

LINK-80™

OPERATOR'S GUIDE



DIGITAL RESEARCH®

LINK-80™ OPERATOR'S GUIDE

LINK-80 OPERATOR'S GUIDE

Copyright (c) 1980

**Digital Research
P.O. Box 579
801 Lighthouse Avenue
Pacific Grove, CA 93950
(408) 649-3896
TWX 910 360 5001**

All Rights Reserved

COPYRIGHT

Copyright (c) 1980 by Digital Research. All rights reserved. No part of this publication may be reproduced, transmitted, transcribed, stored in a retrieval system, or translated into any language or computer language, in any form or by any means, electronic, mechanical, magnetic, optical, chemical, manual or otherwise, without the prior written permission of Digital Research, Post Office Box 579, Pacific Grove, California, 93950.

This manual is, however, tutorial in nature. Thus, permission is granted to reproduce or abstract the example programs shown in the enclosed figures for the purposes of inclusion within the reader's programs.

DISCLAIMER

Digital Research makes no representations or warranties with respect to the contents hereof and specifically disclaims any implied warranties of merchantability or fitness for any particular purpose. Further, Digital Research reserves the right to revise this publication and to make changes from time to time in the content hereof without obligation of Digital Research to notify any person of such revision or changes.

TRADEMARKS

CP/M is a registered trademark of Digital Research. PL/I-80, MP/M-80, RMAC, SID, ZSID and TEX are trademarks of Digital Research.

The "LINK-80 Operator's Guide" was prepared using the Digital Research TEX Text formatter.

* Second Printing: December, 1980 *

TABLE OF CONTENTS

1.	LINK LINKAGE EDITOR	1
1.1.	LINK Operation	1
1.2.	LINK Switches	2
1.2.1.	The Additional Memory (A) Switch	2
1.2.2.	The Data Origin (D) Switch	2
1.2.3.	The Go (G) Switch	2
1.2.4.	The Load Address (L) Switch	2
1.2.5.	The Memory Size (M) Switch	3
1.2.6.	The No List (NL) Switch	3
1.2.7.	The No Recording of Symbols (NR) Switch	3
1.2.8.	The Output COM File (OC) Switch	3
1.2.9.	The Output PRL File (OP) Switch	3
1.2.10.	The Program Origin (P) Switch	3
1.2.11.	The '?' Symbol (Q) Switch	4
1.2.12.	The Search (S) Switch	4
1.3.	Creating MP/M PRL Files	4
1.4.	Sample Link	5
1.5.	Error Messages	9
1.6.	Format of REL Files	10
1.7.	Format of IRL Files	13
2.	RMAC RELOCATING MACRO ASSEMBLER	13
2.1.	RMAC Operation	13
2.2.	Expressions	13
2.3.	Assembler Directives	14
2.3.1.	The ASEG Directive	15
2.3.2.	The CSEG Directive	15
2.3.3.	The DSEG Directive	15
2.3.4.	The COMMON Directive	15
2.3.5.	The PUBLIC Directive	15
2.3.6.	The EXTRN Directive	16
2.3.7.	The NAME Directive	16
3.	LIB PROGRAM LIBRARIAN	17
3.1.	LIB Operation	17
3.2.	Error Messages	18
4.	DATA REPRESENTATION AND INTERFACE CONVENTIONS	19
4.1.	Representation of Data Elements	19
4.1.1.	Pointers, and Entry and Label Variables	19
4.1.2.	Fixed Binary Data Format	19
4.1.3.	Bit Data Representation	20
4.1.4.	Character Data Representation	20
4.1.5.	Fixed Decimal Data Representation	21
4.1.6.	Floating Point Binary Representation	21
4.1.7.	File Constant Representation	22
4.2.	Layout of Aggregate Storage	22
4.3.	General Parameter Passing Conventions	23
4.4.	Returning Values from Functions	24
4.4.1.	Returning Pointer, Entry, and Label Variables	28
4.4.2.	Returning Fixed Binary Data	28

4.4.3.	Returning Bit String Data	28
4.4.4.	Returning Character Data	28
4.4.5.	Returning Fixed Decimal Data	29
4.4.6.	Returning Floating Point Numbers	29
5.	PL/I-80 RUNTIME SUBROUTINES	33
5.1.	Stack and Dynamic Storage Subroutines	33
5.1.1.	The TOTWDS and MAXWDS Functions	33
5.1.2.	The ALLWDS Subroutine	34
5.1.3.	The STKSIZ Function	34
5.2.	PL/I-80 Runtime Subroutine Entry Points	39
5.3.	Direct CP/M Function Calls	43

APPENDIXES

A:	Listing of "PLIDIO" Direct CP/M Call Entry Points	46
B:	Listing of "DIOCALLS" Showing the Basic CP/M Direct Interface	59
C:	Listing of "DIOCOPY" Showing Direct CP/M File I/O Operations	67
D:	Listing of "DIORAND" Showing Extended Random Access Calls	73
E:	Description of Overlays and File Location Controls	78
F:	Description of XREF Cross-Reference Utility	90

1. LINK LINKAGE EDITOR.

LINK is a utility used to combine relocatable object modules into an absolute file ready for execution under CP/M or MP/M. The relocatable object modules may be of two types. The first has a filetype of REL, and is produced by PL/I-80, RMAC, or any other language translator that produces relocatable object modules in the Microsoft format. The second has a filetype of IRL, and is generated by the CP/M librarian LIB. An IRL file contains the same information as a REL file, but includes an index which allows faster linking of large libraries.

Upon completion, LINK lists the symbol table, any unresolved symbols, a memory map and the use factor at the console. The memory map shows the size and locations of the different segments, and the use factor indicates the amount of available memory used by LINK as a hexadecimal percentage. LINK writes the symbol table to a SYM file suitable for use with the CP/M Symbolic Instruction Debugger (SID), and creates a COM or PRL file for direct execution under CP/M or MP/M.

1.1. LINK Operation

LINK is invoked by typing

```
LINK filenamel{,filename2,...,filenameN}
```

where filenamel,...,filenameN are the names of the object modules to be linked. If no filetype is specified, REL is assumed. LINK will produce two files: filenamel.COM and filenamel.SYM. If some other filename is desired for the COM and SYM files, it may be specified in the command line as follows:

```
LINK newfilename=filenamel{,filename2,...,filenameN}
```

When linking PL/I programs, LINK will automatically search the run-time library file PLILIB.IRL on the default disk and include any subroutines used by the PL/I programs.

A number of optional switches, provided for additional control of the link operation, are described in the following section.

During the link process, LINK may create up to eight temporary files on the default disk. The files are named:

```
XXABS.$$$ XXPROG.$$$ XXDATA.$$$ XXCOMM.$$$  
YYABS.$$$ YYPROG.$$$ YYDATA.$$$ YYCOMM.$$$
```

These files are deleted if LINK terminates normally, but may remain on the disk if LINK aborts due to an error condition.

1.2. LINK Switches

LINK switches are used to control the execution parameters of LINK. They are enclosed in square brackets immediately following one or more of the filenames in the command line, and are separated by commas.

Example:

```
LINK TEST[L4000],IOMOD,TESTLIB[S,NL,GSTART]
```

All switches except the S switch may appear after any filename in the command line. The S switch must follow the filename to which it refers.

1.2.1. The Additional Memory (A) Switch. The A switch is used to provide LINK with additional space for symbol table storage by decreasing the size of LINK's internal buffers. This switch should be used only when necessary, as indicated by a MEMORY OVERFLOW error, since using it causes the internal buffers to be stored on the disk, thus slowing down the linking process considerably.

1.2.2. The Data Origin (D) Switch. The D switch is used to specify the origin of the data and common segments. If not used, LINK will put the data and common segments immediately after the program segment. The form of the D switch is Dnnnn, where nnnn is the desired data origin in hex.

1.2.3. The Go (G) Switch. The G switch is used to specify the label where program execution is to begin, if it does not begin with the first byte of the program segment. LINK will put a jump to the label at the load address. The form of the G switch is G<label>.

1.2.4. The Load Address (L) Switch. The load address defines the base address of the COM file generated by LINK. Normally, the load address is 100H, which is the base of the Transient Program Area in a standard CP/M system. The form of the L switch is Lnnnn, where nnnn is the desired load address in hex. The L switch also sets the program origin to nnnn, unless otherwise defined by the P switch.

Note that COM files created with a load address other than 100H will not execute properly under a standard CP/M system.

1.2.5. The Memory Size (M) Switch. The M switch may be used when creating PRL files for execution under MP/M to indicate that additional data space is required by the PRL program for proper execution. The form of the M switch is Mnnnn, where nnnn is the amount of additional data space needed in hex.

1.2.6. The No List (NL) Switch. The NL switch is used to suppress the listing of the symbol table at the console.

1.2.7. The No Recording of Symbols (NR) Switch. The NR switch is used to suppress the recording of the symbol table file.

1.2.8. The Output COM File (OC) Switch. The OC switch directs LINK to produce a COM file. This is the default condition for LINK.

1.2.9. The Output PRL File (OP) Switch. The OP switch directs LINK to produce a page relocatable PRL file for execution under MP/M, rather than a COM file. See section 1.3 for more information on creating PRL files.

1.2.10. The Program Origin (P) Switch. The P switch is used to specify the origin of the program segment. If not used, LINK will put the program segment at the load address, which is 100H unless otherwise specified by the L switch. The form of the P switch is Pnnnn, where nnnn is the desired program origin in hex.

1.2.11. The '?' Symbol (Q) Switch. Symbols in the PL/I run-time library begin with a question mark to avoid conflict with user symbols. Normally LINK suppresses listing and recording of these symbols. The Q switch causes these symbols to be included in the symbol table listed at the console and recorded on the disk.

1.2.12. The Search (S) Switch. The S switch is used to indicate that the preceding file should be treated as a library. LINK will search the file and include only those modules containing symbols which are referenced but not defined in the modules already linked.

1.3. Creating MP/M PRL Files

Assembly language programs often contain references to symbols in the base page such as BOOT, BDOS, DFCB, and DBUFF. To run properly under CP/M (or as a COM file under MP/M) these symbols are simply defined in equates as follows:

```
BOOT EQU 0 ;JUMP TO WARM BOOT
BDOS EQU 5 ;JUMP TO BDOS ENTRY POINT
DFCB EQU 5CH ;DEFAULT FILE CONTROL BLOCK
DBUFF EQU 80H ;DEFAULT I/O BUFFER
```

With PRL files, however, the base page itself may be relocated at load time, so LINK must know that these symbols, while at fixed locations within the base page, are relocatable. To do this, simply declare these symbols as externals in the modules in which they are referenced:

```
EXTRN BOOT, BDOS, DFCB, DBUFF
```

and link in another module in which they are declared as publics and defined in equates:

```
PUBLIC BOOT, BDOS, DFCB, DBUFF
BOOT EQU 0 ;JUMP TO WARM BOOT
BDOS EQU 5 ;JUMP TO BDOS ENTRY POINT
DFCB EQU 5CH ;DEFAULT FILE CONTROL BLOCK
DBUFF EQU 80H ;DEFAULT I/O BUFFER
END
```

1.4. Sample Link

A sample link is shown on the following pages. First the sample program GRADE.PLI is compiled, and then a COM file is created by LINK. LINK automatically searches the PL/I run-time library PLILIB.IRL for the subroutines used by GRADE. The Q switch causes the symbols taken from PLILIB.IRL to be included in the symbol table listing (and the SYM file). The memory map following the symbol table indicates the length and location assigned to each of the segments. A use factor of 49 indicates that 49H%, or a little more than a quarter of the memory available to LINK was used.

PL/I-80 V1.0, COMPILATION OF: GRADE

D: Disk Print

L: List Source Program

NO ERROR(S) IN PASS 1

NO ERROR(S) IN PASS 2

PL/I-80 V1.0, COMPILATION OF: GRADE

```
1 a 0000 average:
2 a 0006      proc options (main);
3 a 0006      /* grade averaging program */
4 a 0006
5 c 0006      dcl
6 c 000D      sysin file,
7 c 000D      (grade,total,n) fixed;
8 c 000D
9 c 000D      on error (1)
10 c 0014     /* conversion */
11 d 0014      begin;
12 e 0017      put skip list('Bad Value, Try Again.');
13 e 0033      get skip;
14 e 0044      go to retry;
15 d 0047      end;
16 d 0047
17 c 0047      on endfile (sysin)
18 d 004F      begin;
19 e 0052      if n ^= 0 then
20 e 005B      put skip list
21 e 008A      ('Average is',total/n);
22 e 008A      stop;
23 d 008D      end;
24 d 008D
25 c 008D      put skip list
26 c 00A9      ('Type a List of Grades, End with Ctl-Z');
27 c 00A9      total = 0;
28 c 00AF      n = 0;
29 c 00B9
30 c 00B9      retry:
31 c 00B9      put skip;
32 c 00CA      do while('l'b);
33 c 00CA      get list (grade);
34 c 00E2      total = total + grade;
35 c 00ED      n = n + 1;
36 c 00F7      end;
37 a 00F7      end average;
```

CODE SIZE = 00F7

DATA AREA = 004C

B>link grade[q]

LINK V0.4

AVERAG	0100	/SYSIN/	1B77	?START	1A08	?ONCOP	18AE
?SYSPR	02C5	?SKPOP	0430	?SLCTS	1367	?PPNCOP	01FD
?QIOOP	1987	?SYSIN	02C1	?ID22N	13B3	?QICOP	127E
?PNVOP	0221	?STOPX	1B19	?RECOV	1468	?GNVOP	07D5
?QCIOP	11FB	/?FILAT/	1B9C	/?FPB/	1BA5	?PPNBOP	01F7
?PNCPR	04CF	?IS22N	13F9	?SIOOP	02CA	?SIOPR	02E8
/?FPBST/	1BD3	/SYSPRI/	1BE6	?OIOOP	05A7	?FPBIO	0758
?OIOPR	05C6	?BSL16	131C	?SIGNA	1626	?SKPPR	0439
?GNCPN	094F	?WRBYT	0E36	?PAGOP	07C7	?NSTOP	1322
?SMVCM	1390	?SJ SVM	132D	?SSCFS	137A	?QB08I	11E7
?OPNFI	0D13	/?FMTS/	1C0E	?FPBOU	19DB	?FPBIN	1993
?GNVPR	0812	?RDBYT	0E23	?RDBUF	0E5C	?WRBUF	0E7F
?CLOSE	0F68	?GETKY	0F99	?SETKY	0FBF	?PATH	0F4C
?BDOS	0005	?DFCB0	005C	?DFCB1	006C	?DBUFF	0080
?ALLOP	14D2	?FREOP	1568	?ADDIO	1A64	?SUBIO	1A7B
?WRCHR	19F1	?RFSIZ	10C4	?RRFCB	1136	?RWFCB	113B
?QB16I	11EA	?IN20	13F1	?CNVER	1400	?BSL08	1316
?SJSCM	132F	?SJSTS	1341	?SLVTS	1365	?SMCCM	1394
?ID22	13CB	?IN20N	13F1	?ZEROD	1420	?IS22	13F9
/?CONSP/	1C16	?OFCOP	14B2	?RSBLK	1437	?RECLS	1E79
?ERMSG	1B34	?BEGIN	1E77	/?ONCOD/	1C37	?SIGOP	1616
?STACK	1E71	?ONCPC	194B	?REVOP	1903	/?CNCOL/	1C3A
?BOOT	0000	?CMEM	1B77	?DMEM	1E7B		

ABSOLUTE 0000

CODE SIZE 1A77 (0100-1B76)

DATA SIZE 023F (1C3C-1E7A)

COMMON SIZE 00C5 (1B77-1C3B)

USE FACTOR 49

A>b:grade

Type a List of Grades, End with Ctl-Z
50, 75, 25
^Z

Average is 50
End of Execution

A>b:grade

Type a List of Grades, End with Ctl-Z
50
75
zot,66

Bad Value, Try Again.
25
^Z

Average is 50
End of Execution

A>b:grade

Type a List of Grades, End with Ctl-Z
^Z

End of Execution

1.5. Error Messages

CANNOT CLOSE: An output file cannot be closed. The diskette may be write protected.

COMMON ERROR: An undefined common block has been selected.

DIRECTORY FULL: There is no directory space for the output files or intermediate files.

DISK READ ERROR: A file cannot be read properly.

DISK WRITE ERROR: A file cannot be written properly, probably due to a full diskette.

FILE NAME ERROR: The form of a source file name is invalid.

FIRST COMMON NOT LARGEST: A subsequent COMMON declaration is larger than the first COMMON declaration for the indicated block. Check that the files being linked are in the proper order, or that the modules in a library are in the proper order.

INDEX ERROR: The index of an IRL file contains invalid information.

INSUFFICIENT MEMORY: There is not enough memory for LINK to allocate its buffers. Try using the A switch.

INVALID REL FILE: The file indicated contains an invalid bit pattern. Make sure that a REL or IRL file has been specified.

INVALID SYNTAX: The command line used to invoke LINK was not properly formed.

MAIN MODULE ERROR: A second main module was encountered.

MEMORY OVERFLOW: There is not enough memory to complete the link operation. Try using the A switch.

MULTIPLE DEFINITION: The specified symbol is defined in more than one of the modules being linked.

NO FILE: The indicated file cannot be found.

OVERLAPPING SEGMENTS: LINK attempted to write a segment into memory already used by another segment. Probably caused by incorrect use of P and/or D switches.

UNDEFINED START SYMBOL: The symbol specified with the G switch is not defined in any of the modules being linked.

UNDEFINED SYMBOLS: The symbols following this message are referenced but not defined in any of the modules being linked.

UNRECOGNIZED ITEM: An unfamiliar bit pattern has been scanned (and ignored) by LINK.

1.6. Format of REL Files

The information in a REL file is encoded in a bit stream, which is interpreted as follows:

1) If the first bit is a 0, then the next 8 bits are loaded according to the value of the location counter.

2) If the first bit is a 1, then the next 2 bits are interpreted as follows:

00 - special link item (see 3)

01 - program relative. The next 16 bits are loaded after being offset by the program segment origin.

10 - data relative. The next 16 bits are loaded after being offset by the data segment origin.

11 - common relative. The next 16 bits are loaded after being offset by the origin of the currently selected common block.

3) A special item consists of:

- A 4 bit control field which selects one of 16 special link items described below.

- An optional value field which consists of a 2 bit address type field and a 16 bit address field. The address type field is interpreted as follows:

00 - absolute

01 - program relative

10 - data relative

11 - common relative

- An optional name field which consists of a 3 bit name count followed by the name in 8 bit ASCII characters.

The following items are followed by a name field only.

0000 - entry symbol. The symbol indicated in the name field is defined in this module, so the module should be linked if the current file is being searched (as indicated by the S switch).

0001 - select common block. Instructs LINK to use the location counter associated with the common block indicated in the name field for subsequent common relative items.

0010 - program name. The name of the relocatable module. LINK checks that the first item in each module is a program name, and issues an error if it is not.

0011 - unused.

0100 - unused.

The following items are followed by a value field and a name field.

0101 - define common size. The value field determines the amount of memory to be reserved for the common block described in the name field. The first size allocated to a given block must be larger than or equal to any subsequent definitions for that block in other modules being linked.

0110 - chain external. The value field contains the head of a chain which ends with an absolute 0. Each element of the chain is to be replaced with the value of the external symbol described in the name field.

0111 - define entry point. The value of the symbol in the name field is defined by the value field.

1000 - unused.

The following items are followed by a value field only.

1001 - external plus offset. The following two bytes in the current segment must be offset by the value of the value field after all chains have been processed.

1010 - define data size. The value field contains number of bytes in the data segment of the current module.

1011 - set location counter. Set the location counter to the value determined by the value field.

1100 - chain address. The value field contains the head of a chain which ends with an absolute 0. Each element of the chain is to be replaced with the current value of the location counter.

1101 - define program size. The value field contains the number of bytes in the program segment of the current module.

1110 - end module. Defines the end of the current module. If the value field contains a value other than absolute 0, it is to be used as the start address for the program being linked. The next item in the file will start at the next byte boundary.

The following item has no value field or name field.

1111 - end file. Follows the end module item of the last module in the file.

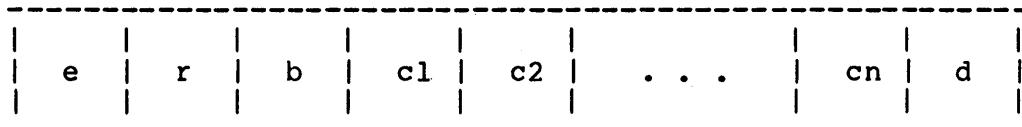
1.7. Format of IRL Files

An IRL file consists of three parts: a header, an index and a REL section.

The header contains 128 bytes defined as follows:

byte 0 - extent number of first record of REL section.
byte 1 - record number of first record of REL section.
bytes 2-127 - currently unused.

The index consists of a number of entries corresponding to the entry symbol items in the REL section. The entries are of the form:



where:

e = extent offset from start of REL section to start of module
r = record offset from start of extent to start of module
b = byte offset from start of record to start of module
c1-cn = name of symbol
d = end of symbol delimiter (0FEH)

The index is terminated by an entry in which c1 = 0FFH. The remainder of the record containing the terminating entry is unused.

The REL section contains the relocatable object code as described in the previous section.

2. RMAC RELOCATING MACRO ASSEMBLER.

The CP/M Relocating Macro Assembler, called RMAC, is a modified version of the CP/M Macro Assembler (MAC). RMAC produces a relocatable object file (REL), rather than an absolute object file (HEX), which may be linked with other modules produced by RMAC, or other language translators such as PL/I-80, to produce an absolute file ready for execution.

The differences between RMAC and MAC are described in the following sections. For a complete description of the assembly language and macro facilities, see CP/M MAC Macro Assembler: Language Manual and Application Guide.

2.1. RMAC Operation

RMAC is invoked by typing

RMAC filename.filetype

followed by optional assembly parameters. If the filetype is not specified, ASM is assumed. RMAC produces three files: a list file (PRN), a symbol file (SYM), and a relocatable object file (REL). Characters entered in the source file in lower case appear in lower case in the list file, except for macro expansions.

The assembly parameter "H" in MAC, used to control the destination of the HEX file, has been replaced by "R", which controls the destination of the REL file. Directing the REL file to the console or printer (RX or RP) is not allowed, since the REL file does not contain ASCII characters.

Example:

RMAC TEST \$PX SB RB

directs RMAC to assemble the file TEST.ASM, send the PRN file to the console, and put the symbol file (SYM) and the relocatable object file (REL) on drive B.

2.2. Expressions

The operand field of a statement may consist of a complex arithmetic expression (as described in the MAC manual, section 3) with the following restrictions:

- 1) In the expression A+B, if A evaluates to a relocatable value or

- an external, then B must be a constant.
- 2) In the expression A-B, if A is an external, then B must be a constant.
- 3) In the expression A-B, if A evaluates to a relocatable value, then:
- B must be a constant, or
 - B must be a relocatable value of the same relocation type as A (both must appear in a CSEG, DSEG, or in the same COMMON block).
- 4) In all other arithmetic and logical operations, both operands must be absolute.

An expression error ('E') will be generated if an expression does not follow the above restrictions.

2.3. Assembler Directives

The following assembler directives have been added to support relocation and linking of modules:

ASEG	use absolute location counter
CSEG	use code location counter
DSEG	use data location counter
COMMON	use common location counter
PUBLIC	symbol may be referenced in another module
EXTRN	symbol is defined in another module
NAME	name of module

The directives ASEG, CSEG, DSEG and COMMON allow program modules to be split into absolute, code, data and common segments, which may be rearranged in memory as needed at link time. The PUBLIC and EXTRN directives provide for symbolic references between program modules.

NOTE: While symbol names may be up to 16 characters, the first six characters of all symbols in PUBLIC, EXTRN and COMMON statements must be unique, since symbols are truncated to six characters in the object module.

2.3.1. The ASEG Directive. The ASEG statement takes the form

label ASEG

and instructs the assembler to use the absolute location counter until otherwise directed. The physical memory locations of statements following an ASEG are determined at assembly time by the absolute location counter, which defaults to 0 and may be reset to another value by an ORG statement following the ASEG statement.

2.3.2. The CSEG Directive. The CSEG statement takes the form

label CSEG

and instructs the assembler to use the code location counter until otherwise directed. This is the default condition when RMAC begins an assembly. The physical memory locations of statements following a CSEG are determined at link time.

2.3.3. The DSEG Directive. The DSEG statement takes the form

label DSEG

and instructs the assembler to use the data location counter until otherwise directed. The physical memory locations of statements following a DSEG are determined at link time.

2.3.4. The COMMON Directive. The COMMON statement takes the form

COMMON /identifier/

and instructs the assembler to use the COMMON location counter until otherwise directed. The physical memory locations of statements following a COMMON statement are determined at link time.

2.3.5. The PUBLIC Directive. The PUBLIC statement takes the form

```
PUBLIC      label{,label,...,label}
```

where each label is defined in the program. Labels appearing in a PUBLIC statement may be referred to by other programs which are linked using LINK-80.

2.3.6. The EXTRN Directive. The form of the EXTRN statement is

```
EXTRN      label{,label,...,label}
```

The labels appearing in an EXTRN statement may be referenced but must not be defined in the program being assembled. They refer to labels in other programs which have been declared PUBLIC.

2.3.7. The NAME Directive. The form of the NAME statement is

```
NAME      'text string'
```

The NAME statement is optional. It is used to specify the name of the relocatable object module produced by RMAC. If no NAME statement appears, the filename of the source file is used as the name of the object module.

3. LIB PROGRAM LIBRARIAN.

The function of LIB is to handle libraries, which are files consisting of any number of relocatable object modules. LIB can concatenate a group of REL files into a library, create an indexed library (IRL), select modules from a library, and print module names and PUBLICS from a library.

3.1. LIB Operation

LIB is invoked by typing

```
LIB filename=filename1,...,filenameN
```

This command will create a library called filename.REL from the files filename1.REL,...,filenameN.REL. If filetypes are omitted, REL is assumed.

A filename may be followed by a group of module names enclosed in parentheses. Only the modules indicated will be included in the LIB function being performed. If omitted, all modules in the file are included.

Example:

```
LIB TEST=A(A1,A2),B,C(C1-C4,C6)
```

This command will create a file TEST.REL consisting of modules A1 and A2 from A.REL, all the modules from B.REL, and the modules between C1 and C4, and C6 from C.REL.

Any of several optional switches may be included in the command line for LIB. These switches are enclosed in square brackets and appear after the first filename in the LIB command. The switches are:

I - create an indexed library (IRL)

M - print module names

P - print module names and PUBLICS

Examples:

```
LIB TEST=A,B,C
```

creates a file TEST.REL consisting of A.REL, B.REL and C.REL.

```
LIB TEST=TEST,D
```

appends D.REL to the end of TEST.REL.

LIB TEST[I]

creates an indexed library TEST.IRL from TEST.REL.

LIB TEST[I]=A,B,C,D

performs the same function as the preceding LIB examples, except no TEST.REL file is created.

LIB TEST[P]

lists all the module names and PUBLICS in TEST.REL.

3.2. Error Messages

CANNOT CLOSE: The output file cannot be closed. The diskette may be write protected.

DIRECTORY FULL: There is no directory space for the output file.

DISK READ ERROR: A file cannot be read properly.

DISK WRITE ERROR: A file cannot be written properly, probably due to a full diskette.

FILE NAME ERROR: The form of a source file name is invalid.

NO FILE: The indicated file cannot be found.

NO MODULE: The indicated module cannot be found.

SYNTAX ERROR: The command line used to invoke LIB was not properly formed.

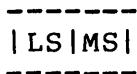
4. DATA REPRESENTATION AND INTERFACE CONVENTIONS.

This section describes the layout of memory used by various Digital Research language processors so that the programmer can properly interface assembly language routines with high level language programs and the PL/I-80 runtime subroutine library. A set of standard subroutine interface conventions is also given so that programs produced by various programmers and language processors can be conveniently interfaced.

4.1. Representation of Data Elements.

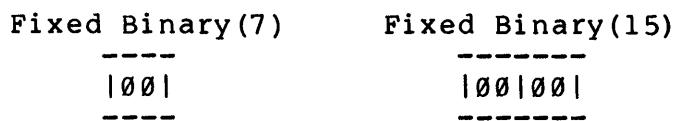
The internal memory representation of data items is presented below.

4.1.1. Pointers, and Entry and Label Variables. Variables which provide access to memory addresses are stored as two contiguous bytes, with the low order byte stored first in memory. Pointer, Entry, and Label data items appear graphically as shown below:



where "LS" denotes the least significant half of the address, and "MS" denotes the most significant portion. Note that MS is the "page address," where each memory page is 256 bytes, and LS is the address within the page.

4.1.2. Fixed Binary Data Format. Simple single and double byte signed integer values are stored in Fixed Binary format. Two modes are used, depending upon the precision of the data item. Fixed Binary values with precision 1-7 are stored as single byte values, while data items with precision 8-15 are stored in a word (double byte) location. As with other 8080, 8085, and Z-80 items, the least significant byte of multi-byte storage appears first in memory. All Fixed Binary data is represented in two's complement form, allowing single byte values in the range -128 to +127, and word values in the range -32768 to +32767. The values 0, 1, and -1 are shown graphically below, where each boxed value represents a byte of memory, with the low order byte appearing before the high order byte:



Fixed Binary(7) Fixed Binary(15)

-----	-----
01	01 00
-----	-----

Fixed Binary(7) Fixed Binary(15)

-----	-----
FE	FE FF
-----	-----

4.1.3. Bit Data Representation. Bit String data, like the Fixed Binary items shown above, are represented in two forms, depending upon the declared precision. Bit Strings of length 1-8 are stored in a single byte, while Bit Strings of length 9-16 occupy a word (double byte) value. Bit values are left justified in the word, with "don't care" bits to the right when the precision is not exactly 8 or 16 bits. The least significant byte of a word value is stored first in memory. The Bit String constant values '1'b, 'A0'b4, and '1234'b4 are stored as shown below

Bit(8) Bit(16)

-----	-----
80	00 80
-----	-----

Bit(8) Bit(16)

-----	-----
A0	00 A0
-----	-----

Bit(8) Bit(16)

N/A	-----
	34 12

4.1.4. Character Data Representation. Two forms of character data are stored in memory, depending upon the declaration. Fixed character strings, declared as CHAR(n) without the VARYING attribute, occupy n contiguous bytes of storage with the first string character stored lowest in memory. Character strings declared with the VARYING attribute are prefixed by the character string length, ranging from 0 to 254. The length of the area reserved for a CHAR(n) VARYING is n+1. Note that in either case, n cannot exceed 254. The string constant

'Walla Walla Wash'

is stored in a CHAR(20) fixed character string as

```
-----  
|W|a|l|l|a| |W|a|l|l|a| |W|a|s|h| | | | |  
-----
```

This same string is stored in a CHAR(20) VARYING data area as

```
-----  
|10|W|a|l|l|a| |W|a|l|l|a| |W|a|s|h|?|?|?|?  
-----
```

where "10" is the (hexadecimal) string length, and "?" represents undefined character positions.

4.1.5. Fixed Decimal Data Representation. Decimal data items are stored in packed BCD form, using nine's complement data representation. The least significant BCD pair is stored first in memory, with one BCD digit position reserved for the sign. Positive numbers have a 0 sign, while negative numbers have a 9 in the high order sign digit position. The number of bytes occupied by a decimal number depends upon its declared precision. Given a decimal number with precision p, the number of bytes reserved is the integer part of

$$(p + 2) / 2$$

where p varies between 1 and 15, resulting in a minimum of 1 byte and a maximum of 8 bytes to hold a decimal data item. Given a decimal number field of precision 5, the numbers 12345 and -2 are represented as shown below

```
----- -----  
|45|23|01| |98|99|99|  
----- -----
```

4.1.6. Floating Point Binary Representation. Floating Point Binary numbers are stored in four consecutive byte locations, no matter what the declared precision. The number is stored with a 24 bit mantissa, which appears first in memory, followed by an 8-bit exponent. Following data storage conventions, the least significant byte of the mantissa is stored first in memory. The floating point number is normalized so that the most significant bit of the mantissa is "1" for non-zero numbers. A zero mantissa is represented by an exponent byte of 00. Since the most significant bit of the mantissa must be "1" for non-zero values, this bit position is replaced by the mantissa sign. The binary exponent byte is biased by 80 (hexadecimal) so that 81 represents an exponent of 1 while 7F represents an exponent of -1. The Floating Point Binary value 1.5 has the representation shown below

00|00|40|81|

Note that in this case, the mantissa takes the bit stream form

0100 0000 0000 0000 0000 0000

which indicates that the mantissa sign is positive. Setting the (assumed) high order bit to "1" produces the mantissa bit stream

1100 0000 0000 0000 0000 0000

Since the exponent 81 has a bias of 80, the binary exponent is 1, resulting in the binary value

1.100 0000 0000 0000 0000 0000

or, equivalently, 1.5 in a decimal base.

4.1.7. File Constant Representation. Each file constant in a PL/I-80 program occupies 32 contiguous bytes, followed by a variable length field of 0 to 14 additional bytes. The fields of a file constant are all implementation dependent and subject to change without notice.

4.2. Layout of Aggregate Storage.

PL/I-80 data items are contiguous in memory with no filler bytes. Bit data is always stored unaligned. Arrays are stored in row-major order, with the first subscript running slowest and the last subscript running fastest. The RMAC COMMON statement is used to share data with PL/I-80 programs which declare data using the EXTERNAL attribute. The following PL/I-80 program is used as an example:

```
declare
  a (10) bit(8) external,
  1 b external,
  2 c bit(8),
  2 d fixed binary(15),
  2 e (0:2,0:1) fixed;
```

The following RMAC COMMON areas share data areas with the program containing the declaration given above.

```

        common /a/
x:    ds      1

        common /b/
c:    ds      1
d:    ds      2
e00: ds      2
e01: ds      2
e10: ds      2
e11: ds      2
e20: ds      2
e21: ds      2

```

where the labels e_{00} , e_{01} , ..., e_{21} correspond to the PL/I-80 subscripted variable locations $e(0,0)$, $e(0,1)$, ..., $e(2,1)$.

4.3. General Parameter Passing Conventions.

Communication between high-level and assembly language routines can be performed using the PL/I-80 general-purpose parameter passing mechanism described below. Specifically, upon entry to a PL/I-80 or assembly language routine, the HL register pair gives the address of a vector of pointer values which, in turn, lead to the actual parameter values. This situation is illustrated in the diagram below, where the address fields are assumed as shown for this example:

H L	Parm Address	Actual Parameters
1000 1000: 2000 2000: parameter #1		
3000 3000: 4000 4000: parameter #2		
4000 4000: 5000 5000: parameter #3		

5000 5000: last parameter		

The number of parameters, and the parameter length and type is determined implicitly by agreement between the calling program and called subroutine.

Consider the following situation, for example. Suppose a PL/I-80 program uses a considerable number of floating point divide operations, where each division is by a power of two. Suppose also that the loop where the divisions occur is speed-critical, and thus an assembly language subroutine will be used to perform the division. The assembly language routine will simply decrement the binary exponent for the floating point number for each power of two in the division, effectively performing the divide operations without the

overhead of unpacking, performing the general division operation, and repacking the result. During the division, however, the assembly language routine could produce underflow. Thus, the assembly language routine will have to signal the UNDERFLOW condition if this occurs.

The programs which perform this function are given on the following pages. The DTEST program, listed first, tests the division operation. The external entry DIV2 is the assembly language subroutine that performs the division, and is defined on line 4 with two parameters: a fixed(7) and a floating point binary value. The test value 100 is stored into "f" on each loop at line 9, and is passed to the DIV2 subroutine on line 10. Each time DIV2 is called, the value of f is changed to $f/(2^{**i})$ and printed using a PUT statement. At the point of call, DIV2 receives a list of two addresses, corresponding to the two parameters i and f, used in the computation.

The assembly language subroutine, called DIV2, is listed next. Upon entry, the value of i is loaded to the accumulator, and the HL pair is set to point to the exponent field of the input floating point number. If the exponent is zero, DIV2 returns immediately since the resulting value is zero. Otherwise, the subroutine loops at the label "dby2" while counting down the exponent as the power of two diminishes to zero. If the exponent reaches zero during this counting process, an UNDERFLOW signal is raised.

The call to "?signal" within DIV2 demonstrates the assembly language set-up for parameters which use the general-purpose interface. The ?signal subroutine is a part of the PL/I-80 subroutine library (PLILIB.IRL). The HL register pair is set to the signal parameter list, denoted by "siglst." The signal parameter list, in turn, is a vector of four addresses which lead to the signal code "sigcode," the signal subcode "sigsub," the file name indicator "sigfil" (not used here), and the auxiliary message "sigaux" which is the last parameter. The auxiliary message is used to provide additional information to the operator when the error takes place. The signal subroutine prints the message until either the string length is exhausted (32, in this case) or a binary 00 is encountered in the string.

The (abbreviated) output from this test program is shown following the assembly language listing. Note that the loop counter i becomes negative when it reaches 128, but the processing within the DIV2 subroutine treats this value as an unsigned magnitude value, thus the underflow occurs when i reaches -123.

4.4. Returning Values from Functions.

As an alternative to returning values through the parameter list, as described in the previous section, subroutines can produce function values which are returned directly in the registers or on the

PL/I-80 V1.0, COMPILEMENT OF: DTEST

L: List Source Program

NO ERROR(S) IN PASS 1

NO ERROR(S) IN PASS 2

PL/I-80 V1.0, COMPILEMENT OF: DTEST

```
1 a 0000 dtest:  
2 a 0006      proc options(main);  
3 c 0006      dcl  
4 c 0006          div2 entry(fixed(7),float),  
5 c 0006          i fixed(7),  
6 c 0006          f float;  
7 c 0006  
8 c 0006          do i = 0 by 1;  
9 c 000A          f = 100;  
10 c 0015         call div2(i,f);  
11 c 001B         put skip list('100 / 2 **',i,'=',f);  
12 c 0063         end;  
13 a 0063         end dtest;
```

CODE SIZE = 0063

DATA AREA = 0018

```

        public  div2
        extrn  ?signal
;
;           entry:
;           ;      p1 -> fixed(7) power of two
;           ;      p2 -> floating point number
;
;           exit:
;           ;      p1 -> (unchanged)
;           ;      p2 -> p2 / (2**p1)
div2:          ;HL = .low(.p1)
        mov     e,m    ;low(.p1)
        inx     h       ;HL = .high(.p1)
        mov     d,m    ;DE = .p1
        inx     h       ;HL = .low(p2)
        ldax   d       ;a = p1 (power of two)
        mov     e,m    ;low(.p2)
        inx     h       ;HL = .high(.p2)
        mov     d,m    ;DE = .p2
        xchg
        ;HL = .p2
;
;           A = power of 2, HL = .low byte of fp num
        inx     h       ;to middle of mantissa
        inx     h       ;to high byte of mantissa
        inx     h       ;to exponent byte
        inr     m
        dcr     m       ;p2 already zero?
        rz
        ;return if so
dby2:          ;divide by two
        ora     a       ;counted power of 2 to zero?
        rz
        ;return if so
        dcr     a       ;count power of two down
        dcr     m       ;count exponent down
        jnz    dby2    ;loop again if no underflow
;
;underflow occurred, signal underflow condition
        lxi    h,siglst;signal parameter list
        call   ?signal ;signal underflow
        ret
        ;normally, no return
;
        dseg
;
0000 0800      siglst: dw      sigcod ;address of signal code
0002 0900      dw      sigsub ;address of subcode
0004 0A00      dw      sigfil ;address of file code
0006 0C00      dw      sigaux ;address of aux message
;
;      end of parameter vector, start of params
0008 03      sigcod: db      3       ;03 = underflow
0009 80      sigsub: db      128    ;arbitrary subcode for id
000A 0000      sigfil: dw      0000    ;no associated file name
000C 0E00      sigaux: dw      undmsg ;0000 if no aux message
000E 20556E6465undmsg: db      32,'Underflow in Divide by Two',0
002A          end

```

A>b:dtest

100 / 2 **	0 =	1.000000E+02
100 / 2 **	1 =	5.000000E+01
100 / 2 **	2 =	2.500000E+01
100 / 2 **	3 =	1.250000E+01
100 / 2 **	4 =	0.625000E+01
100 / 2 **	5 =	3.125000E+00
100 / 2 **	6 =	1.562500E+00
100 / 2 **	7 =	0.781250E+00
100 / 2 **	8 =	3.906250E-01
100 / 2 **	9 =	1.953125E-01
100 / 2 **	10 =	0.976562E-01
100 / 2 **	11 =	4.882812E-02
100 / 2 **	12 =	2.441406E-02
100 / 2 **	13 =	1.220703E-02
100 / 2 **	14 =	0.610351E-02
100 / 2 **	15 =	3.051757E-03
100 / 2 **	16 =	1.525878E-03
100 / 2 **	17 =	0.762939E-03
100 / 2 **	18 =	3.814697E-04
100 / 2 **	19 =	1.907348E-04
100 / 2 **	20 =	0.953674E-04
100 / 2 **	21 =	4.768371E-05
100 / 2 **	22 =	2.384185E-05
100 / 2 **	23 =	1.192E-05
100 / 2 **	24 =	0.540743E-31
100 / 2 **	25 =	0.770372E-31
100 / 2 **	26 =	3.851859E-32
100 / 2 **	112 =	1.925929E-32
100 / 2 **	113 =	0.962964E-32
100 / 2 **	114 =	4.814824E-33
100 / 2 **	115 =	2.407412E-33
100 / 2 **	116 =	1.203706E-33
100 / 2 **	117 =	0.601853E-33
100 / 2 **	118 =	3.009265E-34
100 / 2 **	119 =	1.504632E-34
100 / 2 **	120 =	0.752316E-34
100 / 2 **	121 =	3.761581E-35
100 / 2 **	122 =	1.880790E-35
100 / 2 **	123 =	0.940395E-35
100 / 2 **	124 =	4.701977E-36
100 / 2 **	125 =	2.350988E-36
100 / 2 **	126 =	1.175494E-36
100 / 2 **	127 =	0.587747E-36
100 / 2 **	-128 =	2.938735E-37
100 / 2 **	-127 =	1.469367E-37
100 / 2 **	-126 =	0.734683E-37
100 / 2 **	-125 =	3.673419E-38
100 / 2 **	-124 =	1.836709E-38
100 / 2 **	-123 =	0.918354E-38
100 / 2 **	-122 =	4.591774E-39

UNDERFLOW (128), Underflow in Divide by Two

Traceback: 017F 011B

End of Execution

stack. This section shows the general-purpose conventions for returning data as functional values.

4.4.1. Returning Pointer, Entry, and Label Variables. Variables which provide access to memory addresses occupy a word value, as described in the previous section. In the case of Pointer, Entry, and Label Variables, the values are returned in the HL register pair. If a label variable is returned which can be the target of a GO TO operation, it is the responsibility of the subroutine containing the label to restore the stack to the proper level when control reaches the label.

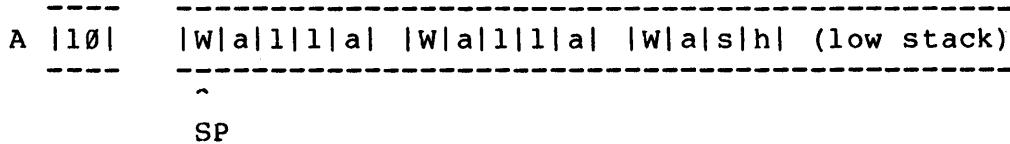
4.4.2. Returning Fixed Binary Data. Functions which return Fixed Binary data items do so by leaving the result in the A register, or HL register pair, depending upon the precision of the data item. Fixed Binary data with precision 1-7 are returned in A, while precision 8-15 items are returned in HL. It is always safe to return the value in HL, with the low order byte copied to the A register, so that register A is equal to register L upon return.

4.4.3. Returning Bit String Data. Similar to Fixed Binary data items, Bit String data is returned in the A register, or the HL register pair, depending upon the precision of the data item. Bit Strings of length 1-8 are returned in A, while precision 9-16 items are returned in the HL pair. Note that Bit Strings are left justified in their fields, so the BIT(1) value "true" is returned in the A register as 80 (hexadecimal). Again, it is safe to return a bit value in the HL register pair, with a copy of the high order byte in A, so that register A is equal to register H upon return.

4.4.4. Returning Character Data. Character data items are returned on the stack, with the length of the string in register A, regardless of whether the function has the VARYING attribute. The string

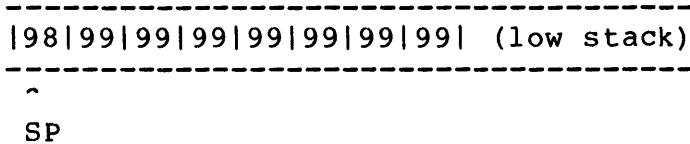
'Walla Walla Wash'

for example, is returned as shown in the diagram below:

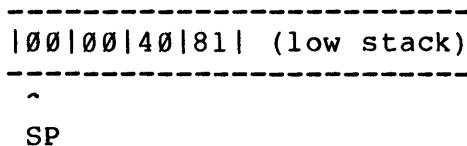


where register A contains the string length 10 (hexadecimal), and the Stack Pointer (SP) addresses the first character in the string.

4.4.5. Returning Fixed Decimal Data. Fixed Decimal data is always returned as a sixteen decimal digit value (8 contiguous bytes) in the stack. The low order decimal pair is stored lowest in memory (at the "top" of the stack), with the high order digit pair highest in memory. The number is represented in nine's complement form, and sign-extended through the high order digit position, with a positive sign denoted by 0, and a negative sign denoted by 9. The decimal number -2, for example, is returned as shown below:



4.4.6. Returning Floating Point Numbers. Floating Point numbers are returned as a four-byte sequence at the top of the stack, regardless of the declared precision. The low order byte of the mantissa is at the top of the stack, followed by the middle byte, then the high byte. The fourth byte is the exponent of the number. The value 1.5 is returned as shown in the following diagram:



The sequence

```

POP D
POP B

```

loads the Floating Point value from the stack for manipulation, leaving the exponent in B, and the 24-bit mantissa in C, D, and E. The result can be placed back into the stack using

```
PUSH B  
PUSH D
```

An example of returning a functional value is shown in the two program listings which follow. The first program, called FDTEST, is similar to the previous floating point divide test, but instead includes an entry definition for FDIV2 which is an assembly language subroutine that returns the result in the stack. The FDIV2 subroutine is then listed, which resembles the previous DIV2 program with some minor changes. First note that the input floating point value is loaded into the BCDE registers so that a temporary copy can be manipulated which does not affect the input value. The exponent field in register B is decremented by the input count, and returned on the stack before the PCHL is executed.

PL/I-80 V1.0, COMPILE OF: FDTEST

L: List Source Program

NO ERROR(S) IN PASS 1

NO ERROR(S) IN PASS 2

PL/I-80 V1.0, COMPILE OF: FDTEST

```
1 a 0000 dtest:  
2 a 0006      proc options(main);  
3 c 0006      dcl  
4 c 0006      fdiv2 entry(fixed(7),float)  
5 c 0006      returns(float),  
6 c 0006      i fixed(7),  
7 c 0006      f float;  
8 c 0006  
9 c 0006      do i = 0 by 1;  
10 c 000A     put skip list('100 / 2 **',i,'=',  
11 c 0055     fdiv2(i,100));  
12 c 0055     end;  
13 a 0055     end dtest;
```

CODE SIZE = 0055

DATA AREA = 0018

```

        public    fd iv2
        extrn    ?signal
        entry:
;
;          p1 -> fixed(7) power of two
;          p2 -> floating point number
;
;          exit:
;
;          p1 -> (unchanged)
;          p2 -> (unchanged)
;          stack: p2 / (2 ** p1)
;
;          fdiv2:                                ;HL = .low(.p1)
        mov      e,m     ;low(.p1)
        inx      h       ;HL = .high(.p1)
        mov      d,m     ;DE = .p1
        inx      h       ;HL = .low(p2)
        ldxax   d       ;a = p1 (power of two)
        mov      e,m     ;low(.p2)
        inx      h       ;HL = .high(.p2)
        mov      d,m     ;DE = .p2
        xchg
;
;          A = power of 2, HL = .low byte of fp num
        mov      e,m     ;E = low mantissa
        inx      h       ;to middle of mantissa
        mov      d,m     ;D = middle mantissa
        inx      h       ;to high byte of mantissa
        mov      c,m     ;C = high mantissa
        inx      h       ;to exponent byte
        mov      b,m     ;B = exponent
        inr      b       ;B = 00?
        dcr      b       ;becomes 00 if so
        jz       fdret   ;to return from float div
;
;          dby2: ;divide by two
        ora      a       ;counted power of 2 to zero?
        jz       fdret   ;return if so
        dcr      a       ;count power of two down
        dcr      b       ;count exponent down
        jnz      dby2   ;loop again if no underflow
;
;          underflow occurred, signal underflow condition
        lxi      h,siglst;signal parameter list
        call    ?signal ;signal underflow
        lxi      b,0     ;clear to zero
        lxi      d,0     ;for default return
;
;          fdret: pop h      ;recall return address
        push    b      ;save high order fp num
        push    d      ;save low order fp num
        pchl
;
;          dseg
        siglst: dw      sigcod  ;address of signal code
        dw      sigsub  ;address of subcode
        dw      sigfil  ;address of file code
        dw      sigaux  ;address of aux message
;
;          end of parameter vector, start of params
        sigcod: db      3       ;03 = underflow
        sigsub: db      128    ;arbitrary subcode for id
        sigfil: dw      0000   ;no associated file name
        sigaux: dw      undmsg ;0000 if no aux message
        000E 20556E6465undmsg: db      32,'Underflow in Divide by Two',0
        002A      end

```

5. PL/I-80 RUNTIME SUBROUTINES.

The PL/I-80 Runtime Subroutine Library (PLILIB.IRL) is discussed in this section, along with the optional subroutines for direct CP/M Input Output. The information given here is useful when PL/I-80 is used as a "systems language," rather than an application language, since direct access to implementation dependent CP/M functions is allowed. Note that the use of these features makes your program very machine and operating system dependent.

5.1. Stack and Dynamic Storage Subroutines.

A number of implementation-dependent functions are included in the PL/I-80 Runtime Library which provide access to stack and dynamic storage structures. The functions are discussed below, with sample programs which illustrate their use. The stack is placed above the code and data area, and below the dynamic storage area. The default value of the stack size is 512 bytes, but can be changed using the STACK(n) option in the OPTIONS portion of the main program procedure heading. In general, the PL/I-80 dynamic storage mechanism maintains a list of all unallocated storage. Upon each request for storage, a search is made to find the first memory segment which satisfies the request size. If no storage is found, the ERROR(7) condition is signalled (Free Space Exhausted). Otherwise, the requested segment is taken from the free area, and the remaining portion goes back to the free space list. In version 1.0 of PL/I-80, storage is dynamically allocated only upon entry to RECURSIVE procedures, upon explicit or implicit OPENS for files which access the disk, or upon executing an ALLOCATE statement. In any case, an even number of bytes, or whole words, is always allocated, no matter what the request size.

5.1.1. The TOTWDS and MAXWDS Functions. It is often useful to find the amount of storage available at any given point in the execution of a particular program. The TOTWDS (Total Words) and MAXWDS (Max Words) functions can be used to obtain this information. The functions must be declared in the calling program as

```
dcl totwds returns(fixed(15));  
dcl maxwds returns(fixed(15));
```

When invoked, the TOTWDS subroutine scans the free storage list and returns the total number of words (double bytes) available in the free list. The MAXWDS subroutine performs a similar function, but returns the size of the largest segment in the free list, again in words. A subsequent ALLOCATE statement which specifies a segment size not

exceeding MAXWDS will not cause the ERROR(7) signal to be raised, since at least that much storage is available. Note that since both TOTWDS and MAXWDS count in word units, the values can be held by FIXED BINARY(15) counters. If, during the scan of free memory, invalid link words are encountered (usually due to a out-of-bounds subscript or pointer store operation), both TOTWDS and MAXWDS return the value -1. Otherwise, the returned value will be a non-negative integer value.

5.1.2. The ALLWDS Subroutine. The PL/I-80 Runtime Library contains a subroutine, called ALLWDS, which is useful in controlling the dynamic allocation size. The subroutine must be declared in the calling program as

```
dcl allwds entry(fixed(15)) returns(ptr);
```

The ALLWDS subroutine allocates a segment of memory of the size given by the input parameter, in words (double bytes). If no segment is available, the ERROR(7) condition is raised. Further, the input value must be a non-negative integer value. The ALLWDS function returns a pointer to the allocated segment.

An example of the use of TOTWDS, MAXWDS, and ALLWDS functions is given in the ALLTST program on the next page. A sample program interaction is given following the program listing.

5.1.3. The STKSIZ Function. The function STKSIZ (Stack Size) returns the current stack size in bytes whenever it is called. This function is particularly useful for checking possible stack overflow conditions, or in determining the maximum stack depth during program testing. The STKSIZ function is declared in the calling program as

```
dcl stksiz returns(fixed(15));
```

A Sample use of the STKSIZ function appears in the listing of the recursive Ackermann test. In this case, it is used to check the maximum stack depth during the recursive function processing. An interaction with this program is given following the program listing.

PL/I-80 V1.0, COMPILATION OF: ALLTST

L: List Source Program

NO ERROR(S) IN PASS 1

NO ERROR(S) IN PASS 2

PL/I-80 V1.0, COMPILATION OF: ALLTST

```
1 a 0000 alltst:  
2 a 0006      proc options(main);  
3 a 0006      /* assembly language interface to  
4 a 0006      dynamic storage allocation module */  
5 c 0006      dcl  
6 c 0006          totwds returns(fixed(15)),  
7 c 0006          maxwds returns(fixed(15)),  
8 c 0006          allwds entry(fixed(15)) returns(ptr);  
9 c 0006  
10 c 0006      dcl  
11 c 0006          allreq fixed(15),  
12 c 0006          memptr ptr,  
13 c 0006          meminx fixed(15),  
14 c 0006          memory (0:0) bit(16) based(memptr);  
15 c 0006  
16 c 0006      do while('1'b);  
17 c 0006          put edit (totwds(),' Total Words Available',  
18 c 004F          maxwds(),' Maximum Segment Size',  
19 c 004F          'Allocation Size? ')  
20 c 004F          (2(skip,f(6),a),skip,a);  
21 c 004F          get list(allreq);  
22 c 0067          memptr = allwds(allreq);  
23 c 0070          put edit('Allocated',allreq,  
24 c 00B2          ' Words at ',unspec(memptr))  
25 c 00B2          (skip,a,f(6),a,b4);  
26 c 00B2  
27 c 00B2          /* clear memory as example */  
28 c 00B2          do meminx = 0 to allreq-1;  
29 c 00CC          memory(meminx) = '0000'b4;  
30 c 00E7          end;  
31 c 00E7          end;  
32 a 00E7      end alltst;
```

CODE SIZE = 00E7

DATA AREA = 0078

A>B:ALLTST

25596 Total Words Available
25596 Maximum Segment Size
Allocation Size? 0

Allocated 0 Words at 250A
25594 Total Words Available
25594 Maximum Segment Size
Allocation Size? 100

Allocated 100 Words at 250E
25492 Total Words Available
25492 Maximum Segment Size
Allocation Size? 25000

Allocated 25000 Words at 25DA
490 Total Words Available
490 Maximum Segment Size
Allocation Size? 490

Allocated 490 Words at E92E
0 Total Words Available
0 Maximum Segment Size
Allocation Size? 1

ERROR (7), Free Space Exhausted
Traceback: 016D
End of Execution

PL/I-80 V1.0, COMPIILATION OF: ACKTST

L: List Source Program

NO ERROR(S) IN PASS 1

NO ERROR(S) IN PASS 2

PL/I-80 V1.0, COMPIILATION OF: ACKTST

```
1 a 0000 ack:
2 a 0006      procedure options(main,stack(2000));
3 c 0006      dcl
4 c 0006      (m,n) fixed,
5 c 0006      (maxm,maxn) fixed,
6 c 0006      ncalls decimal(6),
7 c 0006      (curstack, stacksize) fixed,
8 c 0006      stksiz entry returns(fixed);
9 c 0006
10 c 0006     put skip list('Type max m,n: ');
11 c 0022     get list(maxm,maxn);
12 c 0046     do m = 0 to maxm;
13 c 005F     do n = 0 to maxn;
14 c 0078     ncalls = 0;
15 c 0088     curstack = 0;
16 c 008E     stacksize = 0;
17 c 0091     put edit
18 c 012F     ('Ack(',m,',',n,')=',ackermann(m,n),
19 c 012F     ncalls,' Calls,',stacksize,' Stack Bytes')
20 c 012F     (skip,a,2(f(2),a),f(6),f(7),a,f(4),a);
21 c 012F     end;
22 c 012F     end;
23 c 012F     stop;
24 c 0132
25 c 0132     ackermann:
26 c 0132     procedure(m,n) returns(fixed) recursive;
27 e 0132     dcl
28 e 015C     (m,n) fixed;
29 e 015C     ncalls = ncalls + 1;
30 e 0177     curstack = stksiz();
31 e 017D     if curstack > stacksize then
32 e 018A     stacksize = curstack;
33 e 0190     if m = 0 then
34 e 0199     return(n+1);
35 e 01A1     if n = 0 then
36 e 01AA     return(ackermann(m-1,1));
37 e 01BB     return(ackermann(m-1,ackermann(m,n-1)));
38 c 01DC     end ackermann;
39 a 01DC     end ack;
```

CODE SIZE = 01DC
DATA AREA = 0082

A>B:ACKTST

Type max m,n: 6,6

Ack(0, 0)=	1	1 Calls,	4 Stack Bytes
Ack(0, 1)=	2	1 Calls,	4 Stack Bytes
Ack(0, 2)=	3	1 Calls,	4 Stack Bytes
Ack(0, 3)=	4	1 Calls,	4 Stack Bytes
Ack(0, 4)=	5	1 Calls,	4 Stack Bytes
Ack(0, 5)=	6	1 Calls,	4 Stack Bytes
Ack(0, 6)=	7	1 Calls,	4 Stack Bytes
Ack(1, 0)=	2	2 Calls,	6 Stack Bytes
Ack(1, 1)=	3	4 Calls,	8 Stack Bytes
Ack(1, 2)=	4	6 Calls,	10 Stack Bytes
Ack(1, 3)=	5	8 Calls,	12 Stack Bytes
Ack(1, 4)=	6	10 Calls,	14 Stack Bytes
Ack(1, 5)=	7	12 Calls,	16 Stack Bytes
Ack(1, 6)=	8	14 Calls,	18 Stack Bytes
Ack(2, 0)=	3	5 Calls,	10 Stack Bytes
Ack(2, 1)=	5	14 Calls,	14 Stack Bytes
Ack(2, 2)=	7	27 Calls,	18 Stack Bytes
Ack(2, 3)=	9	44 Calls,	22 Stack Bytes
Ack(2, 4)=	11	65 Calls,	26 Stack Bytes
Ack(2, 5)=	13	90 Calls,	30 Stack Bytes
Ack(2, 6)=	15	119 Calls,	34 Stack Bytes
Ack(3, 0)=	5	15 Calls,	16 Stack Bytes
Ack(3, 1)=	13	106 Calls,	32 Stack Bytes
Ack(3, 2)=	29	541 Calls,	64 Stack Bytes
Ack(3, 3)=	61	2432 Calls,	128 Stack Bytes
Ack(3, 4)=	125	10307 Calls,	256 Stack Bytes
Ack(3, 5)=			

5.2. PL/I-80 Runtime Subroutine Entry Points.

The standard PL/I-80 Runtime Library entry points are listed below. The entry point name is shown to the left, followed by the input value registers and the result registers. A short explanation is given on the right. Note that this list does not include the environmental or I/O operators since these entry points may vary from version to version. Further, the definitions shown below are for general information purposes only, and are subject to change without notice. The register names are given in capital letters, M(r) denotes memory addressed by the register pair r, and ST represents a stacked value.

name	parameters	result	comment or definition
im22n	DE	HL	HL word*word integer multiply
id22n	DE	HL	HL word/word integer divide
is22n	DE	HL	HL word-word integer subtract
in20n	HL		-word
fl40m	HL	ST	fp load from M(HL) to stack
fx44s	ST	HL	M(HL) fp xfer from stack to M(HL)
fx44m	DE	HL	M(HL) fp xfer from M(HL) to M(DE)
fa44s	ST	ST	fp add stack+stack to stack
fa44m	DE	HL	ST fp add M(DE)+M(HL) to stack
fa44l	ST	HL	ST fp add stack+M(HL) to stack
fa44r	HL	ST	ST fp add M(HL)+stack to stack
fs44s	ST	ST	fp sub stack-stack to stack
fs44m	DE	HL	ST fp sub M(DE)-M(HL) to stack
fs44l	ST	HL	ST fp sub stack-M(HL) to stack
fs44r	HL	ST	ST fp sub M(HL)-stack to stack
fm44s	ST	ST	fp mul stack*stack to stack
fm44m	DE	HL	ST fp mul M(DE)*M(HL) to stack
fm44l	ST	HL	ST fp mul stack*M(HL) to stack
fm44r	HL	ST	ST fp mul M(HL)*stack to stack
fd44s	ST	ST	fp div stack/Stack to stack
fd44m	DE	HL	ST fp div M(DE)/M(HL) to stack
fd44l	ST	HL	ST fp div stack/M(HL) to stack
fd44r	HL	ST	ST fp div M(HL)/Stack to stack
fc44s	ST	ST	fp comp stack:stack to stack
fc44m	DE	HL	ST fp comp M(DE):M(HL) to stack
fc44l	ST	HL	ST fp comp stack:M(HL) to stack
fc44r	HL	ST	ST fp comp M(HL):stack to stack
fn40s	ST	ST	fp negate stack
fn40m	HL	ST	fp load from M(HL) and negate
fe40s	ST	A	float p extract sign from stack
fe40m	HL	A	float p extract sign from memory 1 => positive sign (non zero set) 0 => zero result (zero flag set) -1 => negative sign (minus set)
fmodf	ST	ST	floating point mod(x,y)
fabsf	ST	ST	floating point abs(x)
fmaxf	ST	ST	floating point max(x,y)
fminf	ST	ST	floating point min(x,y)

froun	ST	A	ST	floating point round(x,k)
ftrnc	ST		ST	floating point trunc(x)
fflor	ST		ST	floating point floor(x)
fceil	ST		ST	floating point ceil(x)
fexop	ST	A	ST	fp ** k (k pos constant)
ffxop	ST	ST	ST	x ** y (exp(y*log(x)))
bcl2n	D	HL	HL	8/16 bit concatenate, where B=length of d, C=mask
bc22n	DE	HL	HL	16/16 bit concatenate, where B=length of d, C=mask
bsl16	B	HL	HL	bit shift left 16, size in b
bsl08	A	B	A	bit shift left 8, size in b
bst08	A B C	HL	M(HL)	bit substring store bit(8) in A to bit(8) in memory at HL, B = index, C = length
bstl16	B C DE	HL	M(HL)	bit substring store bit(16) in DE to bit(16) in memory at HL
bix08	A B D	H	A/HL	bit index, A=source, B=search D=len(source), E=len(search)
bix16	B C DE	HL	A/HL	bit index, B=len(source), C=len(search), DE=source, HL=search
boolf	B	DE	HL	bool(x,y,b), B = 4-bit mask x,y operands in DE and HL
iel2n	A		HL	sign extend A to HL
iel0n	A		A	integer extract sign (8-bit)
ie20n	HL		A	integer extract sign (16-bit)
imdop	DE	HL	HL	integer mod(x,y)
iab07	A		A	integer 7 abs(i)
iab15	HL		HL	integer 15 abs(i)
imaxf	DE	HL	HL	integer max(x,y)
iminf	DE	HL	HL	integer min(x,y)
iroun	HL	A	HL	integer round(i,k)
ixerop	HL	A	HL	integer ** k (k pos constant)
slvts	HL		A	string load varying to stack A=length of string on return
slcts	A	HL		string load char to stack A=length of char string
ssvfs	A	B	HL	string store varying from stack A=current len, B=max length
sscfs	A	B	HL	string store char from stack
smvvm	A	DE	HL	string move vary to vary in memory A=max target len, DE=source, HL=target
smvcvm	A	DE	HL	string move vary to char in memory A=target length
smcvvm	A	B	DE	string move char to vary in memory A=max target len, B=source len
smccm	A	B	DE	A=target len, B=source len
sjsts	A	ST	ST'	string juxtapose (catenate) stack A=length of left, ST=chars of left ST' = pushed psw with length of right followed by chars of right
sjscm	A	B	HL	string juxtapose stack with char memory A=stacked len, B=char len, HL=.char

sjsvm	A		HL	string juxtapose stack with vary memory
savvm	A	B	HL	string append vary to vary in memory A=char len, B=max target length
sasvm	A	B	HL	string append stack to vary in memory A=stacked length, B=max target length
sacvm	A	B	HL	string append char to vary in memory A=char len, B=max target length
scccm	A	B	DE	string compare char to char in memory A=len right, B=len left, DE = .char left, HL = .char right
sccvm		B	DE	String compare char to vary in memory B=len left, DE=.char, HL=.vary
scvcm	A		DE	string compare vary to char in memory A=len right char, DE=.vary, HL=.char
scvvm			DE	String compare vary to vary in memory DE=.vary left, HL=.vary right
scscm	A	B		string compare stack to char in memory A=len stk, B=len char, HL=.char
scsvm	A			string compare stack to vary in memory A=len stk, HL=.vary
sccms	A	B		string compare char in mem to stack A=len stk, B=len char, HL=.char
scvms	A			string compare vary in mem to stack A=len stk, HL=.vary
scsts	A			string compare stack to stack A=len right element on stack, ST is stack right string, next is pushed psw with len left string, followed by left string, result: sign value & cond if l < r, zero value & cond if l = r, pos value & cond if l >= r, nzer value & cond if l > r.
cs2ad	A	E	HL	char substr(ex,ei) address A=length, E=ei, HL=ex
cs3ad	A	C	E	A=result length on return char substr(ex,ei,el) address C=el
vs2ad			E	A=result length on return vary substr(ex,ei) address E=ei, HL=ex
vs3ad		C	E	A=result length on return vary substr(ex,ei,el) address C=el
cxccm	A	B	DE	A=result length on return str index char to char in memory A=len right, B=len left, DE = .char left, HL = .char right
cxcvm		B	DE	str index char to vary in memory B=len left, DE=.char, HL=.vary
cxvcm	A		DE	str index vary to char in memory A=len right char, DE=.vary, HL=.char
cxvvm			DE	str index vary to vary in memory DE=.vary left, HL=.vary right
cxscm	A	B		str index stack to char in memory

				A=len stk, B=len char, HL=.char
cxsvm	A		A/HL	str index stack to vary in memory
				A=len stk, HL=.vary
cxcms	A B		A/HL	str index char in mem to stack
				A=len stk, B=len char, HL=.char
cxvms	A		A/HL	str index vary in mem to stack
				A=len stk, HL=.vary
cxsts	A			str index stack to stack
				A=len right element on stack, ST is stack right string, next is pushed psw with len left string, followed by left string, result:
				A/HL = Ø if right not found in left, otherwise index returned
verop	A ST ST		A/A/HL	verify(s,c), A=len(c), st has chars(c),len(s),chars(s)
colop			A/ST	collate(), A=128, stack has
xl2op	A ST ST		A/ST	translate(s,t), A=len(t), stack has chars(t), s
xl3op	A ST ST ST		A/ST	translate(s,t,x) A=len(x), stack has chars(x), t, s Ø,1, ..., 127 (ascii chars)
dldop	A	HL	ST	decimal load to stack, A = prec
dasop	A	ST	HL	decimal assign, stack to memory
dadop	ST	ST	ST	decimal add to stack
dzuop	ST	ST	ST	decimal subtract to stack
dngop	ST		ST	decimal negate to stack
dcmop	ST		A	decimal compare operator
dexop	ST	ST	ST	decimal exponentiate to stack
dmuop	ST	ST	ST	decimal multiply to stack
ddvop	ST	ST	ST	decimal divide to stack
dsiop	ST		A	decimal sign extract
dmodf	ST	ST	ST	decimal mod(x,y)
dabsf	ST		ST	decimal abs(x)
dmaxf	ST	ST	ST	decimal max(x,y)
dminf	ST	ST	ST	decimal min(x,y)
droun	ST	A	ST	decimal round(x,k)
dtrnc	ST		ST	decimal trunc(x)
dflor	ST		ST	decimal floor(x)
dceil	ST		ST	decimal ceil(x)
dexop	ST	A	ST	decimal ** k (k pos constant)
qcdop	A B	ST	ST	convert character to decimal A = string length, B = scale ST = character string, returns ST = decimal number
qddsl	A	ST	ST	decimal/decimal left shift A = shift count
qddsr	A	ST	ST	decimal/decimal right shift A = shift count
qicop	A		HL	convert integer to char in stack A=string size, HL=integer value
qvcop	A/ST		A/ST	convert varying to char
qi07d	A		ST	convert fix(7) to decimal
qi15d	HL		ST	convert fix(15) to decimal
qi07f	A		ST	convert fix(7) to float

qi15f	HL	ST	convert fix(15) to float
qfi07	ST	A	convert float to fix(7)
qfil5	ST	HL	convert float to fix(15)
qfcss	A	ST	convert float-char stack to stack A=target length, ST=fp number
qfcms	A	M(HL)	convert float-char memory to stack
qb08c	A	B	convert bit(8) in a, to string in stack, with precision b
qb16c	HL	B	convert bit(16) in HL to string
qb08i	A	B	convert bit(8) in A to fixed with precision B in HL
qb16i	HL	B	convert bit(16) to fixed
qi07b	A	B	convert fix(<8) to bit(8) fixed precision in b
qi15b	HL	B	convert fix(<16) to bit(16)
mdi07	ST	A	convert dec in stack to fix(7)
qdil5	ST	HL	convert dec in stack to fix(15)
qciop	A/ST	HL	convert char in stack to integer
qclop	A/ST	ST	convert char in stack to float
qccop	A B	ST	convert char to char on stack A=len(s), B=converted length return A=b, ST trunc or extend
nstop	BC DE HL	M(HL)	non-computational store, move M(DE) to M(HL) for BC bytes
nc22n	DE	HL	double byte non-computational compare: zero flag set if DE = HL, non-zero otherwise
ncomp	BC DE HL	M(HL)	non-computational compare, M(DE) - M(HL), set flags

5.3. Direct CP/M Function Calls.

Access to all CP/M version 1 and 2 functions, and equivalent MP/M calls, is accomplished through the optional subroutines included in PLIDIO.ASM, given in the listing of Appendix A, and included in source form on the PL/I-80 diskette.

The PLIDIO.ASM subroutines are not included as a part of the standard PLILIB.IRL file because specific applications may require various changes to the direct CP/M functions which either remove operations to decrease space, or alter the manner in which the interface to a specific function takes place. Note that if the interface to a function is changed, it is imperative that the name of the entry point is also changed to avoid confusion when the program is read by another programmer.

The relocatable file, PLIDIO.REL, is created by assembling the source program using RMAC:

```
rmac plidio $pz+
```

(the \$pz+s option avoids production of the listing and symbol files). Given that a PL/I-80 program, such as DIOCOPY.PLI, is present on the disk, the DIOCOPY.REL file is produced by typing:

```
pli diocopy
```

(a listing of the DIOCOPY program is given in Appendix C). These two programs are then linked with the PLILIB.IRL file by typing:

```
link diocopy,plidio
```

resulting in the file DIOCOPY.COM which is a program that directly executes under CP/M.

The file DIOMOD.DCL is a source file containing the standard PLIDIO entry point declarations so that they can be conveniently copied into the source program during compilation using the "include" statement

```
%include 'x:diomod.dcl';
```

where the optional "x:" drive prefix indicates the drive name (A: through P:) containing the DIOMOD.DCL file. The drive prefix need not be present if the DIOMOD.DCL file is on the same drive as the PLI source file. The contents of the DIOMOD.DCL file is shown below, and in the listing of Appendix C.

```
dcl
    memptr entry           returns (ptr),
    memsiz entry           returns (fixed(15)),
    memwds entry           returns (fixed(15)),
    dfcb0 entry             returns (ptr),
    dfcbl entry             returns (ptr),
    dbuff entry             returns (ptr),
    reboot entry,
    rdcon entry             returns (char(1)),
    wrcon entry             returns (char(1)),
    rdrdr entry             returns (char(1)),
    wrpun entry             returns (char(1)),
    wrlst entry             returns (char(1)),
    coninp entry             returns (char(1)),
    conout entry             returns (char(1)),
    rdstat entry             returns (bit(1)),
    getio entry              returns (bit(8)),
    setio entry              returns (bit(8)),
    wrstr entry              (ptr),
    rdbuf entry              (ptr),
    break entry              returns (bit(1)),
    vers entry               returns (bit(16)),
    reset entry,
    select entry             (fixed(7)),
    open entry      (ptr)   returns (fixed(7)),
    close entry      (ptr)   returns (fixed(7)),
    sear entry       (ptr)   returns (fixed(7)),
```

```

searn entry      returns (fixed(7)),
delete entry    (ptr),
rdseq entry     (ptr) returns (fixed(7)),
wrseq entry     (ptr) returns (fixed(7)),
make entry      (ptr) returns (fixed(7)),
rename entry    (ptr),
logvec entry    returns (bit(16)),
curdsk entry    returns (fixed(7)),
setdma entry    (ptr),
allvec entry    returns (ptr),
wpdisk entry,
rovec entry     returns (bit(16)),
filatt entry    (ptr),
getdpb entry    returns (ptr),
getusr entry    returns (fixed(7)),
setusr entry    (fixed(7)),
rdran entry     (ptr) returns (fixed(7)),
wrran entry     (ptr) returns (fixed(7)),
filsiz entry    (ptr),
setrec entry    (ptr),
resdrv entry    (bit(16)),
wrranz entry    (ptr) returns (fixed(7));

```

Three programs are included which illustrate the use of the PLIDIO calls. Appendix B lists the DIOCALLS program that gives examples of all the basic functions, while Appendix C shows how the fundamental disk I/O operations take place, in a program called DIOCOPY which performs a fast file-to-file copy function. The last program, given in Appendix D, illustrates the operation of the random access primitives. These programs are designed to demonstrate all of the PLIDIO entry points, and show various additional PL/I-80 programming facilities in the process.

The file FCB.DCL is used throughout DIOCOPY and DIORAND to define the body of each File Control Block declaration. This file is copied into the source program during compilation using the statement:

```
%include 'x:fcb.dcl';
```

where, again, "x:" denotes the optional drive prefix for the drive containing the FCB.DCL file.

Note that the use of these entry points generally precludes the use of some PL/I-80 facilities. In particular, the dynamic storage area is used by the PL/I-80 system for recursive procedures and file I/O buffering. (Be aware that there are no guarantees that the dynamic storage area will not be used for other purposes as additional facilities are added to PL/I-80.) Thus, the use of the MEMPTR function as shown in Appendix B disallows the use of dynamic storage allocation functions. Further, you must ensure that the various file maintenance functions, such as delete and rename do not access a file which is currently open in the PL/I-80 file system. Simple peripheral access, as shown in these examples, is generally safe since no buffering takes place in this case.

APPENDIX A:

**LISTING OF "PLIDIO"
DIRECT CP/M CALL ENTRY POINTS**

```

        name      'DIOMOD'
        title    'Direct CP/M Calls From PL/I-80'
;
;*****                                     *
;*                                         *
;*      cp/m calls from pl/i for direct i/o   *
;*                                         *
;*****                                     *
public  memptr ;return pointer to base of free mem
public  memsiz ;return size of memory in bytes
public  memwds ;return size of memory in words
public  dfcb0 ;return address of default fcb 0
public  dfcbl ;return address of default fcb 1
public  dbuff ;return address of default buffer
public  reboot ;system reboot (#0)
public  rdcon ;read console character (#1)
public  wrcon ;write console character (#2)
public  rdrdr ;read reader character (#3)
public  wrpun ;write punch character (#4)
public  wrlst ;write list character (#5)
public  coninp ;direct console input (#6a)
public  conout ;direct console output (#6b)
public  rdstat ;read console status (#6c)
public  getio  ;get io byte (#8)
public  setio  ;set i/o byte (#9)
public  wrstr  ;write string (#10)
public  rdbuf   ;read console buffer (#10)
public  break   ;get console status (#11)
public  vers    ;get version number (#12)
public  reset   ;reset disk system (#13)
public  select  ;select disk (#14)
public  open    ;open file (#15)
public  close   ;close file (#16)
public  sear    ;search for file (#17)
public  searn   ;search for next (#18)
public  delete  ;delete file (#19)
public  rdseq   ;read file sequential mode (#20)
public  wrseq   ;write file sequential mode (#21)
public  make    ;create file (#22)
public  rename  ;rename file (#23)
public  logvec  ;return login vector (#24)
public  curdsk  ;return current disk number (#25)
public  setdma  ;set DMA address (#26)
public  allvec  ;return address of alloc vector (#27)
public  wpdisk  ;write protect disk (#28)
public  rovec   ;return read/only vector (#29)
public  filatt  ;set file attributes (#30)
public  getdpb  ;get base of disk parm block (#31)
public  getusr  ;get user code (#32a)
public  setusr  ;set user code (#32b)
public  rdran  ;read random (#33)
public  wrran  ;write random (#34)
public  filsiz  ;random file size (#35)
public  setrec  ;set random record pos (#36)
public  resdrv  ;reset drive (#37)
public  wrranz  ;write random, zero fill (#40)

```

```

;
;
extrn ?begin ;beginning of free list
extrn ?boot ;system reboot entry point
extrn ?bdos ;bdos entry point
extrn ?dfcbo ;default fcb 0
extrn ?dfcbl ;default fcb 1
extrn ?dbuf ;default buffer
;
;*****equates for interface to cp/m bdos
;
;
000D = cr equ 0dh ;carriage return
000A = lf equ 0ah ;line feed
001A = eof equ lah ;end of file
;
0001 = readc equ 1 ;read character from console
0002 = wrtc equ 2 ;write console character
0003 = rdrf equ 3 ;reader input
0004 = punf equ 4 ;punch output
0005 = listf equ 5 ;list output function
0006 = diof equ 6 ;direct i/o, version 2.0
0007 = getiof equ 7 ;get i/o byte
0008 = setiof equ 8 ;set i/o byte
0009 = printf equ 9 ;print string function
000A = rdconf equ 10 ;read console buffer
000B = statf equ 11 ;return console status
000C = versf equ 12 ;get version number
000D = resetf equ 13 ;system reset
000E = seldf equ 14 ;select disk function
000F = openf equ 15 ;open file function
0010 = closef equ 16 ;close file
0011 = serchf equ 17 ;search for file
0012 = serchn equ 18 ;search next
0013 = deletf equ 19 ;delete file
0014 = readf equ 20 ;read next record
0015 = writf equ 21 ;write next record
0016 = makef equ 22 ;make file
0017 = renamf equ 23 ;rename file
0018 = loginf equ 24 ;get login vector
0019 = cdiskf equ 25 ;get current disk number
001A = setdmf equ 26 ;set dma function
001B = getalf equ 27 ;get allocation base
001C = wrprof equ 28 ;write protect disk
001D = getrof equ 29 ;get r/o vector
001E = setatf equ 30 ;set file attributes
001F = getdpf equ 31 ;get disk parameter block
0020 = userf equ 32 ;set/get user code
0021 = rdranf equ 33 ;read random
0022 = wrranf equ 34 ;write random
0023 = filszf equ 35 ;compute file size
0024 = setrcf equ 36 ;set random record position
0025 = rsdrvf equ 37 ;reset drive function
0028 = wrrnzf equ 40 ;write random zero fill

```

```

;
; utility functions
;***** *****
;
;* general purpose routines used upon entry *
;
;***** *****
;
getpl: ;get single byte parameter to register e
0000 5E      mov     e,m          ;low (addr)
0001 23      inx    h
0002 56      mov     d,m          ;high(addr)
0003 EB      xchg
0004 5E      mov     e,m          ;hl = .char
0005 C9      ret
;
getp2: ;get single word value to DE
getp2i: ;(equivalent to getp2)
0006 CD0000  call    getpl
0009 23      inx    h
000A 56      mov     d,m          ;get high byte as well
000B C9      ret
;
getver: ;get cp/m or mp/m version number
000C E5      push   h          ;save possible data adr
000D 0E0C  mvi    c,versf
000F CD0000  call    ?bdos
0012 E1      pop    h          ;recall data addr
0013 C9      ret
;
chkv20: ;check for version 2.0 or greater
0014 CD0C00  call    getver
0017 FE14  cpi    20
0019 D0      rnc          ;return if > 2.0
;
001A C32300  error message and stop
                jmp    vererr      ;version error
;
chkv22: ;check for version 2.2 or greater
001D CD0C00  call    getver
0020 FE22  cpi    22h
0022 D0      rnc          ;return if >= 2.2
vererr:
                ;version error, report and terminate
0023 112E00  lxi    d,vermsg
0026 0E09  mvi    c,printf
0028 CD0000  call    ?bdos      ;write message
002B C30000  jmp    ?boot      ;and reboot
002E 0D0A4C6174vermsg: db      cr,lf,'Later CP/M or MP/M Version Required$'
;
;***** *****
;
;* memptr: ;return pointer to base of free storage
;
0054 2A0000  lhld   ?begin
0057 C9      ret
;

```

```

;*****  

;*  

;*****  

memsiz: ;return size of free memory in bytes  

0058 2A0100    lhld    ?bdos+1      ;base of bdos  

005B EB        xchg  

005C 2A0000    lhld    ?begin      ;beginning of free storage  

005F 7B        mov     a,e          ;low(.bdos)  

0060 95        sub     l             ;-low(begin)  

0061 6F        mov     l,a          ;back to l  

0062 7A        mov     a,d          ;high(.bdos)  

0063 9C        sbb     h             ;hl = mem size remaining  

0064 67        mov     h,a          ;hl = mem size remaining  

0065 C9        ret  

;  

;*****  

;*  

;*****  

memwds: ;return size of free memory in words  

0066 CD5800    call    memsiz      ;hl = size in bytes  

0069 7C        mov     a,h          ;high(size)  

006A B7        ora     a             ;cy = 0  

006B 1F        rar  

006C 67        mov     h,a          ;back to h  

006D 7D        mov     a,l          ;low(size)  

006E 1F        rar  

006F 6F        mov     l,a          ;include ls bit  

0070 C9        ret              ;back to l  

;with wds in hl  

;  

;*****  

;*  

;*****  

dfcb0: ;return address of default fcb 0  

0071 210000    lxi    h,?dfcb0  

0074 C9        ret  

;  

;*****  

;*  

;*****  

dfcbl: ;return address of default fcb 1  

0075 210000    lxi    h,?dfcbl  

0078 C9        ret  

;  

;*****  

;*  

;*****  

dbuff: ;return address of default buffer  

0079 210000    lxi    h,?dbuff  

007C C9        ret  

;  

;*****  

;*  

;*****  

reboot: ;system reboot (#0)  

007D C30000    jmp    ?boot  

;

```

```

;*****
;*
;*****
rdcon: ;read console character (#1)
        ;return character value to stack
        mvi    c,readc
        jmp    chrin           ;common code to read char
;
;*****
;*
;*****
wrcon: ;write console character(#2)
        ;l->char(1)
        mvi    c,wrutc          ;console write function
        jmp    chrout           ;to write the character
;
;*****
;*
;*****
rdrdr: ;read reader character (#3)
        mvi    c,rdrf            ;reader function
chrin:
        ;common code for character input
        call   ?bdos             ;value returned to A
        pop    h                  ;return address
        push   psw                ;character to stack
        inx    sp                 ;delete flags
        mvi    a,l                ;character length is 1
        pchl
        ;back to calling routine
;
;*****
;*
;*****
wrpun: ;write punch character (#4)
        ;l->char(1)
        mvi    c,punf             ;punch output function
        jmp    chrout           ;common code to write chr
;
;*****
;*
;*****
wrlst: ;write list character (#5)
        ;l->char(1)
        mvi    c,listf            ;list output function
chrout:
        ;common code to write character
        ;l-> character to write
        call   getpl              ;output char to register e
        jmp    ?bdos              ;to write and return
;
;*****
;*
;*****
coninp: ;perform console input, char returned in stack
        lxi    h,chrstr           ;return address
        push   h                  ;to stack for return

```

CP/M RMAC ASSEM 0.4 #006 DIRECT CP/M CALLS FROM PL/I-80

```

00A6 2A0100 lhld ?boot+1 ;base of bios jmp vector
00A9 110600 lxi d,2*3 ;offset to jmp conin
00AC 19 dad d
00AD E9 pchl ;return to chrstr
;
chrstr: ;create character string, length 1
00AE E1 pop h ;recall return address
00AF F5 push psw ;save character
00B0 33 inx sp ;delete psw
00B1 E9 pchl ;return to caller
;
;*****
;*
;*****
conout: ;direct console output
;l->char(1)
00B2 CD0000 call getpl ;get parameter
00B5 4B mov c,e ;character to c
00B6 2A0100 lhld ?boot+1 ;base of bios jmp
00B9 110900 lxi d,3*3 ;console output offset
00BC 19 dad d ;hl = .jmp conout
00BD E9 pchl ;return through handler
;
;*****
;*
;*****
rdstat: ;direct console status read
00BE 21EC00 lxi h,rdsret ;read status return
00C1 E5 push h ;return to rdsret
00C2 2A0100 lhld ?boot+1 ;base of jmp vector
00C5 110300 lxi d,1*3 ;offset to .jmp const
00C8 19 dad d ;hl = .jmp const
00C9 E9 pchl
;
;*****
;*
;*****
getio: ;get io byte (#8)
00CA 0E07 mvi c,getiof
00CC C30000 jmp ?bdos ;value returned to A
;
;*****
;*
;*****
setio: ;set i/o byte (#9)
;l->i/o byte
00CF CD0000 call getpl ;new i/o byte to E
00D2 0E08 mvi c,setiof
00D4 C30000 jmp ?bdos ;return through bdos
;
;*****
;*
;*****
wrstr: ;write string (#10)
;l->addr(string)
00D7 CD0600 call getp2 ;get parameter value to DE

```

```

/M RMAC ASSEM 0.4      #007      DIRECT CP/M CALLS FROM PL/I-80

0DA 0E09                mvi      c,printf      ;print string function
0DC C30000                jmp      ?bdos       ;return through bdos
;
;*****
;*
;rdbuf:   ;read console buffer (#10)
;    l->addr(buff)
;    call    getp2i      ;DE = .buff
;    mvi      c,rdconf      ;read console function
;    jmp      ?bdos       ;return through bdos
;
;*****
;*
;*****
break:  ;get console status (#11)
        mvi      c,statf
        call    ?bdos       ;return through bdos
;
rdsret: ;return clean true value
        ora      a           ;zero?
        rz       r             ;return if so
        mvi      a,0ffh      ;clean true value
        ret
;
;*****
;*
;*****
vers:   ;get version number (#12)
        mvi      c,versf
        jmp      ?bdos       ;return through bdos
;
;*****
;*
;*****
reset:  ;reset disk system (#13)
        mvi      c,resetf
        jmp      ?bdos
;
;*****
;*
;*****
select: ;select disk (#14)
        ;l->fixed(7) drive number
        call    getpl      ;disk number to E
        mvi      c,seldf
        jmp      ?bdos       ;return through bdos
;
;*****
;*
;*****
open:   ;open file (#15)
        ;l-> addr(fcb)
        call    getp2i      ;fcb address to de
        mvi      c,openf
        jmp      ?bdos       ;return through bdos
;

```

```

;*****
;*
;*****close: ;close file (#16)
;1-> addr(fcb)
010B CD0600      call    getp2i          ;.fcb to DE
010E 0E10      mvi    c,closef
0110 C30000      jmp    ?bdos           ;return through bdos
;
;*****
;*
;*****sear: ;search for file (#17)
;1-> addr(fcb)
0113 CD0600      call    getp2i          ;.fcb to DE
0116 0E11      mvi    c,serchf
0118 C30000      jmp    ?bdos           ;return through bdos
;
;*****
;*
;*****searn: ;search for next (#18)
mvi    c,serchn      ;search next function
011B 0E12      jmp    ?bdos           ;return through bdos
;
;*****
;*
;*****delete: ;delete file (#19)
;1-> addr(fcb)
0120 CD0600      call    getp2i          ;.fcb to DE
0123 0E13      mvi    c,deletf
0125 C30000      jmp    ?bdos           ;return through bdos
;
;*****
;*
;*****rdseq: ;read file sequential mode (#20)
;1-> addr(fcb)
0128 CD0600      call    getp2i          ;.fcb to DE
012B 0E14      mvi    c,readf
012D C30000      jmp    ?bdos           ;return through bdos
;
;*****
;*
;*****wrseq: ;write file sequential mode (#21)
;1-> addr(fcb)
0130 CD0600      call    getp2i          ;.fcb to DE
0133 0E15      mvi    c,writf
0135 C30000      jmp    ?bdos           ;return through bdos
;
;*****
;*
;*****make: ;create file (#22)

```

```

;1-> addr(fcb)
0138 CD0600      call    getp2i          ;.fcb to DE
013B 0E16        mvi    c,makef
013D C30000      jmp    ?bdos           ;return through bdos
;
;*****
;*
;*****
rename: ;rename file (#23)
;1-> addr(fcb)
0140 CD0600      call    getp2i          ;.fcb to DE
0143 0E17        mvi    c,renamf
0145 C30000      jmp    ?bdos           ;return through bdos
;
;*****
;*
;*****
logvec: ;return login vector (#24)
        mvi    c,loginf
        jmp    ?bdos           ;return through BDOS
;
;*****
;*
;*****
curdsk: ;return current disk number (#25)
        mvi    c,cdiskf
        jmp    ?bdos           ;return value in A
;
;*****
;*
;*****
setdma: ;set DMA address (#26)
;1-> pointer (dma address)
0152 CD0600      call    getp2          ;dma address to DE
0155 0E1A        mvi    c,setwdf
0157 C30000      jmp    ?bdos           ;return through bdos
;
;*****
;*
;*****
allvec: ;return address of allocation vector (#27)
        mvi    c,getalf
        jmp    ?bdos           ;return through bdos
;
;*****
;*
;*****
wpdisk: ;write protect disk (#28)
015F CD1400      call    chkv20          ;must be 2.0 or greater
0162 0E1C        mvi    c,wrprof
0164 C30000      jmp    ?bdos
;
;*****
;*
;*****
rovec: ;return read/only vector (#29)

```

CP/M RMAC ASSEM 0.4 #010 DIRECT CP/M CALLS FROM PL/I-80

```

0167 CD1400    call    chkv20      ;must be 2.0 or greater
016A 0E1D      mvi     c,getrof
016C C30000    jmp     ?bdos       ;value returned in HL
;
;***** *****
;*
;***** *****
filatt: ;set file attributes (#30)
        ;l-> addr(fcb)
        call    chkv20      ;must be 2.0 or greater
        call    getp2i      ;.fcb to DE
        mvi     c,setattr
        jmp     ?bdos
;
;***** *****
;*
;***** *****
getdpb: ;get base of current disk parm block (#31)
        call    chkv20      ;check for 2.0 or greater
        mvi     c,getdpf
        jmp     ?bdos       ;addr returned in HL
;
;***** *****
;*
;***** *****
getusr: ;get user code to register A
        call    chkv20      ;check for 2.0 or greater
        mvi     e,0ffh       ;to get user code
        mvi     c,userrf
        jmp     ?bdos
;
;***** *****
;*
;***** *****
setusr: ;set user code
        call    chkv20      ;check for 2.0 or greater
        call    getpl       ;code to E
        mvi     c,userrf
        jmp     ?bdos
;
;***** *****
;*
;***** *****
rdran: ;read random (#33)
        ;l-> addr(fcb)
        call    chkv20      ;check for 2.0 or greater
        call    getp2i      ;.fcb to DE
        mvi     c,rdranf
        jmp     ?bdos       ;return through bdos
;
;***** *****
;*
;***** *****
wrran: ;write random (#34)
        ;l-> addr(fcb)
        call    chkv20      ;check for 2.0 or greater

```

CP/M RMAC ASSEM 0.4 #011 DIRECT CP/M CALLS FROM PL/I-80

```

01A5 CD0600      call    getp2i          ;.fcb to DE
01A8 0E22        mvi    c,wrranf
01AA C30000      jmp    ?bdos           ;return through bdos
;
;*****
;*
;*****
filsiz: ;compute file size (#35)
01AD CD1400      call    chkv20         ;must be 2.0 or greater
01B0 CD0600      call    getp2          ;.fcb to DE
01B3 0E23        mvi    c,filszf
01B5 C30000      jmp    ?bdos           ;return through bdos
;
;*****
;*
;*****
setrec: ;set random record position (#36)
01B8 CD1400      call    chkv20         ;must be 2.0 or greater
01BB CD0600      call    getp2          ;.fcb to DE
01BE 0E24        mvi    c,setrcf
01C0 C30000      jmp    ?bdos           ;return through bdos
;
;*****
;*
;*****
resdrv: ;reset drive function (#37)
;1->drive vector - bit(16)
01C3 CD1D00      call    chkv22         ;must be 2.2 or greater
01C6 CD0600      call    getp2          ;drive reset vector to DE
01C9 0E25        mvi    c,rsdrvrf
01CB C30000      jmp    ?bdos           ;return through bdos
;
;*****
;*
;*****
wrranz: ;write random, zero fill function
;1-> addr(fcb)
01CE CD1D00      call    chkv22         ;must be 2.2 or greater
01D1 CD0600      call    getp2i         ;.fcb to DE
01D4 0E28        mvi    c,wrrnzf
01D6 C30000      jmp    ?bdos           ;return through bdos
;
;*****
;*
;*****
01D9             end

```

CP/M RMAC ASSEM 0.4 #012

DIRECT CP/M CALLS FROM PL/I-80

015A ALLVEC	00E7 BREAK	0019 CDISKF	0014 CHKV20	001D CHKV22
008C CHRIN	009C CHROUT	00AE CHRSTR	010B CLOSE	0010 CLOSEF
00A2 CONINP	00B2 CONOUT	000D CR	014D CURDSK	0079 DBUFF
0120 DELETE	0013 DELETF	0071 DFCB0	0075 DFCB1	0006 DIOF
001A EOF	016F FILATT	01AD FILSIZ	0023 FILSZF	001B GETALF
017A GETDPB	001F GETDPF	00CA GETIO	0007 GETIOF	0000 GETP1
0006 GETP2	0006 GETP2I	001D GETROF	0182 GETUSR	000C GETVER
000A LF	0005 LISTF	0018 LOGINF	0148 LOGVEC	0138 MAKE
0016 MAKEF	0054 MEMPTR	0058 MEMSIZ	0066 MEMWDS	0103 OPEN
000F OPENF	0009 PRINTF	0004 PUNF	00DF RDBUF	0080 RDCON
000A RDCONF	0197 RDRAN	0021 RDRANF	008A RDRDR	0003 RDRF
0128 RDSEQ	00EC RDSRET	00BE RDSTAT	0001 READC	0014 READF
007D REBOOT	0140 RENAME	0017 RENAMF	01C3 RESDRV	00F6 RESET
000D RESETF	0167 ROVEC	0025 RSDRVF	0113 SEAR	011B SEARN
000E SELDF	00FB SELECT	0011 SERCHF	0012 SERCHN	001E SETATF
0152 SETDMA	001A SETDMF	00CF SETIO	0008 SETIOF	0024 SETRCF
01B8 SETREC	018C SETUSR	000B STATF	0020 USERF	0023 VERERR
002E VERMSG	00F1 VERS	000C VERSF	015F WPDISK	0085 WRCON
0002 WRITC	0015 WRITF	009A WRLST	001C WRPROF	0095 WRPUN
01A2 WRRAN	0022 WRRANF	01CE WRRANZ	0028 WRRNZF	0130 WRSEQ
00D7 WRSTR	0000 ?BDOS	0000 ?BEGIN	0000 ?BOOT	0000 ?DBUFF
0000 ?DFCB0	0000 ?DFCB1			

APPENDIX B:
LISTING OF "DIOCALLS"
SHOWING THE BASIC CP/M DIRECT INTERFACE

PL/I-80 V1.0, COMPILE OF: DIOCALLS

L: List Source Program

```
%include 'diomod.dcl';
NO ERROR(S) IN PASS 1
```

```
NO ERROR(S) IN PASS 2
```

PL/I-80 V1.0, COMPILE OF: DIOCALLS

```
1 a 0000 diotst:
2 a 0006      proc options(main);
3 a 0006      /* external CP/M I/O entry points */
4 a 0006      /* (note: each source line begins with tab chars) */
5+c 0006      dcl
6+c 0006      memptr entry           returns (ptr),
7+c 0006      memsiz entry           returns (fixed(15)),
8+c 0006      memwds entry           returns (fixed(15)),
9+c 0006      dfcb0 entry            returns (ptr),
10+c 0006     dfcbl entry            returns (ptr),
11+c 0006     dbuff entry             returns (ptr),
12+c 0006     reboot entry,
13+c 0006     rdcon entry           returns (char(1)),
14+c 0006     wrcon entry           (char(1)),
15+c 0006     rdrdr entry            returns (char(1)),
16+c 0006     wrpun entry           (char(1)),
17+c 0006     wrlst entry            (char(1)),
18+c 0006     coninp entry           returns (char(1)),
19+c 0006     conout entry           (char(1)),
20+c 0006     rdstat entry           returns (bit(1)),
21+c 0006     getio entry            returns (bit(8)),
22+c 0006     setio entry            (bit(8)),
23+c 0006     wrstr entry           (ptr),
24+c 0006     rdbuf entry            (ptr),
25+c 0006     break entry             returns (bit(1)),
26+c 0006     vers entry              returns (bit(16)),
27+c 0006     reset entry,
28+c 0006     select entry           (fixed(7)),
29+c 0006     open entry              (ptr) returns (fixed(7)),
30+c 0006     close entry             (ptr) returns (fixed(7)),
31+c 0006     sear entry              (ptr) returns (fixed(7)),
32+c 0006     searn entry             (ptr) returns (fixed(7)),
33+c 0006     delete entry             (ptr),
34+c 0006     rdseq entry             (ptr) returns (fixed(7)),
35+c 0006     wrseq entry             (ptr) returns (fixed(7)),
36+c 0006     make entry              (ptr) returns (fixed(7)),
37+c 0006     rename entry             (ptr),
38+c 0006     logvec entry             returns (bit(16)),
39+c 0006     curdsk entry             returns (fixed(7)),
40+c 0006     setdma entry             (ptr),
41+c 0006     allvec entry             returns (ptr),
42+c 0006     wpdisk entry             returns (bit(16)),
43+c 0006     rovec entry              (ptr),
44+c 0006     filatt entry             (ptr),
45+c 0006     getdpb entry             returns (ptr),
```

```

46+c 0006      getusr entry      returns (fixed(7)),
47+c 0006      setusr entry      (fixed(7)),
48+c 0006      rdran entry      (ptr) returns (fixed(7)),
49+c 0006      wrran entry      (ptr) returns (fixed(7)),
50+c 0006      filsiz entry     (ptr),
51+c 0006      setrec entry     (ptr),
52+c 0006      resdrv entry     (bit(16)),
53+c 0006      wrranz entry     (ptr) returns (fixed(7));
54 c 0006      dcl
55 c 0006          c char(1),
56 c 0006          v char(254) var,
57 c 0006          i fixed;

58 c 0006
59 c 0006
60 c 0006
61 c 0006
62 c 0006
63 c 0006
64 c 0006
65 c 0006
66 c 0006
67 c 0006
68 c 0006
69 c 0006
70 c 0006
71 c 0006
72 c 0006
73 c 0006
74 c 0006
75 c 0006
76 c 0006
77 c 0006
78 c 0006
79 c 0006
80 c 0006
81 c 000C
82 c 0012
83 c 0018
84 c 001E
85 c 0024
86 c 004A
87 c 008D
88 c 008D
89 c 0137
90 c 0137
91 c 0137
92 c 0137
93 c 0137
94 c 0137
95 c 0137
96 c 0153
97 c 0153
98 c 016A
99 c 017F
100 c 017F
101 c 017F
102 c 017F
103 c 017F
104 c 017F
105 c 017F

*****+
*      * Fixed Location Tests:      *
*      MEMPTR, MEMSIZ, MEMWDS,      *
*      DFCBØ, DFCB1, DBUFF          *
*      *                         *
*****/
dcl
    memptrv ptr,
    memsizv fixed,
    (dfcbØv, dfcblv, dbuffv) ptr,
    command char(127) var based (dbuffv),
    1 fcbØ based(dfcbØv),
        2 drive fixed(7),
        2 name char(8),
        2 type char(3),
        2 extnt fixed(7),
        2 space (19) bit(8),
        2 cr fixed(7),
    memory (Ø:Ø) based(memptrv) bit(8);
memptrv = memptr();
memsizv = memsiz();
dfcbØv = dfcbØ();
dfcblv = dfcbl();
dbuffv = dbuff();
put edit ('Command Tail: ',command) (a);
put edit ('First Default File:',
          fcbØ.name,'.',fcbØ.type) (skip,4a);
put edit ('dfcbØ ',unspec(dfcbØv),
          'dfcbl ',unspec(dfcblv),
          'dbuff ',unspec(dbuffv),
          'memptr',unspec(memptrv),
          'memsiz',unspec(memsizv),
          'memwds',memwds())
          (5(skip,a(7),b4),skip,a(7),f(6));
put skip list('Clearing Memory');
/* sample loop to clear mem */
do i = 0 repeat(i+1) while (i^=memsizv-1);
memory (i) = 'ØØ'b4;
end;

*****+
*      * REBOOT Test      *
*****+

```

```

106 c 017F *****/
107 c 017F put skip list ('Reboot? (Y/N)');
108 c 019B get list (c);
109 c 01B5 if translate(c,'Y','y') = 'Y' then
110 c 01DD call reboot();
111 c 01E0
112 c 01E0
113 c 01E0 *****/
114 c 01E0 *
115 c 01E0 * RDCON, WRCON Test *
116 c 01E0 *
117 c 01E0 *****/
118 c 01E0 put list('Type Input, End with "$" ');
119 c 01F7 v = '^m^j';
120 c 0204 do while (substr(v,length(v)) ^= '$');
121 c 0220 v = v || rdcon();
122 c 022E end;
123 c 022E put skip list('You Typed:');
124 c 024A do i = 1 to length(v);
125 c 0266 call wrcon(substr(v,i,1));
126 c 028E end;
127 c 028E
128 c 028E
129 c 028E *****/
130 c 028E *
131 c 028E * RDRDR and WRPUN Test *
132 c 028E *
133 c 028E *****/
134 c 028E put skip list('Reader to Punch Test?(Y/N)');
135 c 02AA get list (c);
136 c 02C4 if translate(c,'Y','y') = 'Y' then
137 c 02EC do;
138 c 02EC put skip list('Copying RDR to PUN until ctl-z');
139 c 0308 c = ' ';
140 c 0314 do while (c ^= '^z');
141 c 0323 c = rdrdr();
142 c 032E if c ^= '^z' then
143 c 033D call wrpun(c);
144 c 0346 end;
145 c 0346 end;
146 c 0346
147 c 0346
148 c 0346 *****/
149 c 0346 *
150 c 0346 * WRLST Test *
151 c 0346 *
152 c 0346 *****/
153 c 0346 put list('List Output Test?(Y/N)');
154 c 035D get list(c);
155 c 0377 if translate(c,'Y','y') = 'Y' then
156 c 039F do i = 1 to length(v);
157 c 03BB call wrlist(substr(v,i,1));
158 c 03E3 end;
159 c 03E3
160 c 03E3
161 c 03E3 *****/
162 c 03E3 *
163 c 03E3 * Direct I/O, CONOUT, CONINP *
164 c 03E3 *
165 c 03E3 *****/

```

```

166 c 03E3      put list
167 c 03FA      ('Direct I/O, Type Line, End with Line Feed');
168 c 03FA      c = ' ';
169 c 0406      do while (c ^= '^j');
170 c 0415      call conout(c);
171 c 041B      c = coninp();
172 c 0429      end;
173 c 0429
174 c 0429
175 c 0429      ****
176 c 0429      *
177 c 0429      * Direct I/O, Console Status *
178 c 0429      RDSTAT
179 c 0429      *
180 c 0429      ****
181 c 0429      put skip list('Status Test, Type Character');
182 c 0445      do while (^rdstat());
183 c 044F      end;
184 c 044F      /* clear the character */
185 c 044F      c = coninp();
186 c 045A
187 c 045A
188 c 045A      ****
189 c 045A      *
190 c 045A      *      GETIO, SETIO IObyte *
191 c 045A      *
192 c 045A      ****
193 c 045A      dcl
194 c 045A      iobyte bit(8);
195 c 045A      iobyte = getio();
196 c 0460      put edit ('IObyte is ',iobyte,
197 c 0493      ', New Value: ') (skip,a,b4,a);
198 c 0493      get edit (iobyte) (b4(2));
199 c 04AF      call setio(iobyte);
200 c 04B5
201 c 04B5
202 c 04B5      ****
203 c 04B5      *
204 c 04B5      * Buffered Write, WRSTR Test *
205 c 04B5      *
206 c 04B5      ****
207 c 04B5      put list('Buffered Output Test:');
208 c 04CC      /* "v" was previously filled by RDCON */
209 c 04CC      call wrstr(addr(v));
210 c 04D8
211 c 04D8
212 c 04D8      ****
213 c 04D8      *
214 c 04D8      * Buffered Read RDBUF Test *
215 c 04D8      *
216 c 04D8      ****
217 c 04D8      dcl
218 c 04D8      l inbuff static,
219 c 04D8      2 maxsize bit(8) init('80'b4),
220 c 04D8      2 inchars char(127) var;
221 c 04D8      put skip list('Line Input, Type Line, End With Return');
222 c 04F4      put skip;
223 c 0505      call rdbuf(addr(inbuff));
224 c 0511      put skip list('You Typed: ',inchars);
225 c 0536

```

```

226 c 0536 ****
227 c 0536 *
228 c 0536 *
229 c 0536 *      Console BREAK Test *
230 c 0536 *
231 c 0536 ****
232 c 0536 put skip list('Console Break Test, Type Character');
233 c 0552 do while(~break());
234 c 055C end;
235 c 055C c = rdcon();

236 c 0567 ****
237 c 0567 *
238 c 0567 ****
239 c 0567 *
240 c 0567 *      Version Number VERS Test *
241 c 0567 *
242 c 0567 ****
243 c 0567 dcl
244 c 0567     version bit(16);
245 c 0567 version = vers();
246 c 056D if substr(version,1,8) = '00'b4 then
247 c 0576     put skip list('CP/M'); else
248 c 0595     put skip list('MP/M');
249 c 05B1 put edit(' Version ',substr(version,9,4),
250 c 05F5     '.',substr(version,13,4)) (a,b4,a,b4);

251 c 05F5 ****
252 c 05F5 *
253 c 05F5 ****
254 c 05F5 *
255 c 05F5 *      Disk System RESET Test *
256 c 05F5 *
257 c 05F5 ****
258 c 05F5 put skip list('Resetting Disk System');
259 c 0611 call reset();

260 c 0614 ****
261 c 0614 *
262 c 0614 ****
263 c 0614 *
264 c 0614 *      Disk SELECT Test *
265 c 0614 *
266 c 0614 ****
267 c 0614 put skip list('Select Disk # ');
268 c 0630 get list(i);
269 c 0648 call select(i);

270 c 0654 ****
271 c 0654 *
272 c 0654 *
273 c 0654 * Note: The OPEN, CLOSE, SEAR,
274 c 0654 *      SEARN, DELETE, RDSEQ,
275 c 0654 *      WRSEQ, MAKE, and RENAME
276 c 0654 * functions are tested in the
277 c 0654 *      DIOCOPY program
278 c 0654 *
279 c 0654 ****
280 c 0654 ****
281 c 0654 *
282 c 0654 *
283 c 0654 *      LOGVEC and CURDSK
284 c 0654 *
285 c 0654 ****

```

```

286 c 0654      put skip list ('Login Vector',
287 c 0695          logvec(),'Current Disk',
288 c 0695          curdsk());
289 c 0695
290 c 0695      ****
291 c 0695          *
292 c 0695          * See DIOCOPY for SETDMA Function *
293 c 0695          *
294 c 0695      ****
295 c 0695
296 c 0695      ****
297 c 0695          *
298 c 0695          * Allocate Vector ALLVEC Test *
299 c 0695          *
300 c 0695      ****
301 c 0695
302 c 0695      dcl
303 c 0695          alloc (0:30) bit(8)
304 c 0695          based (allvec()),
305 c 0695          allvecp ptr;
306 c 0695          allvecp = allvec();
307 c 0700          put edit('Alloc Vector at ',
308 c 0700              unspec(allvecp),':',
309 c 0700              (alloc(i) do i=0 to 30))
310 c 0700              (skip,a,b4,a,254(skip,4(b,x(1))));

311 c 0700      ****
312 c 0700          *
313 c 0700          * Note: the following functions *
314 c 0700          * apply to version 2.0 or newer. *
315 c 0700          *
316 c 0700      ****
317 c 0700
318 c 0700      ****
319 c 0700          *
320 c 0700          * WPDISK Test *
321 c 0700          *
322 c 0700      ****
323 c 0700          put skip list('Write Protect Disk?(Y/N)');
324 c 071C          get list(c);
325 c 0736          if translate(c,'Y','y') = 'Y' then
326 c 075E              call wpdisk();

327 c 0761
328 c 0761
329 c 0761
330 c 0761      ****
331 c 0761          * ROVEC Test *
332 c 0761          *
333 c 0761      ****
334 c 0788          put skip list('Read/Only Vector is',rovec()));

335 c 0788
336 c 0788
337 c 0788      ****
338 c 0788          * Disk Parameter Block Decoding *
339 c 0788          * Using GETDPB *
340 c 0788          *
341 c 0788      ****
342 c 0788          ddpbp ptr,
343 c 0788          1 dpb based (ddpb),
344 c 0788          2 spt fixed(15),
345 c 0788          2 bsh fixed(7),

```

```

346 c 0788      2 blm bit(8),
347 c 0788      2 exm bit(8),
348 c 0788      2 dsm bit(16),
349 c 0788      2 drm bit(16),
350 c 0788      2 al0 bit(8),
351 c 0788      2 all bit(8),
352 c 0788      2 cks bit(16),
353 c 0788      2 off fixed(7);
354 c 0788      dpbp = getdpb();
355 c 078E      put edit('Disk Parameter Block:',
356 c 08C6      'spt',spt,'bsh',bsh,'blm',blm,
357 c 08C6      'exm',exm,'dsm',dsm,'drm',drm,
358 c 08C6      'al0',al0,'all',all,'cks',cks,
359 c 08C6      'off',off)
360 c 08C6      (skip,a,2(skip,a(4),f(6)),
361 c 08C6      4(skip,a(4),b4),
362 c 08C6      skip,2(a(4),b,x(1)),
363 c 08C6      skip,a(4),b4,
364 c 08C6      skip,a(4),f(6));
365 c 08C6
366 c 08C6      ****
367 c 08C6      *
368 c 08C6      *      Test Get/Set user Code      *
369 c 08C6      *          GETUSR, SETUSR      *
370 c 08C6      *
371 c 08C6      ****
372 c 08C6      put skip list
373 c 08FC      ('User is',getusr(),' ', New User:');
374 c 08FC      get list(i);
375 c 0914      call setusr(i);
376 c 0920
377 c 0920      ****
378 c 0920      *
379 c 0920      *          FILSIZ, SETREC,      *
380 c 0920      *          RDRAN, WRRAN, WRRANZ are      *
381 c 0920      *          tested in DIORAND      *
382 c 0920      *
383 c 0920      ****
384 c 0920
385 c 0920      ****
386 c 0920      *
387 c 0920      *          Test Drive Reset RESDRV      *
388 c 0920      *          (version 2.2 or newer)      *
389 c 0920      *
390 c 0920      ****
391 c 0920      dcl
392 c 0920      drvect bit(16);
393 c 0920      put list('Drive Reset Vector:');
394 c 0937      get list(drvect);
395 c 094F      call resdrv(drvect);
396 c 0955
397 c 0955      ****
398 c 0955      *
399 c 0955      *
400 c 0955      ****
401 a 0955      end diotst;

```

CODE SIZE = 0958
 DATA AREA = 04BA

APPENDIX C:
LISTING OF "DIOCOPY"
SHOWING DIRECT CP/M FILE I/O OPERATIONS

PL/I-80 V1.0, COMPILE OF: DIOCOPY

L: List Source Program

```
%include 'diomod.dcl';
%include 'fcb.dcl';
%include 'fcb.dcl';
%include 'fcb.dcl';
%include 'fcb.dcl';
NO ERROR(S) IN PASS 1
```

NO ERROR(S) IN PASS 2

PL/I-80 V1.0, COMPILE OF: DIOCOPY

```
1 a 0000 diocopy:
2 a 0006      proc options(main);
3 a 0006      /* file to file copy program */
4 a 0006      /* (all source lines begin with tabs) */
5 a 0006
6 c 0006      *replace
7 c 0006      bufwds by 64,    /* words per buffer */
8 c 0006      quest by 63,   /* ASCII '?' */
9 c 0006      true by '1'b,
10 c 0006      false by '0'b;
11 c 0006
12+c 0006      dcl
13+c 0006      memptr entry      returns (ptr),
14+c 0006      memsiz entry      returns (fixed(15)),
15+c 0006      memwds entry      returns (fixed(15)),
16+c 0006      dfcb0 entry      returns (ptr),
17+c 0006      dfcbl entry      returns (ptr),
18+c 0006      dbuff entry      returns (ptr),
19+c 0006      reboot entry,
20+c 0006      rdcon entry      returns (char(1)),
21+c 0006      wrcon entry      (char(1)),
22+c 0006      rdrdr entry      returns (char(1)),
23+c 0006      wrpun entry      (char(1)),
24+c 0006      wrlst entry      (char(1)),
25+c 0006      coninp entry      returns (char(1)),
26+c 0006      conout entry      (char(1)),
27+c 0006      rdstat entry      returns (bit(1)),
28+c 0006      getio entry      returns (bit(8)),
29+c 0006      setio entry      (bit(8)),
30+c 0006      wrstr entry      (ptr),
31+c 0006      rdbuf entry      (ptr),
32+c 0006      break entry      returns (bit(1)),
33+c 0006      vers entry      returns (bit(16)),
34+c 0006      reset entry,
35+c 0006      select entry      (fixed(7)),
36+c 0006      open entry      (ptr) returns (fixed(7)),
37+c 0006      close entry     (ptr) returns (fixed(7)),
38+c 0006      sear entry      (ptr) returns (fixed(7)),
39+c 0006      searn entry     returns (fixed(7)),
40+c 0006      delete entry     (ptr),
41+c 0006      rdseq entry     (ptr) returns (fixed(7)),
```

```

42+c 0006      wrseq entry      (ptr) returns (fixed(7)),
43+c 0006      make entry      (ptr) returns (fixed(7)),
44+c 0006      rename entry    (ptr),
45+c 0006      logvec entry   returns (bit(16)),
46+c 0006      curdsk entry   returns (fixed(7)),
47+c 0006      setdma entry   (ptr),
48+c 0006      allvec entry   returns (ptr),
49+c 0006      wpdisk entry,  returns (bit(16)),
50+c 0006      rovec entry,   (ptr),
51+c 0006      filatt entry,  returns (ptr),
52+c 0006      qetdpb entry,  returns (fixed(7)),
53+c 0006      getusr entry,  (fixed(7)),
54+c 0006      setusr entry,  (ptr) returns (fixed(7)),
55+c 0006      rdran entry,  (ptr) returns (fixed(7)),
56+c 0006      wrran entry,  (ptr) returns (fixed(7)),
57+c 0006      filsiz entry, (ptr),
58+c 0006      setrec entry, (ptr),
59+c 0006      resdrv entry, (bit(16)),
60+c 0006      wrranz entry, (ptr) returns (fixed(7));
61 c 0006

62 c 0006      dcl
63 c 0006      1 destfile,
64+c 0006      2 namel,
65+c 0006          3 drive fixed(7), /* drive number */
66+c 0006          3 fname char(8), /* file name */
67+c 0006          3 ftype char(3), /* file type */
68+c 0006          3 fext fixed(7), /* file extent */
69+c 0006          3 space (3) bit(8),/* filler */
70+c 0006          2 name2, /* used in rename */
71+c 0006              3 drive2 fixed(7),
72+c 0006              3 fname2 char(8),
73+c 0006              3 ftype2 char(3),
74+c 0006              3 fext2 fixed(7),
75+c 0006              3 space2 (3) bit(8),
76+c 0006          2 crec fixed(7), /* current record */
77+c 0006          2 rrec fixed(15), /* random record */
78+c 0006          2 rovf fixed(7); /* random rec overflow */

79 c 0006
80 c 0006      dcl
81 c 0006          dfcb0p ptr,
82 c 0006          1 sourcefile based(dfcb0p),
83+c 0006          2 namel,
84+c 0006              3 drive fixed(7), /* drive number */
85+c 0006              3 fname char(8), /* file name */
86+c 0006              3 ftype char(3), /* file type */
87+c 0006              3 fext fixed(7), /* file extent */
88+c 0006              3 space (3) bit(8),/* filler */
89+c 0006          2 name2, /* used in rename */
90+c 0006              3 drive2 fixed(7),
91+c 0006              3 fname2 char(8),
92+c 0006              3 ftype2 char(3),
93+c 0006              3 fext2 fixed(7),
94+c 0006              3 space2 (3) bit(8),
95+c 0006          2 crec fixed(7), /* current record */
96+c 0006          2 rrec fixed(15), /* random record */
97+c 0006          2 rovf fixed(7); /* random rec overflow */

98 c 0006
99 c 0006      dcl
100 c 0006          1 dfcblfile based(dfcbl()),
101+c 0006          2 namel,

```

```

102+c 0006      3 drive fixed(7), /* drive number */
103+c 0006      3 fname char(8), /* file name */
104+c 0006      3 ftype char(3), /* file type */
105+c 0006      3 fext fixed(7), /* file extent */
106+c 0006      3 space (3) bit(8),/* filler */
107+c 0006      2 name2,          /* used in rename */
108+c 0006      3 drive2 fixed(7),
109+c 0006      3 fname2 char(8),
110+c 0006      3 ftype2 char(3),
111+c 0006      3 fext2 fixed(7),
112+c 0006      3 space2 (3) bit(8),
113+c 0006      2 crec fixed(7), /* current record */
114+c 0006      2 rrec fixed(15), /* random record */
115+c 0006      2 rovf fixed(7); /* random rec overflow */

116 c 0006
117 c 0006      dcl
118 c 0006      1 renfile,
119+c 0006      2 name1,
120+c 0006      3 drive fixed(7), /* drive number */
121+c 0006      3 fname char(8), /* file name */
122+c 0006      3 ftype char(3), /* file type */
123+c 0006      3 fext fixed(7), /* file extent */
124+c 0006      3 space (3) bit(8),/* filler */
125+c 0006      2 name2,          /* used in rename */
126+c 0006      3 drive2 fixed(7),
127+c 0006      3 fname2 char(8),
128+c 0006      3 ftype2 char(3),
129+c 0006      3 fext2 fixed(7),
130+c 0006      3 space2 (3) bit(8),
131+c 0006      2 crec fixed(7), /* current record */
132+c 0006      2 rrec fixed(15), /* random record */
133+c 0006      2 rovf fixed(7); /* random rec overflow */

134 c 0006
135 c 0006      dcl
136 c 0006      answer char(1),
137 c 0006      extcnt fixed(7);

138 c 0006      dcl
139 c 0006      /* buffer management */
140 c 0006      eofile bit(8),
141 c 0006      i     fixed(15),
142 c 0006      m     fixed(15),
143 c 0006      nbuffs fixed(15),
144 c 0006      memory (0:0) bit(16) based(memptr()));

145 c 0006
146 c 0006
147 c 0006      /* compute number of buffs, 64 words each */
148 c 0006      nbuffs = divide(memwds(),bufwds,15);
149 c 0017      if nbuffs = 0 then
150 c 0020          do;
151 c 0020              put skip list('No Buffer Space');
152 c 003C              call reboot();
153 c 003F              end;

154 c 003F
155 c 003F      /* initialize fcb's */
156 c 003F      dfcb0p = dfcb0();
157 c 0045      destfile = dfcblfile;
158 c 0054
159 c 0054      /* copy fcb to rename file, count extents */
160 c 0054      renfile = destfile;
161 c 0060      /* search all extents by inserting '?' */

```

```

162 c 0060      renfile.fext = quest;
163 c 0065      if sear(addr(renfile)) ^= -1 then
164 c 0076          do;
165 c 0076          extcnt = 1;
166 c 007B          do while(searn() ^= -1);
167 c 0083          extcnt = extcnt + 1;
168 c 008A          end;
169 c 008A      put edit
170 c 00C1          ('OK to Delete ',extcnt,' Extent(s)?(Y/N)')
171 c 00C1          (skip,a,f(3),a);
172 c 00C1      get list(answer);
173 c 00DB          if ^ (answer = 'Y' ! answer = 'y') then
174 c 00FF          call reboot();
175 c 0102      end;
176 c 0102
177 c 0102      /* destination file will be deleted later */
178 c 0102      destfile.ftype = '$$$';
179 c 010E      /* delete any existing x.$$$ file */
180 c 010E      call delete(addr(destfile));
181 c 011A
182 c 011A      /* open the source file, if possible */
183 c 011A      if open(addr(sourcefile)) = -1 then
184 c 012B          do;
185 c 012B          put skip list('No Source File');
186 c 0147          call reboot();
187 c 014A      end;
188 c 014A
189 c 014A      /* source file opened, create $$$ file */
190 c 014A      destfile.fext = 0;
191 c 014F      destfile.crec = 0;
192 c 0154      if make(addr(destfile)) = -1 then
193 c 0165          do;
194 c 0165          put skip list('No Directory Space');
195 c 0181          call reboot();
196 c 0184      end;
197 c 0184
198 c 0184      /* $$$ temp file created, now copy from source */
199 c 0184      eofile = false;
200 c 0189      do while (^eofile);
201 c 0190          m = 0;
202 c 0196          /* fill buffers */
203 c 0196          do i = 0 repeat (i+1) while (i<nbuffs);
204 c 01A6          call setdma(addr(memory(m)));
205 c 01B9          m = m + bufwds;
206 c 01C3          if rdseq(addr(sourcefile)) ^= 0 then
207 c 01D4              do;
208 c 01D4              eofile = true;
209 c 01D9              /* truncate buffer */
210 c 01D9              nbuffs = i;
211 c 01E9          end;
212 c 01E9
213 c 01E9          end;
214 c 01EF          m = 0;
215 c 01EF          /* write buffers */
216 c 0206          do i = 0 to nbuffs-1;
217 c 0219          call setdma(addr(memory(m)));
218 c 0223          m = m + bufwds;
219 c 0234          if wrseq(addr(destfile)) ^= 0 then
220 c 0234              do;
221 c 0250              put skip list('Disk Full');
              call reboot();

```

```
222 c 0260           end;
223 c 0260           end;
224 c 0260           end;
225 c 0260
226 c 0260 /* close destination file and rename */
227 c 0260 if close(addr(destfile)) = -1 then
228 c 0271   do;
229 c 0271     put skip list('Disk R/O');
230 c 028D     call reboot();
231 c 0290   end;
232 c 0290
233 c 0290 /* destination file closed, erase old file */
234 c 0290 call delete(addr(renfile));
235 c 029C
236 c 029C /* now rename $$$ file to old file name */
237 c 029C destfile.name2 = renfile.name1;
238 c 02AB call rename(addr(destfile));
239 c 02B7 call reboot();
240 a 02BA end diocopy;
```

CODE SIZE = 02BD

DATA AREA = 00EF

APPENDIX D:
LISTING OF "DIORAND"
SHOWING EXTENDED RANDOM ACCESS CALLS

PL/I-80 V1.0, COMPILED BY: DIORAND

L: List Source Program

```
%include 'diomod.dcl';
%include 'fcb.dcl';
NO ERROR(S) IN PASS 1

NO ERROR(S) IN PASS 2
```

PL/I-80 V1.0, COMPILED BY: DIORAND

```
1 a 0000 diorand:
2 a 0006      proc options(main);
3 a 0006      /* random access tests for 2.0 and 2.2 */
4 a 0006
5+c 0006      dcl
6+c 0006          memptr entry      returns (ptr),
7+c 0006          memsiz entry      returns (fixed(15)),
8+c 0006          memwds entry      returns (fixed(15)),
9+c 0006          dfcb0 entry      returns (ptr),
10+c 0006         dfcbl entry      returns (ptr),
11+c 0006         dbuff entry      returns (ptr),
12+c 0006         reboot entry,
13+c 0006         rdcon entry      returns (char(1)),
14+c 0006         wrcon entry      (char(1)),
15+c 0006         rdrdr entry      returns (char(1)),
16+c 0006         wrpun entry      (char(1)),
17+c 0006         wrlst entry      (char(1)),
18+c 0006         coninp entry     returns (char(1)),
19+c 0006         conout entry     (char(1)),
20+c 0006         rdstat entry     returns (bit(1)),
21+c 0006         getio entry      returns (bit(8)),
22+c 0006         setio entry      (bit(8)),
23+c 0006         wrstr entry      (ptr),
24+c 0006         rdbuf entry      (ptr),
25+c 0006         break entry      returns (bit(1)),
26+c 0006         vers entry      returns (bit(16)),
27+c 0006         reset entry,
28+c 0006         select entry     (fixed(7)),
29+c 0006         open entry      (ptr) returns (fixed(7)),
30+c 0006         close entry     (ptr) returns (fixed(7)),
31+c 0006         sear entry      (ptr) returns (fixed(7)),
32+c 0006         searn entry     returns (fixed(7)),
33+c 0006         delete entry    (ptr),
34+c 0006         rdseq entry     (ptr) returns (fixed(7)),
35+c 0006         wrseq entry     (ptr) returns (fixed(7)),
36+c 0006         make entry      (ptr) returns (fixed(7)),
37+c 0006         rename entry    (ptr),
38+c 0006         logvec entry    returns (bit(16)),
39+c 0006         curdsk entry    returns (fixed(7)),
40+c 0006         setdma entry    (ptr),
41+c 0006         allvec entry    returns (ptr),
42+c 0006         wpdisk entry   returns (bit(16)),
43+c 0006         rovec entry     (ptr),
44+c 0006         filatt entry   (ptr),
```

```

45+c 0006      getdpb entry           returns (ptr),
46+c 0006      getusr entry           returns (fixed(7)),
47+c 0006      setusr entry           (fixed(7)),
48+c 0006      rdran entry            (ptr) returns (fixed(7)),
49+c 0006      wrran entry             (ptr) returns (fixed(7)),
50+c 0006      filsiz entry            (ptr),
51+c 0006      setrec entry             (ptr),
52+c 0006      resdrv entry            (bit(16)),
53+c 0006      wrranz entry             (ptr) returns (fixed(7));
54 c 0006

55 c 0006      dcl
56 c 0006          1 database,
57+c 0006          2 name1,
58+c 0006              3 drive fixed(7), /* drive number */
59+c 0006              3 fname char(8), /* file name */
60+c 0006              3 ftype char(3), /* file type */
61+c 0006              3 fext fixed(7), /* file extent */
62+c 0006              3 space (3) bit(8),/* filler */
63+c 0006          2 name2,           /* used in rename */
64+c 0006              3 drive2 fixed(7),
65+c 0006              3 fname2 char(8),
66+c 0006              3 ftype2 char(3),
67+c 0006              3 fext2 fixed(7),
68+c 0006              3 space2 (3) bit(8),
69+c 0006          2 crec fixed(7), /* current record */
70+c 0006          2 rrec fixed(15), /* random record */
71+c 0006          2 rovf fixed(7); /* random rec overflow */

72 c 0006
73 c 0006      dcl
74 c 0006          lower char(26) static initial
75 c 0006              ('abcdefghijklmnopqrstuvwxyz'),
76 c 0006          upper char(26) static initial
77 c 0006              ('ABCDEFGHIJKLMNOPQRSTUVWXYZ');

78 c 0006
79 c 0006      dcl
80 c 0006          /* simple variables */
81 c 0006          i      fixed,
82 c 0006          fn     char(20),
83 c 0006          c      char(1),
84 c 0006          code   fixed(7),
85 c 0006          mode   fixed(2),
86 c 0006          zerofill bit(1),
87 c 0006          version bit(16);

88 c 0006
89 c 0006      dcl
90 c 0006          /* overlays on default buffer */
91 c 0006          bitbuf (128) bit(8) based(dbuff()),
92 c 0006          buffer char(127) var based(dbuff());

93 c 0006
94 c 0006          put skip list('Random Access Test');
95 c 0022          /* check version number for 2.0 */
96 c 0022          version = vers();
97 c 0028          if substr(version,9,8) < '20'b4 then
98 c 0031              do;
99 c 0031                  put skip list('You Need Version 2');
100 c 004D
101 c 0050
102 c 0050          stop;
103 c 006C
104 c 0086          end;
102 c 0050          put skip list('Zero Record Fill?');
103 c 006C          get list(c);
104 c 0086          zerofill = (c = 'Y' ! c = 'y') &

```

```

105 c 00B5           substr(version,9,8) >= '22'b4;
106 c 00B5
107 c 00B5
108 c 00B5
109 c 00D1
110 c 00EB
111 c 0110
112 c 0110
113 c 0110
114 c 0120
115 c 0129
116 c 0131
117 c 0131
118 c 013B
119 c 013B
120 c 013B
121 c 0153
122 c 016C
123 c 016C
124 c 0188
125 c 018B
126 c 018B
127 c 01A4
128 c 01A4
129 c 01A4
130 c 01A4
131 c 01B4
132 c 01BD
133 c 01BD
134 c 01BD
135 c 01CA
136 c 01D9
137 c 01D9
138 c 01D9
139 c 01D9
140 c 01F5
141 o 020F
142 c 020F
143 c 020F
144 c 020F
145 c 0214
146 c 0214
147 c 0225
148 c 0225
149 c 0241
150 c 0252
151 c 0252
152 c 026E
153 c 0274
154 c 0274
155 c 0274
156 c 0274
157 c 0274
158 c 0280
159 c 02B2
160 c 02B2
161 c 02B2
162 c 02B2
163 c 02B2
164 c 02BE

           substr(version,9,8) >= '22'b4;

/* read and process file name */
put skip list('Data Base Name: ');
get list(fn);
fn = translate(fn,upper,lower);

/* process optional drive prefix */
i = index(fn,':');
if i = 0 then
    drive = 0;
else
if i = 2 then
    do;
        /* convert character to drive code */
        drive = index(upper,substr(fn,1,1));
        if drive = 0 ! drive > 16 then
            do;
                put skip list('Bad Drive Name');
                stop;
            end;
        fn = substr(fn,i+1);
    end;

/* get file name and optional type */
i = index(fn,'.');
if i = 0 then
    do;
        /* no file type specified, use .DAT */
        fname = fn;
        ftype = 'DAT';
    end;
else
    do;
        fname = substr(fn,1,i-1);
        ftype = substr(fn,i+1);
    end;

/* clear the extent field */
fext = 0;

if open(addr(database)) = -1 then
    do;
        put skip list('Creating New Database');
        if make(addr(database)) = -1 then
            do;
                put skip list('No Directory Space');
                stop;
            end;
    end;
else
    do;
        call filsiz(addr(database));
        put skip list('File Size:',rrec,' Records');
    end;

/* main processing loop */
do while('1'b);
    call setrec(addr(database));
    put skip list('Current Record',rrec);

```

```

165 c 02E5      put skip list('Read(0),Write(1),Quit(2)? ');
166 c 0301      get list(mode);
167 c 031A      if mode < 2 then
168 c 0322          do;
169 c 0322          put skip list('Record Number? ');
170 c 033E          get list(rrec);
171 c 035B          rovf = 0;
172 c 0360          end;
173 c 0360      if mode = 0 then
174 c 0367          do;
175 c 0367          code = rdran(addr(database));
176 c 0376          if code = 0 then
177 c 037D              do;
178 c 037D              if bitbuf(1) = '00'b4 then
179 c 0386                  put skip list('Zero Record');
180 c 03A5          else
181 c 03A5              put skip list(buffer);
182 c 03C2          end;
183 c 03C2      else
184 c 03C2          put skip list('Return Code',code);
185 c 03F0      end;
186 c 03F0      else
187 c 03F0      if mode = 1 then
188 c 03F7          do;
189 c 03F7          put skip list('Data: ');
190 c 0413          get list(buffer);
191 c 042F          if zerofill then
192 c 0436              code = wrranz(addr(database));
193 c 0448          else
194 c 0448              code = wrran(addr(database));
195 c 0457          if code ^= 0 then
196 c 045E              put skip list('Return Code',code);
197 c 048C          end;
198 c 048C      else
199 c 048C      if mode = 2 then
200 c 0494          do;
201 c 0494          if close(addr(database)) = -1 then
202 c 04A5              put skip list('Read/Only');
203 c 04C1              stop;
204 c 04C7              end;
205 c 04C7      end;
206 a 04C7      end diorand;

CODE SIZE = 04C7
DATA AREA = 0183

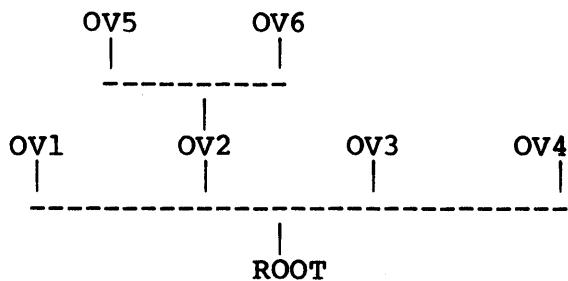
```

APPENDIX E
OVERLAYS AND FILE LOCATION CONTROLS

This appendix describes several additional features incorporated into LINK-80 and LIB-80 in release versions later than 1.0, including extensions to process run-time overlays, and controls for location of source, intermediate, and destination files. Use of the automatic PL/I-80 library search "request item" is included, along with a description of new command line error reporting formats. Additional LIB-80 facilities are also included for deleting or replacing various modules in a subprogram library.

E.1.0. OVERLAYS

LINK may be used to produce a simple tree structure of overlays as shown in the diagram below:



In addition to producing ROOT.COM and ROOT.SYM files, LINK will produce an OVL file and a SYM file for each overlay specified in the command line. The OVL file consists of a 256-byte header containing the load address and length of the overlay, followed by the absolute object code. The origin of an overlay is the highest address of the module below it on the 'tree' rounded up to the next 128-byte boundary. The stack and free space for the PL/I program will be located at the top of the highest overlay linked, rounded up to the next 128-byte boundary. This address is written to the console upon completion of the entire link and is patched into the root module in the location '?MEMORY'. The SYM file contains only those symbols which have not been declared in another module lower in the 'tree'.

The following restrictions must be observed when producing a system of overlays with PL/I-80 and LINK:

Each overlay has one entry point by which it is entered. This entry point is assumed by the overlay manager to be at the base (load address) of the overlay.

No upward references are allowed from a module to an entry point in an overlay higher on the tree, other than the main entry point of the overlay as described in 1. Downward references to entry points in overlays lower on the tree or in the root module are allowed.

The overlays are not relocatable. Hence the root module must be a COM file.

Common blocks (Externals in PL/I) which are declared in one module may not be initialized by a module higher in the tree. Any attempt to do so will be ignored by LINK.

Overlays may be nested to a depth of 5 levels.

The default buffer located at 80H is used by the overlay manager, so user programs should not depend on data stored in this buffer.

E.1.1. USING OVERLAYS IN PL/I PROGRAMS

There are two ways to use overlays in a PL/I program. The first method is very straightforward, and will suffice for most applications. However, it has the restrictions that all overlays must be on the default drive, and overlay names may not be determined at run-time. The second method does not have these restrictions, and involves a slightly more complicated calling sequence.

To use the first method, an overlay is simply declared as an entry constant in the module where it is referenced. As an entry constant, it may have parameters declared in a parameter list. The overlay itself is simply a PL/I procedure, or group of procedures. For example, the following program is a root module having one overlay:

```
root: procedure options (main);
  declare ovl entry (char (15));
  put skip list ('root');
  call ovl ('overlay 1');
end root;
```

The overlay OVL.PLI appears as follows:

```
ovl: procedure (c);
    declare c char (15);
    put skip list (c);
end ovl;
```

Note that if parameters are passed to an overlay, it is the programmer's responsibility to ensure that the number and type of the parameters are the same in the calling program and the overlay itself.

To link these two programs into an overlay system, the following link command would be used:

```
LINK ROOT(OVL)
```

(The command line syntax for linking overlays is described in detail in a later section.)

LINK will produce four files from this command: ROOT.COM, ROOT.SYM, OVL.OVL and OVL.SYM. When ROOT.COM is executed, it will first put the message 'root' out at the console. The 'call ovl' statement will transfer control to the overlay manager. The overlay manager loads the file OVL.OVL from the default drive at the proper location above ROOT.COM and transfers control to it, passing the char (15) parameter in the normal manner. The overlay then executes, producing the message 'overlay 1' at the console. It then returns directly to the statement following the 'call ovl' in root.pli, and execution continues from that point.

Using this method, if the overlay manager determines that the requested overlay is already in memory, the overlay will not be reloaded before control is transferred to it. There are several important notes regarding this first overlay method:

The name associated with the overlay in the call and entry statements is the actual name of the OVL file loaded by the overlay manager, so the two names must agree. Since symbol names are truncated to 6 characters in the REL file produced by PL/I-80, the names of the OVL files must be limited to 6 characters.

The name of the entry point to an overlay (the name of the procedure) need not agree with the name used in the calling sequence. The same name should be used to avoid confusion.

The overlay manager will only load overlays from the default drive (the drive which was the default drive when execution of the root module began, regardless of any changes to the default drive which may have occurred since then).

The names of the overlays are fixed - the source program must be edited, recompiled and relinked to change the names of the overlays.

No non-standard PL/I statements are needed (the program is transportable to other systems).

In some applications it is useful to have greater flexibility with overlays, such as the ability to load overlays from different drives, or the ability to determine the name of an overlay at run-time, say from the keyboard or from a disk file. This is accomplished using a second overlay method.

In this case, an explicit entry point into the overlay manager must be declared in the PL/I program as follows:

```
declare ?ovlay entry (char (10), fixed (1));
```

The first parameter is a character string specifying the name of the overlay to load and an optional drive code in the standard CP/M format 'd:filename'. The second parameter is the load flag. If the load flag is 1, the overlay manager will load the specified overlay whether or not it is already in memory. If the load flag is 0, the overlay will only be loaded if it is not already in memory.

The 'call ?ovlay' statement tells the overlay manager to load the requested overlay, if needed. The overlay manager returns to the calling program, which must then perform a dummy call to execute the overlay just processed by the overlay manager. This allows a parameter list to be passed to the overlay.

The example shown in the first method above would appear as follows:

```
root: procedure options (main);
    declare ?ovlay entry (char (10), fixed (1));
    declare dummy entry (char (15));
    declare name char (10);
    put skip list ('root');
    name = 'OVL';
    call ?ovlay (name, 0);
    call dummy ('overlay 1');
end root;
```

OVL.PLI would be the same as before.

At run-time the overlay manager would load OVL.OVL from the default drive, since that is the current value of the variable 'name', and then return to the calling program (in this case, root). At this point, the argument 'overlay 1' would be set up according to the PL/I-80 parameter passing conventions. The 'call dummy' transfers control to the overlay manager, which would simply transfer control to the base address of the overlay whose name was just processed. When OVL is finished, it returns to the statement following the 'call dummy' statement. Note that while in the example above, 'name' was set to 'OVL' in an assignment statement, the overlay name could have been supplied as a character string derived from some other source,

such as the operator's keyboard. Several important points must be observed when using the second overlay technique:

A drive code may be specified so overlays may be loaded from drives other than the default drive. If no drive is specified, the default drive is used as described in Method 1.

Since the name of the overlay is specified in the character string (and not by the entry symbol), it may be up to 8 characters in length.

If there are any parameters in the dummy call following the 'call ?ovlay', they must agree in number and type with the parameters in the procedure declaration in the overlay.

E.1.2. SPECIFYING OVERLAYS IN THE COMMAND LINE

The syntax for specifying overlays is similar to that for linking without overlays, except that each overlay specification is enclosed in parentheses. An overlay specification may be in one of the following forms:

```
link root(ovl)  
link root(ovl,part2,part3)  
link root(ovl=part1,part2,part3)
```

The first command produces the file OVL.OVL from a file OVL.REL, while the second command produces the OVL.OVL file from OVL.REL, PART2.REL, and PART3.REL. In the last case, the OVL.OVL file is produced from PART1.RLE, PART2.REL, and PART3.REL.

Note that a left parenthesis, which indicates the start of a new overlay specification, also indicates the end of the group preceding it. In other words, the following command line is invalid and will be flagged as an error:

```
LINK ROOT(OVL),MOREROOT
```

All files to be included at any point on the 'tree' must appear together, without any intervening overlay specifications. Thus the following command is valid:

```
LINK ROOT,MOREROOT(OVL)
```

Any filename in the command line may be followed by a number of link switches enclosed in square brackets, as described in the LINK-80 Operator's Guide. Note that the overlay specifications are not set

off from the root module or from each other with commas. Spaces may be used to improve readability.

Nesting of overlays is indicated in the command line by nesting parentheses. The following command line could be used to link the overlay system shown on the first page of the overlay description:

```
LINK ROOT (OV1) (OV2 (OV5) (OV6)) (OV3) (OV4)
```

E.1.3. SAMPLE LINK EXECUTION

In the following sample link operation, notice that OV1 is flagged as an undefined symbol. LINK is simply indicating that OV1 has not been defined in the current module, so it is assumed to be either the name of an overlay or a dummy entry point to an overlay. When linking overlays, each entry variable which refers to an overlay (by actual name or a dummy entry) will appear as an undefined symbol. No symbols other than these actual or dummy overlay entry points should be undefined.

```
A>LINK ROOT(OV1)
LINK 1.1

PLILIB RQST ROOT 0100 /SYSIN/ 1A15 /SYSPRI/ 1A3A
```

UNDEFINED SYMBOLS:

OV1

```
ABSOLUTE 0000
CODE SIZE 18BC (0100-19BB)
DATA SIZE 02A9 (1A90-1D38)
COMMON SIZE 00D4 (19BC-1A8F)
USE FACTOR 4E
```

LINKING OV1.OVL

```
PLILIB RQST

ABSOLUTE 0000
CODE SIZE 0024 (1D80-1DA3)
DATA SIZE 0002 (1DA4-1DA5)
COMMON SIZE 0000
USE FACTOR 09

MODULE TOP 1E00
```

A>ROOT

```
root
overlay 1
End of Execution
A>
```

E.1.4. RUN-TIME ERROR MESSAGES

The overlay manager may produce one of the following error messages:

ERROR (8) OVERLAY, NO FILE d:filename.OVL
The indicated file could not be found.

ERROR (9) OVERLAY, DRIVE d:filename.OVL
An invalid drive code was passed as a parameter to ?ovlay.

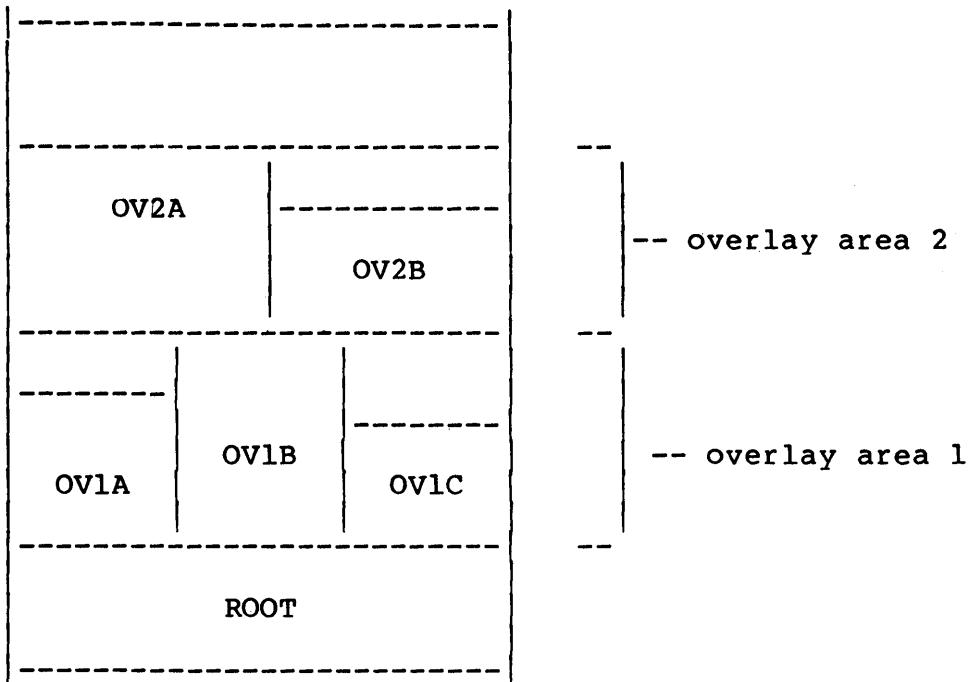
ERROR (10) OVERLAY, SIZE d:filename.OVL
The indicated overlay would overwrite the PL/I stack and/or free space if it were loaded.

ERROR (11) OVERLAY, NESTING d:filename.OVL
Loading the indicated overlay would exceed the maximum nesting depth.

ERROR (12) OVERLAY, READ d:filename.OVL
Disk read error during overlay load, probably caused by premature EOF.

E.1.5. OTHER OVERLAY SYSTEMS

A system of overlays may also be produced which is not a tree structure, but rather contains a number of separate overlay areas, as shown in the figure below:



In such a system, the root module can reference any of the overlays. An overlay may reference entry points in the root module or the main entry point of any overlay which is not in the same overlay area.

Linking a system of overlays as shown above is done in a number of steps. One link must be performed for each overlay area, since the address of the top of the overlay area must be supplied to LINK when linking the next higher overlay area. For example, the command

```
LINK ROOT (OV1A) (OV1B) (OV1C)
```

generates the three overlays in overlay area 1, and indicates the top address of the module. This address is supplied as the load address in the next command:

```
LINK ROOT (OV2A[Lmod top]) (OV2B [Lmod top])
```

This command creates the overlays for overlay area 2 at the appropriate address. Note that the overlay area which is the highest in memory should be linked last, since the module top address is always written into the root module at the end of the link.

At some point after the entire system has been linked, it may be desirable to relink only one overlay, which may not be at the top overlay area. This may be done using the \$OZ switch to prevent generation of a root module which would contain an erroneous ?MEMORY value.

It is the responsibility of the programmer to ensure that none of the overlays overlap, and that no overlay attempts to reference

another overlay in the same overlay area.

E.1.6. THE LINK-80 "\$" SWITCH

The '\$' switch is used to control the source and destination devices under LINK-80. The general form of the switch is:

\$td

where 't' is a type and 'd' is a drive specifier. There are five types:

C - console

I - intermediate

L - library

O - object

S - symbol

The drive specifier may be a letter in the range 'A' thru 'P' corresponding to one of sixteen logical drives, or one of the following special characters:

X - console

Y - printer

Z - byte bucket

\$Cd - Console

Messages which normally appear at the console may be directed to the list device (\$CY) or may be suppressed (\$CZ). Once \$CY or \$CZ has been specified, \$CX may be used later in the command line to redirect console messages to the console device.

\$Id - Intermediate

Intermediate files generated by LINK are normally placed on the default drive. The \$I switch allows the user to specify another drive to be used by LINK for intermediate files.

\$Ld - Library

LINK normally searches on the default drive for library files

which are automatically linked because of a request item in a REL file. The \$L switch instructs LINK to search the specified drive for these library files.

\$Od - Object

LINK normally generates an object file on the same drive as the first REL file in the command line, unless an output file with an explicit drive is included in the command. The \$O switch instructs LINK to place the object file on the drive specified by the character following the \$O, or to suppress the generation of an object file if the character following the \$O is a 'Z'.

\$Sd - Symbol

LINK normally generates a symbol file on the same drive as the first REL file in the command line, unless an output file with an explicit drive is included in the command. The \$S switch instructs LINK to place the symbol file on the drive specified by the character following the \$S, or to suppress the generation of a symbol file if the character following the \$S is a 'Z'.

'td' character pairs following a '\$' must not be separated by commas. The entire group of \$ switches is set off from any other switches by a comma, as shown below:

```
LINK PART1[$SZ,$OD,$LB,Q],PART2
```

```
LINK PART1[$SZODLB,Q],PART2
```

```
LINK PART1[$SZ OD LB],PART2[Q]
```

The three command lines above are equivalent.

The \$I switch specifies the drive to be used for intermediate files during the entire link operation. The other '\$' switches may be changed in the command line. The value of a '\$' switch will remain in effect until it is changed as the command line is processed from left to right. This is generally useful only when linking overlays. For example:

```
LINK ROOT (OV1[$SZCZ]) (OV2) (OV3) (OV4[$SACX])
```

will suppress the SYM files and console output generated when OV1, OV2 and OV3 are linked. When OV4 is linked, the SYM file will be placed on drive A: and the console output will be sent to the console device.

The NR and NL switches used in LINK 1.0 to suppress the recording and listing of the symbol table are not recognized by LINK 1.1, since \$SZ and \$CZ can be used to perform these functions.

E.1.7. THE REQUEST ITEM

Version 1.1 of PL/I-80 uses the request item (a specific bit pattern in a REL file) to indicate to LINK that the PLILIB is to be searched. This is also how the Microsoft compilers link their run-time libraries. When LINK processes a library request, it first searches for an IRL file with the specified filename. If there is no IRL file, it searches for a REL file of that name. Failing in both searches, the error message

NO FILE: filename.REL

is produced, and LINK aborts. Libraries requested in this manner will appear in the symbol table listed at the console with a value of 'RQST'.

E.1.8. COMMAND LINE ERRORS

The error messages 'FILE NAME ERROR' and 'INVALID SYNTAX' are no longer generated. Instead, when a command line error of any kind is detected the command tail is echoed up to the point where the error occurred, followed by a question mark. For example:

```
LINK A, B, C; D
A, B, C;?
```

```
LINK LONGFILENAME
LONGFLEN?
```

E.1.9. ADDITIONAL LIB-80 FACILITIES

Modules in a library may be deleted or replaced in a single command. The names of the modules to be affected are enclosed in angle brackets immediately following the name of the source file containing the modules. The following examples demonstrate the use of this feature.

```
lib newlib=oldlib<mod1>
lib newlib=oldlib<mod1=file1>
lib newlib=oldlib<mod1=>
lib newlib=oldlib<mod1,mod2=file2,mod3=>
```

In the first case, a new library NEWLIB.REL is created which is the same as OLDDLIB.REL except that the module MOD1 is replaced by the

contents of the file MOD1.REL. This form should be used if the name of the module being replaced is the same as the filename of the REL file replacing the module.

In the second case, the module MOD1 is replaced by the contents of the file FILE1.REL in the new library NEWLIB.REL. This form is used to replace a module when the name of the module is not the same as the name of the file which is to replace it. Note that this form must be used if the filename has more than 6 characters, since module names in the REL file are truncated to 6 characters.

When the third command is used, NEWLIB.REL is created from OLDDLIB.REL without the module MOD1.

The last command form demonstrates that a number of replace and/or delete instructions may be included within the angle brackets.

E.2.0. MULTI-LINE COMMANDS

If a command does not fit on a single line (126 characters), the command may be extended by terminating the command line with an ampersand (&). The ampersand may appear after any character of the command, and need not follow a file name. LINK-80 responds with an asterisk (*) on the next line. At this point the command line may be continued. Any number of lines ending with an ampersand may be entered. The last line of the command is terminated with a carriage return. Note that XSUB may be used to submit multi-line LINK-80 commands.

Example:

```
A>link main, iomod1, iomod2, iomod3, iomod4, iomod5,&
LINK 1.3
*lib1[s], lib2[s], lib3[s], lib4&
*[s], lastmod\b2000&
*,d200]
( . . . symbol table and memory map. . . )
A>
```

APPENDIX F

XREF

XREF is an assembly-language cross reference utility that can be applied to print (PRN) files produced by MAC or RMAC in order to provide a summary of variable usage throughout the program. The purpose of this appendix is to provide the information necessary for operation of the XREF utility.

F.1.0. XREF OPERATION

XREF is normally invoked by issuing the command:

```
XREF filename
```

where the "filename" refers to two input files prepared using MAC or RMAC with assumed (and unspecified) file types of "PRN" and "SYM" and one output file with an assumed (and unspecified) file type of "XRF". Specifically, XREF reads the file "filename.PRN" line by line, attaches a line number prefix to each line and writes each prefixed line to the output file "filename.XRF". During this process, each line is scanned for any symbols that exist in the file "filename.SYM". Upon completion of this copy operation, XREF appends to the file "filename.XRF" a cross reference report that lists all the line numbers where each symbol in "filename.SYM" appears. In addition, each line number reference where the referenced symbol is the first token on the line is flagged with a "#" character. Also, the value of each symbol, as determined by MAC or RMAC and placed in the symbol table file "filename.SYM", is reported for each symbol.

As an option, the "filename" specification can be prefaced with a drive code in the standard CP/M format [d:]. When the drive code is specified all the files described above are associated with the specified drive. Otherwise, the files are associated with the default drive. Another option allows the user to direct the output file directly to the "LST:" device instead of to the file "filename.XRF". This option is invoked by adding the string "\$p" to the command line as follows:

```
XREF filename $p
```

XREF allocates space for symbols and symbol references dynamically during execution. If no memory is available for an attempted symbol or symbol reference allocation, an error message is issued and XREF is terminated.

F.1.1. XREF ERROR MESSAGES

No SYM file - This message is issued if the file "filename.SYM" is not present on the default or specified drive.

No PRN file - This message is issued if the file "filename.PRN" is not present on the default or specified drive.

Symbol table overflow - This message is issued if no space is available for an attempted symbol allocation.

Invalid SYM file format - This message is issued when an invalid "filename.SYM" file is read. Specifically, a line in the SYM file not terminated with a CRLF will force this error message.

Symbol table reference overflow - This message is issued if no space is available for an attempted symbol reference allocation.

"filename.XRF" make error - This message is issued if BDOS returns an error code after a "filename.XRF" make request. This error code usually indicates that no directory space exists on the default or specified drive.

"filename.XRF" close error - This message is issued if BDOS returns an error code after a "filename.XRF" close request.

"filename.XRF" write error - This message is issued if BDOS returns an error code after a "filename.XRF" write request. This error code usually indicates that no unallocated data blocks are available or no directory space exists on the default or specified drive.

