file.
F	Display the currently active file parameters, file name, starting and ending memory locations.
FS	Display the file parameters of all memory files.
W	Write the currently active source file to disk. The executive will respond with the query FILE. The user must then type the disk file to receive the source.
R	Read source code from disk into the currently active memory file. The executive responds with the FILE query.
C n	Append a disk file to the currently active memory file, renumbering all source code lines by the increment n. Improperly formed disk operations, disk read errors, or insufficient disk file capacity result in the DISK ERROR diagnostic.

D nnnn mmmmm Delete lines numbered nnnn up to and including mmmmm from the source file. If mmmmm is omitted only nnnn is deleted.

B (BYE) Return to disk operating system.

I  Initialize the system, clearing all source files. The initialization is automatically performed upon initial entry. No lines of source code can be entered until a new source file has been defined.

Pc nnnn Print a formatted listing of the current source file, starting at line number nnnn.

Lc nnnn Print an unformatted listing, suppressing line numbers, of the current source file.
The optional modifying character, c, can be an ASCII digit in the range 0 - 7. The numerical value of this modifier will be present in the accumulator for all subsequent I/O, or until redefined by the user. The value is initialized to zero.

G nnnn Execute at location nnnn. A user program may return to the system executive by a simple return statement.

U Execute at location D000. This command is reserved for entry to the DEBUG control system.

A^c nnnn mmmmm  Assemble the current source file using implied origin (ORG) nnnn and place resulting object code into memory starting at location mmmmm. The second parameter is optional; if absent, the object code is placed into memory at nnnn.

AS  Mark existing symbol table for future global reference. (Save symbol table resulting from last assembly.) This command must follow an assembly: a symbol table must have been generated.

AE nnnn mmmmm Assemble, as above, displaying only source code lines containing an assembler diagnostic.

AK Release (kill) the global symbol table.

AT Print symbol table resulting from previous assembly.

E nnnn

Enter the mini-editor to edit the currently active source file beginning at line nnnn.

The mini-editor enables the user to scroll through the source file, changing source lines on the fly.

Upon entry, the mini-editor displays source line nnnn or the first source line if nnnn is omitted. The mini-editor then awaits keyboard input. Depressing any key except ESCAPE (1BH) advances the file pointer to display the next successive line. The escape key allows the user to re-enter the source line starting at character position two. (At the label field, no line number is required.) The user-entered line, terminated by carriage return, then overlays the old line. The mini-editor cannot insert new source lines into the file. Return to system executive via Control C.

E /STRNG/

Enter the mini-editor to edit the currently active source file beginning at the first occurrence of character string STRNG. The string may be at most five characters long and may contain no blanks. The string search is operable for the P and L commands as well.

N nn

Renumber source lines, starting at nn and incrementing by nn. The value nn is a decimal parameter.

There is space in the ASMB command table for five additional user commands. Available space starts after the 55 00 D0 byte string. New commands must be entered in the format

Command character, byte-reversed branch address

For each such command entered, the command count must be increased.

Search for the byte string 06 0E 3E 01 and increase the byte 0E for each new command entered. A hex parameter, if present, is passed to the user routine in the DE registers. A second hex parameter can be passed in the BC registers. The user routine can re-enter ASMB via a RET instruction.

EDITOR

Source lines are entered into the currently active source file under control of the file editor. The system executive recognizes a source line by a four-digit decimal line number, which must precede every line in the source file. Modifications to the source file consist of one or more whole lines. Lines may be deleted by the D control command. Lines may be modified by retyping the line number and entering the new source line. The editor adjusts the source file to accommodate line length without any wasted file space. Character deletion is accomplished by the underline (5F) key.

Source program lines consist of a four-digit line number followed by a terminating blank. The first character of the source line may contain identifiers '*' or ';'. These identifiers proclaim the entire line to be a comment. The label field of the source line must be separated by exactly one blank from the line number. Identifying labels can be from one to five characters long and may contain no special characters. The operation field must be separated from the label field by one or more blanks. The operand field, if present, must be separated from the operation by a single blank. Two blanks following the last operand separate the comment field, which should start with a semicolon. Source lines may be up to 72 characters in length.

The user can invoke automatic line numbering for lines entered into the source file. In the automatic mode, line numbers are incremented by one from the starting value. Automatic line numbering is initiated by entering the starting line number followed by > (greater than). Subsequent entries begin in character position two. The automatic mode is exited by typing < (less than) following the carriage return for the last source line. Failure to properly exit the automatic mode can result in erroneous source lines. Lengthy insertions can be made into an existing source file by renumbering the file before entering the automatic mode. 

The mini-editor allows text lines in the source file to be modified. When under control of the mini-editor, typing the Escape key switches from the scroll mode to the modify mode. Editing of the source line begins at the first character of the label field. Characters typed in under the modify mode are used to build the new source line. The old source line can be used as a model for generating the new source line: characters can be retrieved from the old line and placed in the new line. In the modify mode, the following control characters are recognized:

- CONTROL-A Fetch the next character from the old line and place it in the new line.
- CONTROL-Z Delete the next character from the old line.
- CONTROL-Q Back up one character in both the old and new lines.
- CONTROL-G Transfer the remainder of the old line to the new line.
- CONTROL-S Reads a character from the console, and transfers all characters from the old line up to, but not including, the input character.
- CONTROL-Y An insert toggle. Between successive toggles, input characters are inserted into the new line.

Any other characters typed in under the modify mode are entered into the new line, overriding the corresponding character from the old line.

ASSEMBLER OPERATION

The assembler operates upon the currently active source file only. The source file consists of a sequence of source lines composed of the four fields: label, operation, operand, and comment.

The label field, if present, must start in the second character position after the line number. Entries present in the label field are maintained in a symbol table. These entries are assigned a value equal to the program counter at the time of assembly, except that for the SET and EQU pseudo operations the variable defined by the label field is assigned the value of the operand field. The variables defined by the label field can be used in the operand field of other instructions either as data constants or locations.

The operation field, separated from the label field by one or more blanks or a colon, cannot appear before the third character following the line number. Entries in the operation field must consist of either a valid Z80 instruction or one of the several pseudo-operations.

The operand field, separated by a blank from the operation field, consists of an arithmetic expression containing one or more program variables, constants, or the special character \$ connected by the operators + or -. Evaluation of the operand field is limited to a left to right scan of the expression, using 16 bit integer arithmetic. Operations requiring multiple operands (e.g., MOV A,B or BIT 3,IX,4) expect the operands to be separated by a comma.

The special operand \$ refers to the program counter at the start of the instruction being assembled.* The program variable \$ can be used as any other program variable except that its value changes constantly throughout assembly. The location counter \$ allows the user to employ program relative computations.

Assembler constants may be either decimal or hexadecimal character strings. Valid hexadecimal constants must begin with a decimal digit, possibly 0, and be terminated by the suffix H.

* NOTE: Some assemblers interpret \$ as the start of the next instruction.

REGISTER MNEMONICS

All of the Z80 registers have been assigned predefined mnemonics. These assignments agree with those given by INTEL and ZILOG.

The predefined register set is defined as:

<u>Register</u>	<u>Definition</u>	<u>Value</u>
A	Accumulator	7
B	8 or 16 bit	0
C	8 bit	1
D	8 or 16 bit	2
E	8 bit	3
H	8 or 16 bit	4
L	8 bit	5
M	Memory Indirect (HL)	6
SP	Stack Pointer	6
PSW	Program Status Word	6
IX	16 bit Index	none x
IY	16 bit Index	none x
RF	Refresh Register	none x
IV	Interrupt Vector	none x

These register assignments may not be redefined.

ASSEMBLY LANGUAGE

As a consequence of favoring the INTEL mnemonic set over that of ZILOG, the Z80 instruction superset has been invented. One consideration in the definition of instruction mnemonics is standard assembly language convention. In the instruction mnemonics which follow

pp qq refers to an arbitrary 16 bit datum;
 yy refers to an arbitrary 8 bit datum;
 d refers to a Z80 displacement except for relative jumps;
 R refers to an 8 bit register (A, B, C, D, E, H, L, M)
 RP refers to a 16 bit register pair (B, D, H, SP)
 QP refers to a 16 bit register pair (PSW, B, D, H)

<u>MNEMONIC</u>	<u>ZILOG</u>	<u>REMARKS</u>
<u>8 BIT LOAD</u>		
MOV R,R	LD R,R	Register to register (to, from)
MOV R,IX,d	LD R,(IX+d)	Register indirect (R≠M)
MOV R,IY,d	LD R,(IY+d)	"
MOV IX,d,R	LD (IX+d),R	Memory indirect (R≠M)
MOV IY,d,R	LD (IY+d),R	"
MOV A,IV	LD A,I	Fetch interrupt vector
MOV A,RF	LD A,R	Fetch refresh register
MOV IV,A	LD I,A	Load interrupt vector
MOV RF,A	LD R,A	Load refresh register

<u>ACCUMULATOR LOAD/STORE</u>		
LDA pp qq	LD A,(nn)	Accumulator direct
LDAX B	LD A,(BC)	Accumulator extended
LDAX D	LD A,(DE)	"
STA pp qq	LD (nn),A	Accumulator direct
STAX B	LD (BC),A	Accumulator extended
STAX D	LD (DE),A	"

<u>8 BIT LOAD IMMEDIATE</u>		
MVI R,yy	LD R,n	Register immediate
MVI IX,d,yy	LD (IX+d),n	Memory indirect immediate
MVI IY,d,yy	LD (IY+d),n	"

MNEMONICZILOGREMARKS

16 BIT LOAD/STORE RP = B, D, H, SP

QP = PSW, B, D, H

LXI RP,pp qq
LXI IX,pp qq
LXI IY,pp qqLD RP,nn
LD IX,nn
LD IY,nn

Extended immediate

LHLD pp qq
LBCD pp qq
LDED pp qq
LIXD pp qq
LIYD pp qq
LSPD pp qqLD HL,(nn)
LD BC,(nn)
LD DE,(nn)
LD IX,(nn)
LD IY,(nn)
LD SP,(nn)

Extended indirect load

SHLD pp qq
SBCD pp qq
SDED pp qq
SIXD pp qq
SIYD pp qq
SSPD pp qqLD (nn),HL
LD (nn),BC
LD (nn),DE
LD (nn),IX
LD (nn),IY
LD (nn),SP

Extended indirect store

SPHL
SPIX
SPIYLD SP,HL
LD SP,IX
LD SP,IY

Set stack pointer

PUSH QP
PUSH IX
PUSH IYPUSH QP
PUSH IX
PUSH IY

To stack

POP QP
POP IX
POP IYPOP QP
POP IX
POP IY

From stack

EXCHANGE, BLOCK TRANSFER, AND SEARCHXCHG
EX
EXX
XTHL
XTIX
XTIYEX DE,HL
EX AF,AF'
EXX
EX (SP),HL
EX (SP),IX
EX (SP),IY

Exchange

LDI
LDIR
LDD
LDDRLDI
LDIR
LDD
LDDR

Transfer

CPD
CPDR
CPII
CPIRCPD
CPDR
CPI
CPIR

Search

<u>MNEMONIC</u>	<u>ZILOG</u>	<u>REMARKS</u>
<u>8 BIT ARITHMETIC AND LOGICAL</u>		
ADD R	ADD R	Add register
ADI yy	ADD A,yy	Add immediate
ADD IX,d	ADD (IX+d)	Add indirect
ADD IY,d	ADD (IY+d)	
ADC R	ADC R	Register with carry
ADC IX,d	ADC (IX+d)	Memory indirect with carry
ADC IY,d	ADC (IY+d)	
ACI yy	ADC n	Immediate with carry
SUB R	SUB R	Subtract Register
SUB IX,d	SUB (IX+d)	Subtract memory indirect
SUB IY,d	SUB (IY+d)	
SBB R	SBC R	Register with carry
SBB IX,d	SBC (IX+d)	Memory indirect with carry
SBB IY,d	SBC (IY+d)	
ANA R	AND R	Logical and register
ANA IX,d	AND (IX+d)	Memory indirect
ANA IY,d	AND (IY+d)	
ORA R	OR R	Logical OR register
ORA IX,d	OR (IX+d)	Memory indirect
ORA IY,d	OR (IY+d)	
XRA R	XOR R	Exclusive OR register
XRA IX,d	XOR (IX+d)	Memory indirect
XRA IY,d	XOR (IY+d)	
CMP R	CP R	Register compare
CMP IX,d	CP (IX+d)	Memory indirect
CMP IY,d	CP (IY+d)	
INR R	INC R	Register increment
INR IX,d	INC (IX+d)	
INR IY,d	INC (IY+d)	
DCR R	DEC R	Register decrement
DCR IX,d	DEC (IX+d)	
DCR IY,d	DEC (IY+d)	
ANI yy	AND yy	Accumulator immediate
XRI yy	XOR yy	
CPI yy	CP yy	
ORI yy	OR yy	
SUI yy	SUB yy	
SBI yy	SBC A,yy	

MNEMONICZILOGREMARKSGENERAL PURPOSE ARITHMETIC AND CPU CONTROL

DAA	DAA	Decimal adjust accumulator
CMA	CPL	Complement accumulator logical
NEG	NEG	Negate accumulator
CMC	CCF	Complement carry flag
STC	SCF	Set carry flag
NOP	NOP	No operation
HLT	HALT	HALT CPU
DI	DI	Disable interrupts
EI	EI	Enable interrupts
IM 0	IM 0	Set interrupt mode
IM 1	IM 1	
IM 2	IM 2	

16 BIT ARITHMETIC GROUP RP = B, D, H, SP

DAD RP	ADD HL,RP	16 bit add
CAD RP	ADC HL,RP	16 bit add with carry
SBC RP	SBC HL,RP	16 bit subtract with carry
DAD IX,RP	ADD IX,RP	16 bit add register pair to IX (RP ≠ H or IY)
DAD IY,RP	ADD IY,RP	16 bit add register pair to IY (RP ≠ H or IX)
INX RP	INC RP	16 bit increment
INX IX	INC IX	
INX IY	INC IY	
DCX RP	DEC RP	16 bit decrement
DCX IX	DEC IX	
DCX IY	DEC IY	

<u>MNEMONIC</u>	<u>ZILOG</u>	<u>REMARKS</u>
<u>ROTATE AND SHIFT GROUP R = B, C, D, E, H, L, M, IX+d, IY+d</u>		
RLC	RLCA	Accumulator left circular
RAL	RLA	Left circular through carry
RRC	RRCA	Accumulator right circular
RAR	RRA	Right circular through carry
SLC R	RLC R	Register left circular
SLC M	RLC (HL)	Memory left circular
SLC IX,d SLC IY,d	RLC (IX+d) RLC (IY+d)	Left circular memory indirect
RL R	RL R	Register left through carry
SRC R	RRC R	Register right circular
RR R	RR R	Register right through carry
SLA R	SLA R	Left linear bit 0 = 0
SRA R	SRA R	Right linear bit 7 = extended
SRL R	SRL R	Right linear bit 7 = 0
RLD	RLD	Left decimal
RRD	RRD	Right decimal

<u>MNEMONIC</u>	<u>ZILOG</u>	<u>REMARKS</u>
<u>BIT MANIPULATION</u> <u>b = bit number</u> <u>$0 \leq b \leq 7$</u>		
BIT b,R	BIT b,R	Zero flag = bit b of register R
BIT b,M	BIT b,(HL)	
BIT b,IX,d	BIT b,(IX+d)	
BIT b,IY,d	BIT b,(IY+d)	
STB b,R	SET b,R	Set (1) bit b of register or
STB b,M	SET b,(HL)	memory
STB b,IX,d	SET b,(IX+d)	
STB b,IY,d	SET b,(IY+d)	
RES b,R	RES b,R	Reset (\emptyset) bit b of register or
RES b,M	RES b,(HL)	memory
RES b,IX,d	RES b,(IX+d)	
RES b,IY,d	RES b,(IY+d)	

<u>INPUT/OUTPUT GROUP</u>	<u>P = port number</u>	<u>R = register</u>
IN P	IN A,(P)	Input to accumulator
CIN R	IN R,(C)	Register R from port (C) (R \neq M)
INI	INI	Input and increment
INIR	INIR	Repeated input and increment
IND	IND	Input and decrement
INDR	INDR	Repeated input and decrement
OUT P	OUT (P),A	Output accumulator
COU R	OUT (C),R	Register R to port (C) (R \neq M)
OUTI	OUTI	Output and increment
OUTIR	OUTIR	Repeated output and increment
OUTD	OUTD	Output and decrement
OUTDR	OUTDR	Repeated output and decrement

MNEMONICZILOGREMARKS

JUMP GROUP V = location (16 bit) dest = destination (±128 bytes displacement)

JMP V	JP V	Jump
JNC V	JP NC,V	No carry
JC V	JP C,V	Carry
JNZ V	JP NZ,V	Not zero
JZ V	JP Z,V	Zero
JPO V	JP PO,V	Parity odd
JPE V	JP PE,V	Parity even
JP V	JP P,V	Positive
JM V	JP M,V	Negative
JR dest	JR d	Jump relative
JRC dest	JR C,d	Carry
JRNC dest	JR NC,d	No carry
JRZ dest	JR Z,d	Zero
JRNZ dest	JR NZ,d	Not zero
PCHL	JP (HL)	Branch to location in HL
PCIX	JP (IX)	Branch to IX
PCIY	JP (IY)	Branch to IY
DJNZ dest	DJNZ,d	Decrement and jump relative if not zero

MNEMONICZILOGREMARKSCALL AND RETURN GROUP

V = address

CALL V	CALL V	Subroutine transfer
CNC V	CALL NC,V	No carry
CC V	CALL C,V	Carry
CNZ V	CALL NZ,V	Not zero
CZ V	CALL Z,V	Zero
CPE V	CALL PE,V	Parity even
CPO V	CALL PO,V	Parity odd
CP V	CALL P,V	Positive
CM V	CALL M,V	Negative
RET	RET	Return
RNC	RET NC	No carry
RC	RET C	Carry
RNZ	RET NZ	Not zero
RZ	RET Z	Zero
RPE	RET PE	Parity even
RPO	RET PO	Parity odd
RP	RET P	Positive
RM	RET M	Negative
RETI	RETI	Return from interrupt
RETN	RETN	Return from non-maskable interrupt
RST n	RST n	Restart

PSEUDO OPERATIONS

<u>ASSEMBLER</u>	<u>PSEUDO OPERATIONS</u>	expr = arithmetic expression
ORG expr	Define program counter to nnnn	
DS expr	Reserve n bytes of storage	
DW expr	16 bit datum definition	
DB expr	8 bit datum or ASCII character string definition. The operand may be an ASCII character string enclosed in single quotation marks. Examples: DB 5,6,7 DB 'ASCII STRING',0DH,0AH	
EQU	The operand defined by the label field is set equal to the expression defined by the operand field. This operation is performed in pass one of the assembler and the variable definition is fixed by the first such definition encountered.	
SET	The operand defined by the label is set equal to the expression defined by the operand field. This operation is performed in both pass 1 and pass 2 and the replacement is effected upon every encounter.	
IF expr	expr is evaluated. If the result is zero the scanner skips to the next ENDIF, END, or end of file before resuming assembly. If the expression evaluates to any non-zero value, assembly proceeds. Operation is performed in both passes.	
ENDIF	Identifies the end of a conditional assembly block.	
END	Terminates assembly.	
USE operand	Allows program assembly to proceed with multiple location counters. The operation is skipped if the operand has not previously been defined; however, the definition can appear after the reference, to be used by pass 2. The USE operation is best explained by example.	

```

AORG  SET  0A000H
BORG  SET  0B000H
      USE  AORG;   SET code origin to AORG
      { code at 0A000H }
      USE  BORG;   SET value of AORG to PC
                        SET PC to BORG
      { code at 0B000H }
    
```

USE AORG; Resume code at end of previous
block which started at A000.

{ code }

USE BORG; Resume code at END of block
which started at B000.

ASSEMBLER ERRORS/DIAGNOSTICS

Assembler error and diagnostic messages consist of single character identifiers which flag some irregularity discovered either during pass 1 or pass 2 of the assembly. The single character precedes the line number of the formatted assembly listing.

- P Phase error: the value of the label has changed between the two assembly passes.
- L Label error: label contains illegal or too many characters, e.g., LB#1:
- U Undefined program variable.
- V Value error: the evaluated operand is not consistent with the operation e.g., MVI A, 1000H (not a valid 8 bit operand).
- S Syntax error e.g., MOV A+B
- O Opcode error, e.g. DCS B
- M Missing label field.
- A Argument error.
- R Register error.
- D Duplicate label error.

EXISTING SOURCE FILES

ASMB is compatible with programs generated under SP#1 or its many descendents, SCS 1,2, ESP-1, ALS-8, etc. These related source programs can be included in the ASMB disk system by the following procedure:

1. Load ASMB and create a memory file at a convenient memory location.
2. Exit from ASMB and load the existing source file into memory starting at the memory location defined in step 1.
3. Re-enter ASMB and examine the file with the P command.
4. Delete and re-enter the last line of the source code.
5. Save the memory file on disk via the W command.
6. EDIT will re-format the source file for MAKRO via the N command.

While all such files are compatible with ASMB, EDIT may be unable to effect the reformat. A failure may arise if EDIT does not encounter the ASMB end-of-file 01 (catastrophic).

MAKRO

An extraordinary disk-based macro assembler
for the development of large programs on
Z80 or 8080 machines.

Copyright 1978

Allen Ashley
395 Sierra Madre Villa
Pasadena, California 91107

(213) 793-5748

INTRODUCTION

MAKRO is a powerful disk-based macro assembler for the development of large programs whose source files may exceed available memory. Both the source and object files of MAKRO reside on disk, freeing all available memory for macro storage and the construction of symbol tables. MAKRO is an extraordinarily powerful development tool incorporating many features not commonly available. The assembler is a working tool which has evolved under the demands generated by its use.

Program development with MAKRO is a two-step process: the source file is created, modified and saved on disk using the text editor EDIT; MAKRO reads the source file and creates the corresponding object file.

MAKRO assembles all Z80 and 8080 instructions. Departing from the cumbersome ZILOG assembly language, MAKRO features instruction mnemonics which are logically and syntactically similar to the more widely familiar INTEL instruction set. Mnemonics for the 8080 subset of the Z80 instruction set are identical to those defined by INTEL, and users already familiar with INTEL assembly language will readily acquire the additional Z80 commands.

MAKRO is written entirely in the 8080 instruction set and is fully operational on either 8080 or Z80 machines. MAKRO can therefore serve as a two-way cross-assembler -- assembling 8080 programs on a Z80 machine or Z80 programs on an 8080 machine. The versatility and power of MAKRO make it an ideal development tool for those owning a Z80 machine or anticipating a future expansion of their 8080 machine to the more powerful Z80 processor.

MAKRO INPUT/OUTPUT

MAKRO is a two-pass assembler, reading the source file first to construct a symbol table, then generating the object file on the second pass.

Source code for MAKRO consists of the four fields: Label, Operation, Operand, and Comments.

- (1) A line starting with a semi-colon is interpreted as a comment.
- (2) Entries in the label field must be terminated by a colon. The label identifier starts with the first non-blank character and ends with the colon. The colon requirement applies to SET and EQU operations, and macro definitions.
- (3) If a label is present, the operation field begins with the first non-blank character after the colon.
- (4) If no colon (hence no label) is detected, the operation field begins with the first non-blank character.
- (5) A comment field must be preceded by a semi-colon. Trailing comments preceded by a double semi-colon ;; are tabbed to the right of the operand field. Comments are not allowed on source lines containing a macro call.
- (6) Source lines must be terminated by carriage return/line feed.

The MAKRO user must identify the origin of the object code by an ORG operation at the start of his source code. Failure to do so will result in the code being assembled at location 0.

The list output of MAKRO displays the program counter, object code, and a well-formatted source display. Horizontal tab sets align the label, operation and operand fields for all source lines. An alphabetized symbol table is presented at the conclusion of pass 2 of the assembly.

MAKRO utilizes all available memory after the load address. Program constants and assembler symbol tables reside in memory immediately after MAKRO. Macro text is stored at highest available memory. The region between is used for macro processing operations.

ASSEMBLER OPERATION

Entries present in the label field are maintained in a symbol table. These entries are assigned a value equal to the program counter at the time of assembly, except that for the SET and EQU pseudo-operations, the variable defined by the label field is assigned the value of the operand field. Entries created in the symbol table by the macro definition refer to the storage location assigned to the text of the macro body. The variables defined by the label field can be used in the operand field of other instructions either as data constants or locations.

The operation field is separated from the label field by the colon. If no label field is present, the operation field may begin anywhere on the line. Entries in the operation field must consist of either a valid Z80 instruction, one of the several pseudo-operations, or a previously defined macro.

The operand field, separated by a blank from the operation field, consists of an arithmetic expression containing one or more program variables, constants, or the special characters \$, @ or %, connected by valid operators. Evaluation of the operand field is performed using 16-bit integer arithmetic. Operations requiring multiple operands (e.g., MOV A,B or BIT 3,IX,4) expect the operands to be separated by a comma. Parameters passed in a macro call are separated by commas and terminated by a carriage return.

The special operand \$ refers to the program counter at the start of the instruction being assembled. (NOTE: some assemblers interpret \$ as the start of the next instruction.) The program variable \$ can be used as any other program variable except that its value changes constantly throughout assembly. The location counter \$ allows the user to employ program-relative computations.

MAKRO recognizes two other special operands. The @, when used as an operand, refers to the repetition counter index. The %, as an operand, refers to the number of actual parameters in the current macro call.

Assembler constants may be decimal, hexadecimal, octal, or binary. Valid hexadecimal constants must begin with a decimal digit, possibly 0, and be terminated by the suffix 'H.' Binary constants are terminated by 'B' and may contain only the digits 0 and 1. Octal constants are terminated by 'O' and may contain only the digits 0 - 7.

After completion of an assembly, MAKRO may not be re-entered.

REGISTER MNEMONICS

All of the Z80 registers have been assigned predefined mnemonics. These assignments agree with those given by INTEL and ZILOG.

The predefined register set is defined as:

<u>Register</u>	<u>Definition</u>	<u>Value</u>
A	Accumulator	7
B	8 or 16 bit	0
C	8 bit	1
D	8 or 16 bit	2
E	8 bit	3
H	8 or 16 bit	4
L	8 bit	5
M	Memory Indirect (HL)	6
SP	Stack Pointer	6
PSW	Program Status Word	6
IX	16 bit Index	none
IY	16 bit Index	none
RF	Refresh Register	none
IV	Interrupt Vector	none

These register assignments may not be redefined.

ASSEMBLY LANGUAGE

As a consequence of favoring the INTEL mnemonic set over that of ZILOG, the Z80 instruction superset has been invented. One consideration in the definition of instruction mnemonics is standard assembly language convention. In the instruction mnemonics which follow

pp qq refers to an arbitrary 16 bit datum;
 yy refers to an arbitrary 8 bit datum;
 d refers to a Z80 displacement except for relative jumps;
 R refers to an 8 bit register (A, B, C, D, E, H, L, M)
 RP refers to a 16 bit register pair (B, D, H, SP)
 QP refers to a 16 bit register pair (PSW, B, D, H)

<u>MNEMONIC</u>	<u>ZILOG</u>	<u>REMARKS</u>
<u>8 BIT LOAD</u>		
MOV R,R	LD R,R	Register to register (to, from)
MOV R,IX,d	LD R,(IX+d)	Register indirect (R≠M)
MOV R,IY,d	LD R,(IY+d)	"
MOV IX,d,R	LD (IX+d),R	Memory indirect (R≠M)
MOV IY,d,R	LD (IY+d),R	"
MOV A,IV	LD A,I	Fetch interrupt vector
MOV A,RF	LD A,R	Fetch refresh register
MOV IV,A	LD I,A	Load interrupt vector
MOV RF,A	LD R,A	Load refresh register

ACCUMULATOR LOAD/STORE

LDA pp qq	LD A,(nn)	Accumulator direct
LDAX B	LD A,(BC)	Accumulator extended
LDAX D	LD A,(DE)	"
STA pp qq	LD (nn),A	Accumulator direct
STAX B	LD (BC),A	Accumulator extended
STAX D	LD (DE),A	"

8 BIT LOAD IMMEDIATE

MVI R,yy	LD R,n	Register immediate
MVI IX,d,yy	LD (IX+d),n	Memory indirect immediate
MVI IY,d,yy	LD (IY+d),n	"

MNEMONICZILOGREMARKS

16 BIT LOAD/STORE RP = B, D, H, SP QP = PSW, B, D, H

LXI RP,pp qq	LD RP,nn	Extended immediate
LXI IX,pp qq	LD IX,nn	
LXI IY,pp qq	LD IY,nn	
LHLD pp qq	LD HL,(nn)	Extended indirect load
LBCD pp qq	LD BC,(nn)	
LDED pp qq	LD DE,(nn)	
LIXD pp qq	LD IX,(nn)	
LIYD pp qq	LD IY,(nn)	
LSPD pp qq	LD SP,(nn)	
SHLD pp qq	LD (nn),HL	Extended indirect store
SBCD pp qq	LD (nn),BC	
SDED pp qq	LD (nn),DE	
SIXD pp qq	LD (nn),IX	
SIYD pp qq	LD (nn),IY	
SSPD pp qq	LD (nn),SP	
SPHL	LD SP,HL	Set stack pointer
SPIX	LD SP,IX	
SPIY	LD SP,IY	
PUSH QP	PUSH QP	To stack
PUSH IX	PUSH IX	
PUSH IY	PUSH IY	
POP QP	POP QP	From stack
POP IX	POP IX	
POP IY	POP IY	

EXCHANGE, BLOCK TRANSFER, AND SEARCH

XCHG	EX DE,HL	Exchange
EX	EX AF,AF'	
EXX	EXX	
XTHL	EX (SP),HL	
XTIX	EX (SP),IX	
XTIY	EX (SP),IY	
LDI	LDI	Transfer
LDIR	LDIR	
LDD	LDD	
LDDR	LDDR	
CPD	CPD	Search
CPDR	CPDR	
CPII	CPI	
CPIR	CPIR	

<u>MNEMONIC</u>	<u>ZILOG</u>	<u>REMARKS</u>
<u>8 BIT ARITHMETIC AND LOGICAL</u>		
ADD R	ADD R	Add register
ADI yy	ADD A,yy	Add immediate
ADD IX,d	ADD (IX+d)	Add indirect
ADD IY,d	ADD (IY+d)	
ADC R	ADC R	Register with carry
ADC IX,d	ADC (IX+d)	Memory indirect with carry
ADC IY,d	ADC (IY+d)	
ACI yy	ADC n	Immediate with carry
SUB R	SUB R	Subtract Register
SUB IX,d	SUB (IX+d)	Subtract memory indirect
SUB IY,d	SUB (IY+d)	
SBB R	SBC R	Register with carry
SBB IX,d	SBC (IX+d)	Memory indirect with carry
SBB IY,d	SBC (IY+d)	
ANA R	AND R	Logical and register
ANA IX,d	AND (IX+d)	Memory indirect
ANA IY,d	AND (IY+d)	
ORA R	OR R	Logical OR register
ORA IX,d	OR (IX+d)	Memory indirect
ORA IY,d	OR (IY+d)	
XRA R	XOR R	Exclusive OR register
XRA IX,d	XOR (IX+d)	Memory indirect
XRA IY,d	XOR (IY+d)	
CMP R	CP R	Register compare
CMP IX,d	CP (IX+d)	Memory indirect
CMP IY,d	CP (IY+d)	
INR R	INC R	Register increment
INR IX,d	INC (IX+d)	
INR IY,d	INC (IY+d)	
DCR R	DEC R	Register decrement
DCR IX,d	DEC (IX+d)	
DCR IY,d	DEC (IY+d)	
ANI yy	AND yy	Accumulator immediate
XRI yy	XOR yy	
CPI yy	CP yy	
ORI yy	OR yy	
SUI yy	SUB yy	
SBI yy	SBC A,yy	

MNEMONICZILOGREMARKSGENERAL PURPOSE ARITHMETIC AND CPU CONTROL

DAA	DAA	Decimal adjust accumulator
CMA	CPL	Complement accumulator logical
NEG	NEG	Negate accumulator
CMC	CCF	Complement carry flag
STC	SCF	Set carry flag
NOP	NOP	No operation
HLT	HALT	HALT CPU
DI	DI	Disable interrupts
EI	EI	Enable interrupts
IM 0	IM 0	Set interrupt mode
IM 1	IM 1	
IM 2	IM 2	

16 BIT ARITHMETIC GROUP RP = B, D, H, SP

DAD RP	ADD HL,RP	16 bit add
CAD RP	ADC HL,RP	16 bit add with carry
SBC RP	SBC HL,RP	16 bit subtract with carry
DAD IX,RP	ADD IX,RP	16 bit add register pair to IX (RP ≠ H or IY)
DAD IY,RP	ADD IY,RP	16 bit add register pair to IY (RP ≠ H or IX)
INX RP	INC RP	16 bit increment
INX IX	INC IX	
INX IY	INC IY	
DCX RP	DEC RP	16 bit decrement
DCX IX	DEC IX	
DCX IY	DEC IY	

MNEMONICZILOGREMARKSROTATE AND SHIFT GROUP

R = B, C, D, E, H, L, M, IX+d, IY+d

RLC	RLCA	Accumulator left circular
RAL	RLA	Left circular through carry
RRC	RRCA	Accumulator right circular
RAR	RRA	Right circular through carry
SLC R	RLC R	Register left circular
SLC M	RLC (HL)	Memory left circular
SLC IX,d SLC IY,d	RLC (IX+d) RLC (IY+d)	Left circular memory indirect
RL R	RL R	Register left through carry
SRC R	RRC R	Register right circular
RR R	RR R	Register right through carry
SLA R	SLA R	Left linear bit 0 = 0
SRA R	SRA R	Right linear bit 7 = extended
SRL R	SRL R	Right linear bit 7 = 0
RLD	RLD	Left decimal
RRD	RRD	Right decimal

<u>MNEMONIC</u>	<u>ZILOG</u>	<u>REMARKS</u>
<u>BIT MANIPULATION</u> <u>b = bit number</u> <u>$0 \leq b \leq 7$</u>		
BIT b,R	BIT b,R	Zero flag = bit b of register R
BIT b,M	BIT b,(HL)	
BIT b,IX,d	BIT b,(IX+d)	
BIT b,IY,d	BIT b,(IY+d)	
STB b,R	SET b,R	Set (1) bit b of register or
STB b,M	SET b,(HL)	memory
STB b,IX,d	SET b,(IX+d)	
STB b,IY,d	SET b,(IY+d)	
RES b,R	RES b,R	Reset (0) bit b of register or
RES b,M	RES b,(HL)	memory
RES b,IX,d	RES b,(IX+d)	
RES b,IY,d	RES b,(IY+d)	

<u>INPUT/OUTPUT GROUP</u>	<u>P = port number</u>	<u>R = register</u>
IN P	IN A,(P)	Input to accumulator
CIN R	IN R,(C)	Register R from port (C) (R≠M)
INI	INI	Input and increment
INIR	INIR	Repeated input and increment
IND	IND	Input and decrement
INDR	INDR	Repeated input and decrement
OUT P	OUT (P),A	Output accumulator
COUT R	OUT (C),R	Register R to port (C) (R≠M)
OUTI	OUTI	Output and increment
OUTIR	OUTIR	Repeated output and increment
OUTD	OUTD	Output and decrement
OUTDR	OUTDR	Repeated output and decrement

MNEMONICZILOGREMARKS

JUMP GROUP V = location (16 bit) dest = destination (±128 bytes displacement)

JMP V	JP V	Jump
JNC V	JP NC,V	No carry
JC V	JP C,V	Carry
JNZ V	JP NZ,V	Not zero
JZ V	JP Z,V	Zero
JPO V	JP PO,V	Parity odd
JPE V	JP PE,V	Parity even
JP V	JP P,V	Positive
JM V	JP M,V	Negative
JR dest	JR d	Jump relative
JRC dest	JR C,d	Carry
JRNC dest	JR NC,d	No carry
JRZ dest	JR Z,d	Zero
JRNZ dest	JR NZ,d	Not zero
PCHL	JP (HL)	Branch to location in HL
PCIX	JP (IX)	Branch to IX
PCIY	JP (IY)	Branch to IY
DJNZ dest	DJNZ,d	Decrement and jump relative if not zero

MNEMONICZILOGREMARKS

CALL AND RETURN GROUP V = address

CALL V	CALL V	Subroutine transfer
CNC V	CALL NC,V	No carry
CC V	CALL C,V	Carry
CNZ V	CALL NZ,V	Not zero
CZ V	CALL Z,V	Zero
CPE V	CALL PE,V	Parity even
CPO V	CALL PO,V	Parity odd
CP V	CALL P,V	Positive
CM V	CALL M,V	Negative
RET	RET	Return
RNC	RET NC	No carry
RC	RET C	Carry
RNZ	RET NZ	Not zero
RZ	RET Z	Zero
RPE	RET PE	Parity even
RPO	RET PO	Parity odd
RP	RET P	Positive
RM	RET M	Negative
RETI	RETI	Return from interrupt
RETN	RETN	Return from non-maskable interrupt
RST n	RST n	Restart

PSEUDO OPERATIONS

<u>ASSEMBLER</u>	<u>PSEUDO OPERATIONS</u>	expr = arithmetic expression
ORG expr	Define program counter to nnnn.	
DS expr	Reserve n bytes of storage. The first and last bytes of the reserved storage area are modified. An unmodified reserved area can be created by ORG \$+SIZE.	
DW expr	16-bit datum definition.	
DB expr	8-bit data or ASCII character string definition. The operand may be an ASCII character string enclosed in single quotation marks. Examples: DB 5,0DH,'FILE' DB 'ASCII STRING',0DH	
EQU	The operand defined by the label field is set equal to the expression defined by the operand field. This operation is performed in pass 1 of the assembler and the variable definition is fixed by the last such definition encountered in pass 1.	
SET	The operand defined by the label is set equal to the expression defined by the operand field. This operation is performed in both pass 1 and pass 2 and the replacement is effected upon every encounter.	
IF expr	expr is evaluated. If the result is zero the scanner skips to the next ENDIF, END, or end-of-file before resuming assembly. If the expression evaluates to any non-zero value, assembly proceeds. Operation is performed in both passes. Read IF as "SKIP IF ZERO."	
NIF expr	expr is evaluated. If the result is not zero the scanner skips to the next ENDIF, END, or end-of-file before resuming assembly. Equivalent to NOT IF. Read NIF as "SKIP IF NOT ZERO."	
ENDIF	Identifies the end of a conditional assembly block.	
END expr	Terminates assembly. expr is an optional execution address to which the hex loader will branch after completion of the load.	

ASSEMBLERPSEUDO OPERATIONS

expr = arithmetic expression

USE operand

Allows program assembly to proceed with multiple location counters. The operation is skipped if the operand has not previously been defined; however, the definition can appear after the reference, to be used by pass 2. The USE operation is best explained by example:

```
AORG: SET 0A000H
BORG: SET 0B000H
      USE AORG;           SET code origin to AORG
      [ CODE AT 0A000H ]
      USE BORG;           SET value of AORG to PC
                          SET PC to BORG
      [ CODE AT 0B000H ]
      USE AORG;           Resume code at end of previous
                          block which started at A000.
      [ CODE ]
      USE BORG;           Resume code at END of block which
                          started at B000.
```

The USE instruction can be used to insert program data at the end of instruction code:

```
AFTR: SET LAST;         Not known on pass 1.
      ORG Start;        Somewhere.
      [ CODE ]
RESUM: SET $;           Remember where we are.
      USE AFTR
STRING: DB 'CHARACTERS'
      USE RESUM;        Resume in-line coding.
      [ CODE ]
      USE AFTR
      [ MORE DATA ]
      USE RESUM;        Continue
LAST: SET $
      END
```

MACROSignifies macro definition.

<u>ASSEMBLER</u>	<u>PSEUDO OPERATIONS</u>	expr = arithmetic expression
MACND	Signifies end of macro definition	
LOCAL	Signifies the start of an assembly block. All labels generated within a local block are confined to that block.	
LOCND	Signifies the end of an assembly block, global assembly resumes. LOCAL/LOCND assembly blocks allow temporary macro definitions.	
GOTO label	Directs assembler to skip forward to label before resuming assembly. If label is reached via a GOTO branch, the symbol will not be entered into the symbol table. If label is reached via a normal assembly sequence it is treated as an ordinary statement label. GOTO is used in conjunction with conditional assembly to effect complex assembly sequences. GOTO allows forward references only. An invalid label terminates the assembly pass.	
IFGZ expr;label	If expr evaluates to zero, the assembler branches forward to label; otherwise assembly continues.	
IFGNZ expr;label	If expr evaluates to non-zero, the assembler branches forward to label; otherwise assembly continues. Labels reached by IFGZ and IFGNZ branches are not entered into the symbol table. Note that label must be separated by a semi-colon from the end of expr.	
REPT expr	Repeat block. The value of expr determines the number of times the repeat block is executed.	
REPND	Defines the end of a repeat block. The portion of source code bracketed by REPT/REPND is assembled repeatedly.	
USR expr	Assembly-time branch to user routine. MAKRO branches to the address given by the value of expr. The user routine may utilize all registers. MAKRO may be re-entered by a return RET. Upon entry to the user routine, the zero flag is set for pass 1 of the assembly, and the DE registers contain the address, within MAKRO, at which assembly must resume. This pseudo-operation provides the means for controlling output.	
IFEQ STR1,STR2;LABEL	Branch to LABEL if character string STR1 is identical to STR2.	
IFNE STR1,STR2;LABEL	Branch to LABEL if character string STR1 is not identical to STR2.	

ASSEMBLERPSEUDO OPERATIONS

expr = arithmetic expression

IFNEG expr;LABEL	Branch to LABEL if expr results in a negative value.
IFDEF SYMBL;DEFND	Branch to DEFND if SYMBL has been entered in the symbol table.
LIST	Turns on full assembly listing, restoring any pass options.
NOLST	Turns off full assembly listing, retaining diagnostic and error messages.
COMPS STR1,STR2;LABEL	Branch to LABEL if character string 2 is greater than character string 1.
LINK FILENAME	Merges disk file FILENAME into the current assembly. The LINK pseudo-operation enables the assembly to include previously developed program modules.
INPUT	MAKRO allows the user to define program variables at assembly time. The INPUT pseudo-operation accepts an expression from the console input, evaluates that expression, and assigns the computed value to the variable defined by the label field.
XPAND	Display macro expansion (default case).
NOEXP	Suppress macro expansion.
APUSH expr	Places the value of expr on the internal assembly stack.
LABEL:APOP	Similar to SET pseudo-op except that value of LABEL is recovered from assembly stack. APUSH and APOP are primarily used within nested control macros as in FOR/NEXT loops. Such nesting requires that the starting address of FOR loops be recovered in reverse sequence by the following NEXT macros.
PAGE	Causes page eject (via form feed).
TITLE 'PAGE HEADING'	Causes corresponding heading to appear on subsequent pages of the assembly listing. If the TITLE field is empty, MAKRO will prompt the user during pass 2 for the page heading. The prompt option is exercised by terminating the TITLE pseudo-op with a carriage return.
SETQ expr	Sets internal label-generating assembly variable to value of expr. A question mark appearing in the label field is expanded as the character string representing the hex value defined by SETQ. This operation was implemented to allow communication between macros.

RELOCATION PSEUDO-OPERATIONS

The relocating assembler, MAKRO version AMA.2, additionally recognizes the following pseudo-operations or directives to the loader LINKED

- | | |
|------------------|---|
| LABEL:ENTRY | Loader directive which defines LABEL for reference in another (independent) assembly. |
| LABEL:EXTRN | Loader directive which defines LABEL as a point created in another assembly, which must be found by the linkage editor. |
| FILE:LIBRY | Loader directive which defines FILE as an object library within which one or more external references may be found. |
| LABEL:ABSNT expr | Loader directive which defines LABEL as a fixed location to be used as an external reference in another assembly. The ABSNT directive operates as an assembly EQUate which can be changed at load time. |

ASSEMBLER ERRORS/DIAGNOSTICS

Assembler error and diagnostic messages consist of single character identifiers which flag some irregularity discovered during either pass 1 or pass 2 of the assembly.

- P Phase error: the value of the label has changed between the two assembly passes.
- L Label error: missing operation field or invalid destination label.
- U Undefined program variable.
- V Value error: the evaluated operand is not consistent with the operation, e.g., MVI A, 1000H (not a valid 8-bit operand).
- S Syntax error, e.g., MOV A+B
- O Opcode error, e.g., DCS B
- M Missing label field.
- A Argument error.
- R Register error.
- D Duplicate label.

MAKRO CONDITIONAL ASSEMBLY

The conditional assembly features of MAKRO include

COMPS	String comparison
IFEQ	Character string equality
IFNE	Character string inequality
IFNEG	Branch on negative
IFDEF	Branch if defined symbol
IF	Skip if zero
NIF	Skip if not zero
ENDIF	Termination of conditional block
IFGZ	Branch to label if zero
IFGNZ	Branch to label if not zero
GOTO	Unconditional branch

These pseudo-operations enable the programmer to direct the assembly by performing assembly time computations. In the simplest application, conditional assembly allows a program to be written with a number of options, such as various input/output modes, with the desired array of options selected by program switches. A single source code module can thus be used for a variety of applications. More powerful application of conditional operations directs the assembly according to results generated during the assembly process. An example of this application is given in the discussion of macro processing.

The conditional assembly operations effect their branching upon the results of evaluating an arithmetic expression. The expression begins with the first non-blank character after the operation field and ends with a carriage return or semi-colon. The label directed branches IFGZ and IFGNZ include a destination field following the expression. A semi-colon must separate the destination from the expression. The destination field is terminated by a blank or carriage return. Branching is performed in a forward direction only, the assembler skipping over source code until the destination label or end-of-file is detected.

Treatment of the destination label in label-directed branches requires discussion. The general form is

```
Branch  expr; There
else    here

[ CODE ]

There:
```

If the branch condition is not satisfied, assembly proceeds in sequence with else, in which case the destination label (There) may be reached in the course of assembly. In this, the fall-through case, the destination label is treated as an ordinary statement label and is entered into the symbol table. However, if the branch condition is satisfied, the label is reached via a skip, and normal assembly proceeds with the first character following the colon at the destination. The destination label is not seen by the assembler.

The IF/ENDIF and NIF/ENDIF assembly blocks bracket portions of code which are conditionally assembled or disregarded. The IF block is disregarded if the corresponding expression evaluates to zero. The NIF block is disregarded if the expression evaluates to not-zero. Mnemonically, these conditions refer to the skip rather than the assembly.

Nested IF/NIF blocks cannot generally be assembled correctly. Consider blocks nested as

```
a  IF          expr1
b  IF          expr2
c  ENDIF       hopefully for the inner
   [ CODE ]    some code in here
d  ENDIF       hopefully for the outer
```

Assembly proceeds as follows:

expr1 is evaluated, the assembler skipping to the first ENDIF (c) if expr1 is zero. If expr1 is not zero, expr2 is evaluated, the assembler reaching the ENDIF (c) regardless of the results. It is seen that CODE is assembled regardless of the contents of either expression. The second ENDIF (d) is superfluous, and is ignored. There may be applications of such behavior, but the operation seems more likely to be a source of confusion. Complicated conditional branching is more easily and clearly generated by the label-directed operations.



A cautionary flag must be raised regarding conditional assembly. Phase changes of assembly variables (change in value between the two assembly passes) can result in a totally invalid assembly. If such phase changes cause the course of the assembler through the source code to differ for pass 1 and pass 2, the resulting assembly is almost certain to fail. You must remember that any and all branches performed in pass 1 must be repeated in pass 2.

The character string tests, IFNE and IFEQ, perform a character-by-character test of the first two parameter strings, conditionally effecting the branch upon the outcome of the comparison. The forms of these operations are:

```
IFEQ STR1,STR2;LABEL
IFNE STR1,STR2;LABEL
```

String 1 begins with the first non-blank character after the operation code and extends to the character preceding the comma. String 2 includes the character following the comma through that preceding the semi-colon.

Remember that the destination field must be preceded by a semi-colon and that the destination label vanishes if the branch is true.

IFNEG expr;LABEL

expr is evaluated. If the result is negative (15-bit signed arithmetic) the assembler branches to LABEL. IFNEG, IFGZ and IFGNZ can be combined to effect any computational branch.

IFDEF SYMBOL;LABEL

The symbol table is searched for symbol. If the entry is found, assembly skips to LABEL. IFDEF is used to provide automatic type declaration.

COMPS STR1,STR2;LABEL

A character-by-character comparison is made between STR1 and STR2. If STR2 is greater than STR1, assembly branches to LABEL. The COMPS pseudo-op is used to test parameter type in a macro call.

MAKRO MACRO CAPABILITY

INTRODUCTION TO MACROS

A macro can be considered an assembly language super-instruction with which the user can invoke many elementary assembly language statements with a single macro call. Users familiar with FORTRAN utilize a macro in the FORTRAN statement function. BASIC programs using the DEF FN operation capitalize upon an economical feature similar to a macro. The PL/1 pre-processing pass is a macro phase.

Assembly language programming is distinguished from such high level languages on the basis of the translation from the programmer-oriented language to the machine-oriented object code. This translation is performed on an approximately one-to-one basis for assembly language programs -- one machine instruction for each assembly language instruction. Programs written in a high level language enjoy greater leverage in that a high level language statement may result in the generation of many elementary machine code instructions.

A macro assembler can be regarded as bridging the gap between rudimentary assembly and high level language programming. Indeed, several high level languages have been implemented upon an underlying macro structure. A high level language implemented by macros can furnish the efficiency of assembly language and the ease of high level programming. Via macros, the user can design his own open-ended high level language.

MACRO PROCESSING

Interpretation of a macro involves the three steps:

- macro definition
- macro call
- macro expansion

The macro definition is the means by which the programmer informs the assembler of the instruction sequence to be effected. Briefly, in the macro definition the programmer informs the assembler that "when I say this, I mean that." The macro definition associates a name (label) with the sequence of instructions. Subsequent to the definition, the macro name is used as an entry in the op-code field to invoke the entire instruction sequence. In order to provide more power and flexibility to the macro, beyond that which can be furnished by a text editor, the macro definition allows certain parameters (dummy) to be included in the definition. These dummy parameters appear in the operand field of the macro definition. The assembler recognizes the dummy parameters when they

appear in the sequence of instructions comprising the body of the macro.

The macro definition thus consists of the following:

```
NAME: MACRO          dummy parameter list
      [ MACRO BODY ]
      MACND           signals end of definition
```

The macro call consists of the macro name appearing in the operation (opcode) field of a subsequent instruction. Actual parameters, appearing in the operand field of the macro call, replace the dummy parameters of the macro definition.

In the macro expansion phase, the instruction sequence representing the body of the macro is delivered to the assembler. Dummy parameters appearing in the macro body are replaced, in sequence, by the actual parameters included in the call. With the single macro call, the user has invoked an entire instruction sequence.

MAKRO deals with the macro definition during pass 1 of the assembly. Source text, comprising the macro body, is transferred to a temporary buffer following the symbol table. The source text is scanned for occurrences of the dummy parameters which are replaced by the parameter sequence number. The compressed macro text is then stored uppermost in memory.

Macro expansion must be performed for both passes of the assembly. After recognizing a macro call, the body of the macro is expanded into the buffer area, with actual parameters replacing the parameter sequence values. Assembler input is directed to the expanded text (away from the mass storage device). Input from the mass storage device is resumed when the body of the macro is exhausted.

MAKRO IDIOSYNCRACIES

The treatment of macros by MAKRO differs somewhat from conventional technique. The differences, however, stem from careful consideration, and MAKRO processing is considerably more powerful than alternative methods. The primary departure from convention arises in the treatment of macro parameters. MAKRO delays the binding of parameter values until object code is generated (all parameters are call by name, not value). Dummy parameters appearing in the macro definition are treated as character strings which are recognized in the macro body regardless of their context. Thus, in the definition

```
MAX1: MACRO          String 1, String 2
      [ BODY ]
MACND
```

any occurrence of String 1 in the macro body is regarded as a reference to the first dummy parameter. For example

```
MAX1: MACRO          THIS, THAT
DB 'THIS'           ;THIS or THAT
DW THAT
LXI H, THIS
MACND
```

is treated as reference to the dummy parameters as

```
DB '1'              ;1 or 2
DW 2
LXI H, 1
```

in which the digits represent the parameter sequence.

Actual parameters, in the macro call, are likewise treated without regard to context in the expansion phase. Character strings representing actual parameters directly replace the dummy sequence values. Thus the call

```
MAX1 ALFA, BETA
```

generates

```
DB 'ALFA'           ;ALFA or BETA
DW BETA
LXI H, ALFA
```

The revised and expanded body is then delivered to the assembler for interpretation.

PROCEDURAL AND SYNTACTICAL RULES

1. Dummy parameters must be at least two characters in length. All characters, including blanks, in both actual and dummy parameter strings, are considered significant.
2. Dummy and actual parameter strings begin with the first non-blank character in the operand field. Parameter strings are separated by a comma.
3. All labels generated within the macro body assume global status. The special character # appearing in the macro body is regarded as a reference to a four-digit hex number which is unique for each macro expansion. Labels generated for which global status is undesirable should be suffixed with the # character.

Thus, within the macro expansion,

LABEL:	assumes global status
L#:	is local to the current expansion

4. As a consequence of pass 1 treatment of the definition, a macro cannot be globally redefined.
5. No macro definition may appear within the body of another macro expansion.
6. Macro expansions may be nested up to ten deep, i.e., up to ten macro calls can be simultaneously active. (Refer to REPEAT BLOCK discussion).
7. Scanning for a macro call precedes the search through the op-code table. Thus a macro can be used to redefine a machine operation. For example, to trace jump operations the JMP instruction may be replaced by a macro as

```
JMP: MACRO ADDRESS
      PUSH PSW
      MVI A, 'J'
      CALL CHOUT
      CALL CHIN
      POP PSW
      DB 0C3H
      DW ADDRESS
      MACND
```

which causes the program to display 'J' and await keyboard input before effecting any JMP.

8. The number of actual parameters ordinarily agrees with the number of dummy parameters. Excess actual parameters are ignored. Insufficient actual parameters default to the null parameter.
9. The parameter separation character (default ',') in macro calls can be redefined at the time of macro definition. If the formal parameter list begins with a comma (,) the character immediately following is taken to be the parameter separation character for subsequent calls of that macro. The first formal parameter begins with the character following the separation character. This option is provided to allow syntactically more attractive macro usage.

10. The macro definition must precede any reference.
11. A null actual parameter, represented by two consecutive commas in the parameter string of the macro call, results in a null replacement string in the macro expansion. The first actual parameter is considered null if the calling parameter string begins with a comma.
12. The MACND pseudo-instruction may not be preceded by a label field.
13. MAKRO actual parameters, or portions thereof, enclosed in square brackets [], are treated as literal blocks and expanded without regard to any delimiters contained therein. Each such expansion strips off a matching pair of square brackets. The brackets must be balanced.

USING MACROS

Macro calls are typically used to alleviate tiresome sequences of instructions, such as in table generation or monitor function references. Thus

```
CHOUT: MACRO
        CALL OUTCH
        MACND
```

or

```
STATUS: MACRO PORT,STBIT
S#:     IN PORT
        ANI STBIT
        JZ S#
        MACND
```

illustrate the least imaginative exploitation of macro power. Computer literature is filled with awesome examples of the heights which can be reached by sophisticated macro use. See P.J. Brown, MACRO PROCESSORS, in which it is revealed that SNOBOL 4 is implemented by macros.

The following illustration of a high level language (BASIC) is presented in order to suggest more penetrating application of the macro:

TYPE DECLARATION

```
WORD: MACRO LABEL,VALUE
LABEL: DW VALUE
MACND
```

```
STRING: MACRO LABEL,DATA
```

```
LABEL: DB 'DATA'
```

```
NLABEL: EQU $+1-LABEL
MACND
```

If you want string length

```
LOOPVR: MACRO LOOP
```

Loop index variable

```
LOOPNM: DS 2
```

Loop start
Rep counter

```
MACND
```

PROGRAM LOOPING

```
FOR: MACRO LOOP,REPS
LXI H,REPS
SHLD LOOPNM
LOOPST: SET $
MACND
```

```
NEXT: MACRO LOOP
LHLD LOOPNM
DCX H
SHLD LOOPNM
MOV A,H
ORA L
JNZ LOOPST
MACND
```

ARITHMETIC OPERATIONS

```
ADDITION: MACRO LEFTARG,RTARG,ANSWER
LXI B,LEFTARG
LXI D,RTARG
LXI H,ANSWER
CALL FPADD
MACND
```

Macro expansion in conjunction with conditional assembly offers an especially powerful assembly combination. To illustrate, refer to the previously defined ADDITION macro. Now assume that we wished to address the destination (ANSWER) either directly as shown, or indirectly (LHLD instead of LXI). Further, assume that we wish to avoid the generation of the instruction entirely if the destination location is unchanged from a previous operation. Reflect upon the following complex:

```
ADDITION: MACRO LARG,RARG,ANS,FLAG
LXI B,LARG
LXI D,RARG
NIF HCON-ANS                      Check for valid H
GOTO ADDND
ENDIF
IF 1-FLAG                          Flag is 0 for indirect
GOTO INDIR
ENDIF
LXI H,ANS                          Direct
GOTO ADDND
INDIR:LHLD ANS                      Indirect
GOTO ADDND                          Gobble label
ADDND: CALL FPADD
HCON: SET ANS
MACND
```

This macro was designed to illustrate many of the novel features of MAKRO. Some economy of code could have been effected by use of IFGZ and IFGNZ pseudo-operations. Note that no labels are generated by a call to this macro since the destinations INDIR and ADDND are invariably reached by a GOTO branch. Quite clearly the macro could be expanded to treat the left and right arguments as well. Complex macro usage greatly reduces the chance of coding error, since without macro expansion the chance of correctly entering a number of such sequences is minimal. A set of such complex macros need only be developed once and then merged into the current file. MAKRO, in conjunction with your macro file, becomes your high level language.

REPETITION CONTROL

MAKRO allows assembly time repetition (looping). A block of assembly code may be replicated up to 255 times by enclosing the block in REPT/REPND brackets. The form of the repeat block is

```
REPT expr
```

```
[ CODE ]
```

```
REPND
```

in which expr is evaluated, truncated to an 8-bit value, and used as a loop repetition factor. Repeat blocks may be nested, and may occur within a macro expansion. MAKRO maintains a control stack of length 80 bytes. The maximum depth of nesting is determined by the stack limit.

An active repeat block consumes 10 bytes of the control stack, and an active macro expansion consumes 8 bytes. Repeat blocks and macro expansions may be nested in any way so long as the total stack depth does not exceed 80 bytes.

In order to provide some flexibility to the repeat block, MAKRO recognizes two special operands:

@ is a repeat loop index, counting up from zero, marking progression of the repeat block.

% is a count of the number of active parameters in the most recent macro expansion.

MAKRO also allows looping over the actual parameters in a macro expansion. Such looping is governed by three special characters appearing in the macro body:

```
↑N Control-N   Parameter flag (Press Control and N simultaneously)
↑S Control-S   Start of macro loop
↑Q Control-Q   End of macro loop
```

The start and end of the macro loop must be bracketed by ↑S/↑Q; the loop is then repeated over all the actual parameters occurring in the macro call. Within such a loop, the elements of the parameter sequence are referenced by two ↑N's in sequence.

To illustrate the macro loop, assume we have a series of ASCII strings we wish to print, and that the sequence and number of these strings to be printed must vary within our program. Define the macro print all:

```
PNALL: MACRO
↑S                               Start loop over all actual parameters
LXI H,↑N↑N
CALL PRINT
↑Q                               End the loop
MACND
```

Now we use this macro as

```
PNALL S1,S2,S3
PNALL S6,S1,S9,S2,S7
```

The loop control automatically handles the counting and parameter referencing.

MAKRO BLOCK STRUCTURED ASSEMBLY

The LOCAL/LOCND pseudo-operations allow the user to bracket portions of the assembly, treating such portions as isolated units. Macro definitions, addresses, equates, and sets generated within such blocks may not be accessed from outside the block. Consider such blocks as FORTRAN subroutines or procedures in PL/1 or ALGOL. The insulation of such blocks from one another is nearly complete; the blocks may not contain references to elements outside the block (exception coming).

The treatment of such blocks is effected by limiting the scope of the symbol table. During pass 1 of the assembly, LOCAL restricts access to the symbol table to only those entries following. LOCND, on pass 1, resets global access to the symbol table. On pass 2, LOCND causes all entries generated between the two bracketing LOCAL/LOCND operations to be deleted from the symbol table.

Now the exception promised earlier: An attempt is made during pass 2 to satisfy a reference to an undefined element by searching symbol table entries after the block. Local symbols must remain in the symbol table until the procedural block completes pass 2, and these symbols may be accessed in an attempt to resolve an undefined element, global or local.

ASSEMBLY TIME INPUT

The INPUT pseudo-operation allows the user to define program variables at assembly time. Critical program variables, such as the assembly origin or I/O port numbers, may be entered as input variables, with their value determined by console input during pass 1 of the assembly.

As an example, assume that we have developed a program requiring input from a serial port; however, neither the port number or status mask can be standardized. We may therefore write the source program with these variables defined by input:

```
IPORT:INPUT  
IMASK:INPUT
```

and the status check portion of the program would be

```
READY:IN IPORT  
ANI IMASK  
JZ READY
```

The INPUT pseudo-operation is performed in pass 1 of the assembly. MAKRO displays the source line and awaits console input. The user may enter any valid expression which is terminated by a carriage return.

COMMUNICATION BETWEEN MACROS

The operations APUSH/APOP and SETQ allow communication between related macros. The function of these operations is exemplified by a conceptual DOIF macro.

As the name implies, the DOIF macro is to generate execution time instructions to selectively execute the following block of code. For cosmetic considerations, this macro will utilize '.' as the parameter separation character.

```
DOIF .,ARG1.RELATION.ARG2
```

The macro is invoked as:

```
DOIF X.GT.Y
```

The macro must translate into a logical test of RELATION between the operands ARG1 and ARG2, and JUMP ahead if RELATION is false. While a backward reference can be effected by the SET pseudo-op, forward references cannot. (Why?)

The forward reference is implemented within the DOIF macro as

```
APUSH 0#H  
JUMP IF FALSE TO D#
```

in which the # is uniquely expanded.

A subsequent IFEND macro generates the required label as

```
QVAL:APOP  
SETQ QVAL  
D?:
```

Test your understanding of the above by defining an ELSE macro to be inserted optionally between the DOIF and IFEND macros.

RELOCATION

MAKRO offers two different methods of achieving relocation: at assembly time via the LINK operation, or at load time via relocatable code.

A relocating assembler monitors object code generated by the assembler, and flags portions of that code whose values depend upon the execution address of the program. Object code generated by a relocating assembler is not ready for execution, requiring address modification by another program -- the loader.

A special type of loader -- a linking loader -- will allow program modules to reference previously developed modules (externals). The linking loader performs a library search to find and include all the necessary program modules. The output of the linking loader is an absolute, executable program.

Such techniques are necessary on multi-user machines in which several programs may be executing simultaneously and the execution address of any program is dictated by available memory space. On a micro-computer, the practical advantage of relocation and linkage is that large programs may be developed in small discrete modules which can be created and checked out independently. Commonly used modules, such as floating point routines, need be developed only once.

There are, however, drawbacks to the relocating assembly/link loader:

1. A linking loader and link edit phase is required.
2. Restrictions are placed upon the structure of the source code to enable relocation. These restrictions vary from a minor nuisance to considerable pain, and occasionally force inefficiency into the resulting code.
3. Certain operations (masking) and certain quantities (8-bit values) cannot easily be handled by a relocating assembler.

MAKRO provides the features of a relocating assembler and linking loader via the LINK pseudo-operation, with no restriction placed upon the source code. The LINK operation is performed at assembly time, producing an executable object module, with no need for the linkage/edit or address modification phase. With MAKRO, the user need not restrict his source code to relocatable form, since all MAKRO source is relocatable by the LINK operation.

Relocation and linking are typically performed at the object code level, after assembly has been completed. The MAKRO LINK operation is performed at the source code level. The LINK pseudo-operation extends the assembly to include the named source file(s).

Suppose a main program is being developed which will require library modules FPPACK (a floating point package) and FPOUT (an input/output package). The main program should then include

```
LINK FPPACK  
LINK FPOUT
```

Assembly proceeds through the main program and continues through the link modules in the order given. The LINK pseudo-operation may appear anywhere in the source code, and LINK modules may themselves contain the LINK operation.

The LINK command, without a file name, acts as the INPUT pseudo-operation. The source line is displayed, prompting the definition of the link file at assembly time. Macro library files may be terminated by such a LINK command to chain the assembly to the current source file. In this case the macro library file should be specified as the input file.

The LINK file name must be terminated with a carriage return.

RELOCATION

The LINKED and KWIK Loaders

MAKRO version AMA.2 generates a relocatable object module for source code conforming to certain addressing restrictions. The relocatable object module is loaded into memory, for execution, by:

LINKED	linkage editor/relocating loader
KWIK	relocating loader

Either of these loaders will perform all necessary address modification to relocate the object module for execution anywhere in memory, provided that address constants satisfy the restrictions given below. In addition to relocation, the LINKED linkage editor will perform a library search to include previously assembled object modules required for execution.

Three MAKRO pseudo-operations provide loader directives for the LINKED loader:

ENTRY	Defines the label field of the instruction to be an entry point when this module is referenced elsewhere.
EXTRN	Defines the label field to be a requisite module to complete an executable load.
LIBRY	Defines the label field to be a library containing certain of the requisite external modules.

If none of these three directives is present in the assembly, the object module may be loaded by an INTEL hex loader for execution at the absolute address given by the assembly or by the KWIK loader for relocation. In the absence of the loader directives, object code generated by MAKRO conforms to INTEL hex standards, except that relocation information is passed in the two bytes following the load address. These bytes (7 and 8 following the colon) are ignored by the INTEL hex loader.

The object code produced by MAKRO consists of four types of records:

	<u>Byte Number</u>	<u>Contents</u>
1. <u>DATA RECORD</u>		
	1	':'
	2,3	byte count
	4,5	load address (high)
	6,7	load address (low)
	8,9	relocation information
	10 to n-1	data bytes
	n	checksum

	<u>Byte Number</u>	<u>Contents</u>
2.	<u>LIBRARY DIRECTIVE</u>	
	1	7AH
	2-n	Library file name (ASCII)
3.	<u>ENTRY DIRECTIVE</u>	
	1	ØBAH
	2,6	entry name (ASCII)
	7,8	entry point, relative to start
4.	<u>EXTERNAL DIRECTIVE</u>	
	1	ØFAH
	2,6	external name (ASCII)
	7,8	tail address of linked list
5.	<u>ABSOLUTE ENTRY</u>	
	1	ØBBH
	2,6	entry name
	7,8	entry value

ENTRY DIRECTIVE

MAKRO allows commonly used program modules to be assembled and stored in an object library. Entry points to these modules are defined by the ENTRY directive, which are output along with the object code. These object modules may be referenced in a later assembly by the ENTRY point name. The form of the ENTRY directive is:

```
LABEL:ENTRY
```

which is similar to

```
LABEL:EQU $
```

except that the ENTRY pseudo-operation generates loader information during pass 1 of the assembly.

ABSOLUTE ENTRY DIRECTIVE

The ABSNT directive functions as ENTRY except that the value of the entry is defined by the operand field (as in EQU) and is not subject to relocation by the loader.

Example:

```
DOS:EQU 2000H
CHARIN:ABSNT DOS+10H
```

EXTERNAL DIRECTIVE

The EXTRN directive allows the current assembly to reference an ENTRY point defined by a previous assembly. The form of the directive is:

```
LABEL:EXTRN
```

which defines LABEL as a routine not present in the current assembly, but which may be found in an object library on a disk file. Having defined LABEL as an external, it may be referenced as any other program variable, except that it may not be used in an expression. Thus

```
CALL LABEL      is valid, while
CALL LABEL+3    is forbidden
```

LIBRARY DIRECTIVE

The library directive, LIBRY, identifies the disk file in which LINKED may seek to satisfy subsequent external directives. One or more external directives follow the library directive. For example, a disk file FPPACK may contain a floating point package with entry points FPADD, FPSUB, FPMUL and FPDIV. A source program requiring these floating point routines as externals would declare FPPACK via a LIBRY directive, and itemize the required entry points:

```
FPPACK:LIBRY
FPADD:EXTRN
FPSUB:EXTRN
FPMUL:EXTRN
FPDIV:EXTRN
```



Entry points, library files, and externals must have unique names. Within the library files the required external references must be defined as entry points.

Library files are included in the order in which they are encountered; the entire object module is included.

The LINKED load map defines the execution address of each entry point. Unsatisfied externals are displayed. At completion of the load, the next available memory address is displayed. A checksum error is signified by '?'. Duplicate ENTRY and unsatisfied EXTRN modules are identified by 'D' and 'U' errors respectively. Library files not found on the designated unit are displayed, and the user may then redefine the file and unit.

SOURCE CODE RESTRICTIONS

1. Labels defined by an EXTRN directive may not be used in an arithmetic expression.
2. Relocatable quantities may only be used in an arithmetic expression containing the operators + and -.
3. Relocatability is limited to 16-bit quantities. The relocatability of such quantities is determined by the form of the expression defining the quantity. Absolute quantities are assigned a relocation value of \emptyset . Thus

```
CONST:EQU 5
```

defines CONST as an absolute with relocation value \emptyset . Program relative values are assigned a relocation value of 1. Thus

```
HERE:LXI H,HERE
```

assigns a relocation value of 1 to the label HERE, and flags the LXI instruction as requiring address modification.

4. Arithmetic expressions containing absolute and relocatable quantities derive their relocation value from the result of the expression. The rules of relocation arithmetic are:
 - a. The sum of an absolute and relocatable quantity is relocatable.
 - b. The difference of two relocatable quantities is absolute.
 - c. Any chain expression, containing absolute and relocatable quantities connected by + or -, must evaluate to either \emptyset or 1 in relocatability. Mentally substitute 1 for program relative quantities, and \emptyset for absolutes, and evaluate the expression. MAKRO does not check the resulting expression for validity. This restriction does not mean that masking or other such address computations may not be used. MAKRO will treat the results of such operations as absolute, and it is the programmer's responsibility to ensure that the resulting object code is valid.
5. Secondary load modules, those containing the ENTRY directive, must be assembled at ORIGIN \emptyset .
6. Load modules should neither begin nor end with the DS pseudo operation.

SYMBOL TABLE

The symbol table displays the value of all program variables together with the relocation flag. The symbol table is printed with five entries per line, each entry consisting of the variable name, variable value, and relocation flag. The legend for these flags is:

Ø	absolute value
1	relocatable value
3	external
83	external library

The value shown for an external variable refers to the last address within the program at which that external was referenced.

PDS RELOCATING LOADERS

A loader is the conduit through which the contents of a disk file are transferred to memory for execution. The most widely available loader for micro-computer use is the INTEL hex loader for which source code listings are easily obtainable. Loaders vary widely in the extent to which they operate upon the data (program) while effecting the transfer from disk to memory.

The INTEL loader, one of the simplest, maintains a checksum to ensure fidelity of the transfer, but otherwise performs no operation on the data being transferred. The next higher level of loader sophistication is the relocating loader. This utilizes relocation information to perform certain modifications upon the data being transferred to enable the program to execute at an address other than that for which the program was assembled. The highest level operation is the linkage editor which can combine one or more incomplete modules, relocating as required, into a unified, executable program. A linkage editor may not necessarily perform the loading function, in that no executable image may be left in memory at completion of its task.

PDS spans this spectrum of loader functions by providing two loaders, KWIK and LINKED, which together with the ubiquitous INTEL loader satisfy all requirements.

The function of the PDS loaders is somewhat dependent upon the operating environment. The KWIK loader is the relocation vehicle for object programs created with the MAKRO assembler version AMA.2. The object file and load address are identified to KWIK which proceeds to create an executable image at the load address. The input file to KWIK must satisfy the coding restrictions defined in the preceding section, and the file may not contain any of the loader directives. Such files may also be loaded with the INTEL loader for absolute execution (at the address for which the program was assembled).

The LINKED loader will perform the relocation function while collecting the independent modules defined by the loader directives. LINKED combines the requisite modules into an executable image in memory at the specified load address and simultaneously creates an INTEL hex compatible object file.

It is anticipated that the INTEL loader, or an equivalent binary loader, will continue to perform the bulk of the loader functions. The KWIK loader is expected to be used for unique applications requiring an object file to execute at more than one memory address. The LINKED linkage editor is expected to be used in the development of large applications programs in which a number of component elements have been independently developed.

KWIK and LINKED are furnished in relocatable form and may thus be relocated to satisfy system requirements.

MAKRO EXPRESSION EVALUATION

Arithmetic expressions appearing in the operand field of MAKRO instructions are evaluated according to standard arithmetic rules. The following table defines the available arithmetic operations and the operator precedence.

<u>Operation</u>	<u>Precedence Value</u>	<u>Definition</u>
(16	Begin parenthetical expression
*	12	Multiplication
/	12	Division
\	12	Modulo, integer remainder
+	11	Addition
-	11	Subtraction
&	8	Logical AND
or † (5E hex)	7	Logical OR
!	7	Logical EXCLUSIVE OR (XOR)
>	6	Right shift, zero fill
<	6	Left shift, zero fill
" (quote)		NOT, logical complement
)	∅	End parenthetical expression

Expressions containing these operators are evaluated from left to right, execution of any operation delayed until all preceding operations of precedence value greater than or equal to the pending operation are performed.

The logical complement refers to the operand or parenthetical expression immediately following.

In the expressions

$$A > B, A < B$$

the left operand (A) is shifted in the indicated direction by B bit positions, with zero bits shifted in.

The modulo operator \ returns the integer remainder after division. Thus $A \setminus B$ yields

$$A - [A/B] * B$$

where the integer part of the bracketed term is taken. The modulo operator has precedence equal to *, /. The expression

22\3 * 5 yields 5 as

(22\3) * 5.

In any expression, the user may insert parentheses to force the intended computational sequence. In the previous expression, execution of the modulo can be delayed by

22\ (3*5) = 7

STRING HANDLING PRIMITIVE

Arithmetic operands and the first argument of the IFEQ and IFNE pseudo-operations may be subject to string segmentation. String segmentation is invoked if the first character of the operand is a left angle bracket '<'. The two characters immediately following the opening bracket are taken as the start/finish segmentation markers. The string argument is taken as the remaining characters up to but not including the right angle bracket '>'. The string handling primitive replaces the entire construct with the characters, if any, contained between the start/finish segmentation characters. Thus

<59123456789> yields 678
<()ARRAY(JI)> yields JI
<Ø(ØARRAY(IJ))> yields ARRAY

The string primitive is also functional when recognized in the label field and macro parameter fields. Use of the segmentation primitive can be illustrated by a conceptual LOAD macro to place the value of the argument on an operand stack. The macro must take appropriate action when the argument is an array reference:

```
LOAD:MACRO ARG
    IFEQ <()ARG>,;SCALAR      test for null index
    LXI H,<()ARG>              else array, get index
    LXI D,<Ø(ØARG>
    DAD D
    GOTO QUIT
SCALAR:LXI H,ARG
    GOTO QUIT
QUIT:                          stack operand
    MACND
```

INTEL SOURCE COMPATIBILITY

Source files created for the INTEL assembler must be modified before assembly by MAKRO. The following table defines the systematic editing required. In the table 'b' refers to a blank.

<u>CHANGE:</u>	<u>TO:</u>
bEQUb	:EQUb
bSETb	:SETb
ENDM	MACND
bANDb	&
bORb	^
bSHRb	>
bSHLb	<
bMACRO	:MACRO
bMODb	\
bXORb	!
bNOTb	" (quote)

Source lines containing multiple labels must be modified to contain only a single label identifier.

The expanded capability of MAKRO generally precludes the inverse operation of converting MAKRO source.

SAMPLE LINKAGE OPERATION

The following example should illustrate the use of the linkage editor.

1. Create a source file CALLRS:

```
CALL EXT1
CALL EXT2
EXTS:LIBRY
EXT1:EXTRN
EXTN:LIBRY
EXT2:EXTRN
END
```

2. Use MAKRO to assemble this file, creating the object file CALLR.

3. Create a source file EXTSS:

```
EXT1:ENTRY
LXI H,EXTG
EXTG:ENTRY
LXI H,2
END
```

4. Assemble this source file, creating object file EXTS.

5. Create a source file EXTNS:

```
EXT2:ENTRY
LXI B,EXTQ
EXTQ:ENTRY
MVI B,'Q'
END
```

6. Assemble EXTNS, creating object file EXTN.

7. Exercise your linkage module ULINK, identifying CALLR as the input file, and any convenient load address.

Note that in Step 1 the code for EXT1 and EXT2 does not reside in the current source module. The LIBRY directives identify to the linkage editor the disk file(s) in which the subsequent external references may be found. The module in Step 1 defined EXT1 and EXT2 as modules which must be resolved during the load.

In Step 3, the source module EXTSS creates the first external EXT1. Note that within this module EXT1 is defined as an entry.

In operation, the linkage editor loads the module CALLR, then opens EXTS to find the location of EXT1. The entire module EXTS is loaded.

Finally the linkage editor opens and loads EXTN, resolving references within CALLR to the entry point EXT2.

MAKRO ABSOLUTE FILES

Object code written to disk by MAKRO is first passed through a format program which incorporates the checksum and relocation information. The formatter calls a direct disk write routine which buffers disk output.

MAKRO can be caused to generate absolute object disk files which can be loaded for execution by the DOS loader by skipping the format routine.

To create this program, load MAKRO without entering the program. Use DEBUG to search for the byte string F5 D5 E5. The start of this string marks the start of the direct disk write. Again use DEBUG to search for the string D5 C5 E5 F5 which marks the start of the formatting routine. At this second address, patch in a JMP to the first address. Save the resulting program as disk file ABSMAKRO.

Certain code restrictions must be followed:

1. The code must flow straight through with a single ORG statement at the start, and no manipulation of the location counter within the program.
2. The DS opcode must be replaced by the macro

```
DS:MACRO COUNT  
REPT COUNT  
DB 0  
REPND  
MACND
```

3. None of the loader directives nor any relocation feature can be used.

EDIT

A very powerful Text Editor for
the creation, modification and
disk storage of character-oriented
material

Copyright 1978

Allen Ashley
395 Sierra Madre Villa
Pasadena, CA 91107

(213) 793-5748

INTRODUCTION

EDIT is a very powerful text editor featuring a full spectrum of text manipulation operations including string search, substitution, insertion, deletion and block move or delete. An elaborate command interpreter allows the definition of command string macros. Segments of an input text file can be drawn from disk into memory, modified, and written back to an output disk file. Large, heavily-commented source files which exceed available memory can be developed and modified easily with the EDIT text editor.

Operationally similar to the editor offered by INTEL, EDIT offers a broader range of functions, approximately three times the speed, and occupies a little more than half the memory space of the INTEL ISIS editor or the TDL ZAPPLE editor.

EDIT is written entirely in the 8080 subset of the Z80 instruction set and is thus fully operational on either machine.

EDIT ORGANIZATION

EDIT operates under control of an executive which is responsible for the transfer of textual material between disk and memory and for the interpretation of user commands to create or modify that material.

Command strings, consisting of decimal repetition factors, alphabetic command characters, character string parameters and control punctuation, dictate the modifications to be performed on the text stored in memory. Portions of an input text file may be drawn into memory, modified and stored back on an output disk file.

Although the command structure of EDIT is consonant with conventional text editors, users unfamiliar with convention may require some practice to become adept at exploiting the many features. It is suggested that the user practice on an empty diskette, creating and modifying text of no particular value. Each of the executive commands should be exercised in all its variety until the operation of EDIT has become second nature.

The majority of software development time is spent either in the debug mode, finding errors, or in the text-edit mode, correcting those errors. The user is advised to become thoroughly familiar with these software development tools.

EDIT EXECUTIVE

Commands to the system executive consist of single upper case alphabetic characters, optionally preceded by a signed decimal repetition factor. Commands can be chained together to form a block-structured command string. Such command strings are punctuated by the escape character (1BH, echoed as \$), while a block command is indicated by enclosing the block in brackets <>. Every command string must be terminated by two successive escape characters.

Command blocks can be nested quite deep, on the order of fifty. A command block is interrupted either when any portion of the block cannot be executed or the block repetition factor is exhausted. The meaning of these features will, hopefully, be made clear in the subsequent material.

While the escape character is always interpreted as punctuation, the block-defining brackets are significant only in the context of an executive command. In addition to these characters the @, in the proper context, has a variety of meanings which depend upon the command being executed.

Generally the @ is interpreted as 'any' or 'all.' When the @ is used as the command repetition factor, preceding the control character, it is interpreted as 'all,' implying that the command is to be repeated as often as possible. When the @ is used as a character (not the first) in a character string under search, it is interpreted as 'any' in that @ will match any character. In the commands defining disk input/output files, @ is interpreted as 'none.' To create a new file, rather than edit an existing file, the request for an input file name should be answered with the @.

In search strings, the special character 'ampersand' (&) represents an arbitrary character string. Just as @ will match any character in the text, & will match an arbitrary character string not including a line feed.

EDIT maintains five pointers to the text file:

- Start of the text buffer
- End of the text buffer
- Start of a defined textual block
- End of the defined textual block
- Pointer to the current activity

The first of these pointers is stationary, the second moves according to ebb and flow of the file size. The block pointers mark the start and end of textual blocks for deletion or relocation.

The pointer to the current activity dictates the operation of EDIT. Executive commands enable the user to move the activity pointer throughout the text file. The editing commands (Search, Delete, Change) are relative to the position of the current activity pointer. The command structure of EDIT is composed of three command types:

Disk input and output operations
Commands to move the activity pointer
Text modification commands relative to the pointer

Executive commands are expressed to EDIT in response to the prompt @ (the all-purpose character). EDIT examines memory to determine the available size of contiguous RAM following the program end. Having determined memory size, EDIT lays claim to all the available space.

Typing errors in a command string can be backspaced over with the rubout key. EDIT echoes the deleted character. The entire command string can be aborted by Control/C. EDIT accepts the entire command string before proceeding to interpret that string. EDIT automatically supplies a line feed after an input carriage return.

The user should note that apart from the control character Escape and the context-dependent characters (@, &, <, >) no other text characters have any special significance to EDIT. Carriage return, line feed, underline, etc., are merely data characters to be manipulated as any other characters.

EXECUTIVE COMMANDS

In the following, n represents a signed decimal repetition factor, defining the number of times the immediately following command is to be executed. When applicable, a negative parameter value directs EDIT operations toward the start of the text file. Spaces may not separate the repetition factor and the immediate command. The punctuation character Escape is represented by \$. By default, an absent repetition factor is assumed to be unity.

COMMAND LIST

nA APPEND n SECTORS FROM THE INPUT FILE TO THE MEMORY BUFFER.

EDIT will terminate the command when the input file is exhausted, when n sectors have been transferred, or when available memory has been filled. The current pointer position is not affected by this operation.

B MOVE THE CURRENT ACTIVITY POINTER TO THE START OF THE MEMORY FILE.

nC CHANGE CHARACTER STRINGS.

The form of the command is

nCSTRING1\$STRING2\$\$

which changes the next n occurrences, following the current pointer, of String 1 to String 2. Every occurrence, after the pointer position, of String 1 is changed to String 2 by the command

@CSTRING1\$STRING2\$\$

An example, not original, is

nCFROG\$PRINCE\$\$

which changes the next n FROG's to PRINCE's. The current activity pointer is moved to the position immediately following the last of the n operations.

The 'any' character @ can be used to ignore any character, except the first, in String 1. Thus

CT@IS\$THAT\$\$

will change THIS as well as TZIS to THAT.

All characters except @ are considered significant in the strings for the Change and Search commands.

A negative repetition factor directs the change function backward toward the start of text.

A character string can be deleted from the memory buffer by

nCSTRING\$\$

Only occurrences of strings after the current pointer position can be changed. There is no practical limit to the length of parameter strings for Change or Insert functions.

nD DELETE THE NEXT n CHARACTERS FOLLOWING THE CURRENT POINTER.

If the parameter is negative, the n characters preceding the pointer are deleted. The command @D will not delete the remaining characters. To clear the buffer use @K; @D deletes the character, if any, preceding the pointer position. An alternate form of the delete command is nDSTRING, in which the block of text from the current activity pointer up to and including the nth occurrence of STRING is deleted. Deletion is toward the start of text if n is negative.

E TERMINATE EDIT, TRANSFER MEMORY CONTENTS AND ANY REMAINING INPUT FILE CONTENTS TO THE OUTPUT FILE.

Control is passed to the warm-start entry point of the disk operating system. Subsequent re-entry to EDIT allows an entirely new edit session.

F CLOSE THE EXISTING DISK INPUT FILE AND OPEN A NEW INPUT FILE.

EDIT responds with the INPUT query. All significance of the previous input file is lost. EDIT may be used to merge disk files by repeated use of the F command.

G SCROLL THE MEMORY FILE.

The scroll is terminated by Control/C or end of file. The scroll is controlled by the space bar. Pressing the space bar will freeze the display; any other key resumes scroll. At termination (except for end of file) the current pointer is positioned approximately 8 lines before the last line of the display.

H SET BLOCK POINTERS.

EDIT allows text blocks to be moved or deleted. The start and end of the text blocks are defined by the two block pointers. The H command sets the start pointer to the end pointer, and the end pointer to the current position of the activity pointer. Successful definition of the block requires that the activity pointer be moved down, from start to end, invoking the H command twice. EDIT checks only that the end pointer is closer to the end of text than the start pointer. The user is cautioned to exercise care in setting the pointers for a block delete. Make sure the pointers are properly set before a block delete.



I INSERT THE INPUT STRING INTO THE TEXT BUFFER IMMEDIATELY FOLLOWING THE ACTIVITY POINTER.

Thus ISTUFFS would insert the characters STUFF into memory at the position of the activity pointer. The pointer is moved to the character following the insertion. The length of the inserted string is limited only by available memory. The insertion may contain any characters except the Escape punctuation character.

J PAUSE.

Execution of the current command is interrupted to await keyboard input. A '?' prompt is issued to signify that EDIT requires user input before proceeding. Typing the ESCAPE key returns EDIT to the input mode; any other key resumes processing. Upon escape, EDIT saves the current command string.

The pause mode is used to interrupt a long command string to display the working area. The command string

```
@<S:$ØLSIT$JSIL$>$$
```

will search every line containing a colon, display the line, and wait for user response. After detecting the escape key, the activity pointer is positioned at the start of the last line displayed (in this case).

nK DELETE LINES FROM THE BUFFER.

If n is positive, n lines following the current pointer position are deleted. If n is negative, n lines preceding the pointer are deleted. Lines in EDIT are defined as the characters following a line feed character up to and including the next line feed. If the pointer is positioned within a line, only the portion of the line on the deleted side is deleted. The command sequence

```
B$$  
@K$$
```

will scratch the entire memory buffer.

nL MOVE THE CURRENT POINTER POSITION BY n LINES.

Direction of motion is toward the start of text for negative parameter values. If n is 'Ø' the pointer is positioned at the start of the current line.

nM MOVE THE CURRENT POINTER POSITION BY n CHARACTERS.

@M moves the pointer back one character. Use 'Z' to position at end of text.

N REFORMAT AN ASMB SOURCE FILE INTO MAKRO FORMAT.



The source files of ASMB are not suitable for text processing. The 'N' command strips the line numbers and inserts line feeds, preparing the input file for input to the MAKRO assembler. The entire file must be resident in memory.

nP WRITE n SECTORS, IF POSSIBLE, FROM THE MEMORY TEXT FILE TO THE OUTPUT FILE.

The pointer position is moved to the start of the text file. EDIT will not clear the text buffer until an end of the input file is detected.

Q RETURN CONTROL TO THE WARM-START ENTRY OF THE DISK OPERATING SYSTEM.

If the End command has not been executed, EDIT may be re-entered without harm to the active memory file.

nS SEARCH FOR THE nth OCCURRENCE OF A SPECIFIED CHARACTER STRING FOLLOWING THE CURRENT POINTER POSITION.

The pointer is positioned after the last occurrence found. The command string

nSEdit\$\$

positions the pointer after the nth occurrence of EDIT. The search proceeds from the pointer position to the end of text. A negative repetition factor searches backward toward the start of text.

The ampersand (&) as a character, not the first, in a search or change string will match an arbitrary character string not including a line feed. Thus the command

S:&Z\$\$

will succeed for either of the following:

:Z
:XXXXZ

but not for

:CALL SUB
ZERO:INX H

since a line feed separates the first colon and the Z.

nT TYPE (DISPLAY) n LINES FROM THE CURRENT POINTER POSITION.

The sign convention for n is followed.

V VIEW ACTIVITY POINTER IN CONTEXT.

W INSERT THE TEXT BLOCK DEFINED BY THE TEXT POINTERS INTO MEMORY AT THE CURRENT POINTER POSITION.

Blocks may be moved up or down in memory, but the source and destination must not overlap. The source block is not modified by the insertion. EDIT monitors the (possibly new) pointer positions to the source block in preparation for a Delete command.

X DELETE THE BLOCK DEFINED BY THE BLOCK POINTER POSITIONS.



No modifications to the source file except Block Move, may be made between the steps to set the block pointers and either Block Delete or Block Move. No operation is performed if the block end pointer is less than or equal to the block start pointer.

Y DISPLAY, IN HEXADECIMAL, STATISTICS OF THE CURRENT MEMORY FILE.

The display format is:

P hex address of activity pointer
L # of lines in file
C # of characters in file
S # of disk sectors required to contain file
T end of file memory location.

Z MOVE CURRENT ACTIVITY POINTER TO THE END OF MEMORY FILE.

SPECIAL CHARACTER COMMANDS

EDIT recognizes the special character commands only when these characters are entered as the first character of the command string.

Control/R As the first character in a command string, Control/R repeats the immediately preceding command string.

Control/U As the first character in a command string, Control/U fetches and executes the command string (up to 32 characters) saved at interruption of the previous pause command (J).

COMMAND STRINGS AND BLOCK COMMANDS

A single EDIT command consists of the repetition factor and the command terminated by two escape characters. As an example, to change the next two occurrences of THIS to THAT the command is

```
2CTHIS$THAT$$
```

Now, suppose it is desired to search for a line containing a colon, and delete the next following line containing the string 'KEY.' This (far-fetched) sequence could be performed by the sequence of atomic commands

```
S:$ $      Find a colon
SKEY$ $   Now find KEY
ØL$ $     Move to start of KEY line
K$ $      Delete the KEY line
```

The same sequence can be performed by the command chain

```
S:$SKEY$ØL$K$$
```

Note that single escape characters are used to identify the end of each element of the command string and a pair of escape characters mark the end of the chain. Inability to execute any element of the command chain terminates further execution of the string.

Certain of the commands, such as insert, do not recognize a repetition factor. Such commands, or indeed a command chain including these commands, can be repeated an arbitrary number of times by enclosing the chain in brackets. For example, to insert XXXX before every occurrence of YY in the text

```
B$ $      Move pointer to start
@<SY$-2M$I$XXXX$2M$>$ $
```

which is equivalent to indefinite repetition of the command block

```
SYY$ $   Find YY
-2M$ $   Go back over the YY
I$XXXX$ $ Now insert the XXXX
2M$ $    Move past the YY so we don't pick it up again
```

Each such command block must be preceded by a repetition factor.

The unattractive appearance of the command block is alleviated by experience and the fact that 99% of the editing tasks are much simpler than this.

Blocks themselves can be nested, but at this point serious examples are difficult to generate.

COMMAND STRING SYNTAX

The syntactical rules of EDIT were designed to avoid execution of a command string which would produce results not intended by the user. Execution of a command string is immediately terminated upon detection of a syntax error. On occasion this may require that the input string be completely re-entered, a burden considered less serious than the loss of an entire source file.

SYNTACTICAL RULES

1. All command strings must be terminated by two successive escape characters.
2. Parameter strings for Search and Change commands must be terminated by an escape character.
3. Block commands must be preceded by a repetition factor, the sign of which is ignored.
4. Scanning of the command string resumes at the character immediately following the closing bracket of a block command. Thus
2<CE\$X\$>\$B\$\$
terminates after executing the block twice; whereas
2<CE\$X\$>B\$\$
executes the B command before terminating.
5. Nested block commands must have their closing brackets in succession. For example
2<2<CE\$X\$>>\$\$
is a valid command to change E to X four times. On the other hand,
2<2<CE\$X\$>\$>\$\$
terminates after executing the inner block twice.
6. The opening and closing brackets in a command string must be balanced. EDIT assumes each closing bracket refers to the immediately preceding opening bracket. Failure to properly close a block command defeats the repetition factor for that block.

ERROR MESSAGES

ILLEGAL

Indicates an invalid command character.

DISK ERROR

Some condition has prevented access to disk.

NO ROOM

An overflow condition has been detected, either insufficient file space on the output file or insufficient memory to continue the current disk operation. Memory overflow can be remedied by dumping one or more sectors of the memory file to disk. Refer to Memory Organization.

ERROR

Some error condition other than those above has been detected (generally a syntax error in the command).

CANNOT FIND

The CANNOT FIND message signals that EDIT was unable to continue a Search or Change command. It is preceded by the (hexadecimal) number of times the command was successfully executed within the current command block. This feature can be used to count the number of occurrences of a character string.

B\$@SEEDIT\$\$

will yield a count of the number of occurrences of EDIT in the text. The counter also indicates whether the command was ever successfully completed, for if

@CSTRING\$NEWSTRING\$\$

results in

0000 CANNOT FIND
STRING

then STRING was never found.

Any of the above error conditions terminate interpretation of the current command sequence.

SAMPLE EDIT OPERATION

A few examples are presented to illustrate EDIT operation. In these examples, the up-arrow illustrates the position of the current activity pointer.

IThere is a tide in the affairs of men\$\$

There is a tide in the affairs of men↑

Caffairs\$business\$\$

CANNOT FIND
affairs

B\$\$

↑There is a tide in the affairs of men

Caffairs\$business\$\$

There is a tide in the business↑ of men

@Co\$pp\$\$

CANNOT FIND
o

There is a tide in the business pp↑f men

Command; insert text into buffer.

Buffer contents; pointer positioned after insert.

Command; change strings.

Response; pointer was positioned after 'affairs'.

Command; move pointer to top of buffer.

Result.

Command; now change affairs.

Result.

Command; change all 'o' to 'pp'

Response; command repeated until 'o' could no longer be found.

Result; note: 'o' was found and changed once.

Remember that "\$" is the echo of the escape key.

SAMPLE BLOCK OPERATIONS

There is a tide in the business pp^f men
B\$\$

Buffer contents

Command; move pointer to start of text.

↑ There is a tide in the business pp^f men
Sde\$\$

Result.

Command; position pointer after 'de'.

There is a tide[↑] in the business pp^f men
H\$\$

Result.

Set block pointers.

There is a tide[↑] in the business pp^f men
Spp\$\$

Result; block pointer 2 positioned at current pointer position; start pointer not yet valid.

Command; position pointer after 'pp'.

There is a tide in the business pp^f men
H\$\$

Result.

Position block pointers.

There is a tide in the business pp^f men
┌──────────────────┐
| |
 block

Result; block pointer 1 set at old position, block pointer 2 set at current position, both pointers valid.

Z\$\$

Command; prepare to insert block at end of text.

There is a tide in the business pp^f men[↑]
┌──────────────────┐
| |
 block

Result

W\$\$

Insert block at current pointer position.

There is a tide in the business pp^f
men[↑] in the business pp

Result; pointer position unchanged.

X\$\$

Delete old block.

There is a tide[↑] men in the business pp

Result; block deleted, pointer moved to start of deleted block, block pointers no longer valid.

CONDITIONAL COMMAND EXECUTION

Considerable thought was expended in an effort to provide the user with some conditional execution capability. As a paradigm for illustration, consider the command sequence:

```
While not at end of file
SEARCH for CALL
  IF next line is RET
    THEN change CALL to JMP AND delete RET
  ELSE continue search for CALL
```

The potential variations of such conditional sequences and the conditions of the test are unfathomable. Any attempt to provide the mechanics for such a wide variety of possible situations would unnecessarily complicate operations for ordinary tasks.

The adopted solution involves and explains the operation of the pause command. The pause command enables the user to execute a sequence of elementary commands and then display the working area. The user may then interrupt the sequence to effect the necessary repairs, and then resume the initial sequence with the Control/U special command.

Admittedly, the user is not entirely relieved of his burden; however, he may be spared the consequences of an ill-posed command sequence.

Our previous model may be effected by the following:

```
@<SRET$-LS2TSJ$2L$>$$      (Search for RET, back up one line and
                               print two lines; pause; skip over
                               the RET if you wish to continue.)
```

If the display reveals the CALL/RET sequence, the user may interrupt execution, make the necessary modifications, and resume the original sequence by the Control/U command.

EDIT maintains two separate command buffers for the Control/U and Control/R commands, and the user may therefore toggle between these two to systematically edit the entire file.

Still pursuing the previous example, assume the text file consisted of the following:

```
CALL SUB1
RET
DAD H
RET
CALL SUB2
RET
INX H
RET
```

with the pointer positioned at the start of the text. Now enter the search

```
@<SRET$-L$2T$J$2L$>$
```

and EDIT responds with

```
CALL SUB1
RET
?
```

our target for change. Type the escape key to recover the input mode and save the seek command string. Now we effect the text modification with

```
CCALLSJMP$L$K$
```

which defines this as the previous command. (Ignore excessive scrolling here.)

At this point the text buffer consists of

```
JMP SUB1
↑DAD H
RET
CALL SUB2
RET
INX H
RET
```

with the pointer positioned at the up arrow.

Now we re-enter the search mode with

```
Control/U
```

and EDIT returns with

```
DAD H
RET
?
```

to which we respond with the space bar, yielding

```
CALL SUB2
RET
?
```

Now type the escape key and Control/R to yield the text contents

```
JMP SUB1  
DAD H  
RET  
JMP SUB2  
↑INX H  
RET
```

Control/U resumes the search. The entire file is searched and patched by

1. Entering the search command with an inspection pause;
2. Entering the patch command when needed;
3. Toggling between the Control/R and Control/U commands.

TEXT REARRANGEMENT

Rearrangement, while not of particularly pressing import, merits mention for illustrative purposes. Assume that we wish to collect all of a certain group of lines together into a single block. As an example, we may wish to move all data statements of the form

```
DB '
```

to the end of text. Consider the command sequence

```
B$$                                move to start of text  
@<SDB '$ØL$H$L$H$B$W$X$>$$
```

which searches for the target string, moves to the start of that line, sets block pointers, advances one line, sets block pointers, moves to start of text, inserts the target line, and finally deletes the line from its initial position.

Having collected all such lines at the start of text, the block may be re-positioned at will. This operation is quite slow for large files, and collects the target lines in reverse order. The reversal of sequence can be avoided, however (an exercise for the reader).

MULTIPLE STATEMENT LABELS

Source files created with the INTEL assembler, or any assembler permitting multiple statement labels, can be patched to MAKRO format by the following command:

```
B$@<S:&:ØL$$:$IEQU $           carriage return  
$>$$
```

in which the \$ in EQU \$ is the dollar sign; everywhere else, it is the echo of ESCAPE.

USE EDIT TO SAVE TYPING

Suppose a certain name, or assembly language command, must be repeated with tiresome frequency throughout a body of text. We may substitute a single, unused character for the nuisance string(s) and systematically edit the entire file to replace the temporary character with the desired string. For example, we may decide to use '#' to represent the character string

```
:DW 0
```

and enter the assembly source code as

```
DATA#  
LABEL#  
KNTR#   etc.
```

Then enter the command string

```
B$@C#$:DW 0$$
```

Move to top of buffer and change all occurrences of # to the desired string.

to yield

```
DATA:DW 0  
LABEL:DW 0  
KNTR:DW 0
```

Similarly, systematic editing can replace a single character with several lines of code. Thus

```
B$@C#$MOV A,H  
ORA L$$
```

Carriage return inserted in input string.

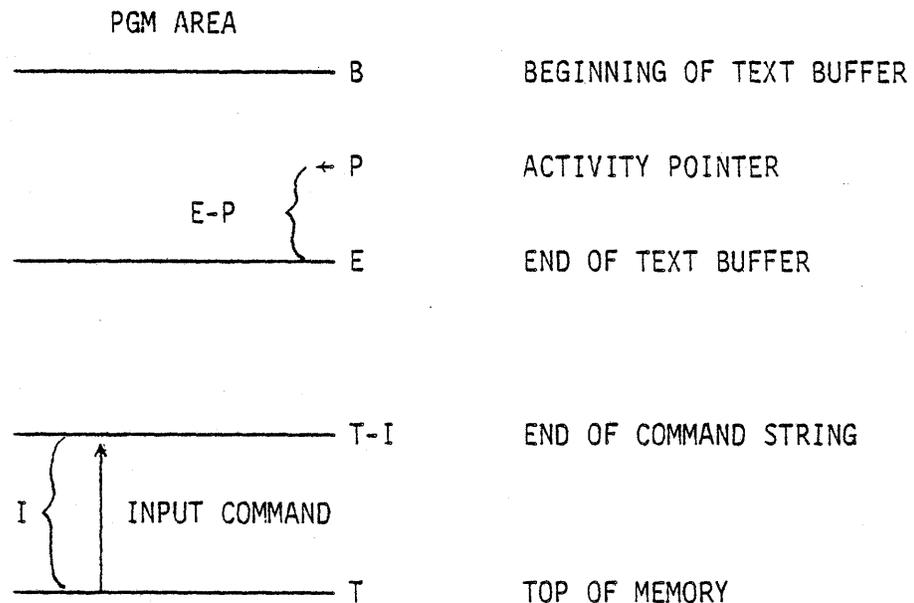
will change every occurrence of '#' to the two lines of code

```
MOV A,H  
ORA L
```

which test the H,L registers for zero.

MEMORY ORGANIZATION

The following diagram illustrates the organization of memory:



The command string, of length I , is stored in reverse at the top of memory. To insert this block at current position P , the text below the pointer (length $E-P$) is first moved down to the end of the command string at $T-I$. The saved text below the pointer extends from $(T-I)$ up to $(T-I)-(E-P)$.

The inserted text is then moved up to P , extending from there to $P+I$. Memory overflow occurs if

$$P+I > (T-I)-(E-P)$$

or equivalently

$$2I > T-E$$

which implies that no single text insertion can ever exceed half the remaining available buffer space. When working with a full memory buffer the user should beware of memory overflow lest his efforts prove fruitless.

DISPLAY

Certain of the commands are followed by a context display showing, when possible, eight lines preceding and eight lines following the current pointer position. For example, the display pops up on the last of any Change or Insert command or the Move Lines command. The context display can be invoked at any time by the $\emptyset L$ command. The display does not appear for block commands.

The position of the current activity pointer is shown as the screen representation of $\emptyset FFH$ (a white block on some monitors). This pointer representation character can easily be changed by the user since $\emptyset FFH$ may delete a character on some monitors. It should be noted that the activity pointer is always assumed to be positioned between two characters.

The pointer character never appears if the pointer is positioned at either end of the memory file.

To change the cursor character, load EDIT into memory without entering the program. Use DEBUG to search for the byte combination 3E FF representing the instruction MVI A, $\emptyset FFH$. Change the FF to any desired character. It is suggested that the cursor character be unique and recognizable at a glance. Save the modified version of EDIT.

DEBUG

An 8080/Z80 debug, monitor and disassembler
program development system.

Copyright 1978

Allen Ashley
395 Sierra Madre Villa
Pasadena, CA 91107

(213) 793-5748

INTRODUCTION

DEBUG is an incomparable software development tool featuring single-step execution of Z80 or 8080 programs with complete display of all register contents, flag status, and trace display, in mnemonic form, of the instruction just executed and the next instruction to be executed. The single-step breakpoint can be located anywhere in the user's program.

DEBUG also allows the user to disassemble Z80 and 8080 programs, examine or modify memory, move or compare blocks of memory, and search for specific byte strings.

DEBUG combines a disassembler, a debug package, and the commonly used monitor routines.

With two exceptions (easily modified by the user) DEBUG is written entirely in the 8080 subset of the Z80 instructions. DEBUG is thus operational on either 8080 or Z80 machines. DEBUG is therefore a recommended development tool for those 8080 owners anticipating a future expansion to the Z80 processor.

DEBUG ORGANIZATION

DEBUG contains an overall executive which interprets user commands and branches to the appropriate module to execute those commands. Upon termination of any DEBUG command, control is returned to the DEBUG executive. Exit from the DEBUG executive returns control to the entry of the disk operating system.



Executive commands consist of single characters which must be entered after the executive prompt (<). Parameters required for any command are entered as a sequence of hexadecimal characters, of which only the last four characters entered are considered valid. A hexadecimal parameter is terminated by any non-valid hexadecimal character.

Pressing Control-C when entering a hex parameter returns control to the monitor.

EXECUTIVE COMMANDS

A DISPLAY CONTENTS OF MEMORY IN ASCII.

DEBUG responds with the @ prompt, requesting an address at which the memory display is to begin. The display consists of a four-digit hexadecimal address followed by 64 bytes displayed as ASCII characters. Invalid ASCII (control) characters are represented by a blank. After each line displayed, the display module awaits keyboard input. Any key except 'Q' advances to the next 64-byte block. The memory pointer can be moved by pressing 'Q' and then typing a new hex address. Depressing 'Q' twice in succession returns control to the DEBUG executive.

B SET BREAKPOINT AND BEGIN EXECUTION

DEBUG responds with the @ prompt twice in succession, requesting two hexadecimal parameters. The first parameter represents the address at which the breakpoint is asserted; the second represents the address at which execution is to begin. Program execution proceeds uninterrupted up to, but not including, the instruction at the breakpoint. NOTE: The first single step executes the instruction at the breakpoint.

Upon reaching the breakpoint, DEBUG displays all the current Z80 registers, the mnemonic of the next instruction to be executed, the CMZ flag status, and memory locations pointed to by each of the registers. Register contents are exhibited as four-character hexadecimal numbers. The format is as follows.

PC	AF	BC	DE	HL	SP	IX	IY	Mnemonic of instruction just executed
(PC)	(AF)	(BC)	(DE)	(HL)	(SP)	(IX)	(IY)	Flags - next instruction to be executed

W — 16 bytes at memory window —

where (REG) represents the (byte reversed) memory contents pointed to by REG, and the memory window displays any desired 16 bytes of memory.



The breakpoint is asserted as Restart 3 (call to 18H). Prior to execution, DEBUG transfers any existing user instruction at 18H, places a jump to DEBUG at 18H, then replaces the user instruction at the breakpoint by a RST 3. Encountering the breakpoint, DEBUG saves the Z80 registers, removes the break, restores the contents at 18H, displays the registers, and jumps to the single-step executive.

The user must not attempt to impose a breakpoint in non-existent or read-only memory. Similarly, the user must not assert a second breakpoint without clearing any former break. If program execution terminates before reaching the break, the breakpoint can be cleared by forcing execution at either the breakpoint or 18H. The breakpoint must be the first byte of a multibyte instruction.

B (Cont'd)

SINGLE STEP EXECUTIVE

When the target program reaches the breakpoint, control is transferred to the single step executive. The single step executive controls further execution of the target program. Commands to the executive consist of a hexadecimal parameter (n) followed by a terminating character. The terminating character defines the command to the executive

SINGLE STEP Commands:

Space bar allows the program to execute the next instruction.

G frees the target program to proceed with uninhibited execution.

W resets the memory window to the position defined by the hex parameter (n).



R asserts a breakpoint at the address given by the top of the user's stack. The target program executes uninterrupted until the new breakpoint is reached. The user must ensure that the top of the stack contains a valid return address.

P resets the breakpoint to the location defined by the hex parameter (n).

Q terminates execution of the target program and returns control to the DEBUG executive.

K abandons single step, but imposes a breakpoint at the instruction just executed. This option is useful for tracking program execution through a loop. The single step executive regains control the next time the program reaches the breakpoint.

O displays only the mnemonic of the next instruction to be executed. The single step executive maintains a toggle which is switched for each execution of the 'O' command. The first execution switches the display to the mnemonics only; the second execution of 'O' resumes the full register display, etc.

B (Cont'd)

Z sets the 8 bit registers. After detecting 'Z' the single step executive awaits a sequence of commands of the form

Rnn

where R is any of the 8 bit registers A, B, C, D, E, F, H, L or M, and nn is a hexadecimal value to be inserted into the register. Control is returned to the single step executive by typing a carriage return instead of a register character.

X executes the next n instructions, without interruption, before returning control to the single step executive.

- * I releases the target program but asserts a break onto the top of the user's stack. The 'R' command places a break in the program at the return address. The 'I' command directs the return to the DEBUG package.
- * N forces program execution to resume at location n, maintaining single step control.
- * J traces transfer instructions (JMP, CALL, etc.) only. The 'J' command is a toggle, as the 'O'.

T sets a program trap. The target program is released for controlled execution. The single step executive will regain control when any 16-bit register contains the value n, or a memory reference is made to address n.

If the target program branches to read-only memory, DEBUG moves the breakpoint to the return address, allowing ROM instructions to be executed and trapping the program upon the return to RAM.

The single step feature of DEBUG will prove to be the user's single most powerful program development tool. It is highly recommended that every effort be made to become familiar with operation of the single step executive.

The single step trace option will prove to be a much more potent analytical device than a simple breakpoint because it allows the user to monitor program evolution.

* Commands available on special DEBUG versions only.

C COMPARE TWO BLOCKS OF MEMORY.

DEBUG responds with the @ prompt thrice in succession. The required parameters are respectively start and end of the first memory block, and start of the second memory block. DEBUG displays the location and contents of all bytes which differ in the two memory blocks. Control is returned to the DEBUG executive. Control-C returns to monitor.

D DISASSEMBLE MEMORY BY SINGLE INSTRUCTIONS.

With the @ prompt, DEBUG requests a starting address. Instructions are disassembled into the MAKRO mnemonics, one instruction at a time, awaiting keyboard input before proceeding. Depressing the space bar will advance to the next sequential instruction. Depressing 'Q' returns control to the DEBUG executive.

Typing any valid hexadecimal address will advance the disassembly pointer to that address and resume sequential disassembly from that point.

E EXAMINE AND MODIFY MEMORY.

The @ prompt requests a starting location. DEBUG displays the current contents and awaits the new hexadecimal value to be inserted in memory. Only the last two hex characters are considered valid. Typing 'Q' returns control to the DEBUG executive. Values to be stored in memory must be terminated by carriage return.

F FILL A BLOCK OF MEMORY WITH A CONSTANT.

DEBUG responds with a # prompt, requesting the constant hexadecimal value. The two @ prompts then following request the starting and ending address of the memory block to be filled. Control is automatically returned to the DEBUG executive.

G EXECUTE.

DEBUG responds with the @ prompt to request the address at which execution is to begin.

M MOVE A BLOCK OF MEMORY.

DEBUG responds with three successive @ prompts representing, respectively, the start and end of the source block, and the start of the destination block. Control is returned to the DEBUG executive.

Q EXIT FROM THE DEBUG EXECUTIVE.

Control is transferred to the disk operating system.

S SEARCH MEMORY FOR SPECIFIED BYTE STRING.

DEBUG accepts the sought-for byte string, up to five bytes in length, immediately after receiving the S command. The byte string is entered as a sequence of the group

2 hex digits followed by a space

The byte string is terminated by a carriage return. Each group of hex digits, including the last, must be followed by a space. Following the carriage return terminating the byte string, DEBUG requests a starting address for the search with the @ prompt.

Memory is searched from the starting address to higher address values, wrapping around to reach the start. The search is interrupted to display the next occurrence of the byte string. The memory pointer to the start of the string is displayed. Successive realizations of the byte string are located by depressing the space bar. At each pause, control can be returned to the DEBUG executive by 'Q'.

An active search can be terminated by Control-C.

The power of the search mode is considerably enhanced by the capability of searching for a given byte string under a specified mask string. The mask string enables the user to include 'don't care bytes' and modified bytes within the string. To illustrate the search-under-mask option, a match between memory byte B and input string byte I is defined as a zero result of the following operation.

(EXCLUSIVE OR OF B AND I) AND NOT MASK

Agreement between the input string and memory is found if and only if a match is found for each byte in the sequence. By default the mask is zero, in which case a match requires identity between the memory and input bytes. If the mask is 0FFH, any memory byte is accepted as a match. The search-under-mask option is enabled by entering the byte string as a sequence of

4 hex digits followed by a space

The first two of these four digits represent the mask byte; the second two digits represent the sought-for byte.

The byte string found in memory can be changed if the user presses 'C' when the search pauses. An input byte string, as that used to define the sought string, can then overlay the memory bytes. The overlay string may be longer, shorter, or equal to the search string. The overlay string is terminated with a carriage return.

T DISASSEMBLE A SEQUENTIAL BLOCK OF MEMORY.

DEBUG responds with the @ prompt twice in succession, representing the start and end of the memory block. The entire block is disassembled without user interaction. Control is returned to the DEBUG executive.

V VIEW MEMORY IN HEXADECIMAL.

The @ prompt requests a starting address. DEBUG displays memory in successive 16-byte groups starting at the input address. Depressing the space bar advances the display to the next 16-byte group. Pressing 'Q' returns control to the DEBUG executive.

USING DEBUG

Experience will prove DEBUG to be an indispensable programming aid. While these notes cannot substitute for that experience, they may assist the user to more rapidly acquire total facility in the operation of DEBUG. The following material adopts, as the measure of programming effort, the time it takes a program to move from the conceptual stage to a fully operational version. It is the intent of these notes to assist the user to exploit DEBUG to minimize that time.

The first point to be made regards programming style: quality software is born in a planning stage. A well-planned program will be up and running long before one poorly conceived, regardless of the development aids. It is altogether too easy to become overly reliant upon DEBUG, in that the user may be drawn into the trap of hastily assembling a program with the assumption that DEBUG will cure all the problems. DEBUG should be used in conjunction with, rather than as a substitute for a planning stage.

From the standpoint of time, however, too much planning may increase the overall development time. As a guideline, one should structure out his concept so that critical program functions are as nearly independent of each other as possible. It is vanity to try to get anything but the simplest programming task to execute properly on the first try. The user should assume that the initial effort will contain errors and structure the program to minimize the extent of the damage caused by any individual error.

Define a major cycle as one trip through the circuit: text edit, assembly, execute/debug. We wish to minimize the total number of such major cycles. Overall development time is minimized not by producing an error-free initial effort, but by limiting the number of development passes.

As much as possible, we want to avoid the serial discovery of errors -- picking up one fatal error on each major cycle. The bulk of the planning effort should be directed to those aspects of the program which must function first.

The first function of the DEBUG package is to bridle the fury of a program error. Let us define a minor cycle as the sequence: reload the program and debug package and try again. Each development pass can contain many such minor cycles, since a simple error can erase memory. The user should learn to manipulate the breakpoint and single step features of DEBUG to maximize the number of errors identified on each minor cycle.

On the first minor cycle, DEBUG should be used to insert a breakpoint before the first subroutine call or major logic branch of the main program. If the program fails before the breakpoint, the minor cycle must be repeated with the break inserted earlier. At a subroutine call, the user should initially trip over the call with the 'R' command to eliminate wasteful single stepping. In the early DEBUG stages, the breakpoint should be used to divide coarsely the program into good and bad zones.

Fatal errors which can be patched without reassembly should be corrected on a fresh copy (newly loaded) of the program, which should then be stored on disk. Minor cycles are much faster than a development pass. All such patches should be noted for the next assembly.

The search to localize an error should be taken in broad steps initially, via the 'R' and 'P' commands, increasing the fineness of the step gradually. If a subroutine call is found to result in an error, then that subroutine should be entered in the single step mode, but any calls out of that routine should be tripped over by the 'R' command.

Whenever possible, the user should try to keep an errant program in execution rather than abort, patch, and start over. Program operations which result in a misdirected branch or faulty register contents should be corrected by the 'N' or 'Z' commands, respectively.

The memory window should be set to monitor a critical memory area away from the current focus; it should be regarded as a rear-view mirror. The memory window may be moved about freely in the single step mode without advancing the program.

Versions of DEBUG supplied for units with software-controlled hardware interrupt (e.g., POLY-88, COMPAL-80) contain a trap feature which will allow the target program to execute until any (16-bit) register contains the trap value or any memory reference is made to the trap address. The trap feature is perfect for finding that program error which results in overwriting memory. In these versions of DEBUG, the 'H' command displays the last five instructions executed. These special versions of DEBUG can single step programs through read only memory.

The 'I' command, implemented only in the special versions of DEBUG, was set up to replace the 'R' command when single stepping the program through ROM. The 'R' command will not work when the return address points to ROM. The 'I' command is outwardly identical to the 'R' command.

The 'K' command is used to keep DEBUG in the simple breakpoint mode, allowing the user to monitor program flow past a critical point.