LC/PRO-LC

Let me start out this issue's column on LC with a reminder. If you are still using LC version 1.0, then you had better send in your disk for an update to LC version 1.1. This update offer is still at no charge to you UNTIL DECEMBER 31, 1983. Beginning January 1, 1984, it will cost you $5 to update your LC disk from 1.0 to 1.1. Patches to bring your EDAS 4.1 diskette up to date for version 4.1 were included in the last issue of NOTES and are continued in this issue of NOTES. If you want us to update your EDAS, there is a $5 charge.

MISOSYS began shipments of the PRO-LC (TRSDOS/LDOS 6.x compatible release of LC 1.1) during October 1983. I am proud that this release is nearly 100% C-source code compatible to the Model I/III version in its support. To my recollection, the only difference in source code function interface is the "ploc(address)" function which had to be different in order to support the non-resident video of LDOS 6.x type systems. The PRO-LC release provides for the user-specification of the stack placement: either high or low memory. This gives the capability of running LC-generated programs with the DOS spooler active in external memory.

Now with the compiler available on the TRS-80 Models I and III, and computers running TRSDOS 6.x or LDOS 6.x, it becomes more important to stress the concepts of portability. All of you C programmers should already be aware that C is a portable language. As long as you keep machine dependent code out of your source, you can easily port a program from one computer to another. All you need is a compatible compiler.

In testing out the PRO-LC implementation, I tried to deal with compiling the programs in the LC interest group library. Unfortunately, most programs were machine dependent. Some used floating point which wasn't yet available with PRO-LC. Some programs were adapted with minor changes. However, let me point out that you really ought to pay more attention to keeping portability over slight variations of machine dependence. As a for-instance, I noticed many C-source programs with the statement:

```
#define CLS (0x1c9)();
```

Now it is a correct statement that programmers have used to clear the video screen since it defines a function which essentially "calls" the Model I or III ROM clear-screen routine. But what happens if you don't have a ROM? Such a program needs to be changed to compile under PRO-LC. It is good that the machine dependent "piece" was implemented as a "#define" which could be easily edited; however, wouldn't it be better to have used something like:

```
#define CLS puts("\x1c\x1f");
```

or include a function, cls(), which does the 'puts("\1c\1f")'? I am sure that many other examples can be illustrated; however, I hope that my point is clear without burdening you with excessive text. As LC is populated onto more machines, the more portable you keep your code, the better off you will be.

Now let me insert an errata to the PRO-LC manual [and the current printing of the LC manual] concerning the call() function. The second

paragraph of call() on page 4-36 should be: "On SuperVisor Call accessible systems, an "address" value of less than 256 will be interpreted as an SVC reference in lieu of a CALL address. The call() function will then use the "address" as the SVC number and ignore regs[0]."

It is important for LC/PRO-LC users to realize that there is an active LC Interest Group. This group was formed in 1982 and is chaired by Earl C. Terwilliger, of Akron Ohio. Earl's dedication and support of the "C"ers has resulted in a very useful focal point for the novice, intermediate, and advanced C-programmer. Earl sums up in his own words, the following synopsis of the group he started.

"An LC interest group (LCIG) has been formed as a hobby by Earl Terwilliger. The group can be used to help new users learn LC, to provide useful programs/functions for the more experienced 'C' programmer and to provide any member with some useful/functional 'C' programs. It provides a central place to collect and distribute LC source programs and related files contributed by its members. The membership fee is $5. The group serves as a non-profit vehicle to share ideas and programs. Programs submitted by members are kept on diskettes. There are now 4 diskettes (CSAMPLER, LCIG8301, LCIG8302, and LCIG8303) [I wouldn't be surprised if by now, Earl has collected material sufficient for a fifth diskette]. A copy of any of the standard LCIG diskettes will be sent to any member for $5 per diskette or $2 per diskette if the member sends in their own media. Members are also encouraged to contribute programs by sending them on a diskette to the group. The address for "The LC Interest Group" is as follows:

The LC Interest Group
c/o Earl C. Terwilliger Jr.
647 North Hawkins Ave.
Akron, Ohio 44313
Phone (216) 836-7389"

Let me go through some of the customer queries received concerning LC that may help you in avoiding problems. The first was submitted by James Campbell, of Aloha, OR. I appreciated the depth with which James explored the problem that he was having. Because of this, I would like to share his letter. James wrote: "I have to tell you how much I enjoy using Elsie, although I am new to the product. I think I have found a bug. You will have to bear with my explanation, since C is not my native language and Z-80 assembler is something I do not know well.

My problem started when I keyed in your sample from Appendix D-9, to show the use of call() [note: this sample program first appeared in the 2nd printing of the LC manual. The example appeared in the LC errata included with the first printing of the manual]. The program worked to a point, telling me the current status of my drives--through drive 8 and climbing. I cut the power about drive 156 or so. Obviously the test was not working [specifically, the test is 'for (d=0;d<8;++d)'].

After checking my text several times, I got an assembler listing and went into DEBUG. I soon determined that the value being used for the drive check was not the same one being used for the test against the value 8. I think the problem is the translation of the inequality test 'd<8'. To get back the current value of D$, the instruction at 01410 should be $GETW D$,

not LD HL,D$. I can say that with some confidence, because I used DEBUG to change the value at 5231 from 210200 to 2A6A5F and the program executed successfully.

Then again, I may be wrong. In any event, I hope the enclosed documents will help you determine where my problem lies."

James included printouts of his source and the assembler listing. He was right about the need for "$GETW LABEL", which is used to fetch a static integer. Assembler code of the form, "LD HL,LABEL" is used to obtain the pointer to a function. So did LC go wrong? My first gut feeling is that it must be a source code error - don't assume a bug in LC. Armed with this hypothesis, I easily discovered the bug. In the "for" statement mentioned above, the inequality was entered as "D<8" instead of "d<8". Since LC is case sensitive, "D" and "d" are two different variables. However, since the assembler requires that all variables be in upper case, LC converts its symbols on output to upper case. Therefore, if you look at an assembler listing, you cannot tell the difference.

The significance of James's mistype goes beyond just the simple error. It sheds light on the importance of keeping all variables in lower case. Recognizing that the compiler is case sensitive, you could have two different variables spelled the same but different cases! At least as far as the compiler is concerned - but not the assembler. It has always been recommended that "defines" use upper case.

I felt it interesting to explore the reason why the compiler generated the "LD HL,D$" instead of the "$GETW D$" macro. Kernighan & Ritchie state on page 209, "There are only two things that can be done with a function: call it, or take its address. If the name of a function appears in an expression not in the function name position of a call, a pointer to the function is generated. ... [the function] must be declared explicitly in the calling routine since its appearance ... was not followed by (." LC doesn't require the explicit declaration; therefore, an identifier which has not been defined, is interpreted as a function reference whether it is followed by a left parenthesis or not. It follows from K&R on page 68 that, "If a name which has not been previously declared occurs in an expression [omitting 'and is followed by a left parenthesis'], it is declared by context to be a function name. Furthermore, by default, the function is assumed to return an int." The LC manual states that, "A function declared within another function body is assumed to have a storage class of external. The compiler regards the declaration as if an 'extern' statement preceded it."

What we then have is an assumed declaration of the form: "extern D();". When "d" was declared, LC converts this to "d$" to avoid assembler restrictions on labels. Since "d" was declared as an "int", LC referenced the variable via the $GETW and $PUTW macros. The assembler requires upper case labels so LC passes "D$" to the /ASM file. The macro argument converts the "D$" to an offset from zero and adds the data origin. When "D" was referenced, it was converted to "D$" to avoid assembler restrictions. Since "D$" was interpreted as an external function name, LC referenced the variable strictly by its name. The assembler did not reject this as an undefined symbol error since "D$" was actually defined based on the original "d". However, "D$" by itself was only a relative offset from zero. If you examine the original C declaration of "int rc,d;", you will understand that the value

of "RC$" should be zero while the value of "D$" should be two. That's exactly what occurred in James's case. The data origin was 5F68 resulting in 5F6A when "d$" was referenced (D$+$$STORG) but only 2 when referencing "D$".

The next problem to discuss was submmitted by P. Bailey, of Derby, England. You should be able to gather the queries raised by Mr. Bailey from my response to him. It went as follows: The "problems" you addressed in your letter postmarked May 23rd were analyzed by our "C" experts. You may be surprised at their conclusions.

The first problem, that of being unable to effect a tab using the "\t" or "\x09" escape sequences, is one of misunderstanding. You actually did insert a tab character in the string. What you may fail to realize is that there is nothing in the output stream that will EXPAND the tab. You can rest assured that the tab was inserted by redirecting the output to a disk file and then listing the file in hex (or in ASCII with the TAB=ON parameter).

In order to assure ourselves of the above, the following program was compiled and run:

```
main()
{ puts("This is a tab ->\t<- character\n");}
```

The compiled and assembled tabtest/ccc program performed as expected. [as an aside, under the LDOS/TRSDOS-type systems, the video driver does not expand the TAB character. This can be accomplished by the LIST command, however, when listing a file]

Your second problem represented another degree of confusion. The illustrated program fragment you supplied can be reduced to the following:

```
main()
{ loop:
        c=getchar();
        goto loop;
}
```

Notice that you have no test for end-of-file in the getchar() function invocation. It is true that the program will pause awaiting a character upon reaching the getchar() function. Even though getchar() will be receiving its input from the keyboard (unless redirected), it nevertheless is still possible to transmit an EOF. This is performed by depressing the BREAK key. Once the BREAK key is depressed, getchar() will continue to return EOF regardless of any further keyboard operation. One possible recoding of this loop is as follows:

```
main()
{ loop:
        if ((c=getchar()) != EOF )
        goto loop;
        else exit;
}
```

The second problem arose from a need to detect an ABORT condition to return to a primary menu prompt. This was clarified in a subsequent letter

which follows:

Without going into a great explanation, your use of BREAK to escape into a menu in a "word processor" type of C application can be accomplished easily. Under C, one needs to be able to maintain device independence; thus, the BREAK key was selected to provide an end-of-file. Considering in your application that the primary means of input would be by keyboard, your need to regain keyboard control after a getchar() and BREAK-EOF is quite important. An elegantly simple solution would be to re-open the standard input device (stdin) with fopen() when the EOF indication is encountered. Then have the C program go to the menu mode.

The reopening of stdin also serves to support the redirection of standard input at the command line to start the input from a disk file and have it automatically switch to the keyboard upon reaching the EOF from the disk file. In any event, all you then need is a command to exit your program.

If our NOTES readers have any other solutions, perhaps they may be submitted.

R. D. Greet, of Lockleys, Australia reports that, "A book that has turned up locally which I would recommend for first reading on C is THE C PRIMER by Hancock & Krieger (Byte Books). This gem made clear in my mind aspects of C that was causing difficulty."

Concerning the issue of books, I have been advised that Paul Chirlian's book, "Introduction to C", will soon be published by Matrix Publishing (Matrix is closely associated with Dilithium Press). Paul, as you may know, is the author of many books concerning microcomputers including the book on Microsoft FORTRAN.

I thought I would relate the findings that Truman Krumholz discovered concerning the "Sieve" program listed in the last issue of NOTES. He writes:

"In Issue 1 of NOTES FROM MISOSYS is the sieve program in C which was taken from the January BYTE. As the program appears, my TRS-80 Model I executes the program in 117 seconds. This agrees with your time taking into consideration different clock speeds. However, if the line containing the main statement and the integer declaration line are reversed in order, the program runs on my Model I in 68 seconds. This is a significant improvement in speed. Being a beginner with C, I tried this because this is the order in which the statements appear in the article in BYTE. [actually, Truman is in error here. The statements are ordered as he states but only in the PASCAL version of the sieve program. The C version orders the statements as shown in NOTES].

For your interest, I also have a home-brewed LNW-80 Model I. The only part of this machine that is LNW are the boards. My machine runs at three speeds, 1.77 mHz, 4mHz, and 5.33 mHz. It also has the capability of switching out the ROM and the ROM wait states and substituting RAM. Running this program on the LNW at 5.33 mHz, it executes in 22 seconds. Both the improved program and the version using multiplication as listed in January BYTE run on my LNW in 18 seconds at 5.33 mHz.

I am enjoying LC very much, but I wish I had more examples to look at. This is the best way for me to learn a language. I copy programs then modify them to see what happens. By the way, my best versions of FORTH, writtern by a local fellow here, ran the program listed in BYTE in 54 seconds (clock speed at 5.33 mHz). I clocked MMSForth at 188 seconds on my TRS-80 Model I and 62 seconds on the LNW at 5.33 mHz."

The reason why Truman detected a significant improvement in execution speed was due to the pronounced effect of re-ordering the declaration of variables. The variables i, prime, k, count, and iter are local to main() when defined within main(). This means that they are referenced from the stack pointer. If their declaration precedes main(), they are global static variables having a fixed address. In order to access an integer that is local, the following code must be performed:

```
LD   HL,OFFSET      ;OFFSET is variable based on current SP
ADD  HL,SP          ;Calculate actual address of storage
CALL @GINT          ;Obtain value
```

The @GINT subroutine performs: \LD A,(HL)\INC HL\LD H,(HL)\LD L,A\RET\. The access of a static global requires only, "LD HL,($$STORG+#NAME)", with the argument referenced at assembly time. If we evaluate t states, I calculate that the local takes 72 t-states versus 16 for the global. At 2 mHz, a t-state is 0.5 microseconds. Thus the additional 56 t-states to access a local take 28 microseconds. That's PER ACCESS. If your program has hundreds of thousands of accesses of its variables, then the time difference is significant.

The utility of local variables arises when dealing with recursive functions. K&R state that "when a function calls itself recursively, each invocation gets a fresh set of all the automatic variables, quite independent of the previous set." This is not true if only statics are used. There are some very definite advantages for using automatic variables - but the advantage is not speed. FORTRAN, for example, has no such thing as an automatic variable. All variable addresses are fixed at compile (and link) time. The same thing is true of compiled BASICs. C gives you a choice. Thus if you are interested in speed, it may be to your advantage to define your variables outside of main().

Darach Foskett of Beecher Falls, VT forwarded an interesting problem which had me going around for awhile. It originates from the text in K&R on page 207 and again on page 86 which concerns token replacement associated with #define and macro substitutions. I must admit, I had to reread K&R a number of times on this point to understand the discussion. As Darach stated, his was not so much an LC question since LC does not support macro substitutions but a general C question. Herewith my response:

The problem stems from a possible misinterpretation of the K&R statement that, "text inside a string or character constant is not subject to replacement". This does not mean that 'identifier arguments' within the #define macro are not subject to replacement but rather if an 'identifier' happens to appear in the program, then it will not be substituted. K&R provide a very brief example of this on page 86 where they illustrate that a '#define YES whatever' will not be substituted for the 'YES' within the C statement, 'printf("YES")'.

On page 9 of THE C PUZZLE BOOK, it uses the following #define statement:

```
#define PRINT(int) printf("int = %d\n",int)
```

The character sequence "int" appears in the #define three times. The first is as an argument identifier. The second is within the printf() string argument. The third happens to be the second argument of the printf(). According to my interpretation of K&R, all three appearances of 'int' will be replaced by the token string. Thus, a statement such as 'PRINT (x < y ? y : x);' will find the 'PRINT' substituted according to the #define to read as follows:

```
printf(" x < y ? y : x  = %d\n", x < y ? y : x );
```

This statement computes to the result printed in PUZZLE. If I had coded a statement, 'printf("This is a PRINT line!\n");', the 'PRINT' within my statement would not be subject to any replacement by the PRINT defined by the #define pre-processor statement.

The answer to your question concerning the use of "int" in the #define is that the formal macro arguments are totally independent of all other parts of the source file. You could have 15 #defines - all using the same dummy parameter name. It is sometimes confusing to look at a program that repeats the use of the same name because one sometimes forgets which edition of the name one is referring to. Such is the case with the reuse of identifiers when you open a new block (i.e. { code }). Although you can repeat the name for a new local variable, it can be confusing to the programmer.

The next topic is one that I am sure the Model I/III LC users will love to read. Let me give you a little background. All of the LC development work done at the MISOSYS office has been on hard drives. In the past, a Lobo 5-meg drive as well as a Radio Shack 5-meg drive have been used. These drives are very quiet in seeking. In fact, you really cannot hear the movement of the head if you are listening in a normal level of room noise. Even when LC was invoked from floppies, the Tandon drives in use were quiet seekers. In fact, the only noisy drives I recollect were the MPI drives.

In any event, when floppies were used, I always was aware that there was a perceptively long time when a program loaded before it seemed to do anything. This effect was just in the back of my mind and the quietness of the various drives in use masked the "excessive" seek activity actually taking place.

I presently use a VR Data package which uses Syquest drives with buffered seek. Although the fan in this cabinet is quite noisy, the drive itself exhibits a kind of "bubbling" sound when the drive is rapidly stepped over distances. Since the PRO-LC package was developed using the Model 4 and VR Data hard drive combination, I became noticably aware of the great number of seeks occuring not only when LC first started, but programs compiled with LC.

Now aware that something bothered me, I decided to look into the matter. It did not take too long to discover that the culprit was the effect of opening the three standard files. Not that opening in itself was the root cause, but rather that @FSPEC followed by @OPEN (or @INIT) was successively

invoked for each of the three standard files. Now @FSPEC is in SYS1/SYS whereas @OPEN and @INIT are in SYS2/SYS. Thus, executing a loop that opened the three files successively resulted in the access and loading of SYS1\SYS2\SYS1\SYS2\SYS1\SYS2. This means directory access as well. When a program first executes, SYS2 is the resident overlay since the DOS just had to OPEN the program file for loading.

After discussing the "problem" with Jim Frimmel (I also talked it over with Karl Hessinger), I decided to add an internal FSPEC-type routine to LC. This was introduced first into PRO-LC at release time. Because I feel that the improvement in apparent (and actual) execution speed is significant, I wanted to apply the same technique to LC. Thus, I have the following procedure to recommend. If we do an LC 1.2 release, this will become a part of it. For now, you can easily achieve the same improvement by adopting the following procedure:

```
. LC SPEED IMPROVEMENT
. This patch modifies LC/CMD so as to improve the
. speed at which the compiler initializes the
. standard files.
. PATCH LC/CMD.LC USING ...
D6A,08=2F BE; WAS 1C 44
D6E,08=E5 D5 7E FE 21 38 0F FE 61 38 06 FE 7B 30 02 EE
D6E,1B=20 12 23 13 18 EC 3E 03 12 D1 E1 C9; WERE ZEROES
. End of patch
.
. This patch modifies LC/LIB so as to improve the
. speed at which compiled programs initialize the
. standard files.
. PATCH LC/LIB.LC USING ...
D2C,BA="@SPEC"; WAS "441CH"
. End of patch
.
. The following routine must be added to LC/ASM
. and should be placed to follow the statement:
.     *SEARCH LC/LIB
. This can be accomplished by loading LC/ASM,
. inserting this routine, then the updated file
. is saved with: W LC.LC
.
        IFDEF   FOPEN
*M
@SPEC   PUSH    HL              ;Save registers
        PUSH    DE
$?1     LD      A,(HL)          ;Exit on space or less
        CP      33
        JR      C,$?3
        CP      97              ;Convert l/c to U/C
        JR      C,$?2
        CP      123
        JR      NC,$?2
        XOR     32
$?2     LD      (DE),A
        INC     HL              ;Bump pointers
        INC     DE
```

```
            JR      $?1             ;Loop until exit char
    $?3     LD      A,3             ;Terminate with ETX
            LD      (DE),A
            POP     DE              ;Restore regs
            POP     HL
            RET
            ENDIF
```

The next thing to offer is a program submitted by Karl Hessinger, of College Park, MD. Karl wrote a conversion program which aids in the conversion of Model I/III BASIC programs to Model 4 BASIC. Not only is this program useful in what it does, it is one more source program to help you learn C. Although it is long, it may prove beneficial, Karl's program works from compressed BASIC program files rather than ASCII program files.

```
#include stdio/csh
#option redirect OFF
FILE *fp1, *fp2;
char *keyword[128];
int remflag,strflag;
main(argc,argv)
int argc, *argv;
{  char c;
   if ( argc != 3 )
      abort("\n** Parameter error **\n");
   puts("\x1c\x1f");
   fp1 = getfile(*++argv,'r');
   fp2 = getfile(*++argv,'w');
   if ((c=getc(fp1)) != '\xff')
      abort("Not a compressed BASIC file!");
   setup();
   while ( !end() )
      {  linenumber();
         remflag = strflag = FALSE;
         while ((c=getc(fp1)) != '\0' )
            if ( remflag || strflag || c != ' ' )
               {  if ( c > 127 && !strflag )
                     {  if (token(c))
                           goto lineend;
                     }
                  else
                     out(c);
                  if ( c == '"' )
                     strflag = strflag ? FALSE : TRUE;
               }
         lineend : if ( strflag )
            out('"');
         out('\n');
      }
   fclose(fp1);
   fclose(fp2);
   exit(0);
}
linenumber()
{  int num;
   num = getc(fp1);
   num += getc(fp1)*256;
   fprintf(fp2,"%d ",num);
   printf("%d ",num);
   return(0);
}
end()
{  int flag; char c;
   if ((c = getc(fp1)) == EOF)
      return TRUE;
   flag = c;
   if ((c = getc(fp1)) == EOF)
      return TRUE;
   flag += c * 256;
   if ( flag == 0 )
      return TRUE;
   else
      return FALSE;
}
token(val)
char val;
{  char c; int pos,flag;
   fputs(keyword[val-128],fp2);
   puts(keyword[val-128]);
   if ( val == 184 )      /* Strip value after CLEAR */
      {  while ((c=getc(fp1)) != ':' && c != '\0' )
            {}
         if (c=='\0')
            return(TRUE);
         else
            {  out(c);
               return(FALSE);
            }
      }
   if ( val == 178 )      /* PRINT change PRINT@ values */
      {  while ((c=getc(fp1)) == ' ')
            {}
         if ( c == '@' )
            {  out(c);
               pos = 0;
               flag = FALSE;
               while (isdigit(c=getc(fp1)))
                  {  flag = TRUE;
                     pos *= 10;
                     pos += c - '0';
                  }
               if (flag)
                  {  fprintf(fp2,"(%d,%d)",pos/64,pos%64);
                     printf("(%d,%d)",pos/64,pos%64);
                  }
            }
         if ( c == '\0' )
            return(TRUE);
         if ( c > 127 )
            return(token(c));
```

```
        else
           {  if ( c=='"' )
                 strflag = strflag ? FALSE : TRUE;
              out(c);
           }
      }
   if ( val == 147 || val == 251 )
      remflag = TRUE;
   return(FALSE);
}
getfile(fname,mode)
char *fname, mode;
{   char *fp;
    if (mode == 'r')
    {   if ((fp=fopen(fname,"r")) == NULL)
        {   printf("Open error - %-20s\n",fname);
            exit(1);
        }
        else return fp;
    }
    else if (mode == 'w')
    {   if ((fp=fopen(fname,"w")) == NULL)
        {   printf("Open error - %-20s\n",fname);
            exit(1);
        }
        else return fp;
    }
}
out(c)
char c;
{  putchar(c);
   if (c != putc(c,fp2))
      {  puts("File I/O error!\n");
         fclose(fp1);
         fclose(fp2);
         exit(1);
      }
   return(0);
}
setup()
{  keyword[0] = "END ";          keyword[1] = "FOR ";
   keyword[2] = "RESET";         keyword[3] = "SET";
   keyword[4] = "CLS ";          keyword[5] = "CMD ";
   keyword[6] = "RANDOM ";       keyword[7] = "NEXT ";
   keyword[8] = "DATA ";         keyword[9] = "INPUT ";
   keyword[10] = "DIM ";         keyword[11] = "READ ";
   keyword[12] = "LET ";         keyword[13] = " GOTO ";
   keyword[14] = "RUN ";         keyword[15] = "IF ";
   keyword[16] = "RESTORE ";     keyword[17] = " GOSUB ";
   keyword[18] = "RETURN ";      keyword[19] = "REM ";
   keyword[20] = "STOP ";        keyword[21] = " ELSE ";
   keyword[22] = "TRON ";        keyword[23] = "TROFF ";
   keyword[24] = "DEFSTR ";      keyword[25] = "DEFINT ";
   keyword[26] = "DEFSNG ";      keyword[27] = "DEFDBL ";
   keyword[28] = "LINE ";        keyword[29] = "EDIT ";
```



```
   keyword[30] = "ERROR ";       keyword[31] = "RESUME ";
   keyword[32] = "OUT ";         keyword[33] = "ON ";
   keyword[34] = "OPEN ";        keyword[35] = "FIELD ";
   keyword[36] = "GET ";         keyword[37] = "PUT ";
   keyword[38] = "CLOSE ";       keyword[39] = "LOAD ";
   keyword[40] = "MERGE ";       keyword[41] = "NAME ";
   keyword[42] = "KILL ";        keyword[43] = "LSET ";
   keyword[44] = "RSET ";        keyword[45] = "SAVE ";
   keyword[46] = "SYSTEM ";      keyword[47] = "LPRINT ";
   keyword[48] = "DEF ";         keyword[49] = "POKE ";
   keyword[50] = "PRINT ";       keyword[51] = "CONT ";
   keyword[52] = "LIST ";        keyword[53] = "LLIST ";
   keyword[54] = "DELETE ";      keyword[55] = "AUTO ";
   keyword[56] = "CLEAR ";       keyword[57] = "CLOAD ";
   keyword[58] = "CSAVE ";       keyword[59] = "NEW ";
   keyword[60] = "TAB(";         keyword[61] = " TO ";
   keyword[62] = "FN ";          keyword[63] = "USING ";
   keyword[64] = "VARPTR ";      keyword[65] = "USR ";
   keyword[66] = "ERL ";         keyword[67] = "ERR ";
   keyword[68] = " STRING$";     keyword[69] = " INSTR";
   keyword[70] = " POINT";       keyword[71] = " TIME$ ";
   keyword[72] = " MEM ";        keyword[73] = " INKEY$ ";
   keyword[74] = " THEN ";       keyword[75] = " NOT ";
   keyword[76] = " STEP ";       keyword[77] = "+";
   keyword[78] = "-";            keyword[79] = "*";
   keyword[80] = "/";            keyword[81] = "^";
   keyword[82] = " AND ";        keyword[83] = " OR ";
   keyword[84] = ">";            keyword[85] = "=";
   keyword[86] = "<";            keyword[87] = " SGN";
   keyword[88] = " INT";         keyword[89] = " ABS";
   keyword[90] = " FRE";         keyword[91] = " INP";
   keyword[92] = " POS";         keyword[93] = " SQR";
   keyword[94] = " RND";         keyword[95] = " LOG";
   keyword[96] = " EXP";         keyword[97] = " COS";
   keyword[98] = " SIN";         keyword[99] = " TAN";
   keyword[100] = " ATN";        keyword[101] = " PEEK";
   keyword[102] = " CVI";        keyword[103] = " CVS";
   keyword[104] = " CVD";        keyword[105] = " EOF";
   keyword[106] = " LOC";        keyword[107] = " LOF";
   keyword[108] = " MKI$";       keyword[109] = " MKS$";
   keyword[110] = " MKD$";       keyword[111] = " CINT";
   keyword[112] = " CSNG";       keyword[113] = " CDBL";
   keyword[114] = " FIX";        keyword[115] = " LEN";
   keyword[116] = " STR$";       keyword[117] = " VAL";
   keyword[118] = " ASC";        keyword[119] = " CHR$";
   keyword[120] = " LEFT$";      keyword[121] = " RIGHT$";
   keyword[122] = " MID$";       keyword[123] = " ' ";
   return;
}
abort(msg)
char *msg;
{  fputs(msg,stderr);
   putc(eol,stderr);
   exit(1);
}
```

Finally, I discussed this last idea with Jim Frimmel and he concurred in its appearance in NOTES. Although C is considered to be a portable language, situations arise whereby it would be useful to be able to write C-source in conditional blocks - much like the assembler permits conditional pseudo-OPs.

There do exist conditional processes in a full C preprocessor. Although not implemented in Elsie, they are #if, #ifdef, #ifndef, #else, and #endif. Now all of these "conditional" operations are supported in EDAS. Since Elsie supports an "#option variable" specification which can define "@_variable" and set it to TRUE or FALSE (actually ON or OFF as used in the assembler), it becomes a simple matter to construct conditional blocks of code that are not conditional as far as the compiler is concerned but rather passed through to the assembler for evaluation of the conditional operations.

To achieve this in the C source, assume that you define an assembler symbol of "@_TEST" an set it to -1 via the C statement, "#option TEST ON". Next, a block of C-source code that is to be included in the resultant program would be surrounded by the statement pair,

```
#asm                          #asm
<TAB>IF @_TEST       and      <TAB>ENDIF
#endasm                       #endasm
```

If you don't want to include the block, specify "#option TEST OFF". Of course, there are variations such as using, "IFDEF @_TEST" and either specifying the #option or not.

This procedure may not be standard K&R; however, under the present implementation of Elsie, it does enable conditional C source.

Speaking of Jim Frimmel, it turns out that the previous item was not the final part of this issue's LC column. Jim has supplied us with the first of his "C'erious Subjects" columns. Although it may be a little advanced for the beginning LC user, if you all spend a little time to explore Jim's column, I am sure you will gain a better understanding of LC.

C'erious Subjects - by Jim Frimmel

Hiya!!! I'd like to discuss with you a few subjects that have been brought to my attention by LC users, and show you some techniques that may be useful to have in your bag of LC tricks.

The first thing is to show you a way to shorten the time required to assemble LC programs. The method shown is illustrated for Model I/III LC; however, it can be easily adapted to PRO-LC. The idea is to pre-assemble the library(s), so that you can use an equate file during each assembly. Then the assembled program is appended to a copy of the pre-assembled code, resulting in a runnable program. This technique saves me 30 seconds on every compile/assembly during development, on a MAX-80 with 8" disks. It will probably save one minute or so on model I/III's. For your final version of a program, use the regular LC/JCL file to get the most compact code possible.

Listing 1 shows the assembly file, LCLIB/ASM, that I used to pre-assemble the LC standard library. The REF macro references the symbol

following it, so that EDAS IV will retrieve it from the library. You can adapt this to pre-assemble any library module or combination of library modules. You don't have to pre-assemble all of a library, and you can pre-assemble ANY function, whether compiled LC functions or assembly language functions that you've written yourself. The LC standard library is trickier than other libraries, so I recommend that you pre-assemble the entire library, as shown here.

Note also that the options that are set in the LCLIB/ASM file, plus the defaults in LCMACS/ASM, will determine the options in effect in the pre-assembled library. Once assembled, the options cannot be changed, so if special options are needed, either pre-assemble a separate module with those options, or don't use this technique.

The transfer address of the pre-assembled /CMD file must be removed so that LC program /CMD files may be appended to it. I actually had to write a program to do this, since I couldn't figure out a way to do it using LDOS's extensive utilities. Why, you may ask, don't I just use Command File Utility to combine the two command files together? I found that it added 15 seconds (MAX-80 time) to the process, just because of all the friendly prompts in CMDFILE. The answers require quite a few JCL lines. Over in listing 2 is the program, XTRA/CCC, to strip the transfer address. Compile and assemble XTRA/CCC using the normal LC/JCL file.

At this point you are ready to actually generate the LCLIB/CMD and LCLIB/EQU files. The JCL file in listing 3 will do all but one step. If you have already typed in and compiled XTRA/CCC, and have typed in LCLIB/ASM, you're ready to DO LCLIB/JCL. The end products will be LCLIB/CMD, the pre-assembled and stripped version of the LC library, and LCLIB/EQU, an equate file that will define addresses of library functions when assembling LC programs for use with LCLIB/CMD. One comment about the LCLIB/JCL file: the I/O redirection in the XTRA command line is a little strange, in that standard input and standard output are the SAME FILE! The reason that I can get away with this is that XTRA writes out exactly what it reads in, minus four bytes. As a general rule, this can be done as long as the program writes out one byte for each one read, and shortens the file, or leaves it the same length. In any other case, results are usually unexpected.

Not all the symbols in the equate file just generated will be useful. No temporary labels, local labels, macro names, etc. need be included. Try and get the equate file looking like the one in listing 4, by going in under EDAS and deleting the unnecessary lines, and adding the few new ones.

We now have to create a modified version of LC/ASM to GET the equate file. The equate file must be included at the beginning of LC/ASM. The actual program being assembled must then be ORG'ed past the end of the pre-assembled code. Since the program may vary in size, the pre-assembled module should reside below the rest of the program. Listing 5 shows my QLC/ASM file ("Quick Levelance Cod").

We can now assemble a program using QLC/ASM. A new JCL file (listing 6) is needed if we're not going to issue the commands manually every time. In addition to the normal stuff LC/JCL does, it must also combine the two /CMD files into the final, runnable form. Note that the assembly's /CMD output file is sent to a temporary file. Then the copy and append are used to

generate the final /CMD file.

One thing that you should notice about the new QLC/ASM, and the pre-assembled standard library: The variables used by the library are placed in upper memory, at 0C000H. This is so that @PROGEND (which determines where alloc() and sbrk() reserve memory) will be defined above all code and variables. This imposes some limitations on the programmer. If your program is greater than about 24K bytes long, the symbol, $$STORG (in LCLIB/ASM), must be equated to a higher value to make room for the larger program. On the other hand, if your program uses large amounts of space reserved by alloc() and sbrk(), $$STORG may be decreased to make more room available to alloc() and free(). In any case, be sure that your program, once assembled, does not overlap the variables used by LCLIB, which begin at $$STORG (in LCLIB/ASM).

Well, it seems that some of us actually write programs with LC that are pushing the limits of the memory size in the TRS-80! I've been asked to explain how to share variables between programs, so I guess someone's program has gotten big enough to need splitting up. Well, this one is easy, since all you have to do is make sure that all programs have the same variables, at the same place. Also, if initialization must be done on variables, it must take place in the first program to be run, and the ZVAR option should be OFF for all subsequent programs. Also, any space gotten by calls to alloc() will no longer be available when the next program runs. This includes open files as well. All files must be re-opened when the next program runs.

To implement this, first collect all variables that are to be shared into a separate file that can be included into all programs. You don't have to put all variables in this file, just the ones to be shared. This file must be #include'd in each program BEFORE ANY OTHER VARIABLES ARE DECLARED! If you define other variables first, then the shared variables will be at the wrong place. Finally, compile and assemble all the programs and check the address given for $$STORG in each, which is the base of variables in the program. Choose the highest value, then add some to allow for growth. Now to make it really happen, make a copy of LC/ASM, and modify the line that equates $$STORG to the program counter, to set $$STORG to the value you have chosen. Reassemble all the programs using the new /ASM file, and you should be all set.

That's all for this issue. Look for details on using true overlays in the next issue of NOTES - if I can get to it. 'Till then, as Olivia says, "Let's get C'erious!".

LISTING 1

```
;   LCLIB/ASM  -  pre-assembly of LC standard library
;
*GET LCMACS
;
;***  If you wish any different options, set them here *
@_ZVAR DEFL    0               ; don't initialize var's
;***   Macro to reference a name  ***
REF    MACRO   #NAME
$TEMP  DEFL    #NAME
       ENDM
;
```



```
;*****   NOTE: This next statement determines how much
;*****         space there is for the program, and
;*****         divides the program (with var's) from
;*****         the areas for alloc() and free().
;*****         Change the value as needed.
@PROGEND       EQU     0C000H          ;24K for program
;***  REFerences to all LC standard library functions ***
;
       ORG     5200H
       REF     @AND
       REF     @ASL
       REF     @ASR
       REF     @COM
       REF     @DIV
       REF     @EQ
       REF     @GE
       REF     @GINT
       REF     @GO
       REF     @GT
       REF     @LE
       REF     @LT
       REF     @MULT
       REF     @NE
       REF     @NOT
       REF     @OR
       REF     @PINT
       REF     @SUB
       REF     @UDIV
       REF     @UGE
       REF     @UGT
       REF     @ULE
       REF     @ULT
       REF     @XOR
       REF     ATOI
       REF     EXIT
       REF     FGETS
       REF     FOPEN
       REF     FPRINTF
       REF     FPUTS
       REF     GETC
       REF     GETCHAR
       REF     GETS
       REF     ISALPHA
       REF     ISDIGIT
       REF     ISLOWER
       REF     ISUPPER
       REF     ITOA
       REF     ITOX
       REF     MOVE
       REF     PRINTF
       REF     PUTC
       REF     PUTCHAR
       REF     PUTS
       REF     STRCAT
       REF     STRCMP
```

```
        REF     STRCPY
        REF     STRLEN
        REF     TOLOWER
        REF     TOUPPER
        REF     XTOI
        REF     ZERO
;*** now get all the modules ***
$$STEMP         DEFL    0               ;init rel. storage
*SEARCH LC/LIB
$$STORG         EQU     $               ;storage areas here
@LCLIBEND       DEFL    $$STORG+$$STEMP ;label to ORG to
        END     402DH           ;back to LDOS if exec'ed
```

LISTING 2

```
#include stdio/csh
char buf[4], c;
main()
{   /* short filter to strip last 4 bytes from stdin */
    /* written to strip transfer address from load   */
    /* modules.      Jim Frimmel   11/2/83           */
    buf[0] = getchar();
    buf[1] = getchar();
    buf[2] = getchar();
    buf[3] = getchar();
    while ((c = getchar()) != EOF)
    {   putchar(buf[0]);
        move(buf+1,buf,3);
        buf[3] = c;
    }
}
```

LISTING 3

```
. LCLIB/JCL - Pre-assemble the LC standard library
.  Format: DO LCLIB (DRIVE=D)   DEFAULT: DRIVE=1
//IF -DRIVE
//ASSIGN DRIVE=1
//END
EDAS (JCL,ABORT)
L LCLIB
A LCLIB:#DRIVE#,LCLIB:#DRIVE# -SL -XR -NM -NC
B
XREF LCLIB:#DRIVE# (EQU)
XTRA <LCLIB/CMD:#DRIVE# >LCLIB/CMD:#DRIVE#
. Pre-assembly completed
.  Now edit the file LCLIB/EQU to prepare it for use
```

LISTING 4

```
@AND    EQU     60D4H
@AP     EQU     5C0BH
@APMOD  EQU     522CH
@ASL    EQU     60DBH
@ASR    EQU     60E3H
@CMP    EQU     613FH
@COM    EQU     60F8H
@DIV    EQU     618FH
@EQ     EQU     6119H
@ERRET  EQU     60B6H
@FCLS9  EQU     5A84H
@FCNT   EQU     0010H
@FTEND  EQU     5222H
@FVTBL  EQU     5202H
@GE     EQU     6133H
@GETFV  EQU     60BCH
@GINT   EQU     6107H
@GO     EQU     522DH
@GT     EQU     6125H
@GTFV1  EQU     60C4H
@LCLIBEND DEFL  6211H
@LE     EQU     612CH
@LOMEM  EQU     5200H
@LT     EQU     6139H
@MOD13  EQU     60B0H
@MULT   EQU     617BH
@NE     EQU     611FH
@NEG    EQU     60F3H
@NOT    EQU     60FFH
@OR     EQU     6112H
@PINT   EQU     610CH
@PUTHL  EQU     60CDH
@RDMOD  EQU     522AH
@SENAM  EQU     5226H
@SINAM  EQU     5222H
@SONAM  EQU     5226H
@SUB    EQU     60EEH
@UCMP   EQU     6169H
@UDIV   EQU     61AAH
@UGE    EQU     614FH
@ULE    EQU     6162H
@ULT    EQU     6155H
@WRMOD  EQU     522BH
@XOR    EQU     6174H
ALLOC   EQU     5ACBH
```



```
ATOI    EQU     5C37H
EXIT    EQU     5300H
FCLOSE  EQU     5A67H
FGETS   EQU     585AH
FOPEN   EQU     598EH
FPRINTF EQU     5378H
FPUTS   EQU     5807H
FREE    EQU     5B4EH
GETC    EQU     590AH
GETCHAR EQU     536FH
GETS    EQU     5346H
ISALPHA EQU     5FE3H
ISDIGIT EQU     5FFBH
ISLOWER EQU     600BH
ISUPPER EQU     601BH
ITOA    EQU     5CF1H
ITOX    EQU     5E2DH
MOVE    EQU     6084H
PRINTF  EQU     5325H
PUTC    EQU     5957H
PUTCHAR EQU     535DH
PUTS    EQU     5334H
SBRK    EQU     5C11H
STDERR  EQU     5206H
STDIN   EQU     5202H
STDOUT  EQU     5204H
STRCAT  EQU     602BH
STRCMP  EQU     603FH
STRCPY  EQU     6059H
STRLEN  EQU     6067H
TOLOWER EQU     5FCDH
TOUPPER EQU     5FB7H
XTOI    EQU     5EE9H
ZERO    EQU     6075H
@UGT    EQU     615BH
```

LISTING 5

```
;LC/ASM - Modifications
;*=*=*
@START EQU      5200H           ;Set to the code origin
*GET LCLIB/EQU                  ;Pre-assembled LC/LIB
        ORG     @LCLIBEND       ;Reference next available code address
;*=*=*
;       Continue with initial LC/ASM code
;*=*=*
        LD      HL,(4049H)      ;Remove "@START" label *****
;
;
        END     @LCLIBEND       ;Change entry point *****
```

LISTING 6

```
. QLC/JCL - Quick compile and assembly of LC programs
. format is:   do lc (file=<progname>,{drive=d},{show})
//if -drive
//assign drive=1
//end
//if show
lc #file#:#drive# +l
//else
lc #file#:#drive#
//end
edas (jcl,abort)
l qlc
c/cprogram/#file#
//if show
atempxxxx:#drive# -we
//else
atempxxxx:#drive# -nl
//end
b
copy lclib/cmd #file#:#drive#
append tempxxxx/cmd:#drive# #file#:#drive#
. completed compilation
```


PaDS/PRO-PaDS VERSION 1.0

In this issue of NOTES, the PaDS section will correct a few typos that appeared in the last issue. Next, there is one additional patch that must be applied to the utility. Following the patches, I will discuss some more uses of this interesting utility. First notice that there is a slight change in the name of the Partitioned Data Set utility. After saying the name of the utility hundreds of times, I felt that I, like others, were pronouncing it as P - D - S. That's three syllables - entirely too long for most folks. Perhaps some of you were already trying to pronounce PDS as a single-syllable "word". My guess would be that "pids" may have been the result. Not liking "pids" as the name of my product, I inserted a lower-case "a". Now the utility can be pronounced "pads". This change was installed at the time that the LDOS/TRSDOS 6.x compatible version, PRO-PaDS, was released. Enough of this smalltalk.

This next patch corrects a problem in the PaDS utility's PDS(APPEND) member when the file you are trying to append is a null file. The patch was listed in the last NOTES. The PATCH as printed was correct; however there was a typo in the procedure. In addition, somehow I did not get this patch into the MASTER DISK. I found that out when Robert J. Newton, of Houston, brought another PaDS problem to my attention. When I gave him a "D"irect patch, it turned out to be in the wrong location. My working copy of PaDS had the PDSA/FIX installed but the MASTER copy did not. Therefore, this patch was not installed until PaDS serial number 220473. Therefore, if your PaDS serial number is between 220001 and 220472, and you have not already installed the PDSA/FIX, please do so. Since this is an X-patch, you will have to patch PaDS by first copying APPEND to workspace, patching the separate file, killing and purging the APPEND in the PaDS, then appending the patched copy back to the PaDS. Do this with a BACKUP copy of PaDS. This is detailed as follows:

1. BUILD the PDSA/FIX file with:

```
. PDSA/FIX - PATCH TO THE APPEND MEMBER OF PDS (PaDS only!)
X'5499'=CD 3D 59
X'593D'=11 8A 58 2A A5 54 AF ED 42 C0 21 4D 59 C3 2E 56
        0A 41 70 70 65 6E 64 69 6E 67 20 66 69 6C 65 20
        69 73 20 6E 75 6C 6C 21 0D
. End of patch
```

2. Execute the command, "PDS(C) PDS(APPEND) APPEND"

3. Execute the command, "PATCH APPEND PDSA"

4. Execute the command, "PDS(K) PDS.PDS(APPEND)"

5. Execute the command, "PDS(P) PDS.PDS"

6. Execute the command, "!APPEND APPEND PDS.PDS"

Note specifically the exclamation point in step 6 - it is required! You now have corrected the PDS(APPEND) member.

The next patch MUST BE APPLIED. It corrects a problem in the PDS(PURGE) utility that trashes the file being purged WHEN A KILLED MEMBER ENDS ON A