7.02.00 INTERFACING WITH ASSEMBLY LANGUAGE (a plug for MACE)

Although PL/9 produces quite efficient code, there are always areas in which improvements could be made by "hand patching". Since the compiler produces object code directly from source in one pass, without an intermediate assembler stage, it is not possible to edit the resulting code (unless a disassembler is used). As this may have imposed unacceptable restrictions on advanced assembly language programmers PL/9 has the capability of accepting in-line assembly language code via the GEN statement (q.v.).

The MACE assembler has three special features that enable the programmer write his own assembly language routines that can be embedded (or INCLUDEd) in a PL/9 program:

(1) The assembler allows comment lines to start with a + as well as the usual *.

(2) There is an extra G option that causes the assembler output to take the form of a series of GEN statements. Those comment lines that started with a + will be delivered intact (but with the + removed) to the output file.

(3) The assembly language source lines will be passed over to the output file as comments against the GEN statements.

For example, the following is a section of a FLEX interface package:

```
+ /* FLEX INTERFACE LIBRARY  V:4.00  */
+
+ASMPROC FLEX;
 JMP $CD03
+
+ASMPROC GET_FILENAME(INTEGER);
 LDX 2,S
 JSR $CD2D
 BCS *+4
 CLR 1,X
 RTS
```

etc. If the assembler command "A:G,L=FLEXIO.LIB" is given, a file called FLEXIO.LIB will be created with the following contents:

```
/* FLEX INTERFACE LIBRARY  V:4.00  */

ASMPROC FLEX;
   GEN $7E,$CD,$03;     /*          JMP   $CD03     */

ASMPROC GET_FILENAME(INTEGER);
   GEN $AE,$62;         /*          LDX   2,S       */
   GEN $BD,$CD,$2D;     /*          JSR   $CD2D     */
   GEN $25,$02;         /*          BCS   *+4       */
   GEN $6F,$01;         /*          CLR   1,X       */
   GEN $39;             /*          RTS             */
```

etc. This file can be incorporated into a PL/9 program or included at compile-time by the statement "INCLUDE FLEXIO".

Important: ASMPROCs must be position-independent if (1) you want the program that uses them to be position-independent and (2) for the tracer to operate properly. In any case a good practice to always write position-independent code for the 6809 since you never know when it might be necessary!

7.03.00 PL/9 VARIABLES

This section will outline the details of, and rules pertaining to, the variables and mathematical expressions in PL/9.

7.03.01 ARITHMETIC QUANTITIES

There are three data sizes recognized by PL/9, viz. BYTE, INTEGER and REAL. BYTE and INTEGER are similar to the 8- and 16-bit quantities used in assembly language programming, there are subtle differences however. REALs provide the programmer with a 32-bit floating point variable.

BYTE is a signed 8-bit number, having a range -128 to +127.

INTEGER is a signed 16-bit number having a range -32768 to +32767.

REAL numbers are 4-byte quantities comprising a signed 8-bit exponent and a signed 24-bit mantissa; they are able to represent numbers between +/-1E-38 and +/-1E37, with a precision of some six to seven decimal digits.

Mixed mode arithmetic, that is where BYTEs, INTEGERs and REALs occur in the same expression, are allowed by PL/9; no checks are made that the operations make sense, it being assumed that the programmer knows what he is doing.

The compiler handles BYTE quantities in the 'B' accumulator and INTEGERs in the 'D' accumulator; conversion between one and the other is therefore a matter either of ignoring the contents of the 'A' accumulator, or of sign extending 'B' into 'A'. REAL numbers are handled by 'D' and 'X'. This structure greatly simplifies the code required to return variables via the ENDPROC and RETURN statements as well as simplifying the code required to compare a byte quantity with an integer quantity.

Vectors, that is, single dimension arrays of one of the three data types, are allowed by PL/9; these may have any number of elements up to 32767 (if memory size allows). To keep run-time overheads as low as possible, no checks are made that a reference to an vector element is consistent with the defined size of the vector, or even that the index is positive - some programmers may indeed have a use for negative vector indices!

Pointers, that is variables that contain addresses of where information is stored, are always 16-bit quantities although they may point to BYTEs, INTEGERs or REALs.

```
* * * * * * *
*  N O T E  *
* * * * * * *
```

It is very important for you to remember that PL/9 treats BYTEs and INTEGERs as SIGNED numbers. When BYTEs are compared with INTEGERs or BYTE quantities are assigned to INTEGERs the BYTE will be sign extended before the operation takes place. This can take a bit of getting used to if you are an assembly language programmer.

The signed nature of the arithmetic in PL/9 has, in the past, caused assembly language programmers the greatest deal of difficulty. PL/9 versions 3.XX onward have improved handling of unsigned numbers, and will, we hope, eliminate a lot of the confusion in this area.

7.03.02 UNSIGNED BYTES AND INTEGERS

As mentioned in the last section PL/9 automatically and inexorably defaults to signed arithmetic when evaluating and assigning BYTE and INTEGER quantities. This is fine when you are performing general purpose arithmetic, flag comparisons, and handling A-D and D-A data. It does, however, pose problems when evaluating or assigning binary bit patterns. These patterns are, for the most part, unsigned. In these circumstances you are usually more interested in the value of the variable as a bit pattern than you are in the signed number this bit pattern may represent.

To provide the programmer with an extra degree of flexibility when performing bit oriented I/O work the following unsigned types have been provided:

BYTE in the range of 0 to 255.

INTEGER in the range of 0 to 65,535.

Since the compiler inherently produces code for signed evaluations and assignments, special mechanisms had to be incorporated to FORCE unsigned evaluations and assignments when desired.

The mechanisms we have adopted are straightforward and easy to use once you understand the principles involved. In order to use unsigned quantities successfully you must THOROUGHLY understand the mechanisms used to FORCE unsigned operations. The point that you should take note of is the fact that these mechanisms are designed to OVERRIDE a fundamental part of the compiler. Like any 'bad habit' that you try to suppress the compiler will switch into signed operations with the slightest excuse.

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                                                                       *
*  Failing to understand what is said in this section will              *
*  UNDOUBTEDLY result in programs that are designed to work with        *
*  unsigned numbers FAILING to achieve the desired results.             *
*                                                                       *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

There are three mechanisms provided in this release of PL/9 to assist you in working with unsigned numbers:

(1) When HEX numbers are treated as numbers between 0 - 255 or 0 - 65,535 except where a BYTE value is assigned to an INTEGER. If you perform this type of operation PL/9 will ALWAYS sign extend the BYTE before it assigns it. If you wish to assign an unsigned BYTE to an unsigned INTEGER you must use the "INTEGER" function (see 3 below). A hex number with one or two digits (e.g. $A, $7E) is considered to be a BYTE and a hex number with three or four digits (e.g. $1A5, $FF7F) is considered to be an INTEGER.

(2) When an expression is preceded with an exclamation mark (!) the compiler will EVALUATE the expression as unsigned regardless of complexity PROVIDED that the expression does not include multiplication (*) division (/) or modulus (\).

7.03.02 UNSIGNED BYTES AND INTEGERS (continued)

This symbol can also be used whenever you want an unsigned EVALUATION but are in doubt about the mixture of BYTEs and INTEGERs in your evaluation. If you had the evaluation right in the first place putting the exclamation mark in will not cause a single extra byte of code to be generated, BUT if the expression was not structured properly for unsigned operations putting the exclamation mark in will force the compiler to do what you want, rather thanwhat you say!

(3) The INTEGER function which is used to ASSIGN an unsigned BYTE to an unsigned INTEGER by setting the top 8-bits of the resulting INTEGER to zero. When an INTEGER is assigned to a BYTE the top 8-bits are ignored thus the result is automatically unsigned.

These mechanisms are explained in more detail in the subsequent sections.

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                                                                       *
*  IF  YOU  INTEND  TO  WORK WITH UNSIGNED NUMBERS READ THE NEXT        *
*  THREE SECTIONS VERY CAREFULLY. THEY CONTAIN SOME VERY  SUBTLE        *
*  COMMENTS ON THE WAY PL/9 HANDLES UNSIGNED NUMBERS.                   *
*                                                                       *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

HEX NUMBERS

When evaluating bit patterns most programmers will automatically use hexadecimal numbers. For example if an 8-bit digital I/O port presented '1000 0000' most programmers, 99% at least, would compare it with $80, NOT -128! Likewise when you wished to assign the bit pattern '1111 1111' you would assign $FF, NOT -1!

PL/9 takes advantage of this fact in the way it treats CONSTANTs, BYTEs, and INTEGERs assigned with, or compared with, HEX numbers. In these circumstances the compiler will assume that if the HEX number has one or two digits (e.g. $E, $1C) that it is a BYTE but if it has three or four (e.g. $1E2, $FFEF) it is an INTEGER.

It is extremely important to note that INTEGER assignments via HEX numbers must be expressed fully. This rule applies to both INTEGER constants and variables. TO THE COMPILER $80 IS NOT THE SAME AS $0080. The compiler will treat the former as a BYTE and sign extend it to form $FF80 before assigning it to, or comparing it with, an INTEGER. If you want to declare an INTEGER value of less than three significant digits you have to EXPLICITLY say so with leading zeros.

PL/9 will treat HEX numbers (i.e. numbers prefixed with '$') as unsigned only when LIKE SIZED quantities are involved in an equal or not equal comparison or when LIKE SIZED quantities are involved in a data assignment. The key words here are 'like sized'. PL/9 will treat HEX numbers just like any other number if you mix INTEGERs and BYTEs in the expression or perform comparisons other than equal/not equal ('=', '<>').

When the >, >=, <, or <= evaluators are involved in a comparison OR the expression contains a comparison between BYTEs and INTEGERs (or vice versa) OR an unsigned addition or subtraction is to be performed, the exclamation mark (q.v.) MUST be used in the expression to force an unsigned evaluation.

Just to confuse the issue, simple EVALUATIONS involving either BYTE or INTEGER variables or constants and HEX numbers do not cause any sign extension. We are not telling you this to encourage you to take advantage of this fact but rather to tell you what happens if you get it slightly wrong.

For example, suppose an integer variable called INTVAR contained $0080. Both of the examples below work as you would expect:

```
IF INTVAR = $0080 THEN...    or    IF INTVAR = $80 THEN...
```

So far so good. Just as you are likely to get complacent the compiler throws a spanner in the works; for, if the expressions are reversed, the compiler takes the second form to mean a comparison is to be made between a BYTE and an INTEGER and will sign extend the BYTE to form $FF80 before the comparison is made. Only the example on the left will work. Due to the sign extension of $80 in the example on the right it will NOT work.

```
IF $0080 = INTVAR THEN...          IF $80 = INTVAR THEN...
```

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                                                                     *
*         INTEGER quantities should ALWAYS be expressed fully.        *
*                                                                     *
*  If you mean $000F then don't say $F, they are not the same thing!  *
*                                                                     *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

A series of examples are presented at the end of this section.

THE EXCLAMATION MARK (!)

To permit the mixing of BYTES and INTEGERS and the use of the greater/less than comparisons a special mechanism (the exclamation mark !) has been provided in PL/9 to assist in the EVALUATION of BYTEs and INTEGERs as unsigned quantities.

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                                                                       *
*  The exclamation mark (!) has one meaning and one meaning only.       *
*                                                                       *
*  It tells the compiler not to sign extend any of the BYTE             *
*  quantities in the following EVALUATION. No more. No less.            *
*                                                                       *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

The key word in the last sentence was 'EVALUATION'. The (!) has absolutely no effect in data assignments that do not contain an evaluation expression before the data assignment. For example, the following expressions mean the same thing, and sign extension of BYTEVAR will take place in both of them.

```
INTVAR = BYTEVAR;                INTVAR = !BYTEVAR;
```

If you wish to assign an unsigned byte or the result of a function that returns a byte to an unsigned integer you MUST use the INTEGER function (q.v.).

However if the compiler must evaluate a mathematical or bit operation between BYTEs and INTEGERs before assigning the result to an INTEGER an exclamation mark WILL make a difference. For example:

```
INTVAR = BYTEVAR + IVAR2;          INTVAR = !BYTEVAR + IVAR2;
INTVAR = BYTEVAR AND IVAR2;        INTVAR = !BYTEVAR AND IVAR2;
```

The expressions on the left mean something entirely different from those on the right. In the expressions on the left BYTEVAR will be sign extended before the operation takes place. In the expressions on the right BYTEVAR will not be sign extended before the operation takes place.

Another area where the (!) MUST be used to force unsigned operation is in ANY expression that involves the (>), (>=), (<), or (<=) evaluators. This rule holds for ALL cases, even if only BYTE quantities or only INTEGER quantities are involved. The (!) must always be used. For example:

```
IF $7F < $80 THEN...
```

will never pass the test! The $80 will be sign extended to form $FF80 and then compared with $007F. The correct construction is:

```
IF !$7F < $80 THEN...
```

Other examples are:

```
IF !$FF > $7F THEN...                IF !BVAR1 < BVAR2 THEN...

IF !BVAR1 > IVAR1 THEN...            IF !BVAR >= IVAR THEN...

IF !BVAR > $7F THEN...               IF !IVAR < $9FFF THEN...

IF !$80 = IVAR THEN...               IF !BVAR = IVAR THEN...

IF !BVAR1 < BVAR2 + BVAR3 THEN ...   (where the sum will overflow a signed
                                      BYTE i.e. > +127 or < -128)
```

INTEGER

This PL/9 function should be used in ANY assignment where a BYTE quantity is being assigned to an INTEGER variable and you wish to have the operation preserve the unsigned value of the BYTE.

For example:

```
BYTEVAR = $FF;
INTVAR  = BYTEVAR;
```

will result in the value $FFFF being assigned to INTVAR. To perform the assignment in an unsigned manner and give the result $00FF the following expression should be used:

```
INTVAR = INTEGER(BYTEVAR);
```

The INTEGER function is particularly important if you are assigning the result of a function procedure (q.v.) to a variable. Suppose, for example, you had a function procedure named READ_PORT that reads an 8-bit I/O port and returns the value of that port as a BYTE. If you assigned it to an INTEGER as follows:

```
INTVAR = READ_PORT;
```

The resulting value in INTVAR would be a sign extended version of what READ_PORT returned. This may not be desirable in many circumstances. In these cases the the following form should be used:

```
INTVAR = INTEGER(READ_PORT);
```

This function is also discussed in section 7.06.00.

EXAMPLES OF UNSIGNED OPERATIONS

```
0001 INCLUDE IOSUBS;
0002
0003     CONSTANT A=$80, B=$7F, C=$0080, D=$007F;
0004
0005
0006 PROCEDURE TEST:BYTE B1,B2,B3,B4:INTEGER I1,I2,I3,I4,I5,I6;
0007
0008    B1=$80;
0009    B2=$7F;
0010    B3=A;
0011    B4=B;
0012
0013    I1=$80;          /* I1 = $FF80! */
0014    I2=B1;           /* I2 = $FF80! */
0015    I3=INTEGER(B1);  /* I3 = $0080 */
0016
0017    I4=$0080;
0018    I5=C;
0019    I6=D;
0020
0021
0022    /* COMPARE BYTES WITH BYTES */
0023
0024
0025    IF B1 = $80 THEN PRINT "\N1";
0026
0027    IF B1 = A THEN PRINT "\N2";
0028
0029
0030    /* UNSIGNED ADDITION AND SUBTRACTION WITH BYTES */
0031
0032    IF B1-1 = B2 THEN PRINT "\N3";
0033
0034    IF B2+8 = B1+7 THEN PRINT "\N4";  (ONLY WORKS BECAUSE OF OVERFLOW!)
0035
0036
0037    /* COMPARE INTEGERS WITH INTEGERS */
0038
0039    IF I5 = $0080 THEN PRINT "\N5";
0040
0041    IF I5 = C THEN PRINT "\N6";
0042
0043    IF I6 = D THEN PRINT "\N7";
0044
0045    IF I4 = $0080 THEN PRINT "\N8";
0046
0047
0048    /* COMPARE INTEGERS WITH BYTES */
0049
0050    IF I5 = $80 THEN PRINT "\N9";
0051
0052    IF I5 = A THEN PRINT "\N10";
0053
0054    IF I5 = B1 THEN PRINT "\N!!";             /* THIS ONE DOES NOT WORK! */
0055
0056    IF !I5 = B1 THEN PRINT "\N11";            /* BUT THIS ONE DOES. */
0057
0058    IF I4 = $80 THEN PRINT "\N12";
0059
0060
```

EXAMPLES OF UNSIGNED OPERATIONS (continued)

```
0061    /* COMPARE BYTES WITH INTEGERS (THEY ALL REQUIRE THE !) */
0062
0063    IF !$80 = I5 THEN PRINT "\N13";
0064
0065    IF !A = I5 THEN PRINT "\N14";
0066
0067    IF !B1 = I5 THEN PRINT "\N15";
0068
0069    IF !$80 = I4 THEN PRINT "\N16";
0070
0071
0072    /* UNSIGNED ADDITION WITH BYTES AND INTEGERS (THEY ALL REQUIRE THE !) */
0073
0074    IF !I5+$10 = B1+$10 THEN PRINT "\N17";
0075
0076    IF !B1+$10 = I5+$10 THEN PRINT "\N18";
0077
0078    IF !I5+B2 = I6+B1 THEN PRINT "\N19";
0079
0080    IF !(I5+B2) - (I6+B1) = (B1+B2) - (I5+I6) THEN PRINT "\N20";
0081
0082
0083    /* COMPARE A BYTE WITH THE SUM OF TWO BYTES WHERE THE RESULT OVERFLOWS
0084       A BYTE VARIABLE.  IN THIS CASE PL/9 WILL CONVERT THE OFFENDING SUM
0085       TO AN INTEGER BY SIGN EXTENSION UNLESS YOU USE THE (!). */
0086
0087     B1 = 100;
0088     B2 = 75;
0089     B3 = 120;
0090
0091     IF !B3 < B1 + B2 THEN PRINT "\N21";
0092
0093
```

If you remove the exclamation marks from any of the lines that contain them the evaluation will not work properly.

The last point is particularly sneaky in that you must have some knowledge of the data stored in the variable before you can make valid comparisons. It's back to the fact that PL/9 expects the programmer to ensure that data resulting from mathematical operations will fit into data size he is using. If B3 was an INTEGER sized quantity the expression would work as expected as PL/9 will automatically sign extend the sum of B1 and B2 to form an INTEGER as the sum overflows a signed BYTE sized variable.

Most programmers would not expect PL/9 to handle the conversion of a REAL number with the value +98676.34 in it to a BYTE sized quantity. On the same token you should not expect PL/9 to handle the overflow of a BYTE sized quantity or an INTEGER sized quantity by converting other variables in the expression to the size required to match the overflow BECAUSE IT WON'T.

7.03.03 DATA TYPES

There are five types of variable/data that can occur in a program:

GLOBAL VARIABLES

Are defined at the start of a program, before the first PROCEDURE, by means of the GLOBAL keyword. They are automatically allocated space on the stack and the 'Y' index register is set to point to them. For this reason, if you call assembly-language routines or embed them using ASMPROC or GEN statements then be sure to preserve the value of 'Y' or the results will be unpredictable. GLOBAL variables are accessible by any procedure; the GLOBAL statement is the best place to put variables that will be used all over a program.

LOCAL VARIABLES

Are defined as part of a PROCEDURE definition. They too reside on the stack but the space allocated is given up when the procedure is terminated. Local variables are private to the procedure in which they are declared, that is they can not be accessed from any other procedure. Their names can be re-used by other procedures as often as the programmer wishes. Temporary variables used for counting or sorting are best declared as local. Parameters passed to a procedure are also local, the only difference being that they are pushed onto the stack before the procedure is entered, so PL/9 generates code to "clean up" the stack after the call.

NOTE: 'LOCAL' variables are truly local. If you have an AT, GLOBAL, CONSTANT, READ-ONLY-DATA, or PROCEDURE name that duplicates the name of a LOCAL variable the compiler will use the LOCAL variable in preference to any other variable of an identical name.

ABSOLUTE VARIABLES

Are declared using the 'AT' keyword, enable the programmer to access any part of his computer and to read and modify the memory contents. They are used generally to access peripheral devices, memory-mapped displays and other parts of the system where the address is fixed, or to provide a "common" area for use by different program modules. AT is also used to define an external vector to be used by 'CALL' or 'JUMP' to gain access to an external subroutine. This allows you to gain access to routines through a RAM or ROM vector table such as those commonly found in system monitors. See sections 7.01.19/20 for details.

CONSTANTS

Allow the programmer to use meaningful names to represent numeric quantities, for example using SPACE rather than 32 or $20. Numbers, e.g. $80, 12, -33, etc., can be considered to be constants when used as part of an evaluation or assignment. CONSTANTS are also used in conjunction with 'CALL' and 'JUMP' to access external subroutines directly. See sections 7.01.19/20 for details.

READ-ONLY DATA

Read-only data statements consists of a list of byte, integer or real values or pointers that are required as constants for use by a program. Data statements (BYTE, INTEGER or REAL keywords) reside outside procedures; data names are therefore global in scope. A data list is in effect a read-only variable, since PL/9 will refuse to assign a value to it.

7.03.04 POINTERS

If an ampersand (&) or a dot (.), the choice is left to the programmer, is placed in front of a variable or procedure name the compiler will use the ADDRESS of the variable (or procedure) rather than the DATA at that address. For example:

```
GLOBAL BYTE VECTORS(16):INTEGER I1;


I1 = VECTORS;
I1 = VECTORS(3);
I1 = .VECTORS;
I1 = .VECTORS(3);
```

The first line above means take the DATA from the first element VECTORS, sign extend it (to form an INTEGER) and place it in I1. The second line above means take the DATA from the third element of VECTORS, sign extend it and place it in I1. The third line above means take the absolute ADDRESS of the first element of VECTORS and place it in I1. The fourth line above means take the absolute ADDRESS of the third element of VECTORS and place it in I1.

The third construction is of considerable use when passing vectors to procedures. It is not usually desirable to pass an entire vector on the stack, as a call by value; normally the procedure being called is going to perform some in-place operation on the vector for the calling program. To enable the procedure to access the vector, all that needs to be passed is its address (i.e. a pointer to the vector) the compiler then selects suitable code to allow the procedure to read or write to any location in the vector.

THERE'S MORE

A variable may also be defined as a pointer. Pointers may be declared in 'AT' or 'GLOBAL' statements or in PROCEDURE declarations, both as passed parameters and as local variables. Pointers may also be declared as READ-ONLY data thereby providing the mechanism for a PL/9 programmer to develop a vector table which contains PROCEDURE addresses, et al. The important point to note is that the name you use does not refer to the place at which the pointer is kept but to the data the pointer refers to. The following are examples of pointer declarations:

```
AT $9000:REAL .RP1:INTEGER .IP1:BYTE .BP1;

GLOBAL REAL .RP2:INTEGER .IP2:BYTE .BP2;

PROCEDURE DEMO(REAL .RP3:INTEGER .IP3:BYTE .BP3)
:REAL .RP4:INTEGER .IP4:BYTE .BP4;

INTEGER PROC_TABLE .DEMO;  /* WARNING: NOT POSITION INDEPENDANT! */
```

Pointers are ALWAYS 16-bit quantities. The variables that pointers point to may be BYTE, INTEGER or REAL (or vectors of these variable types) and must always be declared as such:

```
PROCEDURE DEMO(REAL .RP3:INTEGER .IP3:BYTE .BP4);
```

declares that three pointers are being passed to the procedure. One is pointing to a REAL, one is pointing to an INTEGER, and one is pointing to a BYTE. They can be single variables of the size indicated or they may be elements of a vector.

You must be very careful when declaring the size of the data pointed to. If you

7.03.04 POINTERS (continued)

accidentally tell PL/9 that you are pointing to a REAL that is in fact a BYTE, e.g. (REAL .POINTER) instead of (BYTE .POINTER), PL/9 will produce the code required to read/write 32-bits of data. In this example this will corrupt the data of the three BYTE values at the memory addresses just above the BYTE you are pointing to, as well as producing a completely erroneous result.

This type of programming error (which will not be detected by the compiler) can wreck havoc on programs and cause some really strange problems if the data just happens to be an I/O device. In these cases it will tend to re-initialize the device more often than not!

In order to give the PL/9 programmer complete freedom when accessing vectors or variables via pointers no type (size) checking is done by the compiler. If you want to work through an INTEGER or REAL data table BYTE-by-BYTE with a pointer PL/9 will not stop you, it assumes that you know what you are doing!

BE VERY CAREFUL WHEN YOU SPECIFY THE SIZE OF THE DATA A POINTER POINTS TO!

A variable defined as a pointer can be thought of as a variable that behaves like a "window" to some item of data stored somewhere. For example if we have the following declaration:

```
GLOBAL BYTE .BP1, .BP2, DATA1(120), DATA2(120);
```

we can develop several constructions within a procedure:

```
.BP1 = .DATA1;
.BP2 = .DATA2;
```

This first line means take the absolute ADDRESS of the first element of DATA1 and place it in the variable called '.BP1'. '.BP1' is now pointing to the base of the vector table DATA1. The second line above is similar to the first.

```
BP1 = BP2;
```

This line means take the DATA stored at the location pointed to by the contents of .BP2 (i.e. DATA2(0)) and store it in the location pointed to by the contents of .BP1 (i.e. DATA1(0)). In this example the dot (.) has been left off '.BP1' and '.BP2' so the variable will now behave as a pointer rather than a simple integer variable.

S U M M A R Y

If you define a variable as a pointer by putting a dot (.) or an ampersand (&) before it AT THE TIME YOU DECLARE IT you are telling PL/9 that this variable is capable of being used in one of two ways. If you include the dot (.) when you use the variables name this tells the compiler to manipulate the 16-bit DATA (usually an ADDRESS) held in the variable. If you leave off the dot (.) when you use the variable this tells the compiler to manipulate the DATA pointed to by the address held in the variable. In this case the DATA may be BYTE, INTEGER or REAL sized depending on how you defined the pointer.

If you understood the last paragraph you should understand the fundamental difference in the following two constructions without reading the next paragraph:

7.03.04 POINTERS (continued)

```
.BP1 = .BP1 + 1;
BP1 =  BP1 + 1;
```

The first construction means take the DATA (usually an absolute address in the form of an INTEGER) held in '.BP1' and increment it by one. The second construction means take the DATA (defined to be BYTE sized) pointed to by the absolute ADDRESS held in '.BP1' and increment it by one.

The latter is a much more efficient construction than:

```
DATA1(COUNT) = DATA1(COUNT) + 1;
```

The preceding examples have all illustrated pointers to BYTE vectors but ...

THE SAME RULES APPLY TO POINTERS TO REAL AND INTEGER VARIABLES/VECTORS

Suppose that we wanted to move the 120 bytes of data held in DATA2 to DATA1. There are three basic ways of doing this:

```
PROCEDURE MOVE:BYTE COUNT;
   COUNT = 0;
   REPEAT
      DATA1(COUNT) = DATA2(COUNT);
      COUNT = COUNT + 1;
   UNTIL COUNT = 120;
ENDPROC;
```

OR

```
PROCEDURE MOVE:BYTE COUNT;
   COUNT =  0;
   .BP1  = .DATA1;
   .BP2  = .DATA2;
   REPEAT
      BP1(COUNT) = BP2(COUNT);
      COUNT = COUNT + 1;
   UNTIL COUNT = 120;
ENDPROC;
```

OR

```
PROCEDURE MOVE:BYTE COUNT;
   COUNT = 0;
   .BP1  = .DATA1;
   .BP2  = .DATA2;
   REPEAT
      BP1   =  BP2;
      .BP1  = .BP1 + 1;
      .BP2  = .BP2 + 1;
      COUNT =  COUNT + 1;
   UNTIL COUNT = 120;
ENDPROC;
```

The following will present, by way of more examples, some of the capabilities of pointers.

7.03.04 POINTERS (continued)

The following example is a routine that prints a string, using the character output routine PUTCHAR which is part of the IOSUBS library:

```
0001 PROCEDURE PRINT_STRING (BYTE .STRING) : INTEGER POS;
0002    POS = 0;
0003    WHILE STRING(POS)     /* IMPLICIT <> 0 (NULL) */
0004       BEGIN
0005          PUTCHAR(STRING(POS));
0006          POS = POS + 1;
0007       END;
0008 ENDPROC;
```

Line 3 of this procedure starts a WHILE loop that continues executing as long as STRING(POS) i.e. the POSth character in STRING is non-zero, i.e. not a NULL. Characters are taken one by on from STRING and printed, with POS being incremented each time by line 6. Note so as not to place any restriction upon the length of STRING, POS must be an INTEGER.

The dot at the start of .STRING in line 1 signifies that it is not the actual string itself that has been passed to the procedure but a pointer to it. The actual characters of the string have not been moved; instead their starting address has been supplied to PRINT_STRING. Where a long string is to be printed

this obviously saves a lot of copying.

The 16-bit value that gets passed as .STRING is used only by the procedure; the space it occupies is reserved temporarily on the stack and is given up when the procedure terminates. This means that there is no reason why the pointer should not be modified by the procedure; there will be no effect on the calling program. This fact allows a change to be made that will compile more efficiently:

```
0001 PROCEDURE PRINT_STRING (BYTE .STRING);
0002    WHILE STRING /* implicit <> 0 */
0003       BEGIN
0004          PUTCHAR(STRING);
0005          .STRING=.STRING+1;
006        END;
0007 ENDPROC;
```

As you can see, the local variable POS has disappeared in this version. Instead, the pointer itself is incremented after each character is printed. This is done by line 5. The dot at the start of .STRING in each case tells the compiler that it is the contents of the pointer (.STRING) that are of interest, not the string itself. A significant saving in code is realized by the absence of any vector accesses; in each case it is the element actually pointed to that is wanted. If an index is used, however, then it is added to the CURRENT position of the pointer, wherever that may be.

Where INTEGER or REAL data is being pointed to, it should be noted that an instruction to move to the next item (line 5) would be .PTR=.PTR+2 or .PTR=.PTR+4 respectively. In other words, PL/9 only steps as many bytes as requested; it does not assume that you want the next item unless you specifically tell it.

7.03.04 POINTERS (continued)

To save you having to think about how many bytes to increment by, the construct .PTR=.PTR(N) will cause the pointer to be incremented in 'N' steps according to the data size pointed to. Thus if the declaration had been REAL .PTR and 'N' had been 1 then the step size would be automatically have been computed to be 4. The penalty paid for this convenience is that it is slightly less code-efficient than the method used in the example.

The following summaries the meanings of the various constructs used when dealing with pointers. Suppose that the following declaration has been made:

```
PROCEDURE DEMO: BYTE BUFFER(40), .POINTER;
```

The second variable is a pointer to an as yet unknown BYTE variable or vector.

```
.POINTER = .BUFFER(18);
```

Compute the address of the 18th ELEMENT of BUFFER and place it in the location reserved for '.POINTER'. Now 'POINTER' (not '.POINTER') can be used interchangeably with BUFFER(18) and will have the same effect. The code generated, however, will be different. The point illustrated here is that putting a dot before the pointer name causes the VALUE of the pointer to be used or altered, NOT THE DATA POINTED TO.

```
POINTER = BUFFER(30);
```

Take the 30th element of BUFFER and copy it to the location pointed to by POINTER, in this case the 18th element of BUFFER.

```
BUFFER(30) = POINTER;
```

Do the same thing in reverse.

```
.POINTER = .POINTER + 1;
```

Take the address stored in .POINTER to and increment it by one. In this example it now points to the 19th element of BUFFER.

```
.POINTER = .POINTER(1);
```

Step POINTER to the next ELEMENT of the data it is pointing to. Where the data is BYTE this is the same as '.POINTER = .POINTER + 1;' where the data is INTEGER it is the same as '.POINTER = .POINTER + 2;' and where the data is REAL it is the same as '.POINTER = .POINTER + 4;'. In this example, POINTER now holds the address of BUFFER(20).

```
POINTER(5) = POINTER(6);
```

Take the element at POINTER(6), which is the same as BUFFER(20+6) and copy it to POINTER(5), i.e. BUFFER(20+5).

Lets take another set of declarations and illustrate a few more of the potential uses of pointers:

7.03.04 POINTERS (continued)

```
AT $E060:INTEGER DA_CONV;

PROCEDURE DEMO:INTEGER .POINTER1, .POINTER2, DATA1, DATA2;

   DATA1=$1234;
   DATA2=$4567;

   POINTER1 = .DATA1;
   POINTER2 = .DA_CONV;
```

Supposing that POINTER contains the address of an INTEGER variable. How would you assign the VALUE at the address pointed to by POINTER1 to an INTEGER variable called DATA2? Simple...

```
DATA2 = POINTER1;
```

Now take another construction. Again POINTER1 contains the address of a memory location. This time you want to take the data stored in DATA2 and place it in the memory location pointed to by POINTER1....

```
POINTER1 = DATA2;
```

Lets get carried away and take the data stored in a memory location pointed to by POINTER2 and store it in another location pointed to by POINTER1...

```
POINTER1 = POINTER2;
```

We hope that the previous examples give you some insight into the potential power of using pointers. Refer to the HARDIO library for further examples.

7.03.04 POINTERS (continued)

POINTERS TO STRINGS

A special form of the pointer is provided for handling text strings. For example, suppose that a prompt string is to be printed on the terminal. A procedure called PRINT is supplied in the library file IOSUBS.LIB that takes as a parameter a pointer to a text string and prints the message out to the system console one byte at a time until a null is encountered. The program segment might look like this:

```
PRINT("Choice (Y/N)? ");
```

OR

```
PRINT "Choice (Y/N)?";
```

OR

```
PRINT = "Choice (Y/N)?";
```

The text inside the quotes is generated in line with the program, terminated by a null, and a pointer to it gets passed to the procedure PRINT.

If a message is required several times within a SINGLE procedure the address of the string may be assigned to an INTEGER variable thus:

```
IVAR = "Choice  (Y/N)? ";
```

Whenever the message needs to be printed you simply enter:

```
PRINT(IVAR);
```

If the same prompt is required several times in SEVERAL procedures then the statement can be declared outside of a procedure as read-only data:

```
BYTE PROMPT "Choice (Y/N)? ";
```

Then, whenever you need the message you simply enter:

```
PRINT(.PROMPT);
```

You could also assign the address of PROMPT to an integer variable and use it as in the previous example:

```
IVAR = .PROMPT;

PRINT(IVAR);
```

7.04.00. ARITHMETIC AND OPERATOR PRECEDENCE RULES

The precedence of logical and relational operators is below that of any of the arithmetic operators; the operator precedence table is as follows:

Highest Precedence	Unary Minus (-) and Functions
	* / \
	+ -
	AND
	OR EOR (XOR)
	= <> >= <= > <
	.AND
Lowest Precedence	.OR .EOR (.XOR)

The dots in .AND, .OR, .EOR and .XOR signify that the operator is a logical, not a bitwise, operator.

N O T E

It is very important for you to note that the code that PL/9 produces will evaluate variables in an expression from left to right. This is particularly important when I/O devices are being accessed as often I/O port registers have to be read in a certain order otherwise data is lost. For example:

```
VAR1 = PORT1 AND $7E AND PORT2 AND $5F;
```

would produce code that would:

1. read the data in PORT1
2. bitwise AND the data just read with $7E
3. bitwise AND the result just obtained with the data in PORT2
4. bitwise AND the result just obtained with $5F.
5. place the result in VAR1.

If we used the following construction the precedence rule of a bitwise AND taking
place before a bitwise OR will apply:

```
VAR1 = PORT1 AND $7E OR PORT2 AND $5F;
```

1. read the data in PORT1
2. bitwise AND the data just read with $7E
3. push the result on the stack
4. read the data in PORT2
5. bitwise AND the data just read with $5F
6. bitwise OR the result just obtained with the data on the stack and tidy stack
7. place the result in VAR1.

7.05.00 BIT OPERATORS AND, OR, EOR (XOR)

These operators perform unsigned bit setting and clearing operations with BYTEs or INTEGERs.

A summary of the functions are as follows:

AND ... ANDing a bit with a '1' has no effect on the bit whilst ANDing a bit with a '0' will clear the bit to logical '0'.

OR ... ORing a bit with a '0' has no effect on the bit whilst ORing a bit with a '1' will set the bit to logical '1'.

EOR ... EORing two identical bits will yield '0' in that bit position whilst EORing two different bits will yield '1' in that bit position.

The similarity of the LOGICAL operators with the BIT operators (the only difference being a period (.) preceding the former) requires a certain degree of caution in their use. For example:

```
IF X AND 4 .AND Y AND 2 <> 0 THEN ...
```

is completely different in meaning to

```
IF X AND 4 AND Y AND 2 <> 0 THEN ...
```

The former tests two individual conditions, (X AND 4) and (Y AND 2) and requires both to be non-zero for the test to be true, while the latter tests a single more complex condition.

7.06.00 PL/9 FUNCTIONS

A number of functions are provided to enable conversions to be made between different data types and to perform certain arithmetic operations. The first group all return a BYTE or INTEGER value, although they can operate on any data type:

NOT(A) This function is a ones complement bit inverter which simply changes every one to a zero and every zero to a one in the BYTE or INTEGER being operated on.

SHIFT(A,N) This function performs the equivalent of multiplication or division by powers of two. The expression A is evaluated and shifted left or right according to the value of N, left if N is positive and right if N is negative. N must be a constant, not a variable or an expression. If A is BYTE the value of the function is also BYTE, implying that no automatic extension to INTEGER is performed. If A is REAL it will be FIXed before shifting.

SHIFT can greatly speed some programs by replacing slow multiplications and divisions with fast shifts. For example, SHIFT(VALUE,-2) has the effect of dividing VALUE by four, but is much faster, while SHIFT(X,3) is equivalent to multiplying X by 8.

SHIFT will always preserve the sign bit (bit b15 in an INTEGER and bit b7 in a BYTE) when performing a RIGHT shift, and will shift zeros in on the right when performing a LEFT shift.

SWAP(A) Returns an INTEGER value comprising the reversed bytes of the operand. If a BYTE operand is supplied it will be sign-extended before swapping; a REAL will be FIXed, i.e. converted to the nearest INTEGER.

EXTEND(A) Returns an INTEGER comprising the least significant byte of the operand sign extended into the most. As with SWAP, any data type can be supplied; if the operand is not BYTE it will be converted before sign extension.

INTEGER(A) Works in the same way as EXTEND except that the most significant byte of the operand is set to zero.

BYTE(A) Converts the operand to a BYTE by throwing away the most significant byte. If the operand is REAL it will be FIXed automatically. This function has few uses, since the necessary conversion is usually done automatically. It is included mainly for completeness.

FIX(A) Converts the REAL operand to the nearest INTEGER (rounding where necessary), allowing a REAL sub-expression of an INTEGER expression to be evaluated as REAL before conversion to INTEGER. If FIX were not used then each REAL term would be converted to INTEGER as it is encountered. The mode of evaluation of an expression is always determined by the variable to be assigned (i.e. on the left-hand side of the = operator).

7.06.00 PL/9 FUNCTIONS (continued)

NOTE: If you attempt to 'FIX' a REAL number that holds a value that cannot be held in an INTEGER (-32768 to +32767) the INTEGER returned will be ZERO.

The remaining functions all return a REAL value. They will not be recognized in an integer expression; if for example you need the square root of an INTEGER, use FIX(SQR(A)) to tell the compiler that the word SQR is a REAL operator.

SQR(A) Returns the square root of the operand, which may be of any type, the appropriate conversion being performed automatically.

INT(A) Returns the nearest integer smaller than the operand, which again may be of any type (but it is somewhat pointless using anything other than a REAL). Note that the function returns a REAL integer value.

FLOAT(A) Converts the operand to the nearest REAL, allowing a BYTE or INTEGER sub-expression of a REAL expression to be evaluated as BYTE or INTEGER before conversion to REAL. If FLOAT were not used then each BYTE or INTEGER term would be converted to REAL as it is encountered, which results in extra code and slower processing. This function is complementary to FIX, described above.

7.07.00 ANATOMY OF PL/9 PROGRAMS

This section has been included for programmers wishing to understand the code that PL/9 produces. It should be stressed that in most cases it is possible to write and debug programs without understanding the resulting code, even without understanding the 6809!

Programs can, however, be optimized and made to run faster if careful attention is paid to the constructs used to avoid generating unnecessary code. For example, the SHIFT operator could greatly speed up some programs by replacing (slow) multiplications and divisions by (fast) shifts, while accessing an array with a constant is always faster and more compact than with a variable.

The 'C' compile option (A:T,C) provides a listing of the object code for anyone that wishes to examine it, but the way in which PL/9 handles various constructs deserves some explanation.

7.07.01 VARIABLE ALLOCATION

7.07.02 GLOBAL VARIABLES

Global variables are allocated on the stack before any PROCEDURE is entered. For example:

```
STACK = $8000;
GLOBAL BYTE FLAG, DATA(16):
INTEGER POINTER, TABLE(4):
REAL PI;
```

The code generated is

```
10CE 8000         LDS   #$8000
  34 7F           PSHS  CC,D,DP,X,Y,U
  32 E8 E1        LEAS  -31,S
  1F 42           TFR   S,Y
```

so that the 'Y' index register is now pointing to the global variables, as follows:

```
       7FFA-D           PI
       7FF8-9           TABLE(3)
        .
        .
       7FF2-3           TABLE(0)
       7FF0-1           POINTER
       7FEF             DATA(15)
        .
        .
       7FE0             DATA(0)
'Y' -> 7FDF             FLAG
```

To ensure that the code produced is as compact as possible, organize the GLOBAL statement in such a way that the most commonly-used variables are declared first, resulting in the shortest addressing modes being used for them.

7.07.03 LOCAL VARIABLES

Local variables are allocated space on the stack in the same way as globals, except that it is the hardware stack pointer S that is used to point to the stack frame. At the start of a procedure the stack pointer is decremented to make room for any declared local variables, and at the end of the procedure the space on the stack is released. When data is pushed onto the stack (for intermediate calculations, for instance) the compiler adjusts for the effect this has on the locations of its local variables.

7.07.04 ABSOLUTE VARIABLES

Variables declared using AT or CONSTANT statements are accessed by either Direct or Extended addressing, depending upon whether they reside in the lowest 256 bytes of memory. The DP register is always assumed to be set to zero, unless you use the 'DPAGE' directive to tell the compiler to set it somewhere else.

7.07.05 DATA

Data, as declared in a BYTE, INTEGER or REAL statement, is part of the program (but not part of a procedure) and is accessed by procedures via Program Counter Relative (PCR) addressing.

7.07.06 ASSIGNMENTS

7.07.07 SIMPLE ASSIGNMENTS

In a simple BYTE or INTEGER assignment, the right-hand side is evaluated, getting a result either in B or in D, which is then saved into the variable specified.

If the expression evaluates to an INTEGER where the variable is BYTE, then the high-order byte in A is ignored, while if the expression evaluates to a BYTE where the variable is INTEGER then B is sign-extended into A before saving the variable.

Special mechanisms (see section 7.03.02) are provided to prevent this sign extension from taking place when, for example, you wish to perform unsigned operations.

REAL expressions are handled in much the same way, except that the working accumulator is in fact on the stack, allowing the code generated to still be re-entrant. All processing is handled by subroutines in a package that is loaded in its entirety the first time any of its contents are required. The location of this module may also be pre-assigned by using the 'MATHS' directive.

7.07.08 VECTORS

The mechanism for accessing an element of a vector is as follows:

1. Evaluate the vector element, putting the value obtained into D, if necessary by sign-extending.

2. Set the X index register to point to the base (i.e. zeroth element) of the vector.

3. Add D to X.

4. Load B or D via X.

To assign a value to a BYTE/INTEGER vector, stages 1 to 3 above are identical, then

4. Push the value in X onto the stack.

5. Evaluate the right-hand side of the assignment, getting a value in B or D (depending on the size of the variable to be assigned).

6. Assign the variable by using "STB [,S++"] or "STD [,S++"].

7.07.09 PROCEDURE CALLS

A procedure is called by simply stating its name, followed by any required parameters. If the types of these passed parameters (BYTE, INTEGER or REAL) do not match those of the procedure declaration, an automatic conversion will be made wherever possible, otherwise an error will be reported. The compiler evaluates each of the parameters (there is no practical limit to the number of parameters or the complexity of any of them) and pushes them onto the stack before calling the procedure with a BSR or LBSR as appropriate. Following this the stack is incremented by the number of bytes that were pushed onto it.

7.07.10 PROCEDURES

A procedure may or may not require to have parameters passed to it from the calling routine, and may or may not require local variables. In all cases, the compiler keeps track of the locations of any such parameters or variables. Passed parameters that are declared as "dot variables" are assumed to be pointers to the actual variables, whatever may be passed by a calling routine. This implies that it is up to the programmer to ensure that the parameters passed are sensible; the compiler only checks that the right number of parameters are passed and that they are of the right size (one, two or four bytes). INTEGER variables may therefore be treated as pointers and vice versa.

7.07.11 BUILT-IN ARITHMETIC FUNCTIONS

PL/9 code executes quickly, partly because the 6809 allows the necessary constructs for accessing vectors and passing parameters about to be coded so efficiently that separate library subroutines are unnecessary. The only exceptions to this are where integer multiplication or division are required or where a floating-point calculation is requested. Although the 6809 possesses an 8x8 bit MUL instruction, the integer arithmetic used by PL/9 is signed and often 16 bit, so separate subroutines are used. The multiply subroutine performs signed 16x16 bit multiplication giving a 16 bit result. No overflow checking is done; PL/9 is not intended for "number-crunching". If a more sophisticated integer multiplication routine is required this can be implemented as an assembler procedure. Likewise, the division routine divides a 16 bit dividend by a 16 bit divisor to give a 16 bit quotient, with the remainder ignored.

In both multiplication and division the first use made of the function causes the necessary subroutine to be copied whole from the compiler to the output file; subsequent uses then only require a BSR or LBSR. Both routines accept one parameter (multiplicand or dividend) as a 16 bit number on the stack and the other (multiplier or divisor) as a 16 bit number in the 'D' accumulator; the result in each case is returned in 'D' with the stack cleaned up, requiring minimum overhead from the calling routine. Similarly, when REAL numbers are to be handled, the complete floating-point package, occupying about 1k bytes, is copied from the compiler to the object file so that its contents can be accessed via BSR or LBSR calls. If the 'MATHS' directive is used calls to the REAL math package will be made by JSR calls to preserve the position independence of the PL/9 object file.

7.07.12 REGISTER PRESERVATION

PL/9 does not assume that the 6809 registers are preserved from one construction to the next. This is why the code may seem a bit obtuse to you assembly language programmers out there. The only exception to this is that the 'Y' register must be preserved if you are using GLOBALS and the 'DP' register must be preserved if you are using 'DPAGE'. This means that you can do almost anything you like between constructions, insert GEN statements, JUMP or CALL external routines, etc.

7.07.13 POINTERS

If a variable is declared as a DATA STORAGE variable, i.e. it IS NOT preceded with a period '.' or an ampersand '&', (e.g. INTEGER I2) it may be assigned to another variable in one of two ways:

1. It may have its VALUE assigned to another variable: (e.g. I1 = I2)

2. It may have its ADDRESS assigned to another variable: (e.g. I1 = .I2)

If a variable is declared as a POINTER (i.e. it IS preceded with a period '.' or an ampersand '&', (e.g. INTEGER .I3) it may be assigned in one of four ways:

1. It may have its VALUE (which is usually the ADDRESS of a variable) assigned to another variable: I1 = .I3)

2. It may have the VALUE IT IS POINTING to (which is usually data) assigned to another variable: (e.g. I1 = I3)

3. It may have the VALUE IT IS POINTING to assigned the VALUE of another variable: (e.g. I3 = I1)

4. It may have the VALUE IT IS POINTING to assigned the ADDRESS of another variable: (e.g. I3 = .I1)

A fifth method of using variables defined as pointers is the second and third constructions in combination. For example we have two variables defined as .I3 and .I4 we can have the VALUE I3 is pointing to assigned to the VALUE I4 is pointing to with the construction I3 = I4.

A sixth method of using pointers is the construction above with subscripts, e.g. I3(N) = I4(N), where 'N' is a constant, variable or expression.

Pointers can be treated in very much the same manner as normal variables in that you can increment and decrement them:

```
.I3 = .I3 + 1    or    .I3 = .I3 - 1
```

Note that in this context the '.' in front of 'I3' is treated as though it was part of the variables name.

You can increment or decrement the value POINTED to by them:

```
I3 = I3 + 1      or     I3 = I3 - 1
```

And you can subscript them:

```
I3(VAR) = I4(VAR)    NOTE: you CAN NOT use .I3(VAR) = .I4(VAR)
```

The following is an illustration of the code produced by pointer constructions. For the purposes of comparison the code produced by PL/9 when accessing variables that are not pointers is also shown. Let us assume that we have one of the following declarations at the start of a program:

```
   AT $9000: INTEGER I1, I2, .I3, .I4;
```

OR

```
   GLOBAL    INTEGER I1, I2, .I3, .I4;
```

7.07.13 POINTERS (continued)

Now lets examine the basic constructions possible within each group.

CONSTRUCTION	CODE GENERATED FOR 'GLOBAL'	CODE GENERATED FOR 'AT'
A. I1 = I2;	LDD 2,Y STD ,Y	LDD $9002 STD $9000
B. I1 = .I2;	LEAX 2,Y TFR X,D STD ,Y	LDX #$9002 TFR X,D STD $9000
C. I1 = I3;	LDD [4,Y] STD ,Y	LDD [$9004] STD $9000
D. I1 = .I3;	LDD 4,Y STD ,Y	LDD $9004 STD $9000
E. I3 = I1;	LDD ,Y STD [4,Y]	LDD $9000 STD [$9004]
F. .I3 = I1;	LDD ,Y STD 4,Y	LDD $9000 STD $9004
G. .I3 = .I1;	LEAX ,Y TFR X,D STD 4,Y	LDX #$9000 TFR X,D STD $9004
H. .I3 = .I4;	LDD 6,Y STD 4,Y	LDD $9006 STD $9004
I. I3 = I4;	LDD [6,Y] STD [4,Y]	LDD [$9006] LDD [$9004]
J. .I3=.I3+1;	LDD 4,Y ADDD #$0001 STD 4,Y	LDD $9004 ADDD #$0001 STD $9004
K. I3=I3+1;	LDD [4,Y] ADDD #$0001 STD [4,Y]	LDD [$9004] ADDD #$0001 LDD [$9004]

NOTE: All of the above examples show code for pointers to INTEGERS. The exact same constructions can be used with pointers to BYTES and pointers to REALS but the code produced will be different.

A. This is a simple DATA assignment between two variables assigned as INTEGER data storage variables. The code LDD 2,Y means load the 'D' accumulator with the data contained in the location 2,Y. The code STD ,Y means store the 'D' accumulator in the location 0,Y. In plain English this means take the VALUE of the data at 2,Y and put it in 0,Y.

   The code LDD $9002 means load the 'D' accumulator with the data stored at memory locations $9002/3 (remember it is an INTEGER). The code STD $9000 means store the 'D' accumulator in memory locations $9000/1. In plain English this means take the VALUE of the data at $9002/3 and put it in $9000/1.

7.03.04 POINTERS (continued)

B. This is the assignment of a POINTER to a variable to another variable, i.e. the assigning of the address of a variable to another variable. The code LEAX 2,Y means load the address of the memory location held in 'Y' (the base of the global storage allocation) into the 'X' register adding 2 to it as you do so (i.e. the 'Y' offset of '.I2'). This, in effect, puts the physical memory location of the variable at 2,Y into 'X'. The code TFR X,D means move the contents of the 'X' register into the 'D' register, this is done so that PL/9 can manipulate the data further if necessary. In this case no additional manipulation is required so we simply store it. The code STD ,Y does just this and means store the contents of the 'D' register in the location 0,Y. In plain English this means take the ADDRESS of the value at 2,Y and put it in 0,Y.

   The code LDX #$9002 means load the 'X' register with $9002 (the address of '.I2'). Again PL/9 will move it into 'D' where it can manipulate it if required. STD $9000 means store the contents of 'D' at memory locations $9000/1. In plain English this means take the ADDRESS of the value at $9002/3 and put it in $9000/1.

C. Now the plot thickens! The code LDD [4,Y] means load the 'D' accumulator with the VALUE of the data pointed to by the contents of 4,Y. Say that 4,Y contained $1234 and memory location $1234 contained $ABCD, the 'D' accumulator would contain $ABCD after the instruction was executed. The code STD ,Y has been explained previously.

   The code LDD [$9004] has the same basic meaning as above. It means load the 'D' accumulator with the VALUE of the data pointed to by the contents of memory location $9004. If $9004/5 contained $1234 and memory location $1234 contained $ABCD, the 'D' accumulator would contain $ABCD after the instruction was executed.

D. This construction is identical to the first one (A), only different variables are being used. Note that when you use the full name (i.e. include the period or ampersand) a variable declared as a pointer the variable is no longer treated as a pointer but is treated as any other variable.

E. This construction is the inverse of the third one (C). Here we load 'D' with the VALUE contained in 0,Y and store it in the memory location pointed to by the ADDRESS held in 4,Y.

F. This construction is also identical to the first one (A), only different variables are being used.

G. This construction is identical to the second one (B), only different variables are involved.

7.07.13 POINTERS (continued)

H. This construction is identical to the first one (A), only different variables are involved

I. This construction is a combination of the third one (C) and the fifth one (E), only different variables are involved.

J. This construction simply increments the DATA within .I3.

K. This construction is more sophisticated as it increments the DATA pointed to by the contents of .I3.

POINTERS TO PROCEDURES

The concept of producing vector tables which point to subroutine modules is a familiar one to experienced assembly language programmers. The basic idea behind the concept is that you produce a table of addresses which point to the various subroutines in a particular applications program. This then becomes a 'program module' with all external programs accessing the procedures through the vector table rather than accessing the subroutines directly. This approach enables the programmer to modify ANY or ALL of the subroutine modules WITHOUT having to go back and alter any of the programs that use them.

PL/9 also supports this programming concept albeit crudely. PL/9 allows you to build a subroutine library of procedures which can be accessed through a vector table. The main area we have seen this technique put to best use was in the generation of a large graphics handling library which ran to 32K of code. This module started its life as a standard PL/9 library and was INCLUDEd in the program compilation. As the main application program grew the compile time of the INCLUDEd library became tedious as the file was heavily commented. It was then decided to put the GRAPHICS library into a ROM module and generate a small library of ASMPROCs to gain access to it through a vector table. This was done with the result that the library of ASMPROCs only took a few seconds to compile as opposed to the three minutes required to compile the entire graphics library each time the main program was compiled.

The basic method of generating a READ-ONLY vector table in PL/9 is as follows:

```
PROCEDURE PROC_ONE;
.
.
ENDPROC;


ORIGIN = $XXXX;

INTEGER _ONE .PROC_ONE;
```

WARNING: THESE VECTOR TABLES ARE NOT POSITION INDEPENDANT!

7.07.14 MEMORY USE BY PL/9 DURING COMPILATION

PL/9 is a combined editor, compiler and debugger, so memory space within the system is obviously at a premium. For this reason, some care has been taken to ensure that the symbol table occupies as little room as possible. There are 18 tables in use during compilation, all of which could be any size depending on the program being compiled. To avoid setting arbitrary limits on the number of symbols that can be used, all of these tables are dynamic, that is they each grow according to the number of items contained within them.

7.07.15 DISK BINARY FILES

Because PL/9 is a single-pass compiler, forward references present something of a problem. If you use a construct such as "IF X=5 THEN..." the compiler is unable to tell where to branch to in the event of the test failing. To get around the problem, PL/9 puts in a "LBNE $0000". At the same time it makes a note of the fact that an unresolved forward reference has occurred. When the intervening instructions have all been compiled, the true destination of the branch is now known, and the compiler puts in a "fixup" instruction (in brackets on the A:C listing).

When an object file is being generated on disc, PL/9 maintains two consecutive 255-byte buffers. Initially, both buffers are empty. As object code is generated it goes into the first buffer, and when this is full into the second buffer. When the second buffer is full, the first buffer is written to disc, the second buffer is copied into the first and object code is written again into the second buffer until it is full, when another disc write occurs.

Forward branch fixups can be made directly on the contents of the buffers, but if the fixup refers to a point so far back that the corresponding code has already been written to disk, special action has to be taken. What happens is that a special record is generated, usually 4 bytes long, for the offending fixup. Because of the way that FLEX's MAP utility works, this causes an apparent break in the binary code. If you load the disk file into memory you will find that each of the forward references has been correctly fixed.