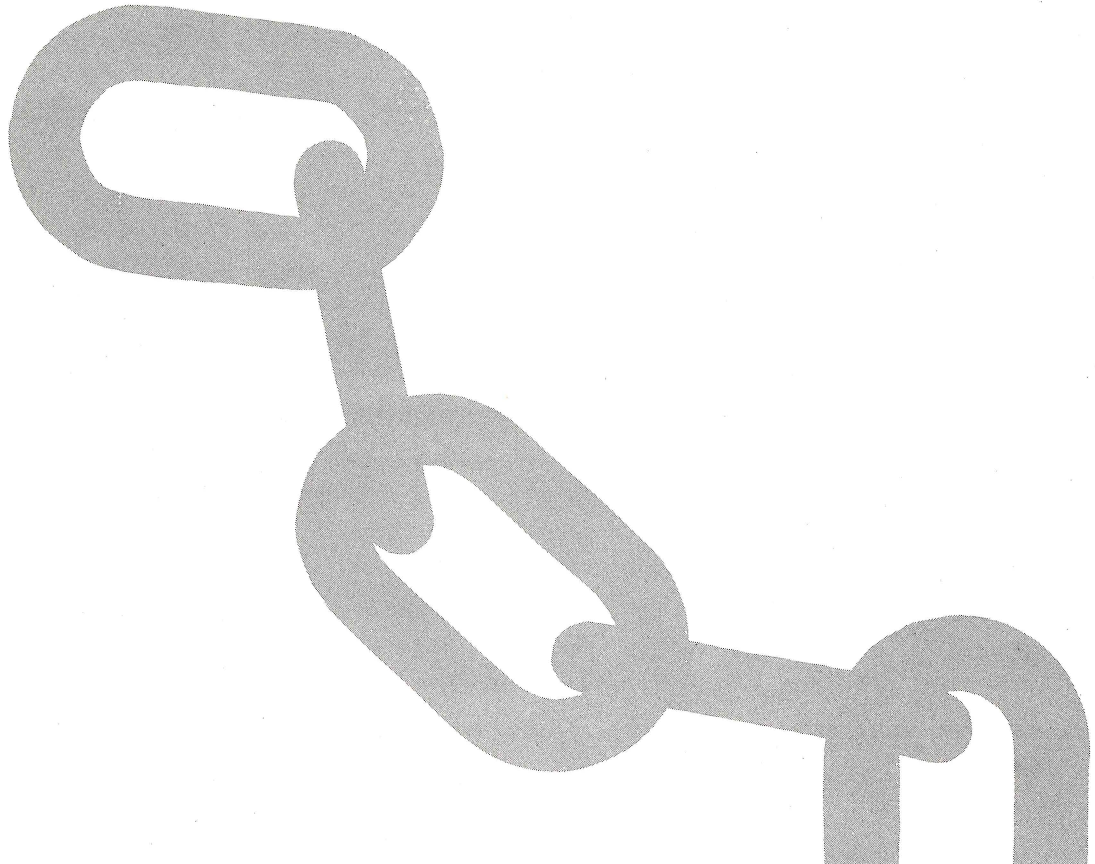
LINKER



The
Software Factory

LINKER

A LINKAGE EDITOR/LOADER FOR THE APPLE II

BY **DON WORTH**

TABLE OF CONTENTS

ASSUMPTIONS AND RECOMMENDATIONS

LINKER requires the following as a minimum:

32K APPLE II or APPLE II PLUS
DISK II
some form of assembler

LINKER is compatable with all versions of DOS up to and including 3.2.1. It should be noted that the above requirements are minimal; a good assembler is very important to productivity, although the integer basic ROM mini-assembler can be used. Throughout this document, examples of assembler language programs are in the TED II + assembler (from the WOZPAK) format. Defined below are the pseudo-ops used by TED II +:

ORG	Sets address where program will run
EQU	Equates a label to a numeric value
DS	Reserves space without assigning it a value
DA	Stuffs address of a label into memory (LO/HI)
DW	Stuffs a byte value into memory
HEX	Stuffs several bytes of hex into memory
ASC	Stuffs several bytes of ASCII into memory

You would be well advised to choose an assembler which will support at least the functions listed above (perhaps with different names).

Another consideration is storage of the assembler output. To use LINKER you must store the memory image created by your assembler as a binary file with a name ending in ".OBJ". Although you can do this by using the DOS BSAVE command, you must then know the length and memory location of the object code. Some assemblers will automatically do this for you and are therefor easier to use with LINKER.

At the Software Factory we use the Programma International full screen editor (PIE), and a modified form of the TED assembler for program development.

LINKAGE EDITORS

In computing, the process of taking a program from initial keying-in to final execution is called the development cycle. This cycle is diagrammed in figure 1.

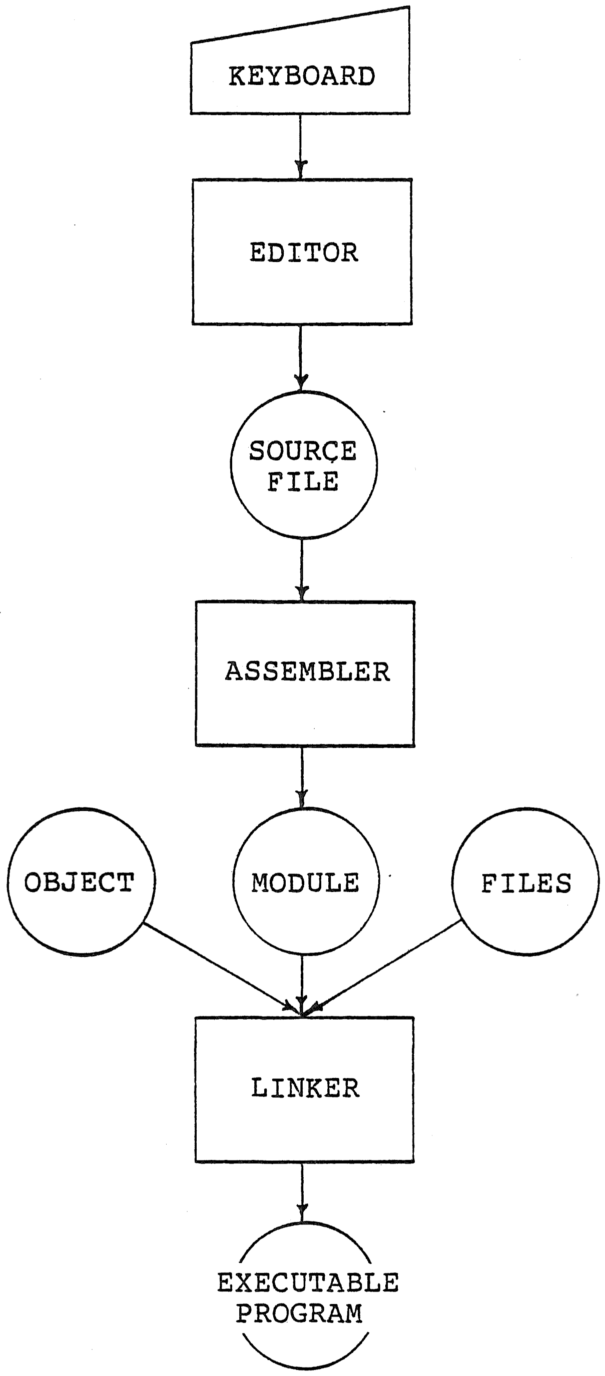


FIGURE 1.

The cycle begins with the programmer keying in his source statements. These are accepted by a program called an EDITOR. The editor usually stores the completed source program as a file on the floppy disk. At the next step, an ASSEMBLER is invoked to translate the source statements into binary machine code. This machine or "object" code is stored on disk as an "object module".

Before LINKER became available, most APPLE program development ended here. An object module was loaded and run at the location for which it was assembled, using the DOS BRUN command. It had to be completely self-sufficient (not requiring any outside subroutines except for those in absolute locations, such as ROM) or it had to be manually combined with other routines at fixed addresses.

With LINKER, however, another step is added, following assembly. The linkage edit step combines and relocates one or more object modules together to form a final, executable module or "load module". LINKER allows the programmer to specify, at linkage edit time, where in memory his module will execute, and relocates the input object modules as necessary, updating their call statements as the locations of their subroutines shift. Once LINKER has finished loading and "connecting" all the object modules, the memory image can be BSAVEd as an executable module or CALLed immediately. If an error occurs in execution, the cycle repeats, starting with the use of the editor to correct the error.

AN EXAMPLE

Suppose you are writing a program called "DEMO" which will act as a musical calculator. It will ask the user for two numbers, multiply them together, and print the answer, producing a tone for each digit printed. Using LINKER and its associated library of subroutines to do this demonstrates their value as a development aid. To see how the demo program works, BRUN DEMO on the LINKER distribution diskette. A source listing of the DEMO program is given in APPENDIX A. DEMO calls seven library subroutines:

BTOA	Converts an internal binary number into printable ASCII numeric digits.
ATOB	Converts ASCII numeric digits, as typed by the user, to a binary number.
DELAY	Waits a period of time or until a keypush before returning to caller.
PRINT	Prints one or more lines of ASCII text.
MULT	Multiplies two numbers together.
TONE	Produces a tone of a given frequency and duration on the APPLE's speaker.
EXIT	Exits to BASIC.

In addition, six subroutines in the monitor ROM are called:

GET	$FD6A	Gets an input line from the keyboard
COUT	$FDED	Prints a single character
CEOS	$FC42	Clears to end of screen
HOME	$FC58	Erases screen
TEXT	$FB39	Sets TEXT mode
VTAB	$FB5B	Does a vertical tab

Since the above six subroutines are at fixed or "absolute" locations, their addresses are assembled into DEMO directly. The other seven will be added to DEMO by LINKER later.

When the program in APPENDIX A is assembled and stored on diskette (using BSAVE) as an object module file (DEMO.OBJ) LINKER may be run. The results of such a run are shown in Figure 2. A BSAVE command is issued to save the final DEMO module and a BRUN executes it. Notice that all of the library subroutines required by DEMO have been added to make the final load module.

```
>BRUN LINKER

L I N K E R - V2.0

COPYRIGHT 1980 THE SOFTWARE FACTORY

(LINKER AT A$1000)

LOW ADDRESS  (IN HEX)        ?800
HIGH ADDRESS (IN HEX)        ?1000
LOAD DOWN FROM HI            ?N
MODULES ON PAGE BOUNDARIES   ?N

LOAD WHAT MODULE             ?DEMO

LOAD WHAT MODULE             ?

MODULE   A$      USE

DEMO     0800      0
BTOA     098B      1
ATOB     09FC      1
DELAY    0A4C      1
PRINT    0A6F      1
MULT     0AA6      1
TONE     0AD7      1
EXIT     0B01      1

TOTAL  A$0800,L$0331

>
```

FIGURE 2.

WRITING ASSEMBLER PROGRAMS FOR USE WITH LINKER

Assembler language modules processed by LINKER must conform to certain conventions. This practice allows the use of any assembler with LINKER rather than requiring a special assembler which can put out relocation and global symbol dictionaries. Although these conventions are fairly easy to follow, it is vital that they be carefully adhered to in order to avoid problems at link or execution time.

A typical object module consists of a single named subroutine or main program, called a "Control Section" (CSECT). A CSECT must begin with a 15 byte header and end with a two byte end-of-section marker. This is so that LINKER will know the name and size of each CSECT it processes. A typical header definition (from the DEMO program) is shown below:

```
+0      JMP     DEMO                JMP TO ENTRY POINT
+3      ASC     "DEMO    "          8 CHARACTER CSECT NAME
+11     HEX     0000                RESERVED FOR LINKER'S...
+13     HEX     0000                ...USE. MUST BE 0.
```

A CSECT always ends with:

```
        HEX     FFF0                $FFF0 END OF CSECT MARKER
```

Within a CSECT, all instructions, data, address constants, and global subroutine references must be collected into separate segments. This is so that LINKER will know what it is dealing with at all times during the relocation of the object module.

All instructions within the CSECT must be contained within one or more instruction segments. Each instruction segment must begin with the following two byte header:

```
        HEX     FFF1                $FFF1 INSTRUCTION SEGMENT
```

This tells LINKER that what follows is a segment of pure instructions. No embedded data is allowed (except as noted below) since LINKER must relocate all absolute JMPs or JSRs (etc.) to labels within the CSECT for the final location of the object module. If data must appear within an instruction segment (as with an inline argument list for PRINT) it must be preceeded by a $DA and end with a $00 byte.

Example:

```
        JSR     PRINT           CALL PRINT SUBROUTINE
        HEX     DA              STARTING EMBEDDED DATA
        ASC     "D E M O"       DATA TO PASS TO PRINT
        HEX     8D              CARRIAGE RETURN
        HEX     00              END OF EMBEDDED DATA
        LDA     #"?             CONTINUE WITH NEXT INSTRUCTION
```

Collect all raw data into data segments. This prevents LINKER from trying to relocate anything imbedded in it that might accidentally look like an address or instruction. Each data segment must begin with the following two byte header:

```
        HEX     FFF4            $FFF4 DATA SEGMENT
```

You must insure that the data does not contain anything which might be interpreted as a valid segment header ($FFF0-$FFF4).

Address constant segments contain one or more two byte address constants which will require relocation. Address constants which are fixed values (such as ROM addresses or $0000) need not be included, since they will not require relocation. Addresses which are labels in the CSECT being assembled should be included, however. The following header marks the beginning of an address constant segment:

```
        HEX     FFF2            $FFF2 ADCON SEGMENT
```

Global symbol reference segments contain JMP instructions to be "tied" to external CSECTS. Thus, if you are calling PRINT from DEMO, you are referencing the globally known symbol "PRINT" from the DEMO CSECT. Global reference segments begin with:

```
        HEX     FFF3            $FFF3 GLOBALS SEGMENT
```

Each global reference has the form:

```
PRINT   JMP     $0000           LINKER WILL FILL IN ADDRESS
        ASC     "PRINT   "      8 BYTE GLOBAL NAME
```

You should take heed of the following warnings:

1) Do not make references to labels in your program with immediate operands, such as:

```
ME      LDA     #<ME            PUT ADDRESS OF ME...
        STA     POINT           INTO POINT
        LDA     #>ME
        STA     POINT+1
```

LINKER will not detect these and they will not be relocated. Use address constants instead, viz:

```
ME      LDA     MEPTR           PUT ADDRESS OF ME...
        STA     POINT           INTO POINT
        LDA     MEPTR+1
        STA     POINT+1
        .
        .       (REST OF PROGRAM)
        .
        HEX     FFF2            ADCON SEGMENT
MEPTR   DA      ME              ADDRESS OF 'ME'
```

2) Avoid moving the program counter forward in an assembly, skipping an area in the object module:

```
BUFF    DS      256
```

or

```
BUFF=*
*=BUFF+256
```

You may be skipping over "garbage" which looks like segment headers. If you must use this practice, first zero the memory which will be used to hold the assembled object code.

3) Avoid data sequences which look like segment headers ($FFF0-$FFF4). Data sequences like this appearing after execution has started, of course, will not affect LINKER.

USING LINKER

To invoke LINKER, boot DOS and type:

```
     BRUN LINKER
```

The following screen will appear:

```
L I N K E R - V2.0
COPYRIGHT 1980 THE SOFTWARE FACTORY


(LINKER AT A$1000)


LOW ADDRESS (IN HEX)         ?
```

Check the LINKER's location (in this case $1000). If your program is to reside in the same general area, you must first relink LINKER itself to another "safe" place in memory. LINKER will not work very well if you have it load your program on top of itself! To avoid this, pick an area of memory you are not using about $1000 long, and follow the instructions below using LINKER as the name of the module to load. Then run the new LINKER to link your module.

Now enter the hex address of the first byte in memory for the output load module (800, for example).

You will now be asked:

```
HIGH ADDRESS (IN HEX)        ?
```

Give the address of the highest byte in memory (+1) which may be used to build the module. Note that the area between the low and high addresses must be large enough to contain the completed load module as well as the LINKER's global symbol dictionary. For safety's sake, assign an area as large as you can (only what is needed will be used) but at least as large as the final load module plus 10 per cent for the dictionary. Let's assume you entered 1000 for this example.

The next question is:

```
LOAD DOWN FROM HI          ?
```

If you want the normal way of loading a module (that is from the low address upward through memory) just hit the return key (or enter "N"). If, on the other hand, you want to pack your module up against some high limit (like HIMEM, for example) and you don't care where it starts in memory (just where it ends), respond "Y" to the question. For our example, we will assume you respond with a "N".

LINKER now asks:

```
MODULES ON PAGE BOUNDARIES  ?
```

If you want all the object modules in your final load module packed together with no space between them, hit return or "N". If it is important to your program that each CSECT starts on an even 256 byte "page" boundary (for timing or whatever), reply with a "Y". Note that forcing page boundaries will use more memory for the load module. In our example, we will reply "N".

At this point, LINKER asks for the module you want to link:

```
LOAD WHAT MODULE           ?
```

Insert the diskette containing the first object module file (usually your 'main' program) in the drive you used to BRUN LINKER. Enter the name of the CSECT (do not include ".OBJ") and hit return. LINKER will proceed to load your object module, relocating it to its new location, then it will search the diskette for each of the subroutines called by your main program and each of their subroutines, loading and relocating them too. If LINKER can not find an object module file on the diskette, it makes an internal notation of this, but continues until all modules that can be loaded are loaded. LINKER then asks for the name of the next module you wish to load. Ordinarily, if all the modules needed were present on the diskette you used, you will just hit return, telling LINKER that the module is complete. If you wish to have LINKER search another diskette for modules it needs, switch diskettes, type the name of any CSECT in your program (your main program again, for example) and hit return. When LINKER has searched all your diskettes, enter a null name, as described above, to exit LINKER. In our example, we entered DEMO as the name of the module to load.

As it exits back to BASIC, LINKER produces a memory map of your load module which might look like this:

```
MODULE   A$       USE


DEMO     0800       0
BTOA     098B       1
ATOB     09FC       1
DELAY    0A4C       1
PRINT    0A6F       1
MULT     0AA6       1
TONE     0AD7       1
EXIT     0B01       1
TOTAL   A$0800,L$0331


>
```

The first object module LINKER loaded, in this example, was DEMO. It was loaded at the low memory address you specified ($0800). The 0 under USE means that this module was 'used' by no other object module as a subroutine, since it is the 'main' program. DEMO makes references to seven other CSECTS; BTOA, ATOB, DELAY, PRINT, MULT, TONE and EXIT. These were loaded one by one right after DEMO in memory and their starting addresses are listed. In each case they were used only by DEMO so their use count is 1. If LINKER was unable to find a CSECT its name will appear in the list with '????' for a starting address. (If you try to run the completed load module and it calls the missing CSECT execution will go to location $0000.) Finally, LINKER gives the lowest address and the total length of the module. To save the finished executable program you could type:

```
    BSAVE DEMO,A$800,L$331
```

and to run it you would type:

```
    BRUN DEMO
```

LINKER ERROR MESSAGES

LINKER can produce the following error messages while loading your module.

INSUFFICIENT MEMORY - The space you alloted in memory for the load module (low address to high address) was not big enough for both the module and the global symbol dictionary. Either lower the low address or raise the high address.

BAD OBJECT MODULE STRUCTURE - One of the object modules LINKER was processing did not conform to the conventions explained in the section WRITING ASSEMBLER PROGRAMS FOR USE WITH LINKER. It is not possible for LINKER to tell you which one it is, but you can try using LINKER to load each CSECT one by one by name to zero in on the culprit.

DISK I/O ERROR - This is a general catch-all error message for any problem having to do with the disk. It could mean you were out of DOS buffers, the object module file ended prematurely, or that a real I/O error occured.

If you accidentally hit reset while running LINKER you may be able to recover what you were doing by calling it at its starting point (normally $1000 unless you've relinked it).

There are at least two ways you can drive LINKER "crazy". If LINKER seems to go into a loop reading the diskette and then finally comes back with an insufficient memory message, its possible that it was trying to load an object module whose DOS file name did not match any of the CSECT names within it. Always make sure that the name of an object module file is the same as one of the programs it contains. If LINKER just freezes up or produces random kinds of garbage or error messages it is likely that you have tried to link your module over the top of LINKER itself. Check its location and move it to another part of memory (by relinking it) if necessary.

LINKER SUBROUTINE LIBRARY

This section describes the subroutines provided for your use on the LINKER distribution diskette. For an example of their use, see APPENDIX A.

PRINT - L$37

This routine allows you to easily print text on the screen (or whatever the output device is) just like you would in BASIC.

INPUT: Simply follow the JSR to PRINT with the text to be printed, preceeded with a $DA and ended with a $00. For example:

```
         JSR     PRINT          CALL PRINT SUBROUTINE
         HEX     DA             EMBEDDED DATA STARTS
         ASC     "LINE 1"
         HEX     8D             CARRIAGE RETURN
         ASC     "LINE 2"
         HEX     8D00           CARRIAGE RETURN/END OF DATA
```

Notice that unless you include carriage returns ($8D) PRINT will not put any in. The $DA and the $00 are not printed. Execution continues with the instruction immediately following the $00.

USES: $3C,$3D

CALLS: $FDED

ATOB - L$50

Converts an ASCII number, as typed in from the keyboard for instance, into its binary equivalent so it can be operated upon arithmetically.

INPUT: $3C/$3D contains the address (LO/HI format) of the first numeric digit of the ASCII string. ATOB will convert up to 5 digits or until a non numeric is encountered.

OUTPUT: $3E/$3F will contain the positive binary result (LO/HI). The Y register will contain the number of valid digits converted.

USES: $3C through $41

BTOA - L$71

BTOA converts binary numbers to ASCII digits for printing. This routine is the reverse of ATOB.

INPUT: The X register should contain the left fill character ($A0 for blank fill), the Y and A registers contain the number to be converted (LO/HI), and $3C/$3D contains the address of the 5 byte output area.

OUTPUT: The output area will contain the number, right adjusted and left filled with the fill character.

USES: $3C through $41

FORMAT - L$3B

This routine performs the same function as BTOA (in fact it calls BTOA) except it prints the number after converting it.

INPUT: The Y and A register should contain the number to be converted (LO/HI).

OUTPUT: Prints 5 characters, left filled with blanks, numeric digits right adjusted.

USES: $3C through $41 and $200-$204

CALLS: BTOA, $FDED

EXIT - L$30

When you want your program to end its execution in a "nice" way, JMP to EXIT. EXIT will give control to the active BASIC (under DOS if it is active). There are no input arguments and no zero page bytes are used.

DELAY - L$23

You can call this routine to cause your APPLE to "spin its wheels" for a period of time while a display is on the screen or to slow down some program's operation. You can set the delay period for anything from 1/10 of a second to 25 seconds in 1/10 of a second intervals. If, prior to the end of the delay period, the APPLE user hits a key, DELAY will return with the time remaining in the A register.

INPUT: A register contains the number of tenths of seconds to delay before returning (0-255, 0=256). For more than 25.6 seconds, call DELAY in a loop.

OUTPUT: A register contains the number of tenths of seconds of the period which have not expired. The zero flag is set so you can disable the keystroke feature by BNEing back to the JSR to DELAY.

CALLS: $FCA8

TONE - L$2A

Calling this subroutine with a frequency and duration (0-255) you can generate simple tones on the APPLE speaker.

INPUT: $3C contains the frequency (0-255) and $3D contains the duration (0-255).

USES: $3C,$3D

RND - L$68

This is a fairly uniform random number generator for generating random integers. It works very much like the RND function in Integer BASIC.

INPUT: Y and A registers contain the highest value the random number can be plus one (LO/HI).

OUTPUT: Y and A registers contain the random number (0 to HIGHVALUE-1).

USES: $50 through $55 and $4E/$4F

CALLS: MULT

OPEN - L$13C

Similar in function to the OPEN command in DOS. You should call this subroutine whenever you want to start reading or writing a text file on the disk. OPEN searches the disk for the proper file, creating it if necessary, and positions to the beginning of the file.

INPUT: Y and A registers contain the address (LO/HI) of a seven byte parameter list:

Y/A -->	+0,+1	Address of a 30 byte file name (LO/HI)
	+2,+3	Record length or 0,0
	+4	Volume number or 0
	+5	Drive to use (0 for last used)
	+6	Slot to use (0 for last used)

The X register should contain either $00 to indicate that the file may be created if it doesn't already exist, or $80 to indicate that it must exist already.

OUTPUT: Unless no DOS buffers were free, the Y and A registers will contain the address (LO/HI) of an allocated DOS buffer for the file. You must save this since it is a required input to POSN, READ, and WRITE, and you must CLOSE it when you are done to free the buffer.
The carry flag is set if an error occured and the X register contains one of the following error codes:

0	NO ERRORS
4	WRITE PROTECTED
5	END OF DATA
6	FILE NOT FOUND
7	VOLUME MISMATCH
8	I/O ERROR
9	DISK FULL
10	FILE LOCKED
12	NO BUFFERS AVAILABLE
13	NOT A TEXT FILE

if return code 12 occurs, Y/A will contain zeros but CLOSE may be called.

USES: $3C through $45 and DOS ZPAGE.

CALLS: FBUFF, POSN, FIO

FIO - Incorporated into OPEN.OBJ

This routine is the linkage subroutine to DOS and is called by OPEN, CLOSE, READ, WRITE, and POSN. It should not be called directly.

CLOSE - L$40

For every OPEN call there must be an eventual CLOSE call to do any final updates on the diskette and free the file buffer.

INPUT: Y and A registers contain the address of the open file buffer (LO/HI) as returned by OPEN.

USES: $3C through $44 and DOS ZPAGE.

CALLS: FIO

POSN/READ/WRITE - L$AC

These three subroutines are all part of the object module, POSN.OBJ. POSN may be called to position the file pointer (the location of the next byte to be read or written). READ is called to read one or more bytes into an area in memory. WRITE is called to write a number of bytes from an area of memory.

INPUT: The calling sequence for all three subroutines is similar. Y and A registers contain the address (LO/HI) of a six byte parameter list:

```
      Y/A -->    +0,+1     Address of open DOS buffer
                 +2,+3     Relative record number for POSN or...
                           Length to READ or WRITE (LO/HI)
                 +4,+5     Byte offset for POSN or...
                           The data address (LO/HI) for READ/WRITE
```

OUTPUT: The carry flag is set if an error occured and the X register contains a return code as defined under the OPEN subroutine.

USES: $3C through $45 and DOS ZPAGE.

CALLS: FIO

FBUFF - L$3B

This routine is called by OPEN to locate a free DOS buffer. If you wish to provide your own buffers to OPEN you may replace FBUFF.OBJ with your own version. DOS buffers have the following format:

+0	30 byte file name area
+30	address of 45 byte workarea
+32	address of 256 byte T/S list area
+34	address of 256 byte data area
+36	address of next buffer on chain

If you are content to use DOS's buffers you may ignore FBUFF.

INPUT: none

OUTPUT: The carry flag is set if no free buffer can be found. Otherwise, $3C/$3D contain the address of the free buffer.

USES: $3C,$3D

MULT - L$31

This routine is similar to the one provided in the NON-AUTOSTART ROM. It is provided so that your programs can work with the AUTOSTART ROM when a multiply routine is needed. Only positive numbers are used.

INPUT: $50/$51/$52/$53 contains a 4 byte binary number (LO to HI) and $54/$55 contains the other number. In general $52/$53 should contain zeros.

OUTPUT: $50/$51/$52/$53 contains the 4 byte result of the multiplication.

USES: $50 through $55.

DIVD - L$33

Divides one number into another. Only positive numbers are used.

INPUT: $50/$51/$52/$53 contains the dividend and $54/$55 contains the divisor.

OUTPUT: $50/$51/$52/$53 contains the quotient.

USES: $50 through $55.

LKED - L$5D1

You may call the functional part of LINKER as a subroutine if you want to dynamically load, relocate, and link programs during the execution of your program. This is especially useful if you want to do overlays. See the section on ADVANCED TOPICS for more information on how this is done.

```
INPUT:  Y and A registers contain address of an 18 byte parameter list:
        Y/A -->   +0         8 character module name (no .OBJ)
                  +8         Slot (or 0 for last used)
                  +9         Drive (or 0 for last used)
                  +10        Flags
                               $80 - Load down from HI
                               $40 - Ignore modules not found
                               $20 - Even page boundaries
                  +11        Return code on output
                               0 - No errors
                               2 - Module not found
                               4 - Out of memory for modules
                               6 - Out of memory for dictionary
                               8 - Inconsistent module structure
                               10- Disk I/O error
                  +12,+13    LO address (LO/HI)
                  +14,+15    HI address (LO/HI)
                  +16,+17    Address of 2 byte anchor
```

The anchor bytes must be preset to zeros. They will be set by LINKER to point to a chain of all modules processed.

OUTPUT: LO and HI addresses are updated to reflect the actual space occupied by the output load module. The carry flag is set if an error occurs. The 2 byte anchor is set to point to the first module loaded and each module is chained to the next (see ADVANCED TOPICS section).

USES: $00 through $09 and $3C through $3F, DOS ZPAGE.

CALLS: OPEN, CLOSE, READ.

DOS TEXT FILE ACCESS

It should be noted that the OPEN, CLOSE, POSN, READ, and WRITE subroutines that constitute LINKER's DOS text file access method call DOS's file management subroutines directly (via the 3-page jump vector). This has several ramifications. Since the usual method of printing DOS commands with a control-D character is not used, there is a significant increase in efficiency. Also, no checking is done within the file manager for 'empty bytes' (hex zeros) within a file so an end of file condition will only occur if no more disk sectors exist (except for random files). With this access method you can dynamically position to any byte in the file and read or write any number of bytes to/from a memory buffer (INPUT/OUTPUT statements are not the medium for data transfer). This means you can store binary values of any kind (control characters, binary zeros, internal flags and binary values) on the disk. The interface to the DOS file manager is a reasonably rigidly defined interface which has remained constant across all versions of DOS to date. No version dependent patches or jumps to DOS are made.

ADVANCED TOPICS

RELINKING A PREVIOUSLY LINKED MODULE

In general it is better to rebuild a load module "from scratch", that is from its component object module files. If this is not possible or undesirable, you can run a previously linked load module (containing several combined CSECTS) through LINKER by renaming it to the ".OBJ" file name format. By doing this you can obtain a map of a previously linked module or add to it or change it.

Suppose you have previously linked DEMO and have misplaced the individual object modules that built it. Now, however, you want to replace the PRINT subroutine in the DEMO load module with a special one you have written. To do this you would rename DEMO to DEMO.OBJ so that LINKER can find it on the diskette. You would put your new PRINT subroutine out as PRINT.OBJ and then BRUN LINKER. When LINKER asks for the first module to load, specify PRINT. When it askes for the next module, specify DEMO. What happens is that LINKER will use the first copy of PRINT it finds (your new version) and will ignore the old version contained in the DEMO load module. There are two disadvantages to this procedure. One is that now PRINT will be the first CSECT in your new load module, meaning the entry point of the load module is now somewhere in the middle. This makes it impossible to use the BRUN command to execute it. Also, LINKER will not recover the space occupied by the duplicate copy of the PRINT subroutine, so, even though it will never be called, it will continue to be part of the output load module. To avoid these problems, you are better off to always construct your load modules from individual object modules.

Occasionally it is advantageous to have several CSECTS in a single object module file. One reason for this is the case of two or more subroutines which all need to share the same code or data. POSN, READ, and WRITE are an example of this. The code for each of these is the same except that a different entry code is used. A seperate CSECT header is set up for each of these "named entry points", followed by a couple of instructions to load a register with the proper entry code value, and a JMP to the common code (contained in the last CSECT, WRITE). The miniature CSECT ends with $FF,$F0 and is followed by the header of the next. When LINKER loads POSN.OBJ it will find and remember the entry points READ and WRITE as well. Remember, however, that a multiple CSECT object module can not be broken apart by LINKER so even if you are only using READ your module will still include POSN and WRITE also. Another thing to think about is that since READ and WRITE are not represented by a diskette file name, if your module does not use POSN, LINKER won't be able to find READ or WRITE on the diskette. Luckily, OPEN uses POSN and you can't use READ or WRITE without first calling OPEN, so this is not a problem with this example.

SETTING UP OVERLAYS

Using overlays is a way to make a very large program fit into a small memory space. Ordinarily, a program that uses overlays consists of a load module called the "root segment" which is always in memory and two or more "overlay segment" load modules which are stored on diskette and take turns being loaded into the same area of memory. An example might be a data entry program. Such a program would probably begin with a menu of the functions it can do (initialize a file, add entries, update an entry, print entries). When the user of the program is doing one of these functions, there is no need to keep the others in memory so they are kept as overlays. Usually the menu part of the program and the variable data would be the root segment and each function would be an overlay. The amount of memory needed would be that occupied by the root segment plus that occupied by the largest of the overlays. This example is diagrammed below, showing the program's memory during the time the user is adding entries to his file:

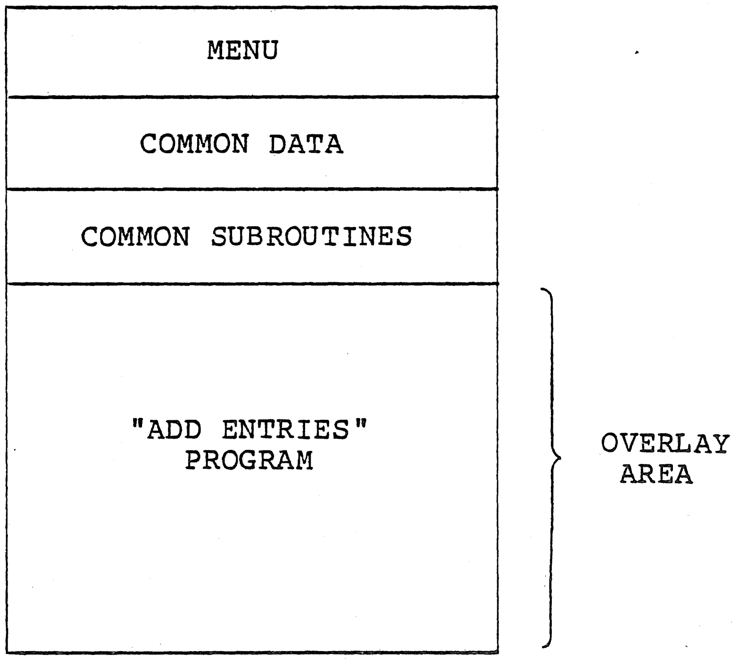


FIGURE 3.

By calling LKED (the functional part of the LINKER program) as a subroutine of the root segment, you can, during execution, load program segments into your "overlay area" as they are needed. In doing so, you can even connect subroutine references in the overlay module to subroutines in the root segment and vice versa. To explain how this is done, consider the following root segment as produced by LINKER:

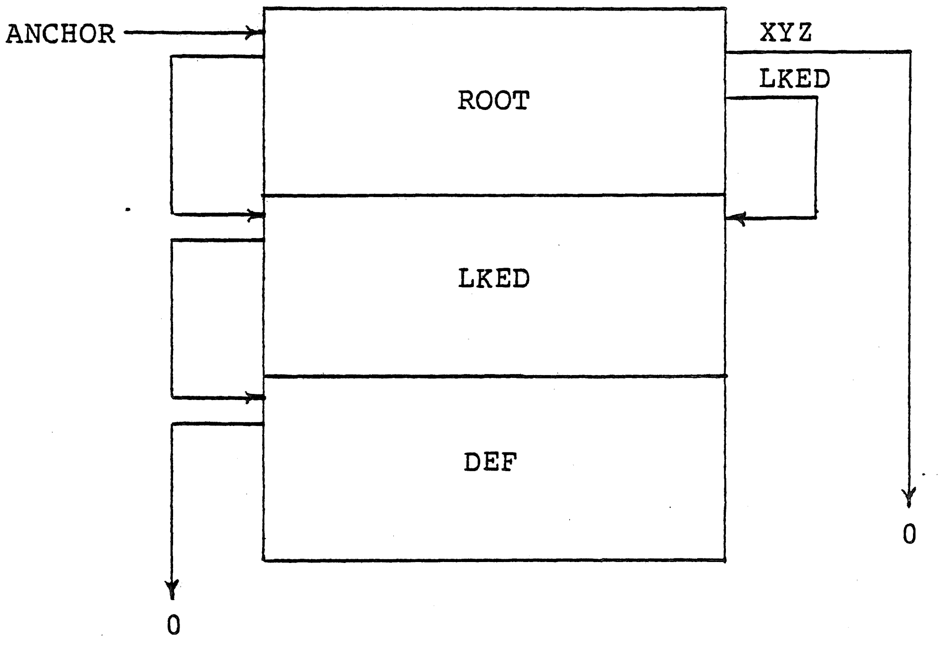


FIGURE 4.

In this diagram, the load module ROOT consists of three CSECTS, ROOT, LKED, and DEF. ROOT references LKED and XYZ. XYZ has been left unresolved (by omitting XYZ.OBJ from the diskette when ROOT was linked) since it will appear in the overlay segment. When LINKER builds a load module, it also links all the CSECTS it finds together in a chain which is pointed to by the anchor bytes (provided in the parmlist to LKED). Each time LKED is called, it first processes all CSECTS it finds on this chain before trying to load the requested module. By manipulating this chain of CSECTS you can control LKED's actions upon your modules. In the example, the anchor points to ROOT, ROOT points to LKED, LKED points to DEF, and DEF points to zeros (end of chain). These link bytes are in the module headers at offset +11 and +12 (Remember the two double bytes in the header? The first one is this pointer and the second is the use count). Normally when you call LKED you pass it a zeroed anchor. This means there are no previously loaded modules to be incorporated into the final module. In our case, however, we want the overlay module to be linked to the root segment so, when LKED is called to load the first overlay, we must pass it an anchor which points to the JMP at the beginning of ROOT. LKED is called to load OVLY1.OBJ in the memory following DEF and the following diagram shows the result.

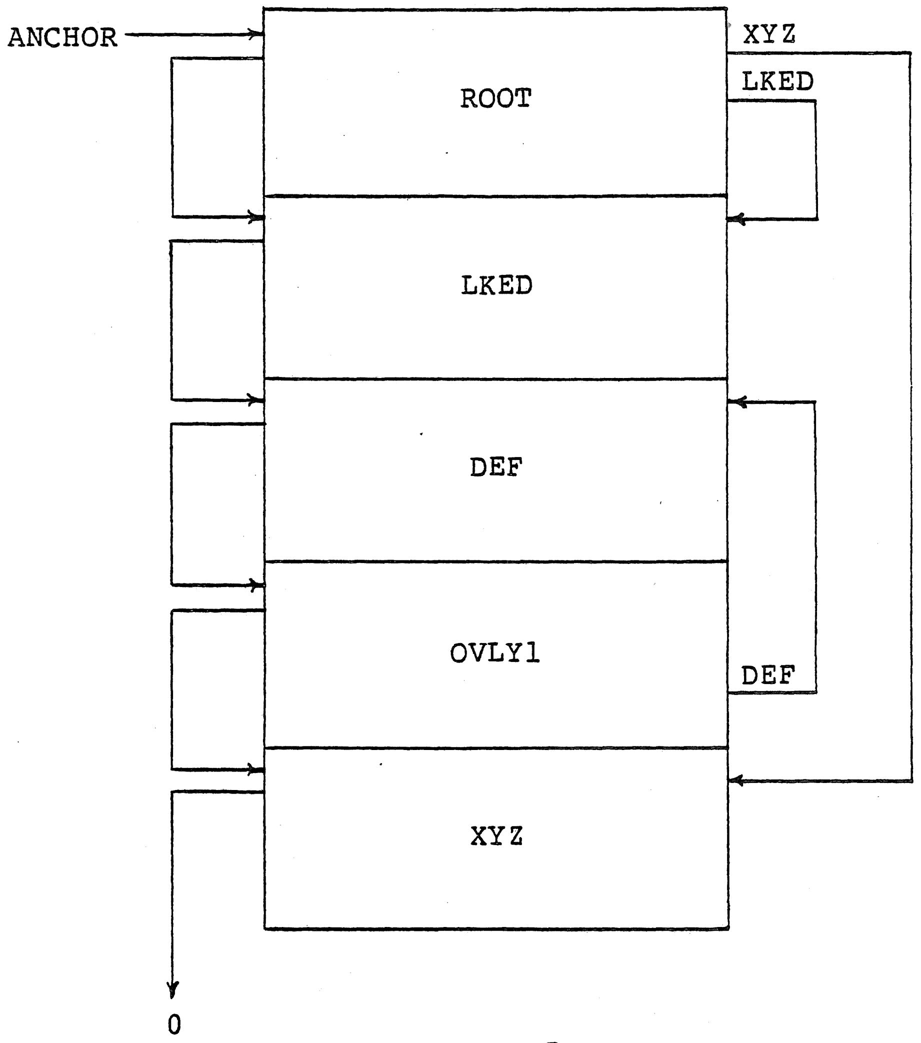


FIGURE 5.

Notice that ROOT's reference to XYZ has been resolved and OVLY1 calls DEF in the root segment. Also note that the anchor chain is longer now. When it is time to load another overlay, the anchor chain must be shortened back to its original length (by storing zeros in the link pointer in the DEF module header) and LKED is called to load OVLY2 into the same memory OVLY1 had occupied. If OVLY2 doesn't have an XYZ subroutine as part of it, ROOT should not attempt to call that CSECT and LKED must be called with the $40 flag set.

APPENDIX A - DEMO PROGRAM LISTING

```
                              ORG   $800
                    *
                    *        DEMO:  DEMONSTRATION OF USE OF
                    *               THE LINKAGE EDITOR AND THE
                    *               ASSOCIATED SUBROUTINE LIBRARY.
                    *
                    IN        EQU   $200               INPUT BUFFER
                    GET       EQU   $FD6A             MONITOR GET SUB
                    COUT      EQU   $FDED             MONITOR OUTPUT SUB
                    CEOS      EQU   $FC42             CLEAR TO END OF SCREEN
                    RETURN    EQU   $8D
                    PROMPT    EQU   $33
                    A1        EQU   $3C
                    A2        EQU   $3E
                    AUX       EQU   $54
                    AC        EQU   $50
                    HOME      EQU   $FC58
                    TEXT      EQU   $FB39
                    VTAB      EQU   $FB5B
                    *
                    *        MODULE HEADER
                    *
0800: 4C 11 08                JMP   DEMO               EPA
0803: C4 C5 CD                ASC   "DEMO      "      NAME
080B: 00 00                   HEX   0000               LINK
080D: 00 00                   HEX   0000               USECOUNT
                    *
080F: FF F1                   HEX   FFF1               INSTRUCTIONS
                    *
0811: 20 39 FB      DEMO      JSR   TEXT               CLEAR SCREEN
0814: 20 58 FC                JSR   HOME               TEXT MODE
0817: 20 51 09                JSR   PRINT
081A: DA                      HEX   DA
081B: C4 C5 CD                ASC   "DEMONSTRATION PROGRAM"
0830: 8D                      DW    RETURN
0831: 8D                      DW    RETURN
0832: 00                      HEX   00
                    *
0833: A9 03         DEMO1     LDA   #3                    ALWAYS AT TOP
0835: 20 5B FB                JSR   VTAB
0838: 20 42 FC                JSR   CEOS
083B: 20 51 09                JSR   PRINT
083E: DA                      HEX   DA
083F: C5 CE D4                ASC   "ENTER FIRST NUMBER"
0851: 8D                      DW    RETURN
0852: 00                      HEX   00
0853: A9 BF                   LDA   #"?
0855: 85 33                   STA   PROMPT           SET PROMPT FOR GET
```

```
0857: 20 6A FD                JSR  GET                   GET LINE OF INPUT
085A: 8A                      TXA  CHECK  LENGTH
085B: F0 D6                   BEQ  DEMO1
085D: AD 00 02                LDA  IN                      END?
0860: C9 C5                   CMP  #"E
0862: D0 03                   BNE  NEND                  NO
0864: 4C 72 09                JMP  EXIT                  YES, EXIT
0867: A9 00          NEND     LDA  #<IN
0869: 85 3C                   STA  A1
086B: A9 02                   LDA  #>IN
086D: 85 3D                   STA  A1+1
086F: 20 3B 09                JSR  ATOB                  CONVERT NUMBER
0872: 98                      TYA  ANYTHING?
0873: F0 BE                   BEQ  DEMO1                NO, START OVER
0875: A5 3E                   LDA  A2                      GET IT
0877: 85 50                   STA  AC
0879: A5 3F                   LDA  A2+1
087B: 85 51                   STA  AC+1
                     *
087D: A9 03          DEMO2    LDA  #3
087F: 20 5B FB                JSR  VTAB                  TAB TO SAME LINE
0882: 20 42 FC                JSR  CEOS
0885: 20 51 09                JSR  PRINT
0888: DA                      HEX  DA
0889: C5 CE D4                ASC  "ENTER SECOND NUMBER"
089C: 8D                      DW   RETURN
089D: 00                      HEX  00
089E: 20 6A FD                JSR  GET                    GET SECOND NUMBER
08A1: 8A                      TXA  NULL   LINE?
08A2: F0 D9                   BEQ  DEMO2                THEN ASK AGAIN
08A4: A9 00                   LDA  #<IN
08A6: 85 3C                   STA  A1                      PASS RESPONSE
08A8: A9 02                   LDA  #>IN
08AA: 85 3D                   STA  A1+1
08AC: 20 3B 09                JSR  ATOB
08AF: 98                      TYA  ANY    DIGITS FOUND?
08B0: F0 CB                   BEQ  DEMO2                NO, ASK AGAIN
08B2: A5 3E                   LDA  A2
08B4: 85 54                   STA  AUX                    SET FOR MULTIPLY
08B6: A5 3F                   LDA  A2+1
08B8: 85 55                   STA  AUX+1
                     *
08BA: A9 03                   LDA  #3
08BC: 20 5B FB                JSR  VTAB
08BF: 20 42 FC                JSR  CEOS                  POSITION FOR ANSWER
08C2: A4 50                   LDY  AC                       PASS FIRST NUMBER
08C4: A5 51                   LDA  AC+1
08C6: 20 F7 08                JSR  FORMAT               TO PRINT SUBROUTINE
08C9: A9 AA                   LDA  #"*
08CB: 20 ED FD                JSR  COUT                  TIMES...
08CE: A4 54                   LDY  AUX
08D0: A5 55                   LDA  AUX+1
08D2: 20 F7 08                JSR  FORMAT               SECOND NUMBER
08D5: A9 BD                   LDA  #"=
08D7: 20 ED FD                JSR  COUT                  EQUALS...
```

```
                 *
08DA: A9 00                LDA   #0                SET HI PART
08DC: 85 52                STA   AC+2             TO ZERO
08DE: 85 53                STA   AC+3
08E0: 20 5C 09             JSR   MULT             AUX*AC
                 *
08E3: A4 50                LDY   AC
08E5: A5 51                LDA   AC+1
08E7: 20 F7 08             JSR   FORMAT          PRINT ANSWER
08EA: A9 8D                LDA   #RETURN         NEW LINE
08EC: 20 ED FD             JSR   COUT
                 *
08EF: A9 1E                LDA   #30               WAIT 3 SECONDS
08F1: 20 46 09             JSR   DELAY
08F4: 4C 33 08             JMP   DEMO1            THEN START ALL OVER
                 *
                 *        FORMAT: PRINT A BINARY NUMBER
                 *
08F7: A2 00      FORMAT    LDX   #<IN             WHERE TO PUT OUTPUT
08F9: 86 3C                STX   A1
08FB: A2 02                LDX   #>IN
08FD: 86 3D                STX   A1+1
08FF: A2 A0                LDX   #"                FILL WITH BLANKS
0901: 20 30 09             JSR   BTOA             Y,A ALREADY LOADED
0904: A2 00                LDX   #0                START OUTPUT LOOP
0906: BD 00 02   FORMLP    LDA   IN,X             GET A CHAR
0909: C9 A0                CMP   #"                BLANK?
090B: F0 1B                BEQ   FORM1            YES, SKIP IT
090D: 20 ED FD             JSR   COUT
0910: 29 0F                AND   #$0F             CONVERT DIGIT
0912: A8                   TAY
0913: 8A                   TXA
0914: 48                   PHA   SAVE   XREG
0915: B9 7F 09             LDA   TONTAB,Y        GET TONE VALUE
0918: 85 3C                STA   A1                PASS AS FREQ
091A: A9 1E                LDA   #30
091C: 85 3D                STA   A1+1             DURATION FOR TONE
091E: 20 67 09             JSR   TONE
0921: A9 01                LDA   #1                WAIT 1/10 SECOND
0923: 20 46 09             JSR   DELAY
0926: 68                   PLA
0927: AA                   TAX   RESTORE XREG
0928: E8         FORM1     INX
0929: E0 05                CPX   #5                END?
092B: 90 D9                BCC   FORMLP
092D: 60                   RTS
                 *
                 *        GLOBALS
                 *
092E: FF F3                HEX   FFF3
                 *
0930: 4C 00 00   BTOA      JMP   $0
0933: C2 D4 CF             ASC   "BTOA      "
093B: 4C 00 00   ATOB      JMP   $0
093E: C1 D4 CF             ASC   "ATOB      "
```

APPENDIX B - PROBLEMS OR QUESTIONS

We hope you find LINKER a useful and reliable product. It has been carefully designed and tested but with every product there is always the possibility that there are minor or esoteric bugs. Naturally, we want to keep LINKER as bug free as possible, so, should you uncover one, please fill out the form below and mail it to us. We will look into the problem you are having and get back to you with a fix if at all possible. If you just have a question on the use of LINKER or on any of its subroutihes, feel free to send the form in with your questions and we will try to answer them for you. Should we make any significant changes to LINKER or its subroutine library, we will issue a new version of the program. To insure that you are kept abreast of such updates, please mail the enclosed postage paid postcard with you name and address and the word LINKER printed on it.

LINKER PROBLEM REPORT

NAME:

ADDRESS:

MACHINE MEMORY SIZE: K NUMBER OF DISK DRIVES:

OTHER DEVICES:

VERSION OF DOS BEING USED:

PLEASE DESCRIBE YOUR PROBLEM/QUESTION BELOW:

MAIL THIS COMPLETED FORM TO

DON WORTH
THE SOFTWARE FACTORY
PO BOX 904
CHATSWORTH, CA 91311

IF POSSIBLE, INCLUDE A PRINTER LISTING AND/OR A DISKETTE CONTAINING YOUR FAILING PROGRAM.

APPENDIX C - OBJECT MODULE STRUCTURE SUMMARY

```
*
*       MODULE HEADER
*
        JMP     NAME                JUMP TO ENTRY POINT
        ASC     "NAME    "          GLOBAL NAME OF CSECT
        HEX     0000                LINK VALUE
        HEX     0000                USE COUNT
*
*       INSTRUCTIONS SEGMENT
*
        HEX     FFF1
*
NAME    .
        JSR     ASUB
        .                           INSTRUCTIONS
        JSR     ZSUB
        .
*
*       DATA SEGMENT
*
        HEX     FFF4
*
        .
        .                           DATA
        .
*
*       ADDRESS CONSTANTS SEGMENT
*
        HEX     FFF2
*
APTR    DA      A                   FIRST ADCON
        .
        .
        .
ZPTR    DA      Z                   LAST ADCON
*
*       GLOBALS SEGMENT
*
        HEX     FFF3
*
ASUB    JMP     $0                  FIRST GLOBAL REFERENCE
        ASC     "ASUB    "
        .
        .
        .
ZSUB    JMP     $0                  LAST GLOBAL REFERENCE
        ASC     "ZSUB    "
*
*       END OF CSECT
*
        HEX     FFF0
```