8.00.00 PL/9 LIBRARIES TECHNICAL REFERENCE MANUAL

This section will describe, in detail, the various libraries we supply with PL/9. The purpose of this section is to provide you with sufficient information not only to assist you in using the library modules as they are supplied but also to give you some insight into how you may adapt them for your own specific purposes.

The architecture of PL/9 lends itself to the production of library modules which may contain the special functions you need for your program development. The

following sums up the capabilities of library procedures:

(1) They eliminate wasting paper when printing listings as the INCLUDEd library modules are not printed.

(2) They reduce the size of the text file resident in memory thereby allowing VERY large programs to be developed.

(3) Any procedure that returns a value can be treated just like any other global or local variable in evaluation expressions.

(4) Any procedure that is passed a value, performs some operation, and returns a value is considered to be a 'function' procedure and can be treated as though it were an actual part of the language itself.

(5) They enable programs that are designed to run under interrupts to be traced by the PL/9 tracer.

(6) They assist in developing programs for a wide variety of hardware environments.

(7) They assist in testing and debugging programs by providing a 'replacement' I/O module that looks to the system console for information rather than hardware devices.

One of the very basic things that you can use library files for is to assist in development of programs on your FLEX based system that the will ultimately run in a dedicated system that bears little, if any, resemblance in your development system.

You can, for example, maintain two program header files, one for the memory and I/O map of your development system and another for the same elements in the dedicated environment. The program can then be tested and debugged to a high confidence level in the development system using the first library file. Then, when you are ready to commit the program to ROM or download it into the dedicated system, you simply substitute the name of the second library file in INCLUDE statement.

This technique can be carried several stages further. Supposing that you are unable to integrate the target hardware into the memory map of the development system or are unable to simulate some of the I/O functions (an A-D or D-A converter for example) for one reason or another. In these instances the I/O operations of the program should be consigned to a library module. You can then build another library module with each I/O procedure given the same name but instead of going to the I/O device for information it prompts or sends data to the system console. The IOSUBS, BITIO, HEXIO, REALCON, REALIO and NUMCON libraries are particularly useful in these circumstances. You can then get on with writing the body of your program and have the capability to simulate most, if not all, of the I/O responses of the target hardware.

8.00.00 PL/9 LIBRARIES TECHNICAL REFERENCE MANUAL (continued)

Normally the library modules should be present on your system disk. You, can, however, speed up compilation and reduce 'head banging' in your disk drives if you put a copy of the library modules you are using on your work drive. If you decide to do this either specify the drive number explicitly in your program, e.g. INCLUDE 1.IOSUBS; or run the 'SETPL9' program to reconfigure your copy of PL/9 so it will automatically look to your work drive for its library modules.

The source files are printed just after the discussion of each library module. We have left in the column of HEX addresses on the left side of the listing to give you an idea of the size of each of the procedures within each library module.

N O T I C E

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                                                                  *
*  The compiler will produce code, and thereby use up memory,     *
*  for every procedure in the library WHETHER YOU USE IT OR NOT.  *
*                                                                  *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

You may remove virtually any procedure in any library module if you are not using it in order to reduce the code generated by a library module. When deleting procedures that you are not using near the top of each library file you should first look to see if they will be used in any subsequent procedures that you will be using. In these cases you must leave the low-level procedures in. Caution is also in order when deleting procedures from one library module that may be required by another library module. The compiler will always tell you if you have deleted a procedure needed by another procedure. These cautions are intended to save you some editing/typing effort!

N O T I C E

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                                                                  *
*  WE RESERVE THE RIGHT TO IMPROVE THE LIBRARY MODULES WE SUPPLY  *
*  ON DISK  WITH  THE  COMPILER.  THE  INITIAL  RELEASE  OF  THE  *
*  COMPILER  LIBRARIES  (V:4.00)  WILL MATCH THE DESCRIPTIONS IN  *
*  THIS MANUAL. IF THE VERSION NUMBER OF THE LIBRARY ON THE DISK  *
*  IS ANYTHING OTHER THAN V:4.00 THERE WILL BE SOME DIFFERENCES.  *
*                                                                  *
*  DON'T ASSUME THAT THE LIBRARIES ON DISK ARE THE SAME AS THOSE  *
*  IN THE MANUAL! CHECK FIRST!                                     *
*                                                                  *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

8.00.01 TRUE - FALSE - MEM DEFINITIONS

N O T I C E

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                                                                 *
*    TRUE, FALSE, and MEM are no longer part of the language.     *
*    If  you are upgrading from an earlier version these must     *
*    be declared explicitly in your program.                      *
*                                                                 *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

There is a small library module called 'TRUFALSE.DEF' that takes care of this declaration for you. All you have to do is INCLUDE it in your existing program.

TRUE - FALSE - MEM DEFINITIONS V:4.00

```
0000  0001 /*  TRUE - FALSE - MEM   DEFINITIONS  V:4.00  */
0000  0002
0000  0003
0000  0004 constant true=-1, false=0;
0000  0005
0000  0006 at $0000:byte mem;
```

EXTERNALS:

```
true                 FFFF
false                0000
mem                  0000   BYTE
```

8.01.00 IOSUBS.LIB

The first thing to note about this library are the contents of line 4 through 12. These CONSTANT declarations must not be duplicated in the main program, or, if you wish to have all constant declarations in the main program, you will have to remove these declarations from the library file.

8.01.01 MONITOR

This procedure jumps to your system monitor warm start entry point through the FLEX 'MONIT' vector. Its primary use will be to assist in the debugging programs as it has very little practical use in a functional program. Simply entering 'MONITOR;' in any procedure will cause the system monitor to be entered whenever that particular line of code is executed. Obviously you can use 'MONITOR' as part of an expression, e.g. IF ERROR THEN MONITOR;

8.01.02 WARMS

This procedure jumps to the FLEX warm start address at $CD03. Its primary use will be to provide a conditional exit from your program back into FLEX. If you write a program that is called from FLEX and expected to return to FLEX simply leaving the 'ENDPROC' off the last procedure in the program will automatically generate the code required. Simply entering 'WARMS;' in any procedure will cause the program to hand control over to FLEX whenever that particular line of code is executed. This function is identical to the 'FLEX' function in the FLEX library and is included here only for completeness.

8.01.03 GETCHAR

This procedure calls the FLEX routine 'GETCHR' and returns the keycode to the calling procedure in the 'B' accumulator (the ENDPROC ACCA statement transfers the contents of the 'A' accumulator to the 'B' accumulator). Since the FLEX routine automatically echo's the incoming character to the system console this routine will as well. This routine will 'hang up' waiting for a key to be pressed. Since this procedure returns a value it may be treated as a variable in the procedure that uses it, for example:

```
CHAR = GETCHAR;   or   IF GETCHAR <> $1B THEN...
```

This low level procedure is the primary link of the IOSUBS package with the keyboard on your system console. It can be reconfigured to use your system monitor input character routine or it may be configured to be a completely self contained routine driving an ACIA or PIA directly, the choice is up to you.

8.01.04 GETCHAR_NOECHO

This routine is identical to 'GETCHAR' except that it uses the FLEX routine pointed to by the FLEX 'INCHNE' vector. This routine, as its name implies, does not echo the incoming character back to the system console.

GETCHAR_NOECHO may be treated in a manner identical to GETCHAR.

NOTE: SOME VERSIONS OF FLEX DO NOT HAVE THE INCHNE VECTOR AT $D3E5 IMPLEMENTED!

8.01.05 GETKEY

This routine is ideally suited to multi-tasking software structures as it does not 'hang up' on the system console keyboard when it is called. If a key has not been hit since this routine was last called the procedure will return with a null ($00) in 'B' (a 'FALSE' condition). If a key has been hit then the procedure will return with the keycode in 'B' (a 'TRUE' condition). This procedure does not echo the key back to the system console. If you wish to ECHO the character back to the system console each time this procedure is used you may alter it by changing the reference to GETCHAR_NOECHO in line 39 to GETCHAR.

GETKEY may be treated just as a variable in a manner identical to GETCHAR.

8.01.06 CONVERT_LC

This procedure is passed a BYTE value and returns a BYTE value. If the value falls into the range of ASCII codes for lower case (a) through lower case (z) the code will automatically be converted to upper case. All other codes pass through the routine without alteration. This routine is present primarily for use by the two subsequent routines but it may be used alone if desired. For example:

```
CHAR=CONVERT_LC(GETKEY);
```

8.01.07 GET_UC

This procedure is very handy when you wish to prompt the console for letters of the alphabet but do not wish to be bothered with making any distinction between upper case and lower case letters in the subsequent evaluation of the key hit by the operator. For example you prompt the operator 'CONTINUE? (Y/N) '. You don't really care if he uses (Y) or (y) or (N) or (n).

This routine, when called, behaves exactly the same way as GETCHAR. It will echo whatever key is hit back to the console but will convert any lower case letter to its upper case equivalent before returning a value to the calling procedure.

GET_UC may be treated just as a variable in a manner identical to GETCHAR.

8.01.08 GET_UC_NOECHO

This routine is identical to the one above except that it does not echo the incoming character back to the system console and behaves very much like GETCHAR_NOECHO.

8.01.09 PUTCHAR

This procedure takes the value passed to it on the stack, transfers it to the 'A' accumulator and then calls the FLEX 'PUTCHR' routine which honours TTYSET.

The syntax of using this procedure, which is passed a value (but does not return one), is as follows:

PUTCHAR(CHAR); or PUTCHAR CHAR; or PUTCHAR = CHAR;

This low level procedure is the primary link of the IOSUBS package with the VDU on your system console. Since the FLEX 'PUTCHR' routine honours the TTYSET parameters any null padding on carriage returns will be taken care of automatically.

This routine may be reconfigured to link up with external assembly language subroutines or directly drive any element of hardware you choose.

This procedure, as it stands, directs output to the system console through FLEX. It could just as easily be vectored to the resident printer driver by substituting a call to $CCE4 instead of the call to $CD18. You would, however, have to call the printer initialization at $CCC0 before you started using the FLEX printer driver routine. Obviously you would have to pre-loaded your printer drivers, e.g.'GET PRINT.SYS'.

8.01.10 PRINTINT

This procedure takes the INTEGER value passed to it on the stack and prints its value on the VDU of your system console. The INTEGER is treated as a signed number and thus it is displayed in the range: -32768 to (+)32767.

The syntax of using this procedure, which is passed a value (but does not return one), is as follows:

PRINTINT(CHAR); or PRINTINT CHAR; or PRINTINT = CHAR;

8.01.11 REMOVE_CHAR

This procedure is provided mainly for use by the INPUT procedure which follows it. The purpose of this procedure is to rub the last ASCII character sent to the system console from the screen and leave the cursor in the position previously occupied by the ASCII character just removed, i.e. a destructive back-space. This procedure assumes that the cursor is immediately to the right of the character to be removed from the screen. This procedure performs essentially the same function as setting the FLEX TTYSET backspace echo (BE) value to $08 (back-space). This procedure may be used on its own if desired. A typical use would be as follows:

... THEN REMOVE_CHAR; in lieu of ... THEN PUTCHAR(BS);

8.01.12 INPUT

This procedure is designed to get a line of data from the system console. When it is called it must be passed a pointer containing the address of a buffer, usually a BYTE vector, and the MAXIMUM number of characters you are willing to accept. It will return with the buffer pointer to facilitate a multi-function construction but this returned value does not have to be used if you don't want to.

The procedure will start off with the console cursor at its last position so if you want it someplace else you will have to position it, using 'CURSOR' for example, before you call INPUT.

The operator is allowed to enter any ASCII code. Control codes, with three exceptions, are ignored. Only ASCII codes greater than or equal to $20 will be placed in the buffer. Hitting the back-space key ($08 as defined by the 'BS' constant) will move the cursor one position to the left and cancel (rub off the screen) the last character entered. If the cursor is in its original starting position no action will be taken.

Data entry is allowed to continue up to the limit of characters specified. If the operator tries to exceed this limit the cursor will simply remain in the last position and keep overtyping the last character in the line. It will also send a bell code ($07) to the console each time you attempt to type in a character past the limit of the buffer.

Hitting the cancel key (CONTROL-X ... $18 as defined by the 'CAN' constant) will erase the entire line and re-position the cursor to the original position.

Once all the data has been entered a carriage-return ($0D as defined by the 'CR' constant) will terminate the procedure. The <CR> is not placed in the buffer nor will it be echoed to the screen. Hence the cursor will remain where it was when you hit the <CR>.

The buffer area pointed to will now contain a string of ASCII characters identical to those on the screen of the system console and will be terminated with an ASCII NULL ($00).

There are several constructions possible when using INPUT; we will present the most common ones for guidance. In the following examples assume that 'BUFFER' has been declared as a GLOBAL or a LOCAL vector as follows: 'BYTE BUFFER(127);', and that BUFPTR has been declared as a GLOBAL or LOCAL integer.

The PRINT routine, which will be discussed in a moment, simply sends the character(s) it finds at the location specified by a pointer that is passed to it. Transmission ends when a null ($00) is encountered.

Each of the following structures does exactly the same thing, i.e. it prompts the operator for information, waits for a carriage return, then prints the exact same data out again. Not particularly brilliant, but it is only meant to demonstrate the structures involved!

```
   INPUT(.BUFFER,120);
   PRINT(.BUFFER);
```

In the above example we are simply using INPUT to fill the vector BUFFER with up to 120 characters supplied by the operator. The returned pointer to BUFFER is not used in this instance.

8.01.12 INPUT (continued)

```
BUFPTR = INPUT(.BUFFER,120);
PRINT(BUFPTR);
```

In the above example the same action takes place but the returned pointer to BUFFER is assigned to an integer called BUFPTR.

```
PRINT(INPUT(.BUFFER,120));
```

In the above example we carry the passing of the pointer returned by INPUT one stage further by passing it directly to PRINT, rather than going through the intermediate variable BUFPTR.

```
POINTER = .BUFFER;
COUNT = 120;

PRINT(INPUT(POINTER,COUNT);
```

In the above example we take the construction one stage further by assigning the address of the buffer to an integer variable called POINTER and assign the line length to another integer variable COUNT.

8.01.13 CRLF

What better name for a routine that simply transmits a carriage return ($0D) followed by a line feed ($0A) to the system console. The standard form is:

```
CRLF;
```


8.01.14 PRINT

This routine performs the task of sending a string of ASCII characters out to the system console. It is passed a pointer containing the address of the first character in the string and will transmit the characters one-by-one until a null ($00) is encountered at which point transmission will be terminated. As with most function procedures PRINT can take many forms depending on how the pointer is to be passed to it, viz:

```
PRINT("HELLO");   or   PRINT "HELLO";   or    PRINT = "HELLO";
```

In each of the above examples the double quotes enclosing a string are a special form of a pointer in PL/9. The actual ASCII code of the message (with its terminating null) will be embedded in the program at that point in the program and the compiler will pass a program counter relative address as the pointer to PRINT. This maintains complete position independence.

```
BYTE MESSAGE CR,LF,BEL,"HELLO";   /* CR, LF & BEL ARE DEFINED AS CONSTANTS */

PROCEDURE PRINT_DEMO:INTEGER POINTER;
PRINT(.MESSAGE);
```

In the above example we have declared a read-only string OUTSIDE of a procedure. Note how you can put CONSTANT declarations before the main body of the string within the double quotes. You may not, however, put CONSTANTS within the body of the string or AFTER the closing quote because they will be ignored by PRINT!.

The previous construction can be taken one stage further, e.g.:

```
POINTER = .MESSAGE;    - or -   POINTER = CR,LF,BEL,"HELLO";
PRINT(POINTER);
```

PRINT also recognizes the following constructions in the middle of strings and takes the action indicated:

```
\0           send null ($00) to console.
\b  or  \B   send bell code ($07) to console.
\e  or  \E   send escape ($1B) to console.
\l  or  \L   send LF ($0A) to console.
\n  or  \N   send CR-LF ($0D, $0A) to console.
\r  or  \R   send carriage return ($0D) to console.
```

As indicated upper or lower case letters may follow the back-slash (\) character. The \0 and \e forms are most commonly used in the transmission of escape sequences to the system console.

If the back-slash character is NOT immediately followed by one of the above characters then the back-slash and the character will be printed, e.g.:

```
PRINT("\b\n");  sends a bell code and a CR-LF to the console, whilst
PRINT("\ ");    sends a back-slash followed by a space to the console, whilst
PRINT("\t\g");  sends \t\g to the console.
```

You can expand the section between lines 131 and 136 to recognize whatever characters you wish and take virtually any desired action.

8.01.14 PRINT (continued)

When you need to send the double quote (") in the middle of a string use the following construction:

```
PRINT("""HELLO"" ""THERE"" ""EVERYBODY""");
```

would result in "HELLO" "THERE" "EVERYBODY" being printed. Note that the string starts and ends with triple quotes ("""). The double quote adjacent to the brackets is the pair of quotes that tells the compiler where the specified string starts and ends. To get a double quote printed in the middle of the string you simply type a pair of double quotes ("").

8.01.15 SPACE

This routine simply outputs a specified number of spaces (up to 32767) to the system console. It is passed a single variable and is used as follows:

SPACE(5); or SPACE(COUNT); where count is a variable or a constant.

Be careful when using this procedure near the right hand side of the system console area as it is not possible to predict how the terminal will behave when you enter a character in the last column. If you accidentally pass this function a negative number no action will be taken.

```
Page  1:  SYSTEM CONSOLE INPUT/OUTPUT ROUTINES  V:4.00           June 1 1984

0000  /*  SYSTEM CONSOLE INPUT/OUTPUT ROUTINES  V:4.00  */
0000
0000
0000  constant nul = $00,
0000           abt = $03,
0000           bel = $07,
0000            bs = $08,
0000            lf = $0a,
0000            cr = $0d,
0000           can = $18,
0000           esc = $1b,
0000            sp = $20;
0000
0000
0000  procedure monitor;
0003    gen $6e,$9f,$d3,$f3;  /* JMP [$D3F3]  ('MONIT') */
0007  endproc;
0008
0008
0008  procedure warms;
0008     jump $cd03; /* FLEX WARM START ENTRY POINT */
000B  endproc;
000C
000C
000C  procedure getchar;
000C     call $cd15; /* FLEX 'GETCHR' */
000F  endproc acca;
0012
0012
0012  procedure getchar_noecho;
0012     gen $ad,$9f,$d3,$e5;  /* JSR [INCHNE] */
0016  endproc acca;
0019
0019
0019  procedure getkey;
0019     call $cd4e; /* FLEX 'STAT' */
001C     if ccr and $04  /* IMPLICIT <> 0 */
001E        then acca = nul;
0029        else getchar_noecho;
002E  endproc acca;
0031
0031
0031  procedure convert_lc(byte char);
0031     if char >= 'a .and char <= 'z
003F        then char = char - $20;
0053  endproc char;
0056
0056
0056  procedure get_uc:byte inchar;
0058  endproc convert_lc(getchar);
0063
0063
0063  procedure get_uc_noecho:byte inchar;
0065  endproc convert_lc(getchar_noecho);
0070
0070
```

Page 2: SYSTEM CONSOLE INPUT/OUTPUT ROUTINES V:4.00 June 1 1984

```
0070 procedure putchar(byte char);
0070    acca = char;
0074    call $cd18; /* FLEX 'PUTCHR'  (HONOURS 'TTYSET' PARAMETERS) */
0077 endproc;
0078
0078
0078 procedure printint(integer n);
0078    if n < 0
007A       then begin
0081               putchar '-;
0089               n = -n;
0091            end;
0091    if n >= 10 then printint n/10;
0109    putchar n\10 + '0;
011C endproc;
011D
011D procedure remove_char;
011D    putchar(bs);
0126    putchar(sp);
012F    putchar(bs);
0138 endproc;
0139
0139
0139 procedure input(byte .buffer,length):byte char:integer pos;
013B    pos = 0;
0140    repeat
0140       char = getchar_noecho;
0145       if char
0145          case bs
014A               then begin
014E                    if pos > 0
0150                       then begin
0157                            remove_char;
0159                            pos = pos - 1;
0160                            end;
0160                       else putchar(bel);
016C                    end;
016C          case can
0171               then begin
0175                       while pos > 0
0177                          begin
017E                            remove_char;
0180                            pos = pos - 1;
0187                          end;
0187                    end;
0189       if char >= sp
018B          then if pos < length
0193                  then begin
01A2                          putchar(char);
01AB                          buffer(pos) = char;
01B5                          pos = pos + 1;
01BC                       end;
01BC                  else putchar(bel);
01C8    until char = cr;
01D0    buffer(pos) = 0;
01D8 endproc .buffer;
```

```
Page  3:  SYSTEM CONSOLE INPUT/OUTPUT ROUTINES  V:4.00           June 1 1984

01DD
01DD
01DD  procedure crlf;
01DD     putchar(cr);
01E6     putchar(lf);
01EF  endproc;
01F0
01F0
01F0  procedure print(byte .string): byte char;
01F2     while string /* IMPLICIT <> 0 (NULL) */
01F2        begin
01FA           if string = '\
01FD              then begin
0203                   .string = .string + 1;
020A                   if string >= 'a .and string <= 'z
021A                      then char = string - $20;  /* FORCE UPPER CASE */
022F                      else char = string;
0237                   if char
0237                      case 'N then crlf;
0242                      case 'B then putchar(bel);
0254                      case 'L then putchar(lf);
0266                      case 'R then putchar(cr);
0278                      case 'E then putchar(esc);
028A                      case '0 then putchar(nul);
029C                      else begin
029F                              putchar('\);
02A8                              putchar(string);
02B2                           end;
02B2                   end;
02B2              else putchar(string);
02BF           .string = .string + 1;
02C6        end;
02C6  endproc;
02CC
02CC
02CC  procedure space(integer n);
02CC     while n > 0
02CE        begin
02D5           putchar(sp);
02DE           n = n - 1;
02E5        end;
02E5  endproc;
```

```
Page  4:  SYSTEM CONSOLE INPUT/OUTPUT ROUTINES  V:4.00          June 1 1984

PROCEDURES:
monitor             0003
warms               0008
getchar             000C   BYTE
getchar_noecho      0012   BYTE
getkey              0019   BYTE
convert_lc          0031   BYTE
get_uc              0056   BYTE
get_uc_noecho       0063   BYTE
putchar             0070
printint            0078
remove_char         011D
input               0139   INTEGER
crlf                01DD
print               01F0
space               02CC

DATA:

EXTERNALS:
nul                 0000
abt                 0003
bel                 0007
bs                  0008
lf                  000A
cr                  000D
can                 0018
esc                 001B
sp                  0020

GLOBALS:
```

8.02.00 INTELLIGENT TERMINAL LIBRARY

The following procedures assume that your system console has some basic 'intelligence' and supports the functions indicated. These functions were previously in the IOSUBS library. As these functions are not used by many PL/9 programmers we have moved them to a separate library in order to reduce the size of the IOSUBS library.

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                                                       *
*  THE   PROCEDURES   IN   THIS   LIBRARY   HAVE  BEEN  *
*  SPECIFICALLY CONFIGURED  FOR  A  SOROC  IQ-120/LEAR  *
*  SIEGLER ADM-5. IF YOU HAVE A DIFFERENT TERMINAL YOU  *
*  MUST RECONFIGURE THIS LIBRARY BEFORE YOU USE IT.     *
*                                                       *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

The codes and code sequences transmitted for each of the functions will vary from terminal to terminal. You will therefore have to configure these routines for the characteristics of your terminal. MANY terminals require that nulls (usually 1 or 2 of them) be sent to the terminal after cursor or screen control codes. The NULLS routine should be configured accordingly.

If your terminal does not support the special functions indicated but supports non-destructive up, down, right, and left cursor movements, but does not support anything more elaborate, it is possible to construct routines that duplicate most of the functions supplied. For example 'HOME' might be implemented by moving the cursor up 24 lines and left 80 spaces assuming that you have an 80 col x 24 line VDU.

CONTROL CODES USED BY THIS LIBRARY

```
CURSOR X,Y    ESCAPE, = , ROW+$20, COL+$20
ERASE EOL     ESCAPE, T
ERASE EOP     ESCAPE, Y
ATTR ON       ESCAPE, )
ATTR OFF      ESCAPE, (
```

One null is sent to the terminal after each escape sequence is sent. If these code sequences are the same ones used by your terminal you will not have to make any changes to this library. If they are not the same, however, you must modify the procedures to match the requirements of your terminal.

8.02.01 NULLS

This routine is used by several of the subsequent routines to provide a small delay after a terminal control command is sent. Many terminals will drop the first two or three characters transmitted after a control command unless this delay is present, particularly when transmitting at high baud rates. As supplied only one null will be sent to the system console when this function is used.

8.02.00 INTELLIGENT TERMINAL LIBRARY (continued)

8.02.02 ERASE_EOL

This command erases the screen from the current cursor position to the end of the current line.

8.02.03 ERASE_EOP

This command erases the screen from the current cursor position to the end of the screen.

8.02.03 CURSOR

This command places the cursor at any desired position on the screen by using the X-Y cursor addressing facilities available in most terminals. This command is supplied two variables when it is called. The first is the desired column number with zero being the far left column. The second is the desired row number with zero being the row at the top of the screen. There are two basic forms:

```
   CURSOR(6,10);   or   CURSOR(COL,ROW);
```

in the second form COL and ROW are either variables or constants.

8.02.04 HOME

This command places the cursor in the top-left corner of the screen.

8.02.05 HOME_N_CLR

This command places the cursor in the top-left corner of the screen and clears the entire screen.

8.02.06 ATTR_ON

This command turns on the console video attributes. The video attribute varies from terminal to terminal. Some use intensified video, others use reduce video and others use reverse video.

8.02.07 ATTR_OFF

This command turn off the console video attributes, i.e. restores the power-up default mode of the console.

Page 1: INTELLIGENT TERMINAL HANDLER LIBRARY V:4.00 June 1 1984

```
0000  /* INTELLIGENT TERMINAL HANDLER LIBRARY  V:4.00  */
0000
0000
0000  include 0.iosubs.lib;
02E8
02E8
02E8  /* THIS LIBRARY IS CONFIGURED FOR A SOROC IQ120/LEAR SIEGLER ADM-5 */
02E8
02E8
02E8  procedure nulls:byte count;
02EA     count = 1;
02EE     while count /* IMPLICIT <> 0 */
02EE        begin
02F5           putchar(nul);
02FE           count = count - 1;
0300        end;
0300  endproc;
0305
0305
0305  procedure erase_eol;
0305     putchar(esc);
030E     putchar('T);
0317     nulls;
0319  endproc;
031A
031A
031A  procedure erase_eop;
031A     putchar(esc);
0323     putchar('Y);
032C     nulls;
032E  endproc;
032F
032F
032F  procedure cursor(byte column,row);
032F     putchar(esc);
0338     putchar('=);
0341     putchar(sp + row);     /* OFFSET OF $20 */
034C     putchar(sp + column);  /* OFFSET OF $20 */
0357     nulls;
0359  endproc;
035A
035A
035A  procedure home;
035A     cursor(0,0);
0366  endproc;
0367
0367
0367  procedure home_n_clr;
0367     home;
0369     erase_eop;
036B  endproc;
036C
036C
```

```
Page  2:  INTELLIGENT TERMINAL HANDLER LIBRARY  V:4.00           June 1 1984

036C  0054 procedure attr_on;
036C  0055    putchar(esc);
0375  0056    putchar(')');
037E  0057 endproc;
037F  0058
037F  0059
037F  0060 procedure attr_off;
037F  0061    putchar(esc);
0388  0062    putchar('(');
0391  0063 endproc;
```

VISA 30/40 CODE SEQUENCES

If you own a VISA 30 or a VISA 40 terminal the following procedures in this library should be modified as indicated below:

```
procedure erase_eol;
putchar(esc);
putchar($0F);  /* SI */
endproc;


procedure erase_eop;
putchar(esc);
putchar($18);  /* CAN */   /* clears foreground */
putchar(esc);
putchar($17);  /* ETB */   /* clears background */
endproc;


procedure cursor(byte column,row);
putchar(esc);
putchar($11);   /* DC1 */
putchar(col);
putchar(row);
nulls;
endproc;


procedure attr_on;   /* BRIGHT */
putchar(esc);
putchar($1F);  /* US */
endproc;


procedure attr_off;  /* NORMAL */
putchar(esc);
putchar($19);  /* EM */
endproc;
```

NOTE: The 'esc' code (e.g. 'putchar(esc)) is determined by a switch on the back of the terminal. In one position the 'esc' code ($1B) will be used as the lead-in character. In the other position 'tilde' ($7E) will be used as the lead-in character.

Page 3: INTELLIGENT TERMINAL HANDLER LIBRARY V:4.00 June 1 1984

PROCEDURES:

monitor	0003	
warms	0008	
getchar	000C	BYTE
getchar_noecho	0012	BYTE
getkey	0019	BYTE
convert_lc	0031	BYTE
get_uc	0056	BYTE
get_uc_noecho	0063	BYTE
putchar	0070	
printint	0078	
remove_char	011D	
input	0139	INTEGER
crlf	01DD	
print	01F0	
space	02CC	
nulls	02E8	
erase_eol	0305	
erase_eop	031A	
cursor	032F	
home	035A	
home_n_clr	0367	
attr_on	036C	
attr_off	037F	

DATA:

EXTERNALS:

nul	0000
abt	0003
bel	0007
bs	0008
lf	000A
cr	000D
can	0018
esc	001B
sp	0020

GLOBALS:

8.03.00 HEXIO LIBRARY

This library contains the basic procedures required to get HEX characters, bytes, or addresses to and from the system console.

It uses the 'GET_UC' and the 'PUTCHAR' routines in the IOSUBS library as its basic communication link with the system console.

In addition it requires that two GLOBAL BYTE variables be declared in your main program. One of the variables is used as an error flag to signify that the operator has supplied a NON-HEX character. The other is used to hold the last character entered by the operator and is usually only used when the error flag is found to be set and the non-hex character input means something else to your program.

W A R N I N G

```
+ + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + 
+  The GLOBAL declaration illustrated  in  the  HEXGLOBL.DEF  file  must  be  +
+  inserted  in  the  file that INCLUDEs this library. Forgetting to declare  +
+  the GLOBAL variables in the main program will cause errors to be reported  +
+  during compilation.                                                        +
+ + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + 
```

HEX-IO LIBRARY GLOBAL DEFINITIONS

The HEXGLOBL.DEF file looks like this:

```
0000  0001 /* HEX-IO LIBRARY GLOBAL DEFINITIONS */
0000  0002
0000  0003
0000  0004 global byte erflag, keychar;
```

GLOBALS:

```
erflag              0000   BYTE
keychar             0001   BYTE
```


8.03.01 GET_HEX_NIBBLE

This is the basic procedure for all of the HEX input routines. It is structured to return one variable directly and two other variables via GLOBALs. This procedure converts the ASCII code representing a valid HEX character (0-9, A-F) to a byte with the lower 4-bits reflecting the HEX value of the value entered. The top 4-bits are not used. If a valid HEX character is supplied by the operator then the GLOBAL variable ERFLAG will be FALSE ($FF). If a non-HEX number is supplied ERFLAG will be TRUE ($00), and the GLOBAL variable KEYCHAR will contain the code of the key the operator entered. This procedure will seldom be used on its own but forms the basis for the two following procedures. The basic form for using this procedure on its own is:

```
 CHAR=GET_HEX_NIBBLE AND $0F;  /* STRIP TOP 4-bits */
```

8.03.02 GET_HEX_BYTE

This procedure uses the procedure GET_HEX_NIBBLE to return a BYTE variable representing the HEX values of TWO keys supplied by the operator. If an non-HEX character is entered on the first keystroke the routine will be terminated, i.e. it will not prompt for the second key. The GLOBAL variable ERFLAG and KEYCHAR will reflect this early termination. In this and the following procedure the error flag becomes very important as it not only provides a method of preventing a subsequent prompt for data when an earlier prompt returned an error but also allows the calling procedure to determine if the data it is getting back is valid or not. For example:

```
   CHAR=GET_HEX_BYTE;
   IF ERFLAG         /* IMPLICIT <> 0 */
      THEN PRINT("  \Bnot HEX!");
      ELSE...
```

Take a look at the 'MINI MONITOR' in the USERS GUIDE for further examples of how ERFLAG and KEYCHAR can be used.

8.03.03 GET_HEX_ADDRESS

This procedure is just an expansion of the previous one and returns an INTEGER value representing the HEX values of FOUR successive keystrokes. As with the procedure GET_HEX_BYTE an early error terminates the procedure immediately.

8.03.04 PUT_HEX_NIBBLE

This procedure uses PUTCHAR in the IOSUBS library as its basic communication link with the system console. This procedure performs the inverse function of GET_HEX_NIBBLE, it sends a single HEX character (in ASCII form) to the system console. This procedure is passed a BYTE variable on the stack. Only the lower 4-bits are assumed to be significant (the top 4-bits are ignored) and it is also assumed that these bits represent the HEX value that is to be printed, for example:

```
   PUT_HEX_NIBBLE($0F);   or   PRINT_HEX_NIBBLE($F);
```

would print 'F' on the system console screen.

8.03.05 PUT_HEX_BYTE

This procedure is simply an expansion of the above and sends two HEX characters out to the system console. It to is passed a byte variable on the stack, and takes the following basic form:

```
PUT_HEX_BYTE($A5);   would print 'A5' on the system console screen.
PUT_HEX_BYTE($7);    would print '07' on the system console screen.
```

8.03.06 PUT_HEX_ADDRESS

Again this procedure is simply an expansion of the previous one, and is passed an INTEGER on the stack which represents the four characters to be transmitted.

The HEXIO library will find the vast majority of its uses in writing HEX number oriented utilities and/or providing a simple method of simulating I/O operations for program development.

Page 1: HEX-IO LIBRARY V:4.00 June 1 1984

```
0000 /*  HEX-IO LIBRARY  V:4.00  */
0000
0000
0000 include 0.hexglobl.def;
0006 include 0.trufalse.def;
0006 include 0.iosubs.lib;
02EE
02EE <---------------------------------------- NOTE:  Sudden    jump    in
02EE procedure get_hex_nibble:byte inchar;           address  caused   by
02F0    inchar = get_uc;                             included libraries
02F5    keychar = inchar;
02F9    erflag = true;
02FD    if inchar >=  '0 .and inchar <= '9
030B       then begin
0319            inchar = inchar - '0;
031F            erflag = false;
0321            end;
0321       else if inchar >=  'A .and inchar <= 'F
0332               then begin
0340                    inchar = inchar - '7;
0346                    erflag = false;
0348                    end;
0348 endproc inchar;
034D
034D
034D procedure get_hex_byte:byte inchar;
034F       inchar = shift(get_hex_nibble,4);
0357       if erflag = true
0359          then return;
0362       inchar = inchar or get_hex_nibble;
0370 endproc inchar;
0375
0375
0375 procedure get_hex_address:integer inchar;
0377       inchar = swap(integer(get_hex_byte));
037E       if erflag = true
0380          then return;
0389       inchar = inchar or integer(get_hex_byte);
0396 endproc inchar;
039B
039B
039B procedure put_hex_nibble(byte outchar);
039B    outchar = (outchar and $0f) + '0;  /* CONVERT TO ASCII */
03A3    if outchar > '9
03A5       then outchar = outchar + 7;       /* A-F OFFSET */
03B1    putchar(outchar);
03BA endproc;
03BB
03BB
03BB procedure put_hex_byte(byte outchar);
03BB    put_hex_nibble(shift(outchar,-4)); /* FIRST DIGIT */
03C7    put_hex_nibble(outchar);           /* LAST DIGIT */
03CF endproc;
03D0
03D0
```

Page 2: HEX-IO LIBRARY V:4.00 June 1 1984

```
03D0  procedure put_hex_address(integer outchar);
03D0     put_hex_byte(swap(outchar)); /* FIRST TWO DIGITS */
03DA     put_hex_byte(byte(outchar)); /* LAST TWO DIGITS */
03E2  endproc;
```


Page 3: HEX-IO LIBRARY V:4.00 June 1 1984

PROCEDURES:

monitor	0009	
warms	000E	
getchar	0012	BYTE
getchar_noecho	0018	BYTE
getkey	001F	BYTE
convert_lc	0037	BYTE
get_uc	005C	BYTE
get_uc_noecho	0069	BYTE
putchar	0076	
printint	007E	
remove_char	0123	
input	013F	INTEGER
crlf	01E3	
print	01F6	
space	02D2	
get_hex_nibble	02EE	BYTE
get_hex_byte	034D	BYTE
get_hex_address	0375	INTEGER
put_hex_nibble	039B	
put_hex_byte	03BB	
put_hex_address	03D0	

DATA:

EXTERNALS:

true	FFFF	
false	0000	
mem	0000	BYTE
nul	0000	
abt	0003	
bel	0007	
bs	0008	
lf	000A	
cr	000D	
can	0018	
esc	001B	
sp	0020	

GLOBALS:

erflag	0000	BYTE
keychar	0001	BYTE

8.04.00 BITIO LIBRARY

This library module provides three sets of routines. The first set is designed to provide bit oriented I/O simulations via your system console. The second set is designed to evaluate or assign the status of a bit in a variable passed to it. The third set is designed to evaluate or assign the status of a bit in a variable pointed to by the calling procedure.

This library module uses 'GETCHAR_NOECHO' and 'PUTCHAR' from the IOSUBS library as its communication link with the system console for the first two routines.

The remaining six routines are free standing. These routines are also very useful if you wish to avoid embedding bit operations in the middle of your procedures. If you have read section 9.07.06 and section 9.07.07 in the USERS GUIDE you should recognize the last six procedures.

The bit positions are numbered as follows:

```
b15  b14  b13  b12  b11  b10  b9   b8   b7   b6   b5   b4   b3   b2   b1   b0
|                                                                            |
+-- (most significant bit)                        (least significant bit) --+
```

8.04.01 BITSIN

This procedure returns an integer value which represents the sequence of ones and zeros supplied by the operator via the keyboard. It is structured to prompt for 16-bits in the following format 'X X X X X X X X X X X X X X X X'. The operator may supply a one or a zero for each position. The standard form is:

```
INTVAR=BITSIN;
```

The input format can be altered to suit your own particular requirements. For example if you wanted to input the binary data as 'XXXX XXXX XXXX XXXX' you would insert the following between line 23 and line 24:

```
IF COUNT=4 .OR COUNT=8 .OR COUNT=12 THEN
```

This procedure could also be configured to work with 8-bits by changing the '16' in line 15 to '8', changing the 'INTEGER' declaration in line 13 to BYTE and restructuring the data table in lines 10 and 11 to:

```
0010 BYTE MASK $01,$02,$04,$08,
0011           $10,$20,$40,$80;
```

8.04.02 BITSOUT

This procedure provides the inverse function of BITSIN. It is passed an integer value and sends a series of ones, zeros and spaces to the system console in the same format as BITSIN. If you look in the program listing in section 9.12.02 you will find a slightly adapted version of this routine that dumps 8-bit data in the following format 'XXXX XXXX'. The standard form for BITSOUT is:

```
BITSOUT(INTVAR);
```

8.04.03 BITIN

This procedure may be passed and INTEGER or BYTE value and a number representing the bit position of interest and will return the status of the specified bit as either a one or a zero. The standard forms are:

```
IF BITIN(VAR,1) = 1   or   IF BITIN VAR,1 = 1
THEN...                    THEN...
```

In the above example we are evaluating the status of bit (b1) and are looking for a logical one. VAR can be either a BYTE or an INTEGER. The number, which must be in the range of 0 - 7 for a BYTE or 0 - 15 for an INTEGER, may be declared as indicated or may be a constant or a variable.

8.04.04 BITOUT

This procedure is also passed a BYTE or an INTEGER variable and a bit position. In addition it is passed the status you wish to impart to the specified bit position and will return the BYTE or INTEGER with the bit modified as requested. The standard forms are:

```
VAR = BITOUT(VAR,4,1);   or   VAR = BITOUT VAR,4,1;
```

In the above example we are assigning a logical one to bit b4.

8.04.05 BITZ8IN

This is the first of two procedures which is similar in concept to BITIN except that instead of passing the variable to the procedure you pass a pointer to the variable to the procedure. Since the variable 'pointed to' must have its size specified, one procedure, this one, is required to operate on BYTE variables, and another procedure, the next one, is required to operate on INTEGER variables.

This is the first in a group of four procedures that are designed primarily to operate with I/O devices specified by 'AT' declarations. They will, of course, also operate on any GLOBAL or LOCAL variable as well. The standard form is:

```
AT $E004: BYTE ACIA_STATUS;

LOOP:
IF BITZ8IN(.ACIA_STATUS,0) = 0   or   IF BITZ8IN .ACIA_STATUS,0 = 0
   THEN GOTO LOOP;
```

In the above example we are testing the status of an MC6850 control register and are looking at the 'Receive Data Register' status flag bit (b0). If it is a 0 we simply go back to the label loop and repeat the process. A better construction would be:

```
WHILE BITZ8IN(.ACIA_STATUS,0) = 0   BEGIN END;

                                     or

REPEAT UNTIL BITZ8IN(.ACIA_STATUS,0) = 1;
```

8.04.06 BITZ16IN

This procedure is simply an expansion of the above and enables you to read the status of a bit on a 16-bit I/O port (or an INTEGER variable) directly. This routine is handy when working with an MC6821 PIA that has had the RS0 and RS1 lines crossed. With RS0 connected to A0 and RS1 connected to A1 the PIA ports stack up as follows: ADATA, ACTRL, BDATA, BCTRL. If these two lines are reversed, i.e. RS0 is connected to A1 and RS1 is connected to A0 the PIA ports stack up as follows: ADATA, BDATA, ACTRL, BCTRL. This configuration allows you to use a PIA as a 16-bit I/O port and therefore take advantage of the MC6809, and PL/9's 16-bit I/O facilities.

8.04.07 BITZ8OUT

This procedure is passed three variables. The first is the address of the variable of interest, the second is the bit position, and the third is the desired status of the bit. This procedure is designed to operate on BYTE size data only. The standard forms are:

```
BITZ8OUT(.PORT,6,1);   or   BITZ8OUT .PORT,6 = 1;
```

8.04.08 BITZ16OUT

This procedure is simply the 16-bit (INTEGER) expansion of the above procedure and has an identical syntax.

Page 1: BIT-IO LIBRARY V:4.00 June 1 1984

```
0000  /*  BIT-IO LIBRARY  V:4.00  */
0000
0000
0000  include 0.iosubs.lib;
02E8
02E8  constant zero = $30,
02E8           one  = $31;
02E8
02E8  integer mask $0001,$0002,$0004,$0008,$0010,$0020,$0040,$0080,
02F4               $0100,$0200,$0400,$0800,$1000,$2000,$4000,$8000;
0308
0308
0308  procedure bitsin:byte count,inchar:integer bitchar;
030A     count = 16;
030E     bitchar = 0;
0313     repeat
0313        repeat
0313          inchar = getchar_noecho;
0318        until inchar = zero  .or  inchar = one
0326        putchar(inchar); /* ECHO 0/1 */
033B        if inchar = one
033D           then bitchar = bitchar or mask(count - 1);
035B        putchar($20);
0364        count = count - 1;
0366    until count = 0;
036C  endproc bitchar;
0371
0371
0371  procedure bitsout(integer bitchar):byte count;
0373    count = 16;
0377    repeat
0377       if bitchar and mask(count - 1)  /* IMPLICIT <> 0 */
038A          then putchar(one);
039E          else putchar(zero);
03AA       putchar($20);
03B3       count = count - 1;
03B5     until count = 0;
03BB  endproc;
03BE
03BE
03BE  procedure bitin(integer data:byte position):byte status;
03C0     if data and mask(position) /* IMPLICIT <> 0 */
03D1        then status = 1;
03E0        else status = 0;
03E5  endproc status;
03EA
03EA
03EA  procedure bitout(integer data:byte position, status);
03EA     if status    /* IMPLICIT <> 0 */
03EA        then data = data or mask(position);
0408        else data = data and not(mask(position));
0424  endproc integer data;
0427
0427
```

```
Page  2:  BIT-IO LIBRARY  V:4.00                                June 1 1984

0427  procedure bitz8in(byte .data:byte position):byte status;
0429     if data and mask(position) /* IMPLICIT <> 0 */
043C        then status = 1;
044B        else status = 0;
0450  endproc status;
0455
0455
0455  procedure bitz16in(integer .data:byte position):byte status;
0457     if data and mask(position)  /* IMPLICIT <> 0 */
0469        then status = 1;
0478        else status = 0;
047D  endproc;
0480
0480
0480  procedure bitz8out(byte .data:byte position, status);
0480     if status  /* IMPLICIT <> 0 */
0480        then data = data or mask(position);
04A1        else data = data and not(mask(position));
04C0  endproc;
04C1
04C1
04C1  procedure bitz16out(integer .data:byte position, status);
04C1     if status  /* IMPLICIT <> 0 */
04C1        then data = data or mask(position);
04E1        else data = data and not(mask(position));
04FF  endproc;
```


Page 3: BIT-IO LIBRARY V:4.00 June 1 1984

PROCEDURES:

monitor	0003	
warms	0008	
getchar	000C	BYTE
getchar_noecho	0012	BYTE
getkey	0019	BYTE
convert_lc	0031	BYTE
get_uc	0056	BYTE
get_uc_noecho	0063	BYTE
putchar	0070	
printint	0078	
remove_char	011D	
input	0139	INTEGER
crlf	01DD	
print	01F0	
space	02CC	
bitsin	0308	INTEGER
bitsout	0371	
bitin	03BE	BYTE
bitout	03EA	INTEGER
bitz8in	0427	BYTE
bitz16in	0455	
bitz8out	0480	
bitz16out	04C1	

DATA:

mask	02E8	INTEGER

EXTERNALS:

nul	0000
abt	0003
bel	0007
bs	0008
lf	000A
cr	000D
can	0018
esc	001B
sp	0020
zero	0030
one	0031

GLOBALS:

8.05.00 HARDIO LIBRARY

This library module provides four procedures which BASIC programmers will recognize; PEEK, DPEEK, POKE, and DPOKE. They are used in the same manner as their BASIC equivalents except that HEX rather than decimal numbers are normally used.

These procedures are extremely fast and code efficient (the ENTIRE library is less than 20 bytes!). All of these procedures use a 'trick' in that these procedures are structured to expect a pointer but may passed a value or a pointer. If they are passed a value it must be the address where the data is expected.

8.05.01 PEEK

This routine is passed a pointer to a BYTE variable and returns the contents of the memory location pointed to:

BVAR=PEEK($1000);   or   BVAR=PEEK(.PORT);   or   BVAR=PEEK(ADDRESS);

The first and last forms are the most common use of PEEK as the one in the middle can be constructed as 'BVAR=PORT;' In the last example ADDRESS would be an INTEGER variable maintained/updated by a procedure and therefore there is no way of telling what its value is. In this instance we are passing the VALUE contained in the variable ADDRESS NOT the memory location of ADDRESS.

8.05.02 DPEEK

This routine is passed a pointer to an INTEGER variable and returns the contents of the memory location, i.e. it returns a 16-bit variable. The syntax is identical to PEEK.

8.05.03 POKE

This routine is passed a pointer to a BYTE variable and a BYTE value to be placed in the memory location pointed to:

POKE($1000,$FF);   or   POKE(ADDRESS,BVAR);

In the second example ADDRESS and BVAR would be an INTEGER variable and a BYTE variable, respectively, which would be updated by a procedure.

8.05.04 DPOKE

This routine is passed a pointer to an INTEGER variable and an INTEGER value to be placed in the memory location pointed to:

DPOKE($1000,$1234);   or   DPOKE(ADDRESS,IVAR);

Page 1: HARD-IO LIBRARY V:4.00 June 1 1984

```
0000  /*  HARD-IO LIBRARY  V:4.00  */
0000
0000
0000  procedure peek(byte .pointer);
0003  endproc pointer;
0007
0007
0007  procedure dpeek(integer .pointer);
0007  endproc pointer;
000B
000B
000B  procedure poke(byte .pointer, char);
000B     pointer = char;
0010  endproc;
0011
0011
0011  procedure dpoke(integer .pointer, char);
0011     pointer = char;
0016  endproc;
```

Page 2: HARD-IO LIBRARY V:4.00 June 1 1984

PROCEDURES:

peek	0003	BYTE
dpeek	0007	INTEGER
poke	000B	
dpoke	0011	

DATA:

EXTERNALS:

GLOBALS:

8.06.00 STRSUBS (STRING SUBROUTINES) LIBRARY

This package provides you with a set of minimal functions to perform string handling and is based on the string functions usually supplied with C compilers. If you are a BASIC programmer have a look at the library module called 'BASTRING.LIB', as it contains functions to perform operations similar to LEFT$, RIGHT$ and MID$.

Strings in PL/9 can be of any length (they are NOT limited to 255 characters) and are by convention terminated in a null. Strings generated by PL/9 always have the null; you don't have to specify one yourself. You can tell PL/9 to use another character in lieu of the null when you run the SETPL9 program if this is what your application requries. If you do this you MUST change all of the library routines that expect to find a null at the end of a string, e.g. 'PRINT' in IOSUBS.

In the following statement:

```
PRINT "HELLO";
```

the string generated is six characters long; five for the word and one for the null at the end. At the risk of being tedious, I should remind you that when you declare a buffer that is to be used to hold a string, always dimension it to one larger than the longest string you will want to put in it.

8.06.01 STRLEN(.STRING)

Measures the supplied string and returns its length as an INTEGER. The argument is shown here as a pointer, but as long as it is 16 bits and points to the string in question it may take any of the following forms:

```
IVAR=STRLEN(.STRING);
POINTER=.STRING;  IVAR=STRLEN(POINTER);
IVAR=STRLEN("How long is this string?");
```

The function does not count the null when measuring it and is smart enough to cope with a null string, returning zero.

8.06.02 STRCOPY(ARG1, ARG2)

Copies the string represented by the SECOND argument into the location pointed to by the FIRST. Note the order; if you don't like it then feel free to re-write the function. If the buffer allocated for ARG1 is smaller than the actual length of ARG2 then the program will happily walk all over your valuable data or eat its own return address, so BE CAREFUL!

8.06.03 STRCAT(ARG1, ARG2)

Puts the second string onto the end of the first. It calls STRLEN and STRCOPY and is a good example of how to write cryptic PL/9 programs! Unfortunately, to make the function more readable also makes it larger and slower, a sad fact of life. Again, no checks are made that the resulting string will fit into the space available. (What do you expect from a FREE library pack?)

8.06.04 STRCMP(ARG1, ARG2)

Compares two strings, for length and for content. The result of the comparison is either -1, 0 or 1 according to these rules:

If ARG1 = ARG2 then return 0.
If ARG1 alphabetically succeeds ARG2 then return 1.
If ARG1 alphabetically preceeds ARG2 then return -1.

NOTE: 'FULL' will preceed 'FULLY'.

8.06.05 STRPOS(ARG1, ARG2)

Searches ARG1 to see if it can find ARG2 contained within. If it can, it returns the position (zero upwards) in ARG1 at which it found the start of ARG2. If ARG1 does not contain ARG2 then the value -1 is returned.


```
Page  1:  STRING FUNCTIONS LIBRARY  V:4.00                          June 1 1984

0000 /*  STRING FUNCTIONS LIBRARY  V:4.00  */
0000
0000
0000 procedure strlen(byte .string): integer len;
0005    len = 0;
000A    while string(len) /* IMPLICIT <> 0 (NULL) */
0012       len = len + 1;
001E endproc len;
0025
0025
0025 procedure strcopy(byte .string1, .string2): byte .return_value;
0027    .return_value = .string1;
002B    repeat
002B       string1 = string2;
0031       .string1 = .string1 + 1;
0038       .string2 = .string2 + 1;
003F    until string1(-1) = 0;
004C endproc .return_value;
0051
0051
0051 procedure strcat(byte .string1, .string2);
0051    strcopy(.string1(strlen(.string1)),.string2);
0067 endproc .string1;
006A
006A
006A procedure strcmp(byte .string1, .string2):
006A    byte c1, c2: integer index;
006C    index = -1;
0071    repeat
0071       index = index + 1;
0078       c1 = string1(index);
0082       c2 = string2(index);
008C       if c1 = 0 .and c2 = 0 then return 0;
00AD       if c1 = 0 then return -1;
00BA       if c2 = 0 then return 1;
00C7    until c1 <> c2;
00CD    if c1 > c2 then return 1;
00DA endproc -1;
00DF
00DF
00DF procedure strpos(byte .string1, .string2):
00DF    byte c1, c2: integer i, j, k;
00E1    i = 0;
00E6    repeat
00E6       j = i;
00EA       k = 0;
00EF       repeat
00EF          c1 = string1(j);
00F9          c2 = string2(k);
0103          if c2 = 0 then return i;
0110          if c1 = 0 then return extend(-1);
011E          j = j + 1;
0125          k = k + 1;
012C       until c1 <> c2;
0132       i = i + 1;
0139    forever;
0139 endproc;
```

Page 2: STRING FUNCTIONS LIBRARY V:4.00 June 1 1984

PROCEDURES:

strlen	0003	INTEGER
strcopy	0025	INTEGER
strcat	0051	INTEGER
strcmp	006A	BYTE
strpos	00DF	INTEGER

DATA:

EXTERNALS:

GLOBALS:

8.07.00 BASTRING (BASIC STRING) LIBRARY

These routines are near equivalents of the BASIC string handling functions LEFT$, RIGHT$, etc. Note that most of these routines alter the string they are given as a parameter. If the string is part of the program (as opposed to being an alterable variable) is not advisable to use one of these functions directly (it would not work at all if the program were in ROM!) Instead you should first use STRCOPY to copy the string to a working buffer (see the example program on the following page). See the REALCON.LIB file for number conversions.

8.07.01 LEFT

This procedure is used in the basic form: X = LEFT(.STRING,N). "LEFT" returns a pointer to the string, truncated to the length specified by 'N' which is defined as an integer.

8.07.02 RIGHT

This procedure is used in the basic form: X = RIGHT(.STRING,N). "RIGHT" returns a pointer to the last N characters of the specified string.

8.07.03 MID

This procedure is used in the basic form: X = MID(.STRING,N,M). "MID" returns a pointer to M characters of the string, starting at character N. To do the equivalent of BASIC's MID$(A$,I) use "MID(.STRING,N,32767)".

8.07.04 SAMPLE PROGRAM

The following program uses all of the functions in this library and should assist the user in generating his own constructions using them.

```
   INCLUDE 0.IOSUBS.LIB;
   INCLUDE 0.STRSUBS.LIB;
   INCLUDE 0.BASTRING.LIB;

   PROCEDURE EXAMPLE: BYTE .X, .Y, .Z, BUF(80);
      STRCOPY(.BUF,"ABCDEFGHIJKLMNOPQRSTUVWXYZ");
      CRLF;
      PRINT .BUF;
      SPACE 10;
      PRINT "ORIGINAL STRING\N";
      .X=LEFT(.BUF,18);
      PRINT .X;
      SPACE 18;
      PRINT ".X=LEFT(.BUF,18);\N";
      .Y=RIGHT(.X,10);
      PRINT .Y;
      SPACE 26;
      PRINT ".Y=RIGHT(.X,10);\N";
      .Z=MID(.Y,3,5);
      PRINT .Z;
      SPACE 31;
      PRINT ".Z=MID(.Y,3,5);\N";
      STRCOPY(.BUF,"ABCDEFGHIJKLMNOPQRSTUVWXYZ");
      CRLF;
      PRINT .BUF;
      SPACE 10;
      PRINT "Original String\N";
      PRINT MID(RIGHT(LEFT(.BUF,18),10),3,5);
      SPACE 31;
      PRINT "MID(RIGHT(LEFT(.BUF,18),10),3,5);\N";
   ENDPROC;
```

Page 1: "BASIC" STRING FUNCTIONS LIBRARY V:4.00 June 1 1984

```
0000  /*  "BASIC" STRING FUNCTIONS LIBRARY  V:4.00  */
0000
0000
0000  include 0.strsubs.lib;
013E
013E
013E  procedure left(byte .string: integer n);
013E     if n < 0
0140        then n = 0;
014C     if strlen(.string) > n
0155        then string(n) = 0;
0163  endproc .string;
0166
0166
0166  procedure right(byte .string: integer n):integer l;
0168     if n < 0
016A        then n = 0;
0176     l = strlen(.string);
0181     if n > l
0183        then n = l;
018D  endproc .string(l - n);
019A
019A
019A  procedure mid(byte .string: integer n, m):integer l;
019C     l = strlen(.string);
01A7     n = n - 1;
01AE     if n < 0
01B0        then n = 0;
01BC     if n > l
01BE        then n = l;
01C8     if m < 0
01CA        then m = 0;
01D6     if n + m > l
01DA        then m = l - n;
01E6     string(n + m) = 0;
01F0  endproc .string(n);
```

Page 2: "BASIC" STRING FUNCTIONS LIBRARY V:4.00 June 1 1984

PROCEDURES:

strlen	0003	INTEGER
strcopy	0025	INTEGER
strcat	0051	INTEGER
strcmp	006A	BYTE
strpos	00DF	INTEGER
left	013E	INTEGER
right	0166	INTEGER
mid	019A	INTEGER

DATA:

EXTERNALS:

GLOBALS:

8.08.00 FLEX LIBRARY

This library file contains procedures that help you to write programs that interface with the FLEX operating system. They are all ASMPROCs, generated by the MACE assembler, and perform the functions indicated. This library, like all of the other library modules, has been processed by 'LCASE' to convert the file to lower case. We mention this because MACE will normally generate upper-case 'GEN' statements and hex numbers. You may alter any of these routines to your own requirements or add more routines that you find useful. The MACE assembler (no apologies for the plug) has facilities for generating ASMPROCs which may save you some work.

8.08.01 FLEX

This is a routine that when called (just use the single word FLEX;) returns you to the operating system via the warm start entry point at $CD03. You could just as easily use JUMP $CD03; (but NOT JUMP FLEX; unless you remove the ASMPROC and put in CONSTANT FLEX=$CD03;). This routine is identical in function to the routine 'WARMS' in the IOSUBS library.

8.08.02 GET_FILENAME(.FCB)

This procedure requires the FLEX Line Buffer Pointer at $CC14 to contain the starting address of a string of characters that are presumed to be a valid filename. FCB is a 320 byte File Control Block that you must allocate (or you may use the System FCB at $C840-C97F). The ASMPROC calls the FLEX routine that checks the filename for validity and copies it into the name area of the FCB. If there is anything wrong with the filename, the FCB Error byte (the second byte into the FCB) will contain 21 (which will cause ILLEGAL FILE SPECIFICATION to be printed if REPORT_ERROR is called), otherwise the byte will be clear.

8.08.03 SET_EXTENSION(.FCB, CODE)

This procedure allows you to add one of the standard filename extensions (see your FLEX manual) if none was specified during a call to GET_FILENAME.

8.08.04 REPORT_ERROR(.FCB)

This procedure causes FLEX to print an error message dependant upon the value of the second byte of the FCB. If ERRORS.SYS is present on the system drive then an error message will be selected from that file, otherwise a numeric error code will be printed. The best way to use this function is put 'IF FCB(1) THEN REPORT_ERROR;' after every call to one of the other routines in this package. This will cause the error byte to be tested, and if non-zero an error message is printed. In all of the example programs that use FLEX.LIB you will find that at the start of the program there is a declaration:

```
AT $C840: BYTE FCB, ERROR(319); possibly followed by
AT $CC14: INTEGER LINE_POINTER;
```

8.08.04 REPORT_ERROR(.FCB) (continued)

The first of these declares the system FCB by a small trick that defines its length as 320 bytes total, 319 of them as error bytes! The effect, however, is that FCB is 320 bytes long and the second element (FCB(1)) is the error byte. You can then say IF ERROR THEN REPORT_ERROR after any call to a FLEX interface routine.

8.08.05 OPEN_FOR_READ(.FCB)

This procedure asks FLEX to open the file already specified in the FCB (by a call to GET_FILENAME) for reading.

8.08.06 OPEN_FOR_WRITE(.FCB)

This procedure asks FLEX to open a file for writing. The file must not already exist on the disk or an error will result.

8.08.07 SET_BINARY(.FCB)

This procedure tells FLEX that the file just opened is binary, not text.

8.08.08 READ(.FCB)

This is a function that reads the next byte from the (already opened) file specified by FCB. If an attempt is made to READ past the end of the file then the error byte will be set to 8. READ returns a BYTE value, being the character read from the file, so you can use CHAR=READ(.FCB).

8.08.09 WRITE(.FCB, CHAR)

This procedure writes the BYTE value CHAR to the (already open) file specified by FCB.

8.08.10 CLOSE_FILE(.FCB)

This procedure closes the file specified by FCB, whether it has been open for read or write.

8.08.11 READ_SECTOR(.FCB, DRIVE, TRACK, SECTOR)

This is a low-level routine that reads the sector at DRIVE, TRACK and SECTOR into the data area of the FCB. This routine should be used with caution and only if you understand what you are doing.

8.08.12 WRITE_SECTOR(.FCB, DRIVE, TRACK, SECTOR)

This is the corresponding write routine.

8.08.13 DELETE_FILE(.FCB)

This procedure deletes the specified file, preserving the filename in FCB (which FLEX would otherwise alter). This routine has its main use in opening an already-existing file for write.

8.08.14 RENAME_FILE(.FCB)

This procedure renames the file specified by FCB, using as the new name the contents of the FCB Scratch Bytes (see your FLEX manual). It is assumed that you will already have set up the FCB correctly.

```
Page  1:  FLEX INTERFACE LIBRARY  V:4.00                              June 1 1984

0000  /*  FLEX INTERFACE LIBRARY  V:4.00  */
0000
0000
0000  asmproc flex;
0003     gen $7e,$cd,$03;     /*          JMP   $CD03     */
0006
0006
0006  asmproc get_filename(integer);
0006     gen $ae,$62;         /*          LDX   2,S       */
0008     gen $bd,$cd,$2d;     /*          JSR   $CD2D     */
000B     gen $25,$02;         /*          BCS   *+4       */
000D     gen $6f,$01;         /*          CLR   1,X       */
000F     gen $39;             /*          RTS             */
0010
0010
0010  asmproc set_extension(integer, byte);
0010     gen $ae,$63;         /*          LDX   3,S       */
0012     gen $a6,$62;         /*          LDA   2,S       */
0014     gen $bd,$cd,$33;     /*          JSR   $CD33     */
0017     gen $39;             /*          RTS             */
0018
0018
0018  asmproc report_error(integer);
0018     gen $ae,$62;         /*          LDX   2,S       */
001A     gen $7e,$cd,$3f;     /*          JMP   $CD3F     */
001D
001D
001D  asmproc open_for_read(integer);
001D     gen $ae,$62;         /*          LDX   2,S       */
001F     gen $86,$01;         /*          LDA   #1        */
0021     gen $a7,$84;         /*          STA   ,X        */
0023     gen $7e,$d4,$06;     /*          JMP   FMS       */
0026
0026
0026  asmproc read(integer): byte;
0026     gen $34,$10;         /*          PSHS  X         */
0028     gen $ae,$64;         /*          LDX   4,S       */
002A     gen $bd,$d4,$06;     /*          JSR   FMS       */
002D     gen $1f,$89;         /*          TFR   A,B       */
002F     gen $35,$90;         /*          PULS  X,PC      */
0031
0031
0031  asmproc open_for_write(integer);
0031     gen $ae,$62;         /*          LDX   2,S       */
0033     gen $86,$02;         /*          LDA   #2        */
0035     gen $a7,$84;         /*          STA   ,X        */
0037     gen $7e,$d4,$06;     /*          JMP   FMS       */
003A
003A
003A  asmproc write(integer, byte);
003A     gen $ae,$63;         /*          LDX   3,S       */
003C     gen $a6,$62;         /*          LDA   2,S       */
003E     gen $7e,$d4,$06;     /*          JMP   FMS       */
0041
0041
```



```
0041  asmproc read_sector(integer, byte, byte, byte);
0041     gen $ae,$65;          /*          LDX   5,S        */
0043     gen $86,$09;          /*          LDA   #9         */
0045     gen $a7,$84;          /*          STA   ,X         */
0047     gen $a6,$64;          /*          LDA   4,S        */
0049     gen $a7,$03;          /*          STA   3,X        */
004B     gen $a6,$63;          /*          LDA   3,S        */
004D     gen $e6,$62;          /*          LDB   2,S        */
004F     gen $ed,$88,$1e;      /*          STD   30,X       */
0052     gen $7e,$d4,$06;      /*          JMP   FMS        */
0055
0055
0055  asmproc write_sector(integer, byte, byte, byte);
0055     gen $ae,$65;          /*          LDX   5,S        */
0057     gen $86,$0a;          /*          LDA   #10        */
0059     gen $a7,$84;          /*          STA   ,X         */
005B     gen $a6,$64;          /*          LDA   4,S        */
005D     gen $a7,$03;          /*          STA   3,X        */
005F     gen $a6,$63;          /*          LDA   3,S        */
0061     gen $e6,$62;          /*          LDB   2,S        */
0063     gen $ed,$88,$1e;      /*          STD   30,X       */
0066     gen $7e,$d4,$06;      /*          JMP   FMS        */
0069
0069
0069  asmproc set_binary(integer);
0069     gen $ae,$62;          /*          LDX   2,S        */
006B     gen $86,$ff;          /*          LDA   #$FF       */
006D     gen $a7,$88,$3b;      /*          STA   59,X       */
0070     gen $39;              /*          RTS              */
0071
0071
0071  asmproc close_file(integer);
0071     gen $ae,$62;          /*          LDX   2,S        */
0073     gen $86,$04;          /*          LDA   #4         */
0075     gen $a7,$84;          /*          STA   ,X         */
0077     gen $7e,$d4,$06;      /*          JMP   FMS        */
007A
007A
007A  asmproc delete_file(integer);
007A     gen $ae,$62;          /*          LDX   2,S        */
007C     gen $86,$0c;          /*          LDA   #12        */
007E     gen $a7,$84;          /*          STA   ,X         */
0080     gen $e6,$04;          /*          LDB   4,X        */
0082     gen $bd,$d4,$06;      /*          JSR   FMS        */
0085     gen $e7,$04;          /*          STB   4,X        */
0087     gen $39;              /*          RTS              */
0088
0088
0088  asmproc rename_file(integer);
0088     gen $ae,$62;          /*          LDX   2,S        */
008A     gen $86,$0d;          /*          LDA   #13        */
008C     gen $a7,$84;          /*          STA   ,X         */
008E     gen $7e,$d4,$06;      /*          JMP   FMS        */
```

Page 3: FLEX INTERFACE LIBRARY V:4.00 June 1 1984

PROCEDURES:

flex	0003	
get_filename	0006	
set_extension	0010	
report_error	0018	
open_for_read	001D	
read	0026	BYTE
open_for_write	0031	
write	003A	
read_sector	0041	
write_sector	0055	
set_binary	0069	
close_file	0071	
delete_file	007A	
rename_file	0088	

DATA:

EXTERNALS:

GLOBALS:

8.09.00 SCIPACK LIBRARY

A C K N O W L E D G E M E N T S

The algorithms used in this library were originally published in '68 Micro Journal and were developed by Ronald Anderson. Subsequent improvements by Ron Anderson have used the SINE and ARCTAN coefficents developed by Matt Scudiere. These were also published in '68 Micro Journal

The routines in this pack provide you with the common scientific functions encountered in engineering programs. Their accuracy is usually better than five significant (decimal) digits and they are reasonably fast, bearing in mind that you are working with an 8-bit micro, not a mainframe! Each of the functions returns a REAL value; the argument supplied may be of any type and is converted, as necessary, to REAL.

The technique used is that of polynomial approximation, whereby a polynomial (A+B*X+C*X*X+....) is computed that fits the function concerned over a limited range. Scaling is done to get the argument into the necessary range. This technique is used by nearly all BASIC interpreters and is more code-efficient than any other method, as well as being faster than most. The polynomial has no more than ten terms, and is arranged in such a way as to require only N multiplications and N additions.

The procedure _POLY does most of the work; it is passed the scaled operand and the address of the appropriate coefficient table, with a BYTE value to tell it how many terms to calculate.

8.09.01 LN(ARG)

Computes the natural (Naperian) logarithm of its argument. The argument must be positive and non-zero, otherwise LN returns zero. It is advisable for you to do your own error checking before calling any of these functions.

8.09.02 LOG(ARG)

Calculates the base 10 logarithm of its argument, by calling LN then multiplying by LOG(E).

8.09.03 EXP(ARG)

Exponentiates its argument, which must be between -88 and +87 or overflow will occur. In such a case the function returns the largest or smallest number it can.

8.09.04 ALOG(ARG)

Evaluates the base 10 antilog of the argument, by multiplying by LN(10) then calling EXP.

8.09.05 XTOY(ARG1, ARG2)

Is the only function that requires two arguments. Its purpose is to raise ARG1 to the power of ARG2, which it does by multiplying the natural log of ARG1 by ARG2 and then exponentiating. ARG2 can have any value, but ARG1 must be positive and non-zero. Note that it is possible to find the square root of a number using XTOY(ARG,0.5), but PL/9's built-in SQR function is much faster and somewhat more accurate.

8.09.06 SIN(ARG)

Returns the sine of the argument, which is assumed to be in radians.

8.09.07 COS(ARG)

Returns the cosine of the argument, also assumed to be in radians.

8.09.08 TAN(ARG)

Computes SIN(ARG)/COS(ARG) but does not check for COS(ARG) being zero.

8.09.09 ATN(ARG)

Computes the arctangent of the argument, giving the result in radians.

Page 1: SCIENTIFIC FUNCTIONS PACKAGE V:4.00 June 1 1984

```
0000  /*  SCIENTIFIC FUNCTIONS PACKAGE  V:4.00  */
0000
0000
0000  /*
0000                      ACKNOWLEDGEMENTS
0000                      ================
0000
0000     The routines in this package were largely written by
0000     Ron Anderson, using data supplied by  Matt  Scudiere
0000     and published in April 1983 Micro Journal.
0000
0000  */
0000
0000
0000  real _pio2 1.5707963;
0007  real _e 2.7182818;
000B  real _log2 0.69314718;
000F
000F  procedure _poly(real op, .table: byte count): real temp;
0011     temp = table(count);
043B     repeat
043B        count = count - 1;
043D        temp = temp * op + table(count);
0460     until count = 0;
0466  endproc real temp;
0470
0470
0470  real log_coeff
0470     0,
0474     0.9999964,
0478    -0.4998741,
047C     0.3317990,
0480    -0.2407338,
0484     0.1676541,
0488    -0.09532939,
048C     0.03608849,
0490    -0.006453544;
0494
0494  procedure ln(real op): byte n;
0496     if op <= 0
04A3        then return real 0;
04B7     gen $e6,$63;   /* LDB 3,S GET EXPONENT OF OP */
04B9     n = accb - 1;  /* ADJUST EXP TO BE 1 */
04BD     accb = 1;
04BF     gen $e7,$63;   /* STB 3,S PUT EXP BACK IN OP */
04C1  endproc real _poly(op - 1, .log_coeff,8) + n * _log2;
04F9
04F9
04F9  procedure log(real op);
04F9  endproc real ln(op) * 0.4342944;
0512
0512
```

```
Page  2:  SCIENTIFIC FUNCTIONS PACKAGE  V:4.00                    June 1 1984

0512  real exp_coeff
0512     0,
0516     0.9999999,
051A     0.4999999,
051E     0.1666700,
0522     0.04165734,
0526     0.00830140,
052A     0.00151500,
052E     0.000116;
0532
0532  procedure exp(real op): real k;
0534     if op > 87
0541        then return real 1E38;
0555     if op < -88
0565        then return real 1E-38;
0579     k = 1;
0586     while op > _log2
058B        begin
0599           op = op - _log2;
05AD           gen $6c,$e4;  /*  INC ,S    K=K*2  */
05AF        end;
05AF     while op < 0
05BE        begin
05C5           op = op + _log2;
05D9           gen $6a,$e4;  /*  DEC ,S    K=K/2  */
05DB        end;
05DB  endproc real (_poly(op, .exp_coeff,7) + 1) * k;
060B
060B
060B  procedure alog(real op);
060B  endproc real exp(op * 2.302585);
0625
0625
0625  procedure xtoy(real op1,op2);
0625     if op2 = 0
0632        then return real 1;
0644     if op1 < 0
0651        then return real 0;
0663  endproc real exp(ln(op1) * op2);
0681
0681
```

Page 3: SCIENTIFIC FUNCTIONS PACKAGE V:4.00 June 1 1984

```
0681 real sin_coeff
0681    1.0,
0685   -0.1666666,
0689    8.333332E-3,
068D   -1.9852E-4,
0691    2.8255E-6,
0695   -3.70E-8;
0699
0699 procedure sin(real op): byte negative, quadrant;
069B    if op = 0
06A8       then return real 0;
06BC    quadrant = fix(int(op / _pio2));
06D3    op = op - quadrant * _pio2;
06F0    negative = quadrant and 2;  /* NON-ZERO FOR QUADS 2,3 */
06F6    if quadrant and 1           /* IMPLICIT <> 0 */
06F8       then op = _pio2 - op;    /* QUADS 1,3 */
0713    op = op * _poly(op * op, .sin_coeff,5);
073E    if negative                 /* IMPLICIT <> 0 */
073E       then op = -op;
0752 endproc real op;
075C
075C
075C real cos_coeff
075C    1.0,
0760   -0.5,
0764    0.041666642,
0768   -1.3888397E-3,
076C    2.47609E-5,
0770   -2.605E-7;
0774
0774 procedure cos(real op): byte negative, quadrant;
0776    if op = _pio2
077B       then return real 0;
0796    quadrant = fix(int(op / _pio2));
07AD    op = op - quadrant * _pio2;
07CA    negative = 0;
07CC    if quadrant = 1 .or quadrant = 2
07DA       then negative = 1;
07EC    if quadrant and 1           /* IMPLICIT <> 0 */
07EE       then op = _pio2 - op;
0809    op = _poly(op * op, .cos_coeff,5);
082C    if negative                 /* IMPLICIT <> 0 */
082C       then op = -op;
0840 endproc real op;
084A
084A
084A procedure tan(real op);
084A endproc real sin(op) / cos(op);
0868
0868
```

```
Page  4:  SCIENTIFIC FUNCTIONS PACKAGE  V:4.00                    June 1 1984

0868  real atn_coeff
0868     1.0,
086C    -0.3333315,
0870     0.1999355,
0874    -0.1420890,
0878     0.1065626,
087C    -0.07528964,
0880     0.04290961,
0884    -0.01616574,
0888     0.002866226;
088C
088C  procedure atn(real op): byte sign, reciprocal;
088E     if op < 0
089B        then begin
08A2                op = -op;
08AF                sign = 1;;
08B3             end;
08B3        else sign = 0;
08B8     if op > 1
08C5        then begin
08CC                op = 1/op;
08E1                reciprocal = 1;;
08E5             end;
08E5        else reciprocal = 0;
08EA     op = op * _poly(op * op, .atn_coeff,8);
0915     if reciprocal               /* IMPLICIT <> 0 */
0915        then op = _pio2 - op;
0930     if sign                     /* IMPLICIT <> 0 */
0930        then op = -op;
0944  endproc real op;
```


Page 5: SCIENTIFIC FUNCTIONS PACKAGE V:4.00 June 1 1984

PROCEDURES:

_poly	000F	REAL
ln	0494	REAL
log	04F9	REAL
exp	0532	REAL
alog	060B	REAL
xtoy	0625	REAL
sin	0699	REAL
cos	0774	REAL
tan	084A	REAL
atn	088C	REAL

DATA:

_pio2	0003	REAL
_e	0007	REAL
_log2	000B	REAL
log_coeff	0470	REAL
exp_coeff	0512	REAL
sin_coeff	0681	REAL
cos_coeff	075C	REAL
atn_coeff	0868	REAL

EXTERNALS:

GLOBALS:

8.10.00 REALCON LIBRARY

This library file contains two routines that are essential when inputting or outputting REAL numbers via the system console. They convert between ASCII strings and the binary format used by PL/9 to hold REAL numbers.

8.10.01 BINARY(.STRINGPTR, .POSITIONPTR)

Is called with two arguments, both BYTE. The first is the address of (i.e. pointer to) the ASCII string that is to be converted into binary. The second argument, also a pointer, is the address of a BYTE location that contains the position in the string at which to start converting. Conversion finished whenever the routine comes across a character that it can't make sense of, for example a space or a null. When this happens, the second argument is left set to the position in the string at which this character was found, allowing the calling program to work its way along a number of items in a buffer.

BINARY returns a REAL value and can therefore be used in any REAL expression in the same way as any other REAL value. Examples of its use:

```
   INPUT(.BUFFER,80);             /* Get a line of input */
   POSITION = 0;                  /* Set up the starting position */
   X=BINARY(.BUFFER,.POSITION);   /* Convert the number */
```

A point to note is that since INPUT returns as its value the first argument it was given, the above can be simplified:

```
   POSITION=0;
   X=BINARY(INPUT(.BUFFER,80),.POSITION);
```

This combines the calls to the two routines.

If you frequently want to get a line of input and read a single number from it, use the following procedure:

```
   PROCEDURE INNUM:BYTE POS, BUF(20);
      POS=0;
   ENDPROC REAL BINARY(INPUT(.BUF,20),.POS);
```

See the routine 'FINPUT' in the 'REALIO' library for further information.

8.10.02 ASCBIN(.STRINGPTR)

This routine uses BINARY to convert an ASCII string pointed to by STRINGPTR to a REAL number. This procedure simplifies converting ASCII strings as the starting position pointer is automatically provided.

8.10.03 ASCII(VAR, .BUFFER)

Is the complementary routine that converts a REAL number into an ASCII string in the BYTE buffer whose address is supplied, terminating it with a null. The chief use for this routine is to print out a REAL number, which is achieved by the following procedure:

```
PROCEDURE OUTNUM(REAL X): BYTE BUF(20);
   PRINT(ASCII(X,.BUF));
ENDPROC;
```

See the routine 'FPRINT' in the 'REALIO' library for further information.

As with INPUT, ASCII returns as its "value" the second parameter passed to it; this is in the example then passed directly to PRINT. Note that PL/9 is happy to accept any of the following:

```
PRINT(ASCII(X,.BUF));                  PRINT ASCII(X,.BUF);
PRINT = ASCII(X,.BUF);
```

```
Page  1:  FLOATING-POINT CONVERSION ROUTINES  V:4.00              June 1 1984

0000  /*  FLOATING-POINT CONVERSION ROUTINES  V:4.00  */
0000
0000
0000  procedure binary(byte .buffer: byte .posptr):
0003     real acc: byte flag, msign, esign, exponent, expt, char;
0005     acc = 0;
0429     flag = 0;   <---------------- NOTE: SUDDEN  JUMP  IN PROGRAM SIZE
042B     msign = 0;                          IS CAUSED BY PL/9  GENERATING
042D     esign = 0;                          THE CODE FOR THE 'REAL' MATHS
042F     expt = 0;                           RUN TIME LIBRARY.
0431     exponent = 0;
0433     if buffer(posptr) = '-
043D        then begin
0443                msign = -1;
0447                posptr = posptr + 1;
044A             end;
044A  loop:
044A     char = buffer(posptr);
0456     posptr = posptr + 1;
0459     while char >=  '0 .and char <= '9
0467        begin
0475           acc = acc * 10 + (char - '0);
049C           if flag                    /* IMPLICIT <> 0 */
049C              then exponent = exponent - 1;
04A5           char = buffer(posptr);
04B1           posptr = posptr + 1;
04B4        end;
04B4
04B4     if char = '. .and flag = 0
04C4        then begin
04D2                flag = -1;
04D6                goto loop;
04D9             end;
04D9
04D9     if char = 'E .or char = 'e
04E7        then begin
04F5                if buffer(posptr) = '-
04FF                   then begin
0505                           esign = -1;
0509                           posptr = posptr + 1;
050C                        end;
050C                char = buffer(posptr);
0518                posptr = posptr + 1;
051B                while char >= '0 .and char <= '9
0529                   begin
0537                      expt = expt * 10 + char - '0;
056E                      char = buffer(posptr);
057A                      posptr = posptr + 1;
057D                   end;
057D                if esign              /* IMPLICIT <> 0 */
057F                   then expt = -expt;
058E                exponent = exponent + expt;
0594             end;
0594
0594     posptr = posptr - 1;
```

Page 2: FLOATING-POINT CONVERSION ROUTINES V:4.00 June 1 1984

```
0597      while exponent > 0
0599         begin
059F            acc = acc * 10;
05B4            exponent = exponent - 1;
05B6         end;
05B6      while exponent < 0
05BA         begin
05C0            acc = acc/10;
05D5            exponent = exponent + 1;
05D7         end;
05D7      if msign                          /* IMPLICIT <> 0 */
05D9         then acc = -acc;
05ED   endproc real acc;
05F7
05F7
05F7   procedure ascbin(byte .buffer):byte posptr;
05F9      posptr = 0;
05FB   endproc real binary(.buffer, .posptr);
060F
060F
060F   procedure ascii(real op: byte .buffer):
060F      integer pointer:
060F      byte ptr, exponent, count, carry, first, last, digit;
0611      pointer = .op;
0617      exponent = -1;
061B      ptr = 0;
061D      if op = 0
062A         then begin
0631                 buffer = '0;
0636                 buffer(1) = 0;
063F                 return .buffer;
0644              end;
0644      if op < 0
0651         then begin
0658                 op = -op;
0666                 buffer(ptr) = '-;
0671                 ptr = ptr + 1;
0673              end;
0673      first = ptr;
0677      while op >=  1
0684         begin
068B            op = op/10;
06A1            exponent = exponent + 1;
06A3         end;
06A3      while op < 0.1
06B2         begin
06B9            op = op * 10;
06CF            exponent = exponent - 1;
06D1         end;
06D1
```

```
Page  3:  FLOATING-POINT CONVERSION ROUTINES  V:4.00            June 1 1984

06D1  /* DE-NORMALIZE THE REAL NUMBER */
06D1
06D1     gen $ae,$e4;         /* DENORM   LDX   ,S        */
06D5     gen $68,$03;         /*          ASL   3,X       */
06D7     gen $69,$02;         /*          ROL   2,X       */
06D9     gen $69,$01;         /*          ROL   1,X       */
06DB     gen $6d,$84;         /*          TST   ,X        */
06DD     gen $27,$0a;         /*          BEQ   *+12      */
06DF     gen $64,$01;         /* :1       LSR   1,X       */
06E1     gen $66,$02;         /*          ROR   2,X       */
06E3     gen $66,$03;         /*          ROR   3,X       */
06E5     gen $6c,$84;         /*          INC   ,X        */
06E7     gen $26,$f6;         /*          BNE   :1        */
06E9
06E9  /* GENERATE SEVEN DECIMAL DIGITS BY REPEATED MULTIPLICATION BY TEN */
06E9
06E9     count = 7;
06ED     repeat
06ED        gen $ae,$e4;         /* DIGIT    LDX   ,S        */
06EF        gen $ec,$84;         /*          LDD   ,X        */
06F1        gen $34,$06;         /*          PSHS  D         */
06F3        gen $ec,$02;         /*          LDD   2,X       */
06F5        gen $8d,$16;         /*          BSR   DOUBLE    */
06F7        gen $8d,$14;         /*          BSR   DOUBLE    */
06F9        gen $e3,$02;         /*          ADDD  2,X       */
06FB        gen $ed,$02;         /*          STD   2,X       */
06FD        gen $35,$06;         /*          PULS  D         */
06FF        gen $e9,$01;         /*          ADCB  1,X       */
0701        gen $a9,$84;         /*          ADCA  ,X        */
0703        gen $ed,$84;         /*          STD   ,X        */
0705        gen $8d,$06;         /*          BSR   DOUBLE    */
0707        gen $e6,$84;         /*          LDB   ,X        */
0709        gen $6f,$84;         /*          CLR   ,X        */
070B        gen $20,$09;         /*          BRA   *+11      */
070D
070D        gen $68,$03;         /* DOUBLE   ASL   3,X       */
070F        gen $69,$02;         /*          ROL   2,X       */
0711        gen $69,$01;         /*          ROL   1,X       */
0713        gen $69,$84;         /*          ROL   ,X        */
0715        gen $39;             /*          RTS             */
0716
0716        digit = accb;
0718        buffer(ptr) = digit + '0;
0725        ptr = ptr + 1;
0727        count = count - 1;
0729     until count = 0;
072F
072F     buffer(ptr) = 0;
0738     ptr = ptr - 1;
073A     last = ptr;
073E     carry = 0;
0740     if buffer(ptr) >= '5
0749        then carry = 1;
```

```
0753      repeat
0753         ptr = ptr - 1;
0755         buffer(ptr) = buffer(ptr) + carry;
0762         if buffer(ptr) > '9
076B            then buffer(ptr) = '0;
077C            else carry = 0;
0781      until carry = 0 .or ptr = first;
079B
079B      if carry                              /* IMPLICIT <> 0 */
079B         then begin
07A2                 count = last + 1;
07A8                 repeat
07A8                    buffer(count+1) = buffer(count);
07BE                    count = count - 1;
07C0                 until count < ptr;
07C6                 buffer(ptr) = '1;
07D1                 exponent = exponent + 1;
07D3              end;
07D3      buffer(last) = 0;
07DC
07DC      if exponent > 5 .or exponent < -2
07EA         then begin
07F8                 count = last + 1;
07FE                 repeat
07FE                    buffer(count+1) = buffer(count);
0814                    count = count - 1;
0816                 until count = first;
081C                 buffer(count+1) = '.;
0829                 while buffer(last) = '0
0832                    last = last - 1;
083A                 if buffer(last) = '.

0845                    then last = last - 1;
084D                 buffer(last+1) = 'E;
085A                 last = last + 2;
0860                 if exponent < 0
0862                    then begin
0868                            exponent = -exponent;
0870                            buffer(last) = '-;
087B                            last = last + 1;
087D                         end;
087D                 if exponent > 9
087F                    then begin
0885                            buffer(last) = exponent/10 + '0;
08FA                            last = last + 1;
08FC                         end;
08FC                 buffer(last) = exponent - exponent/10 * 10 + '0;
0925                 buffer(last+1) = 0;
0930                 return .buffer;
0935              end;
0935
```

```
Page  5:  FLOATING-POINT CONVERSION ROUTINES  V:4.00             June 1 1984

0935        else if exponent >= 0
093A                then begin
0940                        ptr = first + 1;
0946                        while exponent > 0
0948                           begin
094E                              ptr = ptr + 1;
0950                              exponent = exponent - 1;
0952                           end;
0952                        count = last;
0958                        while count >= ptr
095A                           begin
0960                              buffer(count+1) = buffer(count);
0976                              count = count - 1;
0978                           end;
0978                        buffer(ptr) = '.;
0985                     end;
0985
0985                else begin
0988                        if exponent = -2
098A                           then begin
0990                                   count = last + 1;
0996                                   repeat
0996                                      buffer(count+1) = buffer(count);
09AC                                      count = count - 1;
09AE                                   until count < first;
09B4                                   last = last + 1;
09B6                                   buffer(first) = '0;
09C1                                end;
09C1                        count = last + 1;
09C7                        repeat
09C7                           buffer(count+2) = buffer(count);
09DD                           count = count - 1;
09DF                        until count < first;
09E5                        last = last + 1;
09E7                        buffer(first) = '0;
09F2                        buffer(first+1) = '.;
09FF                     end;
09FF
09FF     while buffer(last) = '0
0A08        begin
0A0E           buffer(last) = 0;
0A17           last = last - 1;
0A19        end;

0A19     if buffer(last) = '.
0A24        then buffer(last) = 0;
0A33  endproc .buffer;
```


Page 6: FLOATING-POINT CONVERSION ROUTINES V:4.00 June 1 1984

PROCEDURES:

binary	0003	REAL
ascbin	05F7	REAL
ascii	060F	INTEGER

DATA:

EXTERNALS:

GLOBALS:

8.11.00 REAL I/O LIBRARY

This library provides two procedures to simplify console input and output of REAL numbers. These procedures are the two procedures outlined in the description of the REALCON library. The reason we placed these two procedures in a separate library was to maintain compatibility with older versions of the compiler libraries.

8.11.01 FINPUT

This procedure will expect to get a line of input from the terminal representing a real number. e.g. 23456932, 0.123456, -456.986, 2.47609E-5, -1.388457E-3, etc. The ASCII string input will be returned by this procedure as a REAL number which may be evaluated or assigned as required by your application.

8.11.02 FPRINT

This procedure does the inverse of the above. It is passed a REAL number and prints it out on the system console. If the REAL number is greater than or equal to 1,000,000 or less than 0.01 the response will be scientific notation. If the number falls between these limits the response will be in decimal with all trailing zeros truncated (100.0000) will be 100.

Page 1: FLOATING-POINT INPUT/OUTPUT ROUTINES V:4.00 June 1 1984

```
0000  /*  FLOATING-POINT INPUT/OUTPUT ROUTINES  V:4.00  */
0000
0000
0000  include 0.iosubs.lib;
02E8  include 0.realcon.lib;
0CC0
0CC0
0CC0  procedure finput: byte ptr, buffer(20);
0CC3     ptr = 0;
0CC5  endproc real binary(input(.buffer, 20), .ptr);
0CE5
0CE5
0CE5  procedure fprint(real op): byte buffer(20);
0CE8     print(ascii(op, .buffer));
0CFE  endproc;
```

```
Page  2:  FLOATING-POINT INPUT/OUTPUT ROUTINES  V:4.00           June 1 1984

PROCEDURES:

monitor             0003
warms               0008
getchar             000C   BYTE
getchar_noecho      0012   BYTE
getkey              0019   BYTE
convert_lc          0031   BYTE
get_uc              0056   BYTE
get_uc_noecho       0063   BYTE
putchar             0070
printint            0078
remove_char         011D
input               0139   INTEGER
crlf                01DD
print               01F0
space               02CC
binary              02E8   REAL
ascbin              08DC   REAL
ascii               08F4   INTEGER
finput              0CC0   REAL
fprint              0CE5

DATA:

EXTERNALS:
nul                 0000
abt                 0003
bel                 0007
bs                  0008
lf                  000A
cr                  000D
can                 0018
esc                 001B
sp                  0020

GLOBALS:
```

8.12.00 NUMCON LIBRARY

This library provides procedures to simplify console input and output of INTEGERs.

8.12.01 BINTODEC

BINTODEC is an assembler routine that converts a supplied integer into an ASCII decimal number between 0 and 65535 and puts it into the buffer whose address is supplied, terminating it with a NULL.

8.12.02 PRDEC

PRDEC prints an integer as a decimal number between 0 and 65535.

8.12.03 PRNUM

PRNUM prints an integer as a signed decimal number between -32768 and 32767.

8.12.04 GETNUM

GETNUM works its way along the text in the buffer whose address is supplied and returns the number contained therein. The number may have a leading minus sign; a $ signifies hexadecimal. Examples: 25 -9003 $1234 -$A7 0 50817.

8.12.05 TRY THIS

If you want to get some insight into how these routines work try entering the following and running it under the PL/9 tracer:

```
include 0.trufalse.def;
include 0.iosubs.lib;
include 0.numcon.lib;

procedure test:byte buffer(20):integer i;
   i = getnum(input(.buffer,20));
   crlf; crlf;
   prnum(i);
   space(5);
   prdec(i);
```

```
Page  1:  NUMERICAL CONVERSION AND I/O ROUTINES  V:4.00           June 1 1984

0000  /*  NUMERICAL CONVERSION AND I/O ROUTINES  V:4.00  */
0000
0000
0000  include 0.trufalse.def;
0000  include 0.iosubs.lib;
02E8
02E8
02E8  asmproc bintodec(integer,integer);
02E8     gen $cc,$00,$04;     /*          LDD   #$0004    */
02EB     gen $34,$26;         /*          PSHS  D,Y       */
02ED     gen $ae,$66;         /*          LDX   6,S       */
02EF     gen $ec,$68;         /*          LDD   8,S       */
02F1     gen $31,$8c,$2b;     /*          LEAY  TABLE,PCR */
02F4     gen $6f,$84;         /* L1       CLR   ,X        */
02F6     gen $63,$84;         /*          COM   ,X        */
02F8     gen $6c,$84;         /* L2       INC   ,X        */
02FA     gen $a3,$a4;         /*          SUBD  ,Y        */
02FC     gen $24,$fa;         /*          BCC   L2        */
02FE     gen $e3,$a1;         /*          ADDD  ,Y++      */
0300     gen $34,$04;         /*          PSHS  B         */
0302     gen $e6,$84;         /*          LDB   ,X        */
0304     gen $26,$04;         /*          BNE   L3        */
0306     gen $6d,$61;         /*          TST   1,S       */
0308     gen $27,$04;         /*          BEQ   L4        */
030A     gen $8d,$0e;         /* L3       BSR   L5        */
030C     gen $6c,$61;         /*          INC   1,S       */
030E     gen $35,$04;         /* L4       PULS  B         */
0310     gen $6a,$61;         /*          DEC   1,S       */
0312     gen $26,$e0;         /*          BNE   L1        */
0314     gen $8d,$04;         /*          BSR   L5        */
0316     gen $6f,$84;         /*          CLR   ,X        */
0318     gen $35,$a6;         /*          PULS  D,Y,PC    */
031A     gen $cb,$30;         /* L5       ADDB  #$30      */
031C     gen $e7,$80;         /*          STB   ,X+       */
031E     gen $39;             /*          RTS             */
031F     gen $27,$10,$03,$e8; /* TABLE    FCB   $27,$10,$03,$E8 */
0323     gen $00,$64,$00,$0a; /*          FCB   $00,$64,$00,$0A */
0327
0327
0327  procedure prdec(integer n): byte buffer(6);
0329     bintodec(n, .buffer);
0335     print(.buffer);
033E  endproc;
0341
0341
0341  procedure prnum(integer n): byte buffer(6);
0343     if n < 0
0345        then begin
034C                putchar('-);
0355                n = -n;
035D             end;
035D     bintodec(n, .buffer);
0369     print(.buffer);
0372  endproc;
0375
```

Page 2: NUMERICAL CONVERSION AND I/O ROUTINES V:4.00 June 1 1984

```
0375
0375  procedure getnum(byte .buffer):
0375     integer ptr, n:
0375     byte sign, base, char, done;
0377     ptr = 0;
037C     sign = false;
037E     n = 0;
0383     base = 10;
0387     done = false;
0389     while buffer(ptr) = sp
0391        ptr = ptr + 1;
039E     if buffer(ptr) = '-
03A8        then begin
03AE                sign = true;
03B2                ptr = ptr + 1;
03B9             end;
03B9     if buffer(ptr) = '$
03C1        then begin
03C7                base = 16;
03CB                ptr = ptr + 1;
03D2             end;
03D2     repeat
03D2        char = buffer(ptr) - '0;
03DE        if char > 9 .and char < 17
03EC           then done = true;
03FE        if char > 16
0400           then if base = 10
0408                   then done = true;
0412                   else char = char - 7;
041B        if char < 0 .or char > $f
0429           then done = true;
043B        if not(done)           /* IMPLICIT <> 0 */
043E           then begin
0443                   n = n * base + char;
0476                   ptr = ptr + 1;
047D                end;
047D     until done;
0484     if sign                   /* IMPLICIT <> 0 */
0484        then n = -n;
0493  endproc n;
```

Page 3: NUMERICAL CONVERSION AND I/O ROUTINES V:4.00 June 1 1984

```
PROCEDURES:
monitor          0003
warms            0008
getchar          000C   BYTE
getchar_noecho   0012   BYTE
getkey           0019   BYTE
convert_lc       0031   BYTE
get_uc           0056   BYTE
get_uc_noecho    0063   BYTE
putchar          0070
printint         0078
remove_char      011D
input            0139   INTEGER
crlf             01DD
print            01F0
space            02CC
bintodec         02E8
prdec            0327
prnum            0341
getnum           0375   INTEGER

DATA:

EXTERNALS:
true             FFFF
false            0000
mem              0000   BYTE
nul              0000
abt              0003
bel              0007
bs               0008
lf               000A
cr               000D
can              0018
esc              001B
sp               0020

GLOBALS:
```


8.13.00 SORT LIBRARY

A C K N O W L E D G E M E N T

These routines are based on a series of sorting algorithms presented in the May 1983 issue of BYTE magazine, from page 482 onwards.

This library is provided as a primitive to enable you to develop more complex sorting routines. As supplied the library sorts a VECTOR of REAL numbers into ascending order. It can be easily modified to sort a VECTOR of INTEGERs or a VECTOR of BYTEs.

The standard form is:

```
SHELLSORT(.VECTOR, NUMBER_OF_ELEMENTS_TO_BE_SORTED);
```

```
Page  1:  SHELL SORT LIBRARY ROUTINE  V:4.00                      June 1 1984

0000  /*  SHELL SORT LIBRARY ROUTINE  V:4.00  */
0000
0000
0000  procedure shellsort(real .data: integer size):
0003     integer i, j, d:
0003     real temp;
0005     i = size;
0009     d = 16383;
000E     while i < 16384
0010        begin
0017           d = shift(d,-1);
001D           i = shift(i,1);
0023        end;
0023     repeat
0025        i = 0;
002A        repeat
002A           j = i;
002E           repeat
002E              if data(j) <= data(j + d)
0462                 then break;
046B              temp = data(j);
047D              data(j) = data(j + d);
04A0              data(j + d) = temp;
04BA              j = j - d;
04C0           until j < 0;
04C9           i = i + 1;
04D0        until i = size - d;
04E2        d = shift(d,-1);
04E8     until d = 0;
04F1  endproc;
```


Page 2: SHELL SORT LIBRARY ROUTINE V:4.00 June 1 1984

PROCEDURES:
shellsort 0003

DATA:

EXTERNALS:

GLOBALS:

THIS PAGE INTENTIONALLY LEFT BLANK

