7.00.00 PL/9 LANGUAGE REFERENCE MANUAL

This section of the documentation on PL/9 is designed as a detailed technical reference to the KEYWORDS and VARIABLES used to produce PL/9 programs. It is no accident that the titles used in this section match the equivalent sections in the USERS GUIDE where more detailed information may be found.

The tutorial organisation of this manual has necessitated a certain amount of duplication of topics between this Reference Manual and the Users Guide.

To make this section more readable we have organized it in functional order, i.e. the order you would normally encounter the keywords when writing PL/9 programs. To help improve 'random' access to this section the KEYWORDS section in the main index is organized in alphabetical order.

7.00.01 COMMENTS

PL/9 accepts the '/* ... */' pair (borrowed from PL/M) as start and end markers, respectively, for comment fields. If the first line in the program contains a comment it will be printed as a title at the top of all listings to the printer or to disk files. In this instance the comment field may only be 50 characters long. Anything after the 50th character will simply be ignored; any leading spaces will be suppressed.

In the main body of the program comments may be as long as desired, extending over several lines if necessary. / and * may be used within the comment field but they may never be adjacent to each other, i.e. you cannot use /* or */ inside the comment field. It is also a no-no to place comments inside of text strings! You will see several examples of comments in the sample programs in the Users Guide.

7.00.02 SYMBOLS

Symbols, that is to say constant, variable, or procedure names may comprise any sequence of upper or lower case alphabetic and numeric characters. The name may be optionally punctuated by the underline '_' character, e.g. LABEL, _LABEL, LABEL_, _LA_BE_L_. No other characters, (+ - $; : * \ | / etc) may be used for punctuation in a symbol.

In all cases the first letter MUST be alphabetic or the underline character. A symbol may be anything from 1 to 127 characters long and is unique over its entire length. PL/9 keywords must not be used as symbol names. The compiler does not make any distinction between upper and lower case letters in symbols. For example a procedure named 'TEST' is the same as a procedure named 'test' or 'Test'.

7.01.00 PL/9 KEYWORD DESCRIPTIONS

All PL/9 keywords consist of UPPER-case or lower-case letters and must be followed by at least one space (or in some cases by a semicolon or an equal sign). The following is a list of the PL/9 keywords:

ACCA	CCR	- FLOAT	NMI	STACK
ACCB	CONSTANT	FOREVER	- NOT	- SWAP
ACCD	DPAGE	GEN	+ OR	SWI
+ AND	ELSE	GLOBAL	.OR	SWI2
.AND	END	GOTO	ORIGIN	SWI3
ASMPROC	ENDPROC	IF	PROCEDURE	THEN
AT	ENDPROC END	INCLUDE	REAL	UNTIL
BEGIN	+ EOR	- INT	REPEAT	WHILE
BREAK	.EOR	* INTEGER	RESET	+ XOR
* BYTE	- EXTEND	IRQ	RETURN	.XOR
CALL	FIRQ	JUMP	- SHIFT	XREG
CASE	- FIX	MATHS	- SQR	

\+ Covered in the section on BIT OPERATORS.

\- Covered in the section on FUNCTIONS.

* Two uses; one described in this section, one in the section on FUNCTIONS.

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                                                                       *
*                              W A R N I N G                            *
*                                                                       *
*  DO NOT USE any of these as constant, variable or procedure names,    *
*  even if there may not appear to be any possibility of ambiguity.     *
*                                                                       *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

NOTE: One of the changes in PL/9 that occurred between version 2.XX and 3.XX was the removal of 'TRUE', 'FALSE' and 'MEM' from the keywords list. These facilities may be reinstated by including the library module 'TRUFALSE.DEF' in your program. See the library reference section for further details.

7.01.01 CONSTANT

It is often useful to be able to use meaningful symbols rather than numbers in a program; the readability is considerably improved. For example, any program doing a great deal of terminal I/O will use carriage return, line feed and space characters from time to time; these can be equated to convenient symbols as follows:

```
CONSTANT CR=$0D, LF=$0A, SP=$20, BEL=$07, PROM_BASE=$F800, STACK_INIT=$E700;
```

Later on in the program, if the line feed character is to be assigned to the variable CHAR, "CHAR = LF;" is more readable than "CHAR = $0A;" or CHAR = 10;".

CONSTANT statements may appear at any point in a program as long as it is before any use is made of the symbols defined. Note that constants can only be BYTE or INTEGER; there is at present no way of declaring a REAL constant other than by a read-only data declaration. (see 7.01.07)

Constants do not take up any room in memory and do not increase the size of the source file by any significant amount. REAL constants declared via read-only data declarations will use 4 bytes per constant declared.

CONSTANTs defined by HEX numbers are treated in a special way by the compiler to alleviate some of the confusion that might otherwise occur when attempting to work with unsigned numbers. This means that when you assign a CONSTANT with a HEX number $0F is not the same as $000F. One will be treated as a BYTE quantity, and the other will be treated as an INTEGER quantity by the compiler. This topic will be discussed in detail in section 7.03.02.

If a CONSTANT name duplicates a PROCEDURE name, or vice versa, a warning will be posted during compilation, viz:

```
WARNING: SYMBOL ALREADY USED FOR ANOTHER VARIABLE TYPE
```

7.01.02 AT

AT is used to give names to absolute addresses in the 6809's memory space. For example, suppose that an ACIA is located at $E004:

```
AT $E004: BYTE ACIACONTROL(0), ACIASTATUS, ACIADATA;
```

This statement declares three byte variables. ACIACONTROL is declared to be a vector of nil size; the effect is that ACIASTATUS will be at the same address. ACIADATA will be at the next higher address ($E005).

```
AT $E800: BYTE VDU(1920);
```

In this case we have declared a byte vector of 1920 elements (0-1919) to allow direct access to a 24x80 VDU display.

```
AT $E040: INTEGER PIADDR(0), PIADATA, PIACONTROL;
```

If the RS0 and RS1 register select lines of a PIA (in this case on port 4 in the S-30 section of a Windrush or GIMIX system) are connected to A1 and A0 instead of the usual vice-versa, the device's registers appear in the order A-DATA(DDR), B-DATA(DDR), A-CONTROL, B-CONTROL, allowing 16-bit input/output operations to be performed. The above statement declares the three integer variables corresponding to the three pairs of registers DDR, DATA and CONTROL. (Readers unfamiliar with the MC6821 should not worry too much about this.)

AT statements can appear at any point in a program, as long as it is before the point at which the variables thereby defined are actually used. It is good practice, however, to declare all 'AT' variables near the start of the program.

AT can also be used to define an external address which CONTAINS an address for use with 'STACK', 'JUMP' or 'CALL'. See the appropriate section for further information.

The library module 'TRUFALSE.DEF' declares an 'AT' variable called 'MEM' as follows:

```
AT $0000: BYTE MEM;
```

This allows you to generate constructions such as:

```
MEM(COUNT1) = MEM(COUNT2);
```

to shift data around in the absolute memory map.

7.01.03 ORIGIN

ORIGIN allows the programmer to specify where in memory the object program should be located, on a procedure-by-procedure basis. An ORIGIN may be placed before any procedure or read-only data (BYTE, INTEGER or REAL) statement and has the following syntax:

```
ORIGIN = $C100;
```

Origin may also be assigned via a constant. e.g.:

```
CONSTANT PROM_BASE = $F800;
ORIGIN = PROM_BASE;
```

You are allowed to have as many ORIGIN statements in your program as you desire and they may be declared in any memory address order your application requires. There are two points to note however.

First is that when you declare a second origin you must be sure that it does not overlap the code produced by an earlier origin (unless it is your intention to do so). The way the code produced is stored on disk and/or loaded into memory will cause the latest information (i.e. the information furthest down the source file) to over-write any earlier data.

The second point is when the second ORIGIN statement occurs before the compiler 'sees' a read-only data declaration or a procedure declaration. The compiler uses either of these events as a signal to reserve space for the branch to the last procedure in the file. If you declare a second ORIGIN before either of these events occurs the resulting code will have a gap between the first ORIGIN (where the program is normally entered) and the branch to the last procedure. To prevent this simply declare a dummy read only byte before you declare the second origin, for example:

```
ORIGIN=$1000;
STACK=*;
GLOBAL REAL I;

ORIGIN=$2000;
```

The above construction will cause the program to crash when it is entered at $1000. If you look at the code PL/9 produces it will be obvious why this happens. The following construction works as you would expect.

```
ORIGIN=$1000;
STACK=*;
GLOBAL REAL I;
BYTE DUMMY 0;  <---- This declaration forces the compiler to reserve space
                     for the branch to the last procedure before it adjusts
ORIGIN=$2000;        the program counter to the new origin.
```

When the ORIGIN statement is not present at the beginning of a PL/9 program the compiler will set the origin to $0000.

ORIGINs are ignored by the tracer, which instead locates the program at the top of available memory for testing purposes (see section 3.01.00).

7.01.04 STACK

This keyword allows the programmer to specify where the stack should be located, and thereby all global and local variables. The stack should be one of the first assignments after ORIGIN (if it is used at all). It is essential to ensure that adequate stack space is reserved to cover the program's requirements; this may not be the case if the stack is left where the system monitor allocated it, for example. The syntax may take one of four forms as follows:

```
CONSTANT STACK_INIT = $E700;
AT $CC2B: INTEGER MEM_END;


STACK = $8000;        Generates  LDS  #$8000
STACK = STACK_INIT;   Generates  LDS  #$E700    (STACK_INIT is a CONSTANT)
STACK = MEM_END;      Generates  LDS  $CC2B     (MEM_END is an 'AT' VARIABLE)
STACK = *;            Generates  LEAS *,PCR
```

The third form allows the stack to be assigned to a value determined by an external program. In this example the contents of FLEX 'MEMEND' will determine where the stack will be initialized.

The last form allows the stack to grow downward from the start of the program (as long as it is the first statement other than an ORIGIN). The STACK statement will normally directly follow the ORIGIN, if present, and will usually be before any other program code. The STACK statement is ignored by the tracer, which acts as though "STACK=*" had been specified.

Generally speaking the stack should NEVER be reassigned if the PL/9 program is being called as a subroutine from another program but MUST be assigned if the PL/9 procedure is the main program in a self-starting (from RESET) application.

STACK is also a pseudo register name within PL/9 which allows you to dynamically assign the stack within your program code. The following are examples of use:

```
AT $CC2B: INTEGER MEMEND;
CONSTANT MYSTACK = $A000;

GLOBAL INTEGER I1, .I2, PROGSTACK(6):BYTE COUNT;

INTEGER STACKTABLE $A000,$B000,$C000;  /* READ ONLY DATA */

PROCEDURE STACK_DEMO;
   STACK = MEMEND;
   STACK = MYSTACK;
   STACK = I1;
   STACK = .I1;
   STACK = I2;
   STACK = .I2;
   STACK = PROGSTACK(1);
   STACK = PROGSTACK(COUNT);
   STACK = STACKTABLE(1);
   STACK = STACKTABLE(COUNT);
   I2    = STACK;
```

All of these assignments make use of the 'D' accumulator and a few of them also make use of the 'X' register.

7.01.05 GLOBAL

Global variables are defined before any procedures or data statements. The effect is to reserve space on the stack for variables that can be accessed from anywhere in the program. The GLOBAL declaration also generates the code necessary to push the entire register set (CC, D, DP, X, Y and U) onto the stack before the global variables are allocated. When GLOBAL is used in conjunction with ENDPROC END (q.v.) the entire register set will be preserved and the PL/9 program may therefore be called from a program written in any other language.

Typical GLOBAL declarations are as follows:

```
GLOBAL BYTE DUMMY;  (just to push the register set for use with ENDPROC END)

GLOBAL BYTE BUFFER(80), FLAG, COUNT;

GLOBAL REAL VALUE1, VALUE2: BYTE FLAG;

GLOBAL BYTE     CHAR, SIGN:
       INTEGER  TABLE(500), POSITION:
       REAL     FRACTION:
       BYTE     BUFFER(128);

GLOBAL BYTE     VECTOR_BASE(0), VECTORS(10), DELAY(0), COUNT;
```

In each case, the GLOBAL keyword is followed by a data type descriptor, BYTE, INTEGER or REAL, then a list of variables separated by commas. Variables of another type are declared separately, with the different type declarations separated by colons. Any number of variables, each of any size, may be declared in any order required by the application.

The last example above declares 'VECTOR_BASE' as a vector of nil size. This technique is often used to give a variable TWO names; in this case VECTOR_BASE, VECTORS(0), or VECTORS will all mean the same memory location. The same technique has also been used with DELAY and COUNT, again these two names represent the same memory location.

Note, however, that only one GLOBAL statement is allowed in a program; all global variables must therefore be defined in that one statement.

If you wish to access GLOBAL variables from within interrupt procedures special precautions must be taken to preserve the global pointer (the 'Y' index register). This topic will be discussed in more detail in section 7.01.26.

The PL/9 compiler produces a global symbol table which indicates the offset from the 'Y' index register. This information is provided to help facilitate access to the global variables in external assembly language procedures. The code PL/9 produces will ALWAYS leave the 'Y' index register pointing at the base of the GLOBAL variables. The only time that the status of the 'Y' index register cannot be guaranteed to be pointing at the base of the global variables is when an interrupt occurs.

NOTE 1: The variables appear on the stack in order of declaration, that is the first declared is the lowest on the stack. This fact can be used, by declaring the most frequently-used variables first, to ensure that the compiler will select the most efficient indexed mode of addressing when accessing the variables. The same is also true for local variables (see PROCEDURE).

NOTE 2: If you are using GLOBALs you must preserve the 'Y' register any time you use calls to external procedures that may alter it.

7.01.06 DPAGE

This keyword is used to tell PL/9 where to set the direct page register of the MC6809 to. This declaration is normally placed just after the GLOBAL declaration if it is used.

DPAGE is primarily intended as a mechanism to shorten the code required to access 'AT' variables. For example if your I/O devices were in the range of $E000 through $E0FF (the GIMIX and WINDRUSH SS-30 bus memory map) then DPAGE would be set as follows:

```
DPAGE=$E0;
```

Alternatively you can assign DPAGE via a constant.

```
CONSTANT IO_BASE=$E0;
DPAGE=IO_BASE;
```

WARNING: When writing PL/9 procedures to be called as subroutines from other languages never re-assign the direct page register. There are three exceptions to this rule:

(1) When you are absolutely sure that the calling routine has saved all of the registers (or at least the DP) and will restore them when the PL/9 procedure terminates.

(2) When you are sure that the calling routine will restore the direct page when the PL/9 program terminates or does not make any use of the direct addressing mode.

(3) When you have used the GLOBAL declaration in the PL/9 procedure. (the GLOBAL declaration combined with ENDPROC END will ensure that all registers are preserved)

NOTE: If you are using DPAGE within your PL/9 program you must preserve the 'DP' register any time you make calls to external procedures that may alter it.

7.01.07 BYTE, INTEGER and REAL

These three keywords are used in AT, GLOBAL and PROCEDURE statements to declare variables, but they also have an independent use, that being to create blocks of read-only data within the program (but NOT inside any procedure). Four examples follow; first a message string, then a list of addresses (a vector table), then a list of pointers to procedures (i.e. the addresses of the procedures) and last a declaration of a REAL constant:

```
BYTE MESSAGE CR,LF,"This is a message string";  /* CR & LF are constants! */

INTEGER VECTORS $F800,$F802,$F804; /* RESET, CNTRL, INCHNE */

INTEGER PROCEDURES .PROC1, .PROC2, .PROC3;

REAL PI 3.14159265;

REAL POWERS_OF_TEN 10, 100, 1000, 10000, 100000, 1000000;
```

In each case, the name of the data follows the type identifier. To access the Nth element of MESSAGE, use "CHAR = MESSAGE(N);", to execute the Nth subroutine in VECTORS use "CALL VECTORS(N);", and to execute the Nth procedure in PROCEDURES use "CALL PROCEDURES(N);". To access the real 'constant' use "REALVAR = PI". The compiler generates program counter relative (PCR) addressing.

The third example above is primarily used to produce a vector table of the addresses of a given set of procedures within a library module so that they may be accessed from outside of the module. In this way the external procedure always gains access to the library procedures through a fixed point. This allows you to subsequently alter one of the procedures in the library WITHOUT having to re-compile all of the programs that make reference to it.

7.01.08 INCLUDE

As a program grows, it may be convenient to reduce the size of the source file by writing part or all of it to disc. You may then instruct the compiler to incorporate the file into the program at the appropriate point during compilation. The instruction to do this takes the form:

```
INCLUDE FNAME;
```

where FNAME is the name of the source file (having a default drive number and file extension defined by the SETPL9 program) that is to be included.

The default values can be overriden by providing a full specification, e.g.

```
INCLUDE 0.FNAME.EXT;
```

In addition to reducing the size of the memory-resident code, INCLUDEd files are effectively ignored by the trace-debugger, so INCLUDEd procedures run at full speed - a useful feature when programs have to deal with real-time events.

Interrupt handlers MUST be put into INCLUDE files so that the tracer will operate properly.

There are no restrictions on what may be in an INCLUDEd file except that if it itself contains an INCLUDE statement this will be ignored, i.e. it is not possible to nest INCLUDEd files.

The contents of the file are treated in the same way as the rest of the program being compiled, so a symbol defined in both the main program and the INCLUDE file will cause an error to be reported; the line number indicated in this case is that of the INCLUDE statement no matter where in the file the error may be.

7.01.09 A PL/9 PROGRAM HEADER

A typical order of declarations at the beginning of a PL/9 program is as follows

COMMENT .... if the first line contains a comment (between the /* ... */ pair) this comment will be taken as the title of the program and will be printed at the top of all printed listings. Comments may also be placed anywhere in the program (except in the middle of a text string!) via the /*...*/ pair which may go over several lines.

CONSTANT .... used to define various 'hard' memory locations, hardware related parameters, delay time constants, status flags, etc., that might be altered in time. This technique eliminates time consuming searches of the source file for embedded references.

AT .... defines the 'hard' memory addresses in the system, typically I/O addresses. 'AT' variables are also global in that they are accessible by all procedures and are often used to pass parameters to/from the 'parent' program or other programs, particularly interrupt handling procedures.

MATHS .... tells the compiler where the floating point math package is to be located. (see section 7.01.25).

ORIGIN .... used to tell PL/9 where to locate the program.

STACK .... assigns the hardware stack pointer to the desired location but must be used with caution when the program is to be called from somewhere else as a subroutine.

GLOBAL .... two purposes. (1) to push the entire register set onto the stack so that they can be restored before returning to the calling program and (2) to allocate variable storage that is known by and accessible by all subsequent procedures.

DPAGE .... assigns the MC6809 direct page register to improve code efficiency in accessing 'AT' variables.

DATA .... read-only data declarations in the form of BYTE, INTEGER and REAL. Locating data of this type near the start of the program rather than embedding it in with the code improves the readability of the main program and makes it easy to alter, when, and if required.

INCLUDE .... provides a convenient method of keeping your current work file size down to a manageable size. As various subroutines are tested and debugged they may be saved into library files. INCLUDEd files typically form the basis of the I/O and special function routines required by the programmer. Included files have three benefits: (1) paper is not wasted when printed listings of the main file are made, (2) INCLUDEd files do not take up any memory space in the editor, and (3) INCLUDEd files run at full speed when in the PL/9 tracer.

7.01.10 PROCEDURE

All executable code must reside within one or more PROCEDUREs. A procedure consists of three parts; a declaration, the body of statements that perform its purpose, and one or more exits.

The procedure declaration serves as a label indicating to the compiler where the procedure resides. It comprises declarations of any variables that are required to be passed to or returned from the procedure when it is called, as well as any local variables that are to be used solely within the procedure. Since the declaration can take a number of different forms, a formal definition is too complicated, so a number of examples are given:

```
PROCEDURE ONE;
```

This procedure is named ONE, has no parameters passed to it on the stack and uses no local variables. It can only be passed information and may only act upon global (including AT) variables. It may, however, return variables via global variables or via RETURN or ENDPROC statements.

```
PROCEDURE TWO (BYTE VALUE1, VALUE2);
```

Procedure TWO is passed the values of two byte variables VALUE1 and VALUE2. These values are passed on the stack from the calling program and may be simple byte or integer values or array elements. If the size (BYTE, INTEGER or REAL) of the data in the call does not match that in the procedure declaration then PL/9 will attempt to make an automatic conversion; if it cannot then it will report an error.

```
PROCEDURE THREE (INTEGER .POINTER: BYTE POSITION);
```

Procedure THREE is passed a pointer to an INTEGER variable and a BYTE value. The dot before POINTER indicates that the parameter is assumed to be the address of a variable, not its value (this is discussed in detail in section 7.03.04) Since the actual address of the variable is now known to the procedure its contents may be altered by the procedure. This provides an indirect means of passing values back to the calling program.

Note that no information is passed to define whether POINTER points to a BYTE, an INTEGER, a REAL or a VECTOR comprising BYTEs, INTEGERs or REALs. The keyword INTEGER before .POINTER above tells PL/9 that within this procedure the variable to be operated on should be treated as an INTEGER (it could just as easily been a BYTE or a REAL). It does not, however, know if it is a vector. It is assumed that the programmer knows what he is doing when he accesses an element of a vector.

It is important to note that a variable may be declared as one data type in the main program and treated as a different data type within a procedure by using the pointer mechanism. For example a vector of bytes may have a pointer to one of them subsequently defined as a pointer to an INTEGER. This would enable the procedure to manipulate the byte pointed to plus the one directly above it in a single INTEGER operation.

```
PROCEDURE FOUR (REAL REALVAL, .POINTER:BYTE BYTEVAL):INTEGER COUNT;
```

ANY mixture of integer, byte or real values or pointers to variables may be passed to a procedure AND the specification may be in any desired order. Commas separate items in a list of like-sized values while colons are used to indicates a different size is about to be specified.

7.01.10 PROCEDURE (continued)

Here we are passing a REAL value, a pointer to a REAL variable (or a REAL vector table...it is up to the programmer to know) and a BYTE value. We have also declared a local INTEGER variable.

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. viz: (REAL .POINTER1:INTEGER .POINTER2:BYTE .POINTER3) 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.

```
PROCEDURE FIVE: BYTE ALPHA, BETA(6):
                INTEGER I1, I2, COUNT;
                REAL P, Q(3);
```

Procedure FIVE has no parameters passed to it but uses the variables shown in the list, allocating temporary storage for them on the stack. Note that PL/9 allows a free use of spaces and carriage returns to achieve a tidy and readable program listing.

P L E A S E N O T E

There are two very important points to take note of when passing variables from the calling program to a function procedure:

(1) The variables must be declared in EXACTLY the same order with each item separated from the next by a comma.

(2) The variables must be EXACTLY the same size. If they are not the compiler will attempt to generate the code to convert them, (which may not always be desirable) if it cannot it will report an error.

PROCEDURES AS VARIABLES AND FUNCTIONS

Procedures can also be treated as a VARIABLE or used to manipulate a VARIABLE, i.e a behave as a FUNCTION. ENDPROC (or RETURN) is required to implement both of these these procedures so if you don't understand our use of ENDPROC read the following section and then return here.

```
PROCEDURE PROCVAR1;
ENDPROC 1;
```

Here is about the simplest example I could think of. This is a procedure that behaves just like the number 1. Not very useful but it does serve as the basis of illustrating some of the various constructions possible.

```
VAR1 = PROCVAR1;

VAR1(PROCVAR1) = VAR2;

VAR1 = VAR2*PROCVAR1;
```

get the idea yet?

7.01.10 PROCEDURE (continued)

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                                                       *
*  YOU CAN USE A PROCEDURE THAT RETURNS A VARIABLE      *
*  JUST AS YOU WOULD ANY OTHER VARIABLE IN PL/9         *
*  BUT IT ONLY MAKES SENSE TO PUT IT ON THE RIGHT       *
*  SIDE OF AN ASSIGNMENT EXPRESSION.                    *
*                                                       *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

The procedure may be as simple as the one above or as complex as you desire. Further examples of PL/9 procedures that behave as variables can be found in the library module entitled IOSUBS.LIB (q.v.). The procedures of interest are: 'GETCHAR', 'GETCHAR_NOECHO', 'GET_KEY', 'GET_UC', 'GET_UC_NOECHO';

Since the above construction only RETURNS a value (it is not passed one) it may only appear on the right side of the '=' sign in an assignment expression. For

example the compiler will reject the following construction:

```
PROCVAR1 = 1;
```

A procedure that is passed a variable but does not return one can be used in constructions similar to the one above. Procedures of this nature generally manipulate AT or GLOBAL variables, send data to an external procedure, or send data to an I/O device. For example:

```
PROCEDURE PROCVAR2(BYTE CHAR);
   IO_PORT = CHAR;
ENDPROC;
```

now we can use the construction:

```
PROCVAR2 = 'A;
```

Since the preceeding construction is only PASSED a value (it does not return one) it may only appear on the left side of the '=' sign in an assignment expression. For example the compiler will reject the following construction:

```
VAR1 = PROCVAR2;
```

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                                                       *
*  YOU CAN USE A PROCEDURE THAT IS PASSED A             *
*  VARIABLE JUST AS YOU WOULD ANY OTHER VARIABLE        *
*  IN PL/9 BUT IT MAY ONLY APPEAR ON THE LEFT SIDE      *
*  OF AN ASSIGNMENT EXPRESSION. IF IT APPEARS ON        *
*  THE RIGHT IT MUST RETURN A VALUE.                    *
*                                                       *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

Further examples of procedures that are passed variables but do not return one can be found in IOSUBS.LIB (q.v.). The procedures of interest are: 'PUTCHAR', 'PRINTINT', 'PRINT', 'SPACE' and 'CURSOR'.

7.01.10 PROCEDURE (continued)

A procedure that is passed a value and returns a value after manipulating it is classified as a FUNCTION procedure and may be included in an expression as complex as you wish. For example:

```
PROCEDURE PROCVAR3(BYTE CHAR);
ENDPROC CHAR;
```

This is pretty dumb as the procedure does absolutely nothing! But it does serve to illustrate how a value 'passes through' a function procedure. The following two examples will both result in VAR2 being assigned to VAR1;

```
VAR1 = VAR2;

VAR1 = PROCVAR3(PROCVAR3(PROCVAR3(PROCVAR3(PROCVAR3(PROCVAR3(VAR2))))));
```

It shouldn't be too hard to figure out that if you insert instructions between the procedure declaration and the endproc that the procedure can do virtually anything it wishes to the the incoming 'CHAR' before it returns it. You can also return the address of (a pointer to) the incoming variable with the following basic construction:

```
PROCEDURE PROCVAR3(BYTE CHAR);
ENDPROC .CHAR;
```

Further examples of procedures that behave as functions can be found in IOSUBS.LIB (q.v.) and SCIPACK.LIB (q.v.). The procedures of interest are: 'INPUT' and 'LOG', 'EXP', 'ALOG', 'XTOY', 'SIN', 'COS', 'TAN', 'ATN' respectively. The functions in the 'SCIPACK' library should give you some insight into the complexity permitted in procedures that behave as functions. Examples of procedures that return pointers can be found in the STRSUBS and BASTRING libraries.

7.01.11 ENDPROC, RETURN, and ENDPROC END

These keywords provide the mechanism for terminating PROCEDURES.

The RETURN statement can be used at any point in the procedure (or not at all) while ENDPROC can only be the last statement of the procedure.

In either case virtually anything that can be represented by a single REAL, INTEGER or BYTE may be passed back to the calling program. This makes the procedure a function subroutine. All you have to do is state what incoming, global, local variable, register (ACCA, ACCB, ACCD, XREG, STACK, CCR), constant, value or pointer is to be handed back to the calling program. If you need to pass additional information back to the calling procedure that cannot be represented by a single BYTE, INTEGER, or REAL you will have to do this via GLOBALs.

It is worth remembering that the 'sign' of a number is often an easy way of passing more than one bit of information back to the calling procedure. Suppose, for example, your procedure was designed to return an ASCII code in the range of $00 to $7F, all of which are POSITIVE values. If an error occurred you could simply 'RETURN TRUE' which is -1. The calling procedure would then only have to test to see if it was being returned a negative number to acertain whether an error occurred.

ENDPROC END has a special purpose and, if used, is only used in the last procedure (the main one) of the PL/9 program.

In each of the following examples we are not concerned with what is going on inside the procedures, we are only illustrating the various methods of TERMINATING procedures.

ENDPROC

```
PROCEDURE ONE;
   .
   .
ENDPROC;
```

Procedure ONE is not passed any variables, nor does it use any local variables. This procedure only makes any sense if it is operating on GLOBAL or AT variables. In this case the ENDPROC generates a simple RTS (return from subroutine) instruction.

```
PROCEDURE TWO:REAL DELAY;
   .
   .
ENDPROC;
```

Procedure TWO is not passed any variables but it does declare a local variable, in this case a REAL. As the name of the variable suggests this routine may be structured as a delay routine that uses the local variable as a loop counter. This procedure, like any other non interrupt procedure, can access either GLOBAL or AT variables as well. Since a local variable is declared and the stack pointer offset to accommodate it the ENDPROC in this case generates the code necessary to return the stack pointer to the position it was in when it entered the procedure before generating the RTS instruction.

```
PROCEDURE THREE(INTEGER INPUT:BYTE CHAR):BYTE COUNT;
   .
   .
ENDPROC;
```

7.01.11 ENDPROC, RETURN, and ENDPROC END (continued)
ENDPROC (continued)

Procedure THREE is passed two variables on the stack. One is an integer the other is a BYTE. A local variable, a BYTE, is also declared and room made on the stack. This time the ENDPROC will again remove the local variable allocation from the stack before generating the RTS. The stack allocation for the variables passed to the procedure will be tidied up by the calling program.

```
PROCEDURE FOUR:BYTE DATA;
   .
   .
ENDPROC DATA;
```

Procedure FOUR declares one local variable, a BYTE, which is also returned to the calling program via the 'ENDPROC DATA' declaration. Since this procedure returns a value it is called a 'function procedure'.

When a procedure returns a variable it will return it in the B accumulator, D accumulator or a combination of the D accumulator and the X register depending on whether the variable is a BYTE, an INTEGER or a REAL respectively.

The compiler will choose from BYTE or INTEGER according to the size of the value being returned. REAL values can only be returned if they are specified explicitly.

The type of the variable to be returned should ALWAYS be specified if the procedure is working with a mixture of BYTEs, INTEGERs, or REALs OR if the expression is complicated. It never does any harm to specify the type every time as it does not cause ANY extra code to be generated, it only serves as an instruction to the compiler itself and ensures that the procedure returns the data in the form you want it.

In the above example this would be: 'ENDPROC BYTE DATA'. Since a local variable has been declared the stack will be tidied up by the ENDPROC before generating the RTS.

There is one other very important point to note when passing a variable back to the calling procedure. The size of the data returned must match the size of the data the CALLING procedure is expecting. For example if the calling procedure is expecting a function subroutine to return a REAL and you accidentally (or intentionally) return a BYTE one of two things is going to happen. The first is that the compiler will make an automatic conversion to the data type it expects to get back...this may not be desirable in many instances. The second is that the compiler may reject it.

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                                                               *
*  IT IS UP TO THE PROGRAMMER TO ENSURE THAT RETURNED DATA      *
*  SIZES MATCH THOSE EXPECTED BY THE CALLING PROGRAM.           *
*                                                               *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

```
PROCEDURE FIVE(REAL DATAIN1:INTEGER DATAIN2:BYTE DATAIN3):BYTE COUNT;
   .
ENDPROC REAL DATAIN1;
```

7.01.11 ENDPROC, RETURN, and ENDPROC END (continued)
ENDPROC (continued)

Procedure FIVE illustrates the point just raised. Here we are going to perform an operation comprising a mixture of REALs, INTEGERs and BYTEs. In order to ensure that PL/9 returns the proper data size to the calling program we specified REAL after the ENDPROC. This is particularly important when a REAL variable is to be returned to the calling procedure for, in the absence of any instructions to the contrary, PL/9 will return an INTEGER or BYTE value. When you wish to return a REAL you MUST state so explicitly.

An alternative form, available in PL/9 versions 4.XX onward, allows you to fix the size of the returned variable in the procedure declaration:

```
PROCEDURE FIVE(REAL DATAIN1:INTEGER DATAIN2:BYTE DATAIN3):BYTE COUNT:REAL;
   .
ENDPROC DATAIN1;
```

In this example the ':REAL;' declares a REAL variable without any name. The compiler will interpret this to mean that you wish to force the returned value to be REAL. You can also use BYTE and INTEGER in the same manner. This arrangement is particularly useful in recursive function procedures.

```
PROCEDURE SIX(BYTE CHAR);
   CHAR = CHAR + 1;
ENDPROC CHAR;
```

Procedure SIX illustrates the basic form of a FUNCTION procedure again. This type of simple procedure can be shortened to the following form:

```
PROCEDURE SIX(BYTE CHAR);
ENDPROC CHAR + 1;
```

This latter construction illustrates that you can use the construction:

```
ENDPROC <EXPRESSION>;
```

where <EXPRESSION> may be as complex as required. <EXPRESSION> can take virtually any form, for example:

```
ENDPROC 1;        (return the number 1)

ENDPROC CON;      (where CON is a constant)

ENDPROC REAL SINE((COUNT+100)*(COUNT-3));
```

Where <EXPRESSION> contains a mixture of BYTEs, INTEGERs and REALs or you wish the procedure to be forced to return a particular data size you must specify the size of the variable to be returned in form indicated in the last example above, unless you have forced the size in the procedure declaration. The basic form is as follows:

```
ENDPROC <SIZE> <EXPRESSION>;          where <SIZE> is BYTE, INTEGER or REAL.
```

RETURN

RETURN is virtually identical to ENDPROC in the way it works. It may be used to pass variables back to the calling procedure and it always tidies up the local variable allocation on the stack.

RETURN has one primary purpose...to terminate a procedure early when some condition or conditions are met. It will be most frequently used in multi-tasking type programs that need a method of terminating programs early when some specific event takes place or as a mechanism to prevent a variable from being operated on by subsequent elements of a procedure. RETURN provides a convenient method of terminating a procedure at anytime, or at any point.

RETURN also provides a mechanism for overcoming one of the restrictions governing the use of BREAK in nested loops, that is to break out of a very deep nest of REPEAT...UNTIL or WHILE... loops.

There is absolutely no limit to the number of opportunities you may take to terminate a procedure with RETURN.

There are a couple of points to note when using RETURN in function procedures that are expected to return a variable. Whether you return a variable to the calling procedure via the RETURN statement or not will largely be determined by how your procedure is constructed and what the calling program expects to get back. Take the following two examples to illustrate the point.

(1) When the procedure that is calling the function procedure is part of a series of function procedures operating on some element of data RETURN should always return something. In this instance RETURN will probably be used as one of the conditional exits of the procedure in order to return VALID data and prevent any further operations taking place on the data involved. For example a negative number has been detected and the remainder of the procedure is designed to operate on positive values.

(2) An example of when it makes no difference to the calling procedure that was expecting a variable to be returned is when, for example, a global flag has been set, to inform the calling program that what is being returned is meaningless as, for example, an error has occurred. In this case the calling procedure is probably assigning the result of the function procedure to an intermediate variable and then testing the status of the error flag. If the error flag is set the returned information is ignored.

Since returned values are always returned in 'B', 'D' or 'D & X' (BYTE, INTEGER, and REAL respectively) there is never any problem of stack allocation if you fail to return a value. The point is whether these registers will contain meaningful data or not.

If you can read between the lines of the above statement it implies that only ONE variable, a BYTE, an INTEGER or a REAL may be returned by a function procedure used in an assignment expression (i.e. A=B). If you need to pass back more than one value it is impossible to do it via an assignment expression anyway so your only recourse is to use global variables.

In order to effectively illustrate the points we have just raised we will have to use the IF...THEN...ELSE construction described in the next section and the REPEAT...UNTIL/FOREVER construction described in another section. If what we are doing in the following examples is not obvious then read the relevant sections before going any further.

7.01.11 ENDPROC, RETURN, and ENDPROC END (continued)
RETURN (continued)

```
PROCEDURE ONE;
   IF PORT1 <> 0 THEN RETURN;
   REPEAT
     .
     .
   UNTIL PORT2=SWITCH2;
ENDPROC;
```

In this example we are using RETURN as part of a conditional argument. In this case to abort the procedure the moment it is entered if PORT1 is not equal to zero.

```
PROCEDURE TWO;
   REPEAT
     IF PORT1 <> 0 THEN RETURN;
     .
     .
  UNTIL PORT2=SWITCH2;
ENDPROC;
```

In this example we are again using RETURN as part of a conditional argument but we have put it within the body of a REPEAT...UNTIL loop as a method of terminating the loop AND the procedure should PORT1 not be equal to zero. As there is no telling how many iterations of the loop will have to take place before PORT2=SWITCH2 the RETURN statement is being used as a method of checking PORT1 for some priority condition whilst inside what could be a lengthy loop.

```
PROCEDURE THREE;
   REPEAT
      IF PORT1<>0 THEN RETURN;
      .
   UNTIL PORT2=SWITCH2;

   REPEAT
      IF PORT1<>0 THEN RETURN;
      .
   UNTIL PORT3=SWITCH3;
   .
   .
ENDPROC;
```

This is just an expansion of the above theme and serves to illustrate that RETURN can be used several times as a mechanism for premature termination of a procedure. In this case we are looking for some priority event no matter where we are in the procedure.

7.01.11 ENDPROC, RETURN, and ENDPROC END (continued)
RETURN (continued)

```
PROCEDURE FOUR;
   REPEAT ---------------------------------------+
      IF PORT1<>0 THEN RETURN;                    |
                                                  |
      REPEAT ------------------------------+      |
         IF PORT1<>0 THEN RETURN;          |      |
                                           |      |
         REPEAT ----------------------+    |      |
            IF PORT1<>0 THEN RETURN; |    |      |
            .                      (3)  (2)  (1)
            .                       |    |      |
         UNTIL PORT4=SWITCH4; -------+    |      |
         .                                |      |
         .                                |      |
      UNTIL PORT3=SWITCH3; ---------------+      |
      .                                          |
      .                                          |
   UNTIL PORT2=SWITCH2; ----------------------+
ENDPROC;
```

This procedure illustrates how RETURN can be used to break out of nested loops, and terminate a procedure at ANY time when some specific event takes place. Since we have not covered REPEAT...UNTIL loops the actual loop within a loop within a loop is defined by the markers (1), (2), and (3) with loop (3) being the most deeply nested. If, for example, we were inside of loop (3) and PORT1 was read as being equal to zero the procedure would be terminated instantly.

We mentioned earlier that there is a limitation with BREAK in this respect. If BREAK were used in lieu of RETURN in the above construction and we were inside of loop (3) and the PORT1<>0 statement were true we would only break out of the innermost loop. We would still be trapped within loops (2) and (1). This is due to the fact that the compiler is not clairvoyant! How is it supposed to know that you want to break out of loops (3), (2) and (1)? Maybe you only want to break out of (3). If you want to completely break out of nested loops build them into a procedure and use RETURN to terminate it when you want to. Alternatively use a global flag as a method of signalling the outer loops that a break condition has been detected by an inner loop.

```
PROCEDURE FIVE(REAL COUNT);
   IF COUNT=100 THEN RETURN REAL SINE((COUNT+100)*(COUNT-3));
   IF COUNT=200 THEN RETURN REAL COS((COUNT+10)+(2+(COUNT*5)-4));
   IF COUNT=210 THEN RETURN REAL TAN(COUNT+20);
   .
   .
ENDPROC REAL 3.14;
```

Here is an example of terminating a procedure at any one of several points and returning a value at each one of them.

This construction also amplifies two points that we raised previously. The first is that you can perform arithmetic operations of almost any complexity, including the use of other function procedures (in this case some of the functions in the SCIPACK library), as part of the RETURN operation just as you can with 'ENDPROC <EXPRESSION>. The second is that you can return a value, in this case the 3.14 in the ENDPROC statement. You could also return a CONSTANT that has been declared previously, or a pointer to a variable if this is what was required.

ENDPROC END

A special point concerns the last procedure in a program. If the ENDPROC is left off, then PL/9 generates a "JMP $CD03", causing control to be returned to FLEX when (and if) the program ends.

If just an ENDPROC is placed at the end of the last procedure only the local variables on the stack used by the last procedure will be tidied up. This will then be followed by an RTS ($39). This is fine providing that no GLOBAL storage was allocated and that you are not worried whether the PL/9 program corrupts the 6809 registers or not.

If it is desired, however, that the program behave in its entirety as a subroutine (so that it can be called from BASIC, for example) then it is

important that the stack pointer and 6809 registers are returned with the values the PL/9 program was entered with.

If there is a GLOBAL statement at the start of the program then the entire incoming register set will have been pushed onto the stack and then some space will have been reserved on the stack for the global variables. To tell PL/9 that the program has in fact ended and that the reserved space should now be released and the saved registers recovered, the ENDPROC END form should be used.

ENDPROC END; will cause PL/9 to generate the following basic code:

```
LEAS  n,S                 Release the stack space allocated to the GLOBALs.
PULS  CC,D,DP,X,Y,U,PC    Recover the stacked registers and return address.
```

The PULS PC in this case acts like the RTS instruction. This will allow the entire PL/9 program to behave as a single subroutine. It MUST be called at its first address (not the address given for the main program in the symbol table or compile listing). The STACK command MUST be avoided in this case.

N O T E

Further (technical) information on how variables are passed to and returned from function procedures can be found on the section covering ASMPROCs.

7.01.12 IF...THEN...ELSE IF...CASE1.THEN...CASE2.THEN...

The IF keyword allows program statements to be executed conditionally upon the result of some test. There are two forms of an IF statement:

```
(1) IF <EXPRESSION> THEN <STATEMENT> [ ELSE <STATEMENT> ]


(2) IF <EXPRESSION> CASE <NUMBER> THEN <STATEMENT>
                  [ CASE <NUMBER> THEN <STATEMENT> ]
                     .
                     .
                     .
                     .
                  [ CASE <NUMBER> THEN <STATEMENT> ]

                  [ ELSE <STATEMENT> ]
```

the square brackets implying that the contents are optional. The second form allows a multi-way branch, with the expression being tested against a number of values and an optional default at the end if none of them match. <STATEMENT> may be any simple or compound statement (even another IF statement). <CONDITION> may be any arithmetic, logical or relational expression, such as the following:

```
IF X THEN ...    (implicit <> 0)

IF VALUE = 66.3 THEN ...

IF A*2+B < 14 THEN ...

IF CHAR < '0 .OR CHAR > '9 THEN ...

IF REPLY
   CASE 'Y THEN CONTINUE;
   CASE 'N THEN STOP;
ELSE PRINT("WHAT?\N");
```

In all cases, if the <EXPRESSION> is true the THEN clause (or one of them if the CASE form is used) will be executed; if not, the ELSE clause, if present, will be executed.

It is important to note that when the first 'CASE' is found to be true all subsequent CASE arguments will be skipped and control passed to the line right after the last CASE argument (or after the ELSE statement if present)

If, and only if, all CASE statements are not true will the ELSE statement be executed.

NOTE: The CASE statement may NOT contain any further <EXPRESSION>'s. For example CASE '1 .AND PORT=2 THEN... is a definite no-no. If you wish to expand a CASE argument you must use the BEGIN...END pair (q.v.).

7.01.13 BEGIN...END

These keywords are used to bracket a group of statements so as to become one compound statement. A BEGIN .. END block is indivisible, that is to say that it is equivalent in every way to a simple statement. To clarify this point, an example:

```
IF VALUE = 59
   THEN BEGIN
        COUNT = COUNT + 1;
        NUMBER = VECTOR(COUNT);
        END;
```

If VALUE does happen to be 59, the compound BEGIN ... END statement will be executed; otherwise execution will pass to the statement following the END, that is the entire BEGIN...END block will be skipped.

Note that BEGIN is not followed by a semicolon but the matching END is. The compiler will forgive you, more often than not, if you accidentally put a semicolon after the BEGIN statement. You should, however, try to remember NOT to put the semicolon after BEGIN. This message is primarily for programmers used to working with SPL/M where the 'THEN DO;' statement, which is similar, must be terminated with a semicolon.

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                                                                           *
*  YOU MAY USE ARGUMENTS OR ASSIGNMENTS, REGARDLESS OF COMPLEXITY, WITHIN   *
*  A BEGIN...END PAIR WHEREVER YOU WOULD USE A SIMPLE STATEMENT.            *
*                                                                           *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

The BEGIN...END pair can also be used to expand the operations performed by IF...CASE...THEN...ELSE or WHILE... constructions, for example:

```
IF CHAR1 <> CHAR2
   THEN BEGIN
        CHAR1=2;
        CHAR2=1;
        END;
   ELSE BEGIN
        CHAR1=5;
        CHAR2=2;
        END;
```

```
IF CHAR
   CASE '1 THEN BEGIN
                CHARA=1;
                CHARB=2;
                END;
   CASE '2 THEN BEGIN
                CHARA=3;
                CHARB=6;
                END;
          ELSE BEGIN
                CHARA=0;
                CHARB=0;
                END;
```

7.01.14 LOGICAL .AND, .OR, .EOR (.XOR)

These keywords and their preceeding period (.) form the logical operators. The bitwise functions use the same words but lack the period. The logical forms can be used to greatly simplify control arguments, whilst the bitwise functions (which are described elsewhere) are used for bit manipulation of data. There is no practical limit to the complexity of the argument you can develop using these keywords but ...

LOGICAL OPERATORS MAY NOT BE BRACKETED!

For example to perform a given operation when ALL of several distinct expressions are true:

```
IF A=1 .AND B=2 .AND C=3
   THEN...
```

To perform a given operation when ANY one of several expressions are true:

```
IF A=1 .OR B=2 .OR C=3
   THEN ...
```

To perform a given operation when EITHER (but not both) of two expressions is true:

```
IF A=1 .EOR B=1   (the alternative form .XOR means the same)
   THEN ...
```

If you require to construct a bracketed argument you must do it via IF...THEN statements, for example to construct the following argument in PL/9

```
IF (A=B .AND B=C) .OR (A=C .AND B=D) THEN... you would use


IF A=B .AND B=C
   THEN...
ELSE IF A=C .AND B=D
   THEN...
```

You can avoid duplicating the THEN... statement if you use a flag to signify that one of the arguments was true, i.e.:

```
FLAG=0;
IF A=B .AND B=C
   THEN FLAG=1;
IF A=C .AND B=D
   THEN FLAG=1;
IF FLAG <> 0
   THEN...
```

7.01.15 WHILE

The WHILE construct allows a statement or group of statements to be executed repeatedly for as long as a specified condition is true. The syntax is:

```
WHILE <CONDITION> <STATEMENT>
```

This construction ALWAYS tests <CONDITION> at the start of the loop. If <CONDITION> is not true then the body of the loop, represented by <STATEMENT> will not be entered. This is in contrast to the REPEAT...UNTIL construction, which is in the next section, which ALWAYS executes the body of the loop at least once. Providing these two different loop control constructions provides the programmer with 'the proper tool for the job' as each mechanism has its own particular use in structured programs. NOTE: <STATEMENT> must always be present or it must be replaced by 'BEGIN END;'.

The WHILE construction can be a simple mechanism for locking onto a particular signal from an an I/O port, for example:

```
WHILE PORT <> ESCAPE    BEGIN END;
```

In this case <STATEMENT> is replaced by 'BEGIN ... END'. The program produced will simply sit in a very tight loop until PORT=ESCAPE at which time the loop will terminate. Note that when there is no <STATEMENT> you MUST terminate the construction with 'BEGIN END;'

```
COUNT=$FFFF;
WHILE COUNT <> 0
   COUNT=COUNT-1;
```

This time we have built a simple delay loop by incorporating <STATEMENT> which decrements count on each iteration until the condition <> 0 is met at which time the loop terminates. The '<> 0' may be implicitly stated if desired. This construction has the advantage of producing less code and therefore executing faster. The following is functionally identical to the above:

```
WHILE COUNT  /* implicit <> 0 */
   COUNT=COUNT-1;
```

The while construction can be expanded by BEGIN...END when required. For example, assume that a procedure called ACTION is to be executed repeatedly and COUNT incremented each time, until a character is detected as having been received by ACIA (an MC6850). The code to perform this task might be as follows:

```
COUNT=0;
WHILE (ACIA AND 1) = 0
   BEGIN;
      ACTION;
      COUNT=COUNT+1;
   END;
```

In this case the BEGIN ... END block is equivalent to <STATEMENT> in the formal definition. If it is necessary to terminate the WHILE loop prematurely, a BREAK statement (q.v.) may be used. WHILE loops may be nested inside other WHILE loops, IF statements or REPEAT .. UNTIL loops to a depth only limited by available stack space during compilation.

7.01.16 REPEAT...UNTIL REPEAT...FOREVER

This construct operates in a similar manner to WHILE, except that the test for completion of the loop occurs at the end rather than at the start. The syntax is

```
REPEAT <STATEMENT> UNTIL <CONDITION>
```

For example, suppose that 25 elements of VECTOR1 (elements 0-24) have to be copied to VECTOR2. In BASIC this is where a FOR-NEXT loop might have been used, but that construct is not available in PL/9:

```
COUNT=0;
REPEAT
   VECTOR2(COUNT) = VECTOR1(COUNT);
   COUNT=COUNT+1;
UNTIL COUNT=25;
```

A special case REPEAT ... FOREVER allows the programmer to set up an endless loop that can only be broken out of by means of a BREAK statement when it is part of the MAIN procedure or by RETURN if it is part of a procedure used as a subroutine. For example:

```
REPEAT
   IF (ACIA AND 1)=1 THEN BREAK;
   .
   .
   .
FOREVER;
```

The REPEAT...FOREVER construction is more commonly used to form the MAIN procedure of a control program that is designed to run forever.

7.01.17 BREAK

The BREAK statement causes the current WHILE or REPEAT loop to be broken out of prematurely to its normal termination, as in the previous example. Program execution will continue at the statement following the end of the CURRENT loop.

Up to ten BREAKs may be pending at any time in a given construction. Once the construction ends you may start another, which may also have up to ten BREAKs pending before the main (or only) loop terminates.

```
REPEAT
   IF PORT=1 THEN BREAK;
FOREVER;
```

In the above example the BREAK statement is the only conditional exit of a REPEAT...FOREVER loop.

```
REPEAT
   IF PORT=1 THEN BREAK;
   COUNT=0;
   REPEAT
      COUNT=COUNT+1;
   UNTIL COUNT=25;
FOREVER;
```

This construction will also work properly. In this construction the BREAK statement will pass control to the line just after FOREVER.

```
REPEAT
   REPEAT
      IF PORT=1 THEN BREAK;
   FOREVER;
   IF PORTA=PORTB THEN BREAK;
FOREVER;
```

This example illustrates that BREAK will only terminate the current loop. The only way to get out of the main REPEAT...FOREVER loop is when PORTA = PORTB. When PORT=1 only the inner REPEAT...FOREVER loop will be exited. If you need to break out of both loops when PORT=1 then you have two options. The first is to consign the loops to a separate procedure and use RETURN in lieu of BREAK. The second is to use a flag to inform the outer loop that a break condition occurred in the inner loop, for example:

```
FLAG=FALSE;
REPEAT
   REPEAT
      IF PORT=1
         THEN BEGIN
              FLAG=TRUE;
              BREAK;
              END;
   FOREVER;
   IF FLAG=TRUE .OR  PORTA=PORTB THEN BREAK;
FOREVER;
```

7.01.18 GOTO

GOTO allows the program to jump unconditionally BACK to some previous point in the current procedure. The destination of the GOTO is a label followed by a colon. For example:

```
LOOP:
   IF ACIA AND 1 = 0
      THEN BEGIN;
           COUNT = COUNT+1;
           GOTO LOOP;
           END;
   ELSE .....
```

GOTO may only be used as a control mechanism WITHIN THE CURRENT PROCEDURE. It may not be used to pass control to a point outside of the current procedure, i.e. branch to a location in a previous or subsequent procedure.

GOTO cannot be used to branch forward in a procedure (i.e. branch to a line that follows the GOTO statement. Remember PL/9 is a single pass compiler!

7.01.19 CALL

To call a PL/9 procedure or an ASMPROC (q.v.) it is only necessary to type its name and any parameters that may be required. External subroutines, defined by AT or CONSTANT statements or computed by the program, can be accessed by means of the CALL statement.

If a parameter is to be passed it must be put in one of the 6809's registers using ACCA, ACCB, ACCD, XREG or stored in a 'hard' (AT) memory location. No safeguards exist as to whether it is appropriate to CALL a subroutine; it is up to the programmer to know what he is doing.

CALLing a routine defined by a CONSTANT declaration or in the form CALL $XXXX will produce EXTENDED addressing, i.e. JSR $XXXX. This means control will be passed TO address $XXXX.

CALLing a routine defined by an 'AT' declaration or computed by the program in a vector will produce INDEXED addressing i.e. LDD $XXXX, TFR D,X, JSR 0,X. This means that control will pass to the address CONTAINED in $XXXX.

These two distinctions are very important. The first is for working with external subroutines at fixed addresses. The second is for accessing routines through RAM or ROM vector tables, such as those found in system monitors.

An example of a subroutine at a fixed address is GETCHR at $CD15 in FLEX. This routine actually resides at some place other than $CD15. Since FLEX has a JMP XXXX at this location the effect is the same and may be entered directly at this point in order to access the routine.

An example of an address stored in a vector is MONITR at $D3F3 in FLEX. In this instance memory location D3F3/4 contains the address of the routine we want to enter. If we did a simple CALL $D3F3 the system would probably crash as $D3F3 would not contain sensible executable code...IT CONTAINS AN ADDRESS!. To gain access to the routine pointed to by the contents of $D3F3 we define it via an 'AT' statement, viz: 'AT $D3F3:INTEGER MONITR;', then we can say 'CALL MONITR;'

You may also use CALL to access elements stored in vectors generated by PL/9. There are two cases to consider. The first is when the vectors are read-only variables and the second is when the vectors are in RAM.

When vectors are declared as read-only data they need not be initialized, viz:

```
INTEGER TABLE $F804, $F806, $F808;

CALL TABLE(0); ... CALL TABLE(1); ... CALL TABLE(2);
```

When the vectors are declared as program variables, either global or local, they can be used to dynamically control program flow. In this instance the variables must be initialized by the users program before they are used, viz:

```
GLOBAL INTEGER TABLE(3);

PROCEDURE;
   TABLE(0) = $F804;
   TABLE(1) = $F806;
   TABLE(2) = $F808;

   CALL TABLE(0); ... CALL TABLE(1); ... CALL TABLE(2);
```

NOTE: The only registers you MUST preserve when leaving a PL/9 procedure are 'DP' if DPAGE is being used and 'Y' if GLOBALS are being used.

7.01.20 JUMP

The rules that apply to JUMP are identical to those applicable to CALL. The only difference between the two is that the return address will be preserved in the case of CALL and it won't in the case of JUMP.

JUMP is normally used as a method of either starting a PL/9 program up (from the RESET procedure) or terminating it by jumping to another program. It can also be used as a means of re-starting a PL/9 program when a specific condition is met.

NOTE: The only registers you MUST preserve when leaving a PL/9 procedure are 'DP' if DPAGE is being used and 'Y' if GLOBALS are being used.

7.01.21 GEN

Where assembly language procedures are required as part of a PL/9 program, the GEN statement allows any sequence of 6809 instructions to be inserted into the program. The syntax of GEN is as follows:

```
GEN $7E,$CD,$03;  /* JUMP TO FLEX */
```

GEN statements may also be assigned via BYTE sized constants. INTEGER sized constants MAY NOT be used with GEN statements as they will be truncated to bytes and produce completely erroneous results:

```
CONSTANT JMP=$7E, FLEXHI=$CD, FLEXLO=$03;
GEN JMP,FLEXHI,FLEXLO;
```

This latter feature can improve the readability of frequently used GEN statements or in situations where the address/data involved in the GEN statement needs the flexibility provided by declaring a constant at the beginning of the program.

Note that it is advisable to have a good understanding of how PL/9 accesses variables before attempting to perform operations on the stack, otherwise the program will almost certainly crash.

The PL/9 trace debugger will only work if the compiled code is 100% correct; it will not locate faults caused by faulty understanding of the operation of the compiler!

It is worth noting that our assembler, MACE, has built in capabilities to produce GEN statements from assembly language programs. As an added benefit the assembly language source is passed into the output file as comments against the GEN statements.

GEN statements may be used freely BETWEEN PL/9 control statements. This is because the code generated by PL/9 does not assume that the registers are in any particular state from one control statement to the next. This is why the code produced by PL/9 may seem wasteful to you clever assembly language programmers!

The only consideration is that you preserve 'DP' if you are using the DPAGE directive and that you preserve 'Y' if you are using globals. In fact you can use GEN statements to do just that. For example:

```
TO PRESERVE THE 'DP' REGISTER:    GEN $34,$08       (PSHS DP)

TO PRESERVE THE 'Y'  REGISTER:    GEN $34,$20       (PSHS Y)

TO PRESERVE 'DP' AND 'Y':         GEN $34,$28       (PSHS DP,Y)


TO RECOVER THE 'DP' REGISTER:     GEN $35,$08       (PULS DP)

TO RECOVER THE 'Y' REGISTER:      GEN $35,$20       (PULS Y)

TO RECOVER 'DP' AND 'Y':          GEN $35,$28       (PULS DP,Y)
```

One last consideration is that the routine being called should preserve the state of the 'F' and 'I' bits (the FIRQ and IRQ mask bits) in the CCR if your PL/9 program is using either of these interrupt sources.

7.01.22 ASMPROC

The ASMPROC keyword defines a special kind of procedure which is made up of only GEN statements. It is in effect a label that allows the programmer to include routines written in assembly language, for speed or to provide facilities not available in PL/9, such as I/O device drivers or extended precision arithmetic routines.

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                                                               *
*  ASMPROCS MUST BE WRITTEN IN POSITION INDEPENDANT CODE        *
*  (PIC) IF YOU WANT TO USE THE PL/9 TRACER OR PRESERVE         *
*  THE PIC MODULARITY OF THE PL/9 PROGRAM THAT USES THEM        *
*                                                               *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

If any parameters are to be passed to an ASMPROC then the size, BYTE, INTEGER or REAL, of each parameter must be specified in the declaration, as follows:

```
ASMPROC ONE;                              /*  No parameters passed      */
ASMPROC TWO(BYTE);                        /*  One BYTE parameter passed */
ASMPROC THREE(INTEGER,INTEGER,BYTE);      /*  Three parameters passed;
                                              two INTEGER and one BYTE  */
ASMPROC FOUR(REAL);                       /*  One REAL parameter passed */
```

In the above examples, the INTEGER type is used both for integer-sized values and for pointers to both integer and byte vectors. The important factor is the number of bytes on the stack rather than what the bytes actually represent.

In the second example above 'BYTE' will be at 2,S (the return address is at 0,S and 1,S). Thus it may be accessed by the mnemonic 'LDB 2,S'. This follows the PL/9 convention of passing BYTEs around in 'B' so they can be sign extended to form an INTEGER in 'D'.

In the third example above the first 'INTEGER' will be at 5,S/6,S, the second 'INTEGER' will be at 3,S/4,S and the 'BYTE' will be at 2,S. Thus they may be accessed by 'LDD 5,S', 'LDD 3,S' and 'LDB 2,S' respectively. This register allocations used above follow the PL/9 convention of passing INTEGERs around in 'D' with 'A' containing the most significant byte and 'B' containing the least significant byte. Thus the INTEGER may be converted to a BYTE by simply ignoring 'A' (assuming that the INTEGER does not contain a value that cannot be held in a BYTE).

In the fourth example the 'REAL' will be on the stack at 2,S through 5,S. To get the REAL into the same registers that PL/9 processes REALS in you would use the mnemonics 'LDD 2,S' and 'LDX 4,S'. This follows the PL/9 convention of passing REALs around in 'D' and 'X' with 'D' containing the exponent (8 bits in 'A') and the most significant 8 bits of the mantissa in 'B' and 'X' containing the least significant 16 bits of the mantissa. There is further information on the REAL number format further along in this section.

R E M E M B E R

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                                                               *
*  PL/9 PUSHES THINGS ONTO THE STACK IN THE ORDER               *
*  SPECIFIED. THUS THE FIRST ITEM SPECIFIED WILL BE THE         *
*  HIGHEST ITEM ON THE STACK AND THE LAST ITEM SPECIFIED        *
*  WILL BE JUST ABOVE THE RETURN ADDRESS.                       *
*                                                               *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

7.01.22 ASMPROC (continued)

If the ASMPROC is to return a value to the calling program then this must also be indicated, since no ENDPROC or RETURN statements are allowed:

```
ASMPROC FOUR: BYTE;                    /*  Returns a BYTE value  */
ASMPROC FIVE(BYTE,BYTE): INTEGER;      /*  Two BYTE parameters passed;
                                           INTEGER value returned  */
```

The returned variable will be expected to be in the registers normally used by PL/9:

```
BYTE       'B' REGISTER
INTEGER    'D' REGISTER
REAL       'D' and 'X' REGISTERS
```

We will be discussing the passing of variables in more detail in a moment.

An example of an ASMPROC can be found on the next page. ASMPROCS can be generated by hand, but the MACE assembler has facilities for producing them automatically from standard assembler programs.

A good example of the type of thing that ASMPROCs can be used for is the RANDOM NUMBER GENERATOR program that follows:

```
GLOBAL INTEGER RNDVAL(2);

ASMPROC RND(INTEGER);
   GEN $AE,$62;          /*          LDX   2,S       */ (point to RNDVAL)
   GEN $C6,$08;          /*          LDB   #8        */ (loop counter)
   GEN $A6,$84;          /* LOOP     LDA   0,X       */ (get MS byte)
   GEN $48;              /*          ASLA            */
   GEN $48;              /*          ASLA            */
   GEN $48;              /*          ASLA            */
   GEN $A8,$84;          /*          EORA  0,X       */ (EOR bit 28 with 31)
   GEN $48;              /*          ASLA            */ (put into carry)
   GEN $69,$03;          /*          ROL   3,X       */
   GEN $69,$02;          /*          ROL   2,X       */
   GEN $69,$01;          /*          ROL   1,X       */
   GEN $69,$84;          /*          ROL   0,X       */ (rotate into RNDVAL)
   GEN $5A;              /*          DECB            */ (count off)
   GEN $26,$ED;          /*          BNE   LOOP      */ (8 times)
   GEN $39;              /*          RTS             */

PROCEDURE RANDOM;
   RND(.RNDVAL);
ENDPROC RNDVAL(1) AND $7FFF; /* POSITIVE NUMBERS ONLY */
```

To use the random number generator you simply assign the value passed back by RANDOM to another INTEGER variable for subsequent analysis, e.g.:

```
IVAL=RANDOM;

IF IVAL >200 .AND IVAL <1000
   THEN...
```

7.01.22 ASMPROC (continued)

Or if you only want a single copy of the random number for evaluation you just include RANDOM in the expression, e.g.

```
IF RANDOM >1000
   THEN...
   ELSE...
```

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

The original idea for the above routine came from the TSC GAMES package.

7.01.22 ASMPROC (continued)

HOW DATA IS PASSED TO AND FROM PROCEDURES

This section is fairly technical and meant for those of you who are intending to interface PL/9 to assembly language programs and hence assumes that you understand the 6809 mnemonics.

Aside from the ability to pass variables to and from procedures via GLOBAL variables (including AT variables) PL/9 has a built-in mechanism to pass as many variables (of any size) as required to a procedure via the STACK. PL/9 also has a built-in mechanism to return a single variable (of any size) via the 'B' (BYTE), 'D' (INTEGER) or 'D & X' (REAL) register(s). To illustrate this ability the following is an example of the code produced for such an activity:

```
                                        GLOBAL INTEGER I1, I2, I3;
   PSHS  CC,D,DP,X,Y,U
   LEAS  -6,S (allocate global storage)
   TFR   S,Y  (point 'Y' at base of global storage)
                                        PROCEDURE ADDEM(INTEGER X,Y);
                                        ENDPROC INTEGER X + Y;
   LDD  4,S ('Y')
   ADDD 2,S (add the contents of the 'D' accumulator to 'X')
   RTS      (result is in 'D')
                                        PROCEDURE TEST;
                                        I2 = 10;
   LDB  #10
   SEX      (sign extend to form integer)
   STD  2,Y ('I2')
                                        I3 = 20;
   LDB  #20
   SEX      (sign extend to form integer)
   STD  4,Y ('I3')
                                         I1 = ADDEM(I2,I3);
   LDD  2,Y ('I2')
   PSHS D   (put I2 onto stack)
   LDD  4,Y ('I3')
   PSHS D   (put I3 onto stack)
   BSR  ADDEM
   LEAS 4,S (tidy stack)
   STD  ,Y  (put result into I1)
```

The first point to note is that the variables are pushed onto the stack in the order they are typed (i.e. from left to right). In this example I2 was put on the stack followed by I3. The second point to note is that the procedure that the variables are passed to (ADDEM) assumes that the first item declared, 'INTEGER X' in this case, is the lowest item on the stack, i.e. it is just above the return address at 0,S and 1,S. Thus in this example 'X' will correspond to 'I2' and 'Y' will correspond to 'I3'. REALs and BYTEs are treated in exactly the same manner.

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                                                                             *
*  A PROCEDURE THAT IS BEING PASSED VARIABLES MUST DECLARE THEM IN EXACTLY    *
*  THE SAME ORDER  THAT THE CALLING  PROCEDURE PASSES THEM.  THE VARIABLES    *
*  BEING  PASSED  FROM THE CALLING PROCEDURE MUST ALSO BE THE SAME SIZE AS    *
*  THE VARIABLES DECLARED IN THE PROCEDURE BEING CALLED.                      *
*                                                                             *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

7.01.22 ASMPROC (continued)

THE ARCHITECTURE OF REAL (FLOATING-POINT) NUMBERS

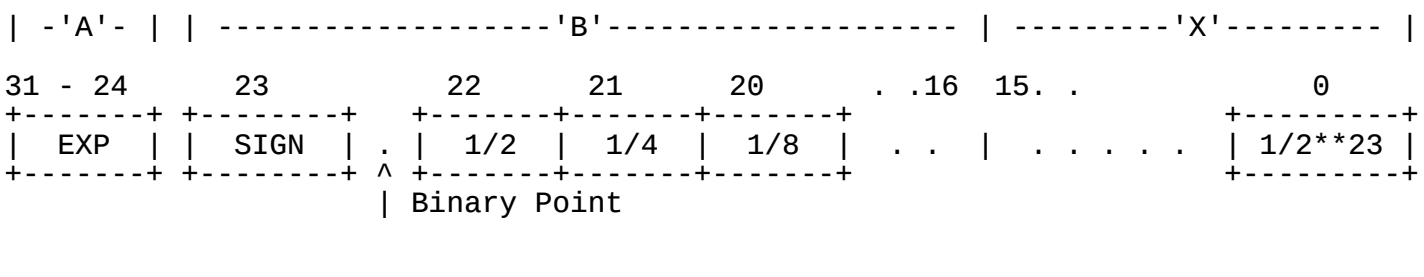
The following is a description of the format used by PL/9 to handle REAL numbers. It was thought to be appropriate to include a discussion on the architecture of REAL numbers in this section as if you wish to manipulate REAL numbers outside of PL/9 (which is what ASMPROCS are all about) you've got to know what makes them tick, right?

It is possible to make speed and code savings by replacing PL/9 expressions with assembly code, particularly where numbers are to be multiplied or divided by powers of two (see the SCIPACK.LIB library for examples). In general, however, it is not worth going to the effort of writing your own routines unless you are unable to perform the required function using PL/9 code.

In the following descriptions, +/- means "plus or minus" and ** means "raised to the power of".

Floating-point numbers occupy four bytes each - that's one for the exponent and three for the mantissa. The exponent is normally held in 'A', the 8 most significant bits of the mantissa in 'B' and the 16 least significant bits of the mantissa in 'X'. The mantissa assumes a binary point just to the right of the most significant bit, so the bit positions rank as follows:

REGISTERS



This gives a range of +/-2**-128 thru +/-2**127 (about +/-10**-37 thru +/-10**37) and a precision of 1 in 2**23 (a little less than seven significant decimal digits).

7.01.22 ASMPROC (continued)

EXAMPLES OF REAL NUMBER BINARY FORMATS

The mantissa for the number 0.5 is 01000000 00000000 00000000, or 40 00 00 in hex, and 0.75 is 01100000 00000000 00000000, or 60 00 00 in hex. In order to maintain 23 bits of precision, bit 22 must be 1, i.e. the number must be left-justified, or "normalised"; the mantissa then represents a number M, where 0.5 <= M < 1.0.

The job of the exponent is to indicate how many times and in which direction the mantissa has been shifted in order for it to be normalised. Therefore, the number 0.5 is represented with a zero exponent:

0.5 is 00 40 00 00 (the first byte is the exponent)

The number 1 is 0.5 times 2, i.e. one shift to the left:

1.0 is 01 40 00 00

Some more numbers:

0.75 is 00 60 00 00
1.5 is 01 60 00 00
3.0 is 02 60 00 00
10 is 04 50 00 00
0.25 is FF 40 00 00

the last one indicating a single shift to the right, i.e. division by 2. Some more examples of numbers less than .5:

0.3125 is FF 50 00 00
0.1 is FD 66 66 66
0.12345678 is FD 7E 6B 76

Negative numbers are represented in 2-s complement form, as follows:

-0.5 is 00 C0 00 00
-1.0 is 01 C0 00 00
-1.5 is 01 E0 00 00
-0.25 is FF C0 00 00

NOTE: When expressing fractional numbers (.5, .33, .678 etc) you must preceed the decimal point with a zero when assigning the number to a REAL variable, e.g 0.5, 0.33, 0.678.

7.01.23 ACCA, ACCB, ACCD, XREG and STACK

These pseudo variables represent the associated MC6809 registers. They are provided primarily to facilitate communication with external assembly language procedures.

ACCA provides amongst other things an interface to operating-system routines that expect data to be passed in the A-Accumulator (PL/9 uses the B-Accumulator for byte quantities). For example, most programs that require terminal I/O through FLEX will need the following two procedures:

```
CONSTANT INEEE=$CD15, /* FLEX GETCHR */
         OUTEE=$CD18; /* FLEX PUTCHR */


PROCEDURE GETCHAR;
   CALL INEEE;                /*  System input character routine  */
ENDPROC ACCA;                 /*  Returns the character typed  */


PROCEDURE PUTCHAR (BYTE CHAR);
   ACCA = CHAR;               /*  Get the character in A  */
   CALL OUTEEE;               /*  System output character routine  */
ENDPROC;
```

Some programmers may be able to make use of ACCB and ACCD in their programs. After any assignment the value saved is always in either the B or D accumulators, depending upon whether it was a BYTE or INTEGER value respectively. If, for example, a number of different variables have to be initialized to the same value, the following can save some code:

```
ACCD = 0;
VAR1 = ACCD;
VAR2 = ACCD;
```

This kind of code-saving is not to be recommended, however, unless you have a good understanding of the code that PL/9 produces. If you really need speed or compactness you should instead consider coding one or more of the more sensitive procedures in assembly language as an ASMPROC.

Normally only one of the pseudo registers may be used at any given time. This is due to the fact that the code produced to load or evaluate a subsequent register is likely to alter the contents of other registers of interest.

See the next section for a set of guidelines on the order of use when you wish to use more than one register to pass data to or receive data from an external program. If the required order restricts you in any way it is best to code the register transfers via GEN statements.

If you don't get the results that you expect from these pseudo variables and CCR, which is described in the next section, examine the code that PL/9 produces and you will probably find what is causing the problem. If you don't understand the code that PL/9 produces you shouldn't be using these pseudo variables in the first place!

7.01.24 CCR

CCR is a psuedo-variable that represents the current contents of the 6809's condition codes register, and that allows assignments to be made and decisions to be taken much as if it were a conventional program variable.

For example, to set the carry flag, use

```
CCR = CCR OR 1;
```

To clear the IRQ interrupt flag, use:

```
CCR = CCR AND IRQMASK;
```

Where IRQMASK has been previously defined as being $EF. A decision can also be made using CCR:

```
IF (CCR AND 1) = 1 THEN ...
```

causes the following statement to be executed if the carry flag is set.

Note the order of the above expressions, it is very important. If we had stated IF (1 AND CCR) = 1 THEN... in the last expression it would not work. This is because the '1 AND' would have loaded 'B' which would destroy the contents of the Z and C bits of the CCR.

Another point to note is that when CCR is involved in a complex argument it is usually best to assign the CCR to a temporary variable before any evaluation takes place. This will ensure that the contents of the CCR will be preserved throughout the evaluation.

If you wish to use CCR or any of the pseudo registers ACCA, ACCB, ACCD, XREG or STACK in combination the order of use is very important. The sequence tables below should explain all:

The following order is to be used when passing variables OUT via registers:

```
STACK = VAR1; (the 'D' accumulator and the Z and C bits of the CCR will be lost)
XREG = VAR1;  (the 'D' accumulator and the Z and C bits of the CCR will be lost)
CCR  = VAR2;  (the 'B' accumulator will be destroyed)
ACCA = VAR3;  (the 'B' accumulator will be destroyed as will Z and C of the CCR)
ACCB = VAR4;  (Z and C bits of the CCR will be destroyed)
ACCD = VAR5;  (Z and C bits of the CCR will be destroyed)
```

The following order is to be used when bringing variables IN via registers:

```
VAR2 = CCR;   (the 'B' accumulator will be destroyed)
VAR4 = ACCB;  (the Z and C bits of the CCR will be destroyed)
VAR3 = ACCA;  (the 'B' accumulator and the Z and C bits of the CCR will be lost)
VAR5 = ACCD;  (the Z and C bits of the CCR will be lost)
VAR1 = XREG;  (the 'D' accumulator and the Z and C bits of the CCR will be lost)
VAR1 = STACK; (the 'D' accumulator and the Z and C bits of the CCR will be lost)
```

It should be obvious from the above that certain combinations, usually involving the CCR or ACCA, are not permitted.

7.01.25 MATHS

There are some programs that have to be built up, piece by piece, perhaps comprising modules written in different languages or by different programmers. In cases such as these, the large-system approach is to use a linking loader to collect all of these modules into a single program. In a control environment, however, it may be that some of the modules required by the application are already loaded, possibly in EPROM, and that the programmer just wishes to alter a small part of the program without having to re-load everything. For this reason the program may well be constructed of a number of parts, each essentially independent of each other but communicating with each other through some agreed area of memory.

In a PL/9 program, subroutines are used to perform integer multiplication and division and all floating-point operations. These routines are copied from the compiler to the output file (or memory) the first time they are invoked. The first two are not very large, but the floating-point pack occupies some 1k bytes. If these math functions are to be included in every module of a fragmented program there will be a considerable waste of memory involved.

The MATHS keyword allows the programmer to load the maths library at any specified address in the 6809's memory space. If all program modules have the same declaration they will each independently load the subroutine package, but since it will always be at the same place only one copy results. A typical declaration might be:

```
MATHS = $E800;
```

and it should appear before any program code, usually before the first ORIGIN statement.

The subroutine package starts with a "jump table" so that future revisions of the PL/9 math pack will not alter the entry points.

NOTE

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                                                               *
*  IF YOU USE THE 'MATHS' DIRECTIVE PL9 WILL GENERATE           *
*  'JMP' INSTRUCTIONS TO THE REAL MATHS MODULE RATHER           *
*  THAN THE 'BRA' AND 'LBRA' INSTRUCTIONS IT WILL               *
*  NORMALLY GENERATE. THIS IS TO PRESERVE THE POSITION          *
*  INDEPENDANCE OF THE PL/9 MODULES.                            *
*                                                               *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

7.01.26 RESET, NMI, FIRQ, IRQ, SWI, SWI2, SWI3

These keywords greatly simplify the integration of interrupt service routines into PL/9 programs and make building a SELF-STARTING program as simple as writing one that is called from FLEX.

The SETPL9 program, discussed in section three, will configure your copy of PL/9 for your systems RAM vectors or the MC6809 hardware vectors at $FFF2 through $FFFF. If you use SETPL9 to configure PL/9 to use the system RAM vectors you may invoke the 'R' option at compile time to tell the compiler to substitute the MC6809 ROM vectors ($FFF2 - $FFFF) for the RAM vectors and produce code for a ROM module.

Once you have configured your copy of PL/9 for the appropriate memory locations PL/9 will automatically produce the overlays required to point to the RAM (A:O ... A:M) or the ROM (A:O,R ... A:M,R) vectors to the appropriate PL/9 procedure.

If the procedure name is not used no overlay will be produced, thus allowing the 'parent' program (if any) to produce its interrupt vector overlays completely independently of the PL/9 procedure without any worry of interference.

It should be noted that the status of the GLOBAL variable pointer (the Y index register) cannot be guaranteed to be pointing at the base of the GLOBAL variables when an interrupt occurs. Therefore references to GLOBAL variables that are assigned by the GLOBAL statement must be treated in a special way if used in interrupt procedures. Other than this restriction all of the interrupt procedures can be treated much the same as any other PL/9 procedures.

It is worth noting that global variables defined by the 'AT' statement do not pose any problems whatsoever for interrupt handling procedures. If you have good control over your system memory map, (or at least know where there is some free memory available) using 'AT' variables to pass information between the interrupt procedures and the main procedures simplifies things in many situations.

Section 9.11 of the USERS GUIDE gives hints on how to gain access to the GLOBAL variables stored on the stack should it be desirable in your application.

RESET is a special case in that it must always finish with a JUMP to the first declared origin of the program where the STACK, GLOBALs, DPAGE, etc., will be initialized. REMEMBER that when RESET occurs the MC6809 is the dumbest thing on the face of the earth. Its stack pointer and registers can contain just about anything. If you are building a self-starting program the first ORIGIN must be followed by a STACK assignment or something VERY STRANGE will happen. What you do after you assign the stack is largely up to you but the stack pointer MUST be initialized before you attempt to do anything. See section 9.12.00 for further details.