WINDRUSH
Micro Systems Ltd.

# PL/9
## USERS GUIDE


WORSTEAD LABORATORIES, (Reg. Office), NORTH WALSHAM, NORFOLK NR28 9SA
TELEPHONE (0692) 404086; TELEX 975548 WMICRO G; CABLES 'WINDRUSH' NORTH WALSHAM

# PL/9

## USERS GUIDE

# (P)rogramming (L)anguage for the Motorola MC680(9)

by Graham Trott



Initial Release: 1 Janaury 1982 with version 2.XX Software.
Second Release: 1 June 1983 with version 3.XX Software.
Third Release: 1 June 1984 with version 4.XX Software.

## COPYRIGHT NOTICE



## SERIAL NUMBER NOTICE

This product has been assigned a unique serial number at the time of manufacture. The ASCII code for this serial number, which is encrypted into the body of the product, is also part of the start-up banner. This product is is therefore traceable to the original purchaser in the event of plagiarized copies being discovered.

This product is sold on the basis of being used on a SINGLE microcomputer system by a SINGLE user.

We shall consider it to be an attempt to criminally plagiarize us if duplicate copies of this manual or the accompanying disk are made available for use by other parties, or on other microcomputers. This consideration also applies to, but is not limited to, duplicate copies being produced for use within the original purchasers organisation, establishment, or home for anything other than archival purposes.

## WARNING

We at Windrush Micro Systems Limited and the author Graham Trott consider the recognition we receive as a result of the sale of our programs and manuals to be of vital importance in remaining in business.

Unless written arrangements to the contrary have been made between authorized agents of Windrush Micro Systems Limited and the purchaser of this manual and the accompanying computer program we shall consider it to be an attempt to criminally plagiarize us if our company name, the program name, or the authors name is altered, changed or removed on or from any of the materials purchased from us regardless of the means by which accomplished. This consideration shall include, but not be limited to, the re-writing of this manual, or its reproduction for distribution under another company, program or trade name, or any like modification of the accompanying computer program.

## WARRANTY NOTICE

Although every effort has been made to insure the accuracy of this material, it is sold AS IS and without warranty. No claim as to the suitability or workability of this material for any particular application or on any particular computer is made. This statement is in lieu of any other statement whether expressed or implied.

TABLE OF CONTENTS

TABLE OF CONTENTS

TABLE OF CONTENTS

# PL/9 USERS GUIDE

## 9.00.01 INTRODUCTION

This is a guide to PL/9, aimed at the complete newcomer. As such it is arranged as a tutorial not a technical reference. This section of the manual assumes that you have at least superficially read the REFERENCE MANUAL.

Most of the examples in this section are complete programs, which should each be tried if you have any doubts about the point under discussion. In order to try out the examples you will need to type them in, using either PL9's built-in editor or the editor you are most familiar with. In either case you will need to read the Editor Manual, which will tell you how to load, edit and save files and the Compiler Manual which will tell you how to compile the resulting program.

This guide does not present EVERY feature of the language; the less important, or seldom used features, are described in the appropriate sections of the Language Reference Manual. You are encouraged to read carefully each of these sections as you learn about each feature in this guide.

## 9.01.00 DATA SIZES AND TYPES IN PL/9

Before we enter the discussion on control structures we are going to briefly describe the numerical quantities that PL/9 can work with.

PL/9 has been designed to produce object code for the Motorola MC6809 8/16-bit microprocessor in real world control applications. The data types PL/9 handles have been optimized for this processor in this type of application.

There are three data sizes and five distinct data types handled by PL/9, viz:

1. 8-bit signed (twos complement) BYTE in the range of -128 to +127.

2. 8-bit unsigned (ones complement) BYTE in the range of 0 to +255.

3. 16-bit signed (twos complement) INTEGER in the range of -32768 to +32767.

4. 16-bit unsigned (ones complement) INTEGER in the range of 0 to +65,535.

5. 32-bit floating point REAL in the range of +/-1 E-38 to +/-1 E+38 with six or seven decimal digits of accuracy.

The unsigned quantities have been provided to simplify the bit evaluation and manipulation typically required in digital I/O.

The range of signed quantities available have been selected to meet the demands of the majority of analogue-digital and digital-analogue work as well as the requirements for most numerical control and scientific calculation applications.

Numerical quantities are assumed to be SIGNED DECIMAL numbers unless the programmer specifies otherwise. The dollar symbol ($) is used whenever a quantity is specified in hexadecimal form. The apostrophe (') is used whenever the quantity is specified an ASCII form. Double quotes (") are used to specify ASCII data strings.

The positive (+) and negative (-) signs retain their conventional meanings. Numbers are is implicitly positive and the compiler will reject any attempt to use the positive sign in front of a number unless it is part of an arithmetic expression and used to mean 'add'.

In addition a special symbol, the exclamation mark (!), and quantities expressed in hexadecimal form ($7E, $FF56, etc.) are used in certain circumstances to tell PL/9 that the numbers involved in an expression are to be treated as unsigned numbers. More on this later.

PL/9 was not designed for programmers who write accounting packages and need accuracies to 1 penny in $999,999,999,999,999.99 and are willing to accept the time it takes to calculate quantities to this accuracy. (even if they only wish to add $1.75 and $2.89!)

In control work speed is often essential and the range of numerical quantities more often than not limited to values which can be represented by less than six or seven digits. PL/9 has been optimized to produce fast and accurate results with numbers of this size.

More information on this topic will be presented further on in this guide. We have decided to discuss program structure in detail before we enter a lengthy discussion of PL/9 arithmetic.

## 9.02.00 PROGRAM STRUCTURE IN PL/9

PL/9 is a "Procedural" language. This means that a PL/9 program is composed of blocks of code called Procedures, or Subroutines. The entire structure of the program is organized around procedures. In BASIC you would use the GOSUB command to cause a particular part of the program to execute and another GOTO to move on from there to somewhere else, and so on. In PL/9 (and Pascal or C), on the other hand, you would put each of the functional blocks into its own procedure and link them all together with a supervisory main program that calls each one as it is needed.

A procedure in PL/9 is started with the word "PROCEDURE" and usually finishes with the word "ENDPROC". The following example introduces some of the features of the language:

```
0001 /* PL/9 INTRODUCTORY EXAMPLE */
0002
0003 INCLUDE IOSUBS;
0004
0005 PROCEDURE EXAMPLE;
0006
0007    PRINT("This is a demonstration program");
```

The example above is a complete program - not perhaps a very useful one but one that will actually run and do something. The line numbers and the single spaces that follow them are supplied by PL/9 when it lists what you typed in - all you need to enter is the code itself. Line 1 is a comment, not part of the executable program but useful for helping to improve the readability of the program. The comment is started by the /* pair and terminated by the */ pair. Note next that a blank line follows; you can use as many blank lines as you wish, to break up the text and again improve readability.

Next we come to "INCLUDE IOSUBS;". It may seem strange to programmers familiar with BASIC, but PL/9 has no I/O functions of its own. There is therefore no PRINT or INPUT command built in to the language. This might seem something of a drawback, but remember that PL/9 is not designed around any particular hardware configuration and therefore does not make any assumptions about the environment the programs it produces will be run in! You may be writing your program to run in a washing machine or a numerically-controlled lathe, in which case the I/O requirements will bear little resemblance to the sort of routines that BASIC uses. This lack of built-in I/O is therefore not necessarily a disadvantage. Fortunately, PL/9 is well equipped with facilities for adding suitable routines in a way that enables them to be used almost as if they WERE part of the language itself.

The library file IOSUBS.LIB supplied on the PL/9 disc contains a set of useful procedures for getting characters to and from a terminal. One of these procedures is PRINT, which if passed the address of a string of characters, terminated with a null, will output the characters one by one to the video display. The mechanism for passing the string address (the "parameter") to PRINT is to enclose it in brackets, as shown. PL/9 allows various constructs to be used, but the one shown is particularly suitable for character strings. The string (everything inside the double quotes) is saved as part of the program, terminated with the required null, and its address gets passed to PRINT. How it gets onto the VDU screen is up to PRINT, just as in BASIC, only in this case you can examine the source of PRINT to find out, if you wish.

9.02.00 PROGRAM STRUCTURE IN PL/9 (continued)

To use any of the routines in IOSUBS.LIB the file must be compiled with your program. The statement INCLUDE IOSUBS tells PL/9 to temporarily take input from the named file until the file is exhausted. The filename extension, .LIB in this case, is assumed to match the filename extension established by the PL/9 configuration program 'SETPL9'. INCLUDEd files may themselves have INCLUDE statements so that they will individually compile without errors, but if PL/9 is already reading an INCLUDEd file it will ignore any such nested INCLUDE statements.

The next point to note here is that all of the code-generating statements end with semicolons. The semicolon tells PL/9 that the statement has finished, which may not otherwise be obvious since PL/9 allows you to carry statements from one line to another. Comments too may carry from one line to another and can be inserted anywhere that a space would be appropriate (except inside a quoted character string).

Lastly, you should have noticed that the example program, although stated above to be complete, does not in fact end with an ENDPROC statement. The explanation for this is that the main program does not usually need an ENDPROC, although every other procedure does. How does PL/9 know that EXAMPLE is the main program? Simple, it's the last procedure in the program! Since every procedure or variable must have been declared before being referenced, there's little point in having anything follow the main program, so it must be the last. The reference manual explains on what occasions the main program does need an ENDPROC and how to pass parameters to procedures and get values back.

> If the ENDPROC is left off the last procedure in your program, PL/9 will put in a jump to the FLEX warm start address, thus avoiding any danger of the program running away and crashing the system. This is ABSOLUTELY ESSENTIAL when using the PL/9 Tracer.

The above example will actually run. First type it in (the editor is described in its own manual) then type T to invoke the tracer (which is also described in its own manual). PL/9 will compile the program and load it into memory (it decides where). You can then run the program by typing G. If any errors were reported during compilation you should return to the editor and correct them. NEVER try to run a program with reported errors or you stand a very good chance of losing your program or bombing PL/9, FLEX or even your disc!

The example program should print its message and return to the tracer command level. To run it again type G again, and to return to the editor hit the ESC key. Try experimenting with different messages and with putting \N (new line) at various places in the text string.

OK, so you've now written and tested your first PL/9 program. The PL/9 reference manual has a section that describes the various ways that parameters can be passed to and from procedures. Read the PROCEDURE, ENDPROC and RETURN sections for further details.

There is one very important point to note about INCLUDED library functions, this is that code will be generated for every function in the library ...

EVEN IF YOU ARE NOT USING THE FUNCTION

If certain library functions are not going to be used in the final application and you are running out of program memory you should make a special library module for your program that has all of the unused functions omitted. Refer to the Library Reference Manual in section eight for further details.

## 9.02.01 BRANCHING AND LOOPING

The power of a computer lies in its ability to test a condition and to take an action based upon the result of the test. Without this ability it would be no more than a calculator with some text printing facilities. Computer languages have instructions that exploit the machine's ability in a number of different ways. PL/9 has several mechanisms for control transfer, each described fully in the reference manual. This tutorial will show the beginner how to use some of these powerful facilities, by means of simple examples that should be tried out if they are not COMPLETELY understood.

## 9.02.02 IF...THEN...ELSE

The first method of controlling program flow is the IF..THEN..ELSE construct, which tells the computer that IF a condition is true THEN do something, otherwise do something ELSE. The last part may not always be needed but the first part always is. As an example, suppose that you wish to input a character from the keyboard, and print a message to say whether its ASCII code is less than decimal 65. The program to do this is as follows:

```
0001 INCLUDE IOSUBS;
0002
0003 PROCEDURE TEST_KEY;
0004
0005    IF GETCHAR < 65 THEN PRINT(" is < decimal 65");
0006                    ELSE PRINT(" is > = decimal 65");
```

This example illustrates a number of useful points. Note that the procedure is called TEST_KEY. Procedure and variable names in PL/9 can be up to 127 characters in length, may include the letters A-Z, the numbers 0-9 and the underscore character ( _ ). Names MUST also start with a letter or underscore. ALL characters are significant, unlike some languages where only the first 6 or 8 are significant.

Upper case or lower case letters may be used interchangeably, the compiler makes no distinction between the two. Thus the above procedure could be named test_key, TEST_KEY, Test_Key, TEST_key, test_KEY, etc.; to the compiler they are all the same. Keywords and hexadecimal numbers may also be written in lower case letters if desired.

The library file IOSUBS.LIB includes a routine GETCHAR whose job it is to get character from the keyboard and return it to the routine or construction which asked for it. Again it is not necessary to know how this works. The effect is that GETCHAR can be used in a program just as if it were a constant or a variable. In this case it is compared with the decimal value 65, then one of two messages is printed, depending upon the typed key.

The example above shows how spaces can be used to help program readability. PL/9 allows you to use as many spaces as you like as long as they are not in the middle of keywords or variable names. They take up virtually no room in memory, since PL/9 uses a space compression technique (as does FLEX when storing them on disk) so there is no excuse for not spacing out programs well. There's nothing like a dense mass of code, with no line indenting or blank lines between procedures, for making a program unreadable, even by the author!

## 9.02.03 BEGIN ... END

In the last example the THEN and ELSE statements were followed by a single instruction to the compiler, to print something. Supposing that you wished to do several things if the IF statement is true and/or several things if the IF statement is not true. You are NOT allowed to make statements like:

```
IF X=0
   THEN PRINT("HELLO");
        X=1;
   ELSE PRINT("GOODBYE");
        X=2;
```

Nor will PL/9 allow you to make statements like:

```
IF X=0
   THEN PRINT("HELLO") X=1;
   ELSE PRINT("GOODBYE") X=2;
```

Where more than one action is to be taken if a condition is true or false the BEGIN ... END pair must be used to tell the compiler where the statement ends. The BEGIN ... END pair tells the compiler to treat everything from BEGIN to END as a single compound statement. The correct construction of the above examples would be:

```
IF X=0
   THEN BEGIN
        PRINT("HELLO");
        X=1;
        END;
   ELSE BEGIN
        PRINT("GOODBYE");
        X=2;
        END;
```

The BEGIN statement should not have a terminating semicolon. The END statement must have a terminating semicolon.

BEGIN ... END also have another very important part to play in IF..THEN..ELSE constructions where nesting occurs, for example:

```
IF X=0
   THEN IF Y=0
           THEN IF Z=0
                   THEN A=1;
                   ELSE A=3;
```

In the above example, as the nesting suggests, the ELSE statement applies only to the last IF. This means that the only way A will be made equal to 3 would be if X=0 and Y=0 and Z is not equal to 0. Supposing you wanted to make A equal to 5 if X was not equal to zero or 4 if X was equal to 0 but Y was not equal to 0?

9.02.03 BEGIN ... END (continued)

Simple, you just add the extra ELSE statements:

```
IF X=0
   THEN IF Y=0
           THEN IF Z=0
                   THEN A=1;
                   ELSE A=3;
           ELSE A=4;
   ELSE A=5;
```

Now supposing that you only want to make A equal to 5 if X is not equal to 0, or make A equal to 1 if X, Y, and Z are equal to 0. This time the construction needs to be expressed in a more complicated fashion as we do not have a matching ELSE statement for the 'IF Y=0 THEN' statement or the 'IF Z=0 THEN' statement as the following example illustrates.

```
IF X=0
   THEN BEGIN
        IF Y=0
           THEN IF Z=0
                   THEN A=1;
        END;
   ELSE A=5;
```

Even though the above construction appears to be more complicated than the previous example it produces less code as it no longer has the two ELSE statements to contend with. The BEGIN ... END pair in this example serve to tell the compiler treat everything from BEGIN to END as a single statement. This provides a simple method of providing an ELSE statement if the first IF fails.

If you are in any doubt when nesting IF statements put in the BEGIN ... END pairs. They do not cause PL/9 to generate any extra code, they only serve to prevent the compiler from becoming 'confused' when analysing your constructions. For example each of the following examples generates exactly the same code.

```
0001 INCLUDE IOSUBS;
0002
0003 PROCEDURE IF_TEST:BYTE A,B,C,D;
0004
0005    A=0; B=1; C=2; D=3;
0006
0007    IF A=0
0008       THEN BEGIN
0009            IF B=1
0010               THEN BEGIN
0011                    IF C=2
0012                       THEN BEGIN
0013                            IF D=3
0014                               THEN BEGIN
0015                                    PRINT("HELLO");
0016                                    END;
0017                            END;
0018                    END;
0019            END;
```

9.02.03 BEGIN ... END (continued)

```
0001 INCLUDE IOSUBS;
0002
0003 PROCEDURE IF_TEST:BYTE A,B,C,D;
0004
0005     A=0; B=1; C=2; D=3;
0006
0007     IF A=0 THEN IF B=1 THEN IF C=2 THEN IF D=3 THEN PRINT("HELLO");
```

To recapitulate the BEGIN ... END pairs are used to tell the compiler that multiple statements are being made or that it should be prepared for possible further conditional branching later in the construction.

There are circumstances where the BEGIN ... END pairs are absolutely necessary. These are generally the constructions where IF...THEN...ELSE nesting occurs and the ELSE statements must be matched to the corresponding THEN statement. For example:

```
IF A=0
   THEN BEGIN
        IF B=1
           THEN BEGIN
                IF C=2
                   THEN BEGIN
                        IF D=3
                           THEN BEGIN
                                PRINT("HELLO");
                                END;
                        END;
                END;
           ELSE BEGIN
                PRINT("HI");
                END;
        END;
   ELSE BEGIN
        PRINT("HI THERE");
        END;
```

Can only be simplified to:

```
IF A=0
   THEN BEGIN
        IF B=1
           THEN BEGIN
                IF C=2 THEN IF D=3 THEN PRINT("HELLO");
                END;
           ELSE PRINT("HI");
        END;
   ELSE PRINT("HI THERE");
```

Again both of the above examples generate the same code.

Other examples of circumstances where BEGIN ... END are necessary will be described in subsequent sections.

## 9.02.04 IF...CASE...THEN...ELSE

There is a second, more complex use of the IF statement, introduced by the following example:

```
0001 INCLUDE IOSUBS;
0002
0003 PROCEDURE TEST_NUMBER;
0004
0005    PRINT("TYPE A NUMBER BETWEEN 0 AND 9\N?");
0006    IF GETCHAR
0007       CASE '0 THEN PRINT(" ZERO");
0008       CASE '1 THEN PRINT(" ONE");
0009       CASE '2 THEN PRINT(" TWO");
0010       CASE '3 THEN PRINT(" THREE");
0011       CASE '4 THEN PRINT(" FOUR");
0012       CASE '5 THEN PRINT(" FIVE");
0013       CASE '6 THEN PRINT(" SIX");
0014       CASE '7 THEN PRINT(" SEVEN");
0015       CASE '8 THEN PRINT(" EIGHT");
0016       CASE '9 THEN PRINT(" NINE");
0017
0018       ELSE PRINT(" IS NOT A NUMBER");
```

This program first prints the TYPE A NUMBER message (note the use of \N to cause a new line). The IF statement waits for you to hit a key. The returned keycode result is compared successively with the ASCII codes for the numbers zero to nine; if a match is found the word for that number is then printed. If you typed anything else the last line will tell you.

The CASE statement can be expanded with the BEGIN...END pair as required. For example supposing you wanted to perform several operations if the number entered was a one:

```
CASE '1 THEN BEGIN
             PRINT(" ONE");
             .
             .
            END;
```

If you wish to add an extra argument to the CASE statement you MUST enclose it within a BEGIN...END pair. For example you cannot say:

```
IF NUMBER
   CASE '1 .AND COUNT <10 THEN........;
```

but you can say:

```
IF NUMBER
   CASE '1 THEN BEGIN
                IF COUNT <10 THEN........;
                END;
```

## 9.02.05 LOGICAL .AND .OR .EOR

These LOGICAL operators considerably improve the power of the IF...THEN...ELSE control statements, and greatly simplify many control arguments.

PL/9 recognizes two forms of AND, OR and EOR. The first is the LOGICAL form in which each keyword is preceeded by a period (.). The second is the ones complement BIT operators which lack the period. Understanding the differences between these two forms is very important if you wish to use them successfully.

The LOGICAL forms (.AND .OR .EOR) are used to form constructions such as IF X is true and Y is true THEN do something. The BIT operators (AND OR EOR) are used to set and clear specific bits in a binary pattern of ones and zeros.

If you are a BASIC programmer you will remember that BASIC uses the same form for both logical and bitwise operations. Context and the use of brackets tells the BASIC compiler/interpreter which is which. If you are an assembly language programmer the the ones complement bit-wise AND, OR, and EOR (which will be explained later) are identical to the AND, OR and EOR you are used to.

Perhaps an example is in order:

```
0001 INCLUDE IOSUBS;
0002
0003 PROCEDURE LOGICAL_AND_OR_DEMO:
0004    BYTE CHAR;
0005
0006    CHAR=GETCHAR;
0007    IF CHAR >= 'A .AND CHAR <= 'Z .OR
0008       CHAR >= 'a .AND CHAR <= 'z
0009       THEN PRINT(" is accepted\N");
0010       ELSE PRINT(" is rejected\N");
```

The purpose of this routine is to accept only upper and lower case letters from A-Z and a-z from the keyboard and reject all others. If you are familiar with the ASCII code tables you will know that there are several codes between the upper case 'Z' and the lower case 'a'.

Once again we include IOSUBS.LIB for the useful routines it contains. The PROCEDURE declaration this time terminates not in a semicolon but in a colon. This tells PL/9 that the declaration is not yet complete, that there is more to come. On the next line is BYTE CHAR, which says "reserve a single byte of storage, on the stack, for the exclusive use of this procedure, and call it by the name CHAR". The semicolon after that completes the procedure declaration.

Line 6 uses GETCHAR as though it were a variable, that is it would appear that we are simply saying the variable CHAR is to be made equal to the variable GETCHAR. In fact GETCHAR is a procedure in IOSUBS that gets a keycode from the keyboard and returns with the code which then may be used as though it were a variable and form part of an evaluation statement (as in the discussion of IF...THEN...ELSE). Alternatively the code GETCHAR returns with may be assigned to another variable as we are doing in this example.

Why should we use GETCHAR any differently here than we did previously I hear you ask? The answer is simple; we wish to evaluate the code that GETCHAR returns with several times in order to test it against several conditions. If we used GETCHAR in place of 'CHAR' in the 'IF...THEN...ELSE' construction in this example we would have to provide four different key inputs, clearly not what we want.

9.02.05 LOGICAL .AND .OR .EOR (continued)

Line 7 states "If the character stored in CHAR is greater than or equal to the ASCII code for the letter 'A' AND the character stored in CHAR is less than or equal to the ASCII code for the letter 'Z' then PRINT ' is accepted'". If the character fails this test it then proceeds to the statement following the .OR which means "if the first condition is not met then try the next one...". Line 8 states "If the character stored in CHAR is greater than or equal to the ASCII code for the letter 'a' AND the character stored in CHAR is less than or equal to the ASCII code for the letter 'z' then PRINT ' is accepted'".

Line 10 is what course of action is to be taken if the statement on line 7 AND the statement on line 8 fail verify that CHAR is within the specified range, i.e to print ' is rejected'.

Logical EXCLUSIVE OR (.EOR or .XOR) only has a few practical applications and was included in the language mainly for completeness. Exclusive OR is a special form of OR that means If condition 1 is true and condition 2 is false OR condition 1 is false and condition 2 is true then do something. If both conditions are true or both conditions are false the 'ELSE' action, if specified, is taken. For example:

```
0001 INCLUDE IOSUBS;
0002
0003 PROCEDURE AND_OR_XOR_DEMO:
0004    BYTE CHAR1,CHAR2;
0005
0006    CHAR1=GETCHAR;
0007    CHAR2=GETCHAR;
0008
0009    IF CHAR1 = '1 .EOR CHAR2 = '1
0010       THEN PRINT(" is accepted");
0011       ELSE PRINT(" is rejected");
```

Again we have the now familiar INCLUDE statement. Line 4 is simply an expansion of what we did in previous example. In this example we are saying reserve two bytes of storage, on the stack, for the exclusive use of this procedure, and call one by the name CHAR1 and the other by the name CHAR2". The semicolon after that completes the procedure declaration.

Line 6 gets a character from the keyboard and assigns it to CHAR1. Line 7 does exactly the same thing except assigns the keyboard character to CHAR2. As we have not assigned any limits to what we are willing to accept as values for CHAR1 and CHAR2 any key on the keyboard may be hit and will be accepted by the program. We'll be showing you how to do limit checks on incoming characters later on in this guide.

Line 9 is the exclusive OR evaluation of CHAR1 and CHAR2 and can be read as: "If CHAR1 is equal to ASCII 1 AND CHAR2 is not equal to ASCII 1 OR if CHAR1 is not equal to ASCII 1 AND CHAR2 is equal to ASCII 1 THEN print ' is accepted' otherwise print ' is rejected'".

What happens in practice is that entry of '01' or '10' will display the message 'is accepted', any other input will display the message ' is rejected'.

9.02.05 LOGICAL .AND .OR .EOR (continued)

Logical .AND, logical .OR and logical .EOR may be nested to any required depth. Bracketing of logical constructions is not permitted however. For example PL/9 will NOT allow you to generate statements like:

```
IF (A=0 .AND B=1 .AND C=3) .OR ((B=1 .AND C=2) .AND (D=4 .OR E=4)) .OR F=5
   THEN PRINT("O.K.");
   ELSE PRINT("NOT O.K.");
```

BUT the construction can be emulated by using a flag and IF...THEN...ELSE statements controlled with BEGIN...END pairs, as follows:

```
FLAG=0;
IF A=0
   .AND B=1
   .AND C=3
        THEN FLAG=1;
        ELSE BEGIN
             IF B=1
                 .AND C=2
                    THEN BEGIN
                         IF D=4
                            .OR E=4
                                THEN FLAG=1;
                                ELSE IF F=5
                                        THEN FLAG=1;
                         END;
                    ELSE BEGIN
                         IF F=5 THEN FLAG=1;
                         END;
             END;

 IF FLAG=1 THEN PRINT("O.K.");
           ELSE PRINT("NOT O.K.");
```

OR

```
FLAG=0;
IF A=0 .AND B=1 .AND C=3
                    THEN FLAG=1;
IF B=1 .AND C=2
            THEN IF D=4 .OR E=4
                        THEN FLAG=1;
IF F=5 THEN FLAG=1;

IF FLAG=1 THEN PRINT("O.K.");
          ELSE PRINT("NOT O.K.");
```

Even though the first construction generates more code than the second it will execute faster than the second if any of the early conditions are true. This is because the structure of the second results in the execution of each IF loop even if the FLAG was set to 1 by the previous IF..THEN statement. The first construction finds the first available point to set the FLAG and then branches over all the remaining statements to the THEN PRINT statement.

## 9.02.06 WHILE...

The second method of controlling program flow has to do with loops, where a sequence of actions must be repeated a given number of times. In English, this is like saying "WHILE this condition is satisfied, do that". The equivalent PL/9 construct looks very similar, as in the following example:

```
0001 INCLUDE IOSUBS;
0002
0003 PROCEDURE WHILE_DEMO:
0004    BYTE COUNT;
0005
0006    COUNT=0;
0007    WHILE COUNT<10
0008    BEGIN
0009       PUTCHAR(COUNT + 'A);
0010       SPACE(1);
0011       COUNT=COUNT+1;
0012    END;
```

The purpose of this example is to print out the first 10 letters of the alphabet. Once again we include IOSUBS.LIB for the useful routines it contains, and reserve a single byte of storage on the stack called COUNT.

Now the main program gets under way. First the variable COUNT is set to zero. The next line tests it to see if has a value less than 10. If so, the next statement is executed. The "next statement" in this case starts with a BEGIN keyword, which has the property of turning everything from that point on, up to a matching END, into a single compound statement, behaving in every way like a simple statement such as COUNT=5.

Initially, the value of COUNT is zero, so the test passes (COUNT is less than 10). The BEGIN..END compound statement first outputs a character to the VDU. The character is 'constructed' from the addition of the current value of COUNT and the ASCII value of the letter 'A' ($41 or decimal 65). Thus a count of 0 will output an 'A', a count of 1 will output a 'B', etc. Line 10 outputs a space, line 11 then increments the value of COUNT. The routines used are PUTCHAR and SPACE, both part of IOSUBS.LIB. After the BEGIN..END block has been executed the computer goes back to the WHILE statement and again tests whether COUNT is less than 10. This process continues until COUNT reaches 10, whereupon the entire BEGIN...END block is skipped and the program ends.

PUTCHAR is a routine in the IOSUBS library module whose job it is to output one ASCII character at a time to the video display.

SPACE is another useful routine in the IOSUBS library module. This routines job is to output the number of spaces specified by the number in the brackets to the video display. Its function is very similar to 'SPC(n)' in BASIC; it moves the cursor along the video display a specified number of spaces so that the next message will start at a particular position relative to the current position.

Caution must be observed when using SPACE near the right hand column limit of video displays. Some displays generate an automatic carriage return line feed the moment something is entered in the last column. Others simply overtype the last character and leave the cursor in the last column. The general rule to follow is to treat a display as having one column less than stated in the manufacturers documentation, i.e. 80 columns becomes 79 columns, etc.

## 9.02.07 REPEAT...UNTIL

The WHILE loop described above tests for the required condition at the head of the loop, that is to say before executing the body of the loop. In cases where the test fails the first time, the body of the loop will never be executed. Although any kind of loop can be constructed by this means, it is convenient to have an alternative loop control mechanism equivalent to "REPEAT this action UNTIL that condition is satisfied". In this case the body of the loop is executed before testing for the condition; if the test then fails the loop is repeated until the test passes. The PL/9 construct that performs this function is demonstrated in the following example;

```
0001 INCLUDE IOSUBS;
0002
0003 PROCEDURE REPEAT_DEMO: BYTE COUNT1, COUNT2;
0004
0005    COUNT1=1;
0006    REPEAT
0007
0008       COUNT2=0;
0009       REPEAT
0010          PUTCHAR(COUNT1 + '0);
0011          COUNT2=COUNT2 + 1;
0012       UNTIL  COUNT2=COUNT1;
0013
0014       CRLF;
0015       COUNT1=COUNT1 + 1;
0016    UNTIL COUNT1=10;
```

This example in fact contains two REPEAT...UNTIL loops, one inside the other. Note the use of indenting and blank lines to present this "nested" structure clearly and the extra spaces in the counter incrementing to improve readability. Line 3 declares two BYTE variables, COUNT1 and COUNT2; once again these are totally private to the procedure they are declared in.

The first REPEAT uses variable COUNT1, which is initialized to one in line 5. The body of the loop (lines 8 through 15) is executed, then COUNT1 is then compared with 10. If it has that value the program is terminated. The body of the loop is therefore executed nine times (1 through 9).

The second REPEAT uses variable COUNT2, initialized to zero in line 8. The body of the loop (lines 10 and 11) is executed, then COUNT2 is compared with COUNT1. The inner loop therefore executes a number of times dependant upon the value of COUNT. Since this increases every time that the outer loop is completed, the number of characters printed on each line increases from one to nine. The actual character printed is that obtained by adding COUNT1 to the ASCII value of the character zero. After each line of characters has been printed, a new line is started by CRLF.

CRLF is another useful routine in the IOSUBS library. Its job is to send a ASCII carriage return (CR) followed by a line feed (LF) to the video display. This has the effect of putting the cursor at the beginning of the next line. PRINT("\N") does exactly the same thing but requires a considerable amount of extra code.

## 9.02.08 FOR...NEXT

One of the constructs most frequently used in BASIC is the FOR-NEXT loop. PL/9 does not allow this construct, but the REPEAT command can be used to produce the same effect. Consider the following:

BASIC

```
FOR I=1 TO 100 STEP 2
   .
   .
   .
NEXT I
```

PL/9

```
LOOPVAR=1;
REPEAT
   .
   .
   LOOPVAR=LOOPVAR+2;
UNTIL LOOPVAR>100;
```

The two examples have the same effect, viz a loop variable is set to one and a step count to two. The body of the loop (represented by the dots) is then executed. After that the loop variable is incremented by the step value and is compared with the terminating value. To ensure that the loop is executed when the loop variable equals the terminating value it is necessary to use > for the test, not =. You could alternatively start the loop variable at zero and test for equal to the terminating value.

## 9.02.09 WHY DIDN'T YOU...

The reason that FOR ... NEXT, counted loops, bracketed logical operators and many other constructions found in other languages are not included in PL/9 is very fundamental.

In the introduction we mentioned that the programmers source file, the co-resident editor-compiler-trace facilities in PL/9 and the program object code must all reside in memory together. PL/9 had to be designed to take up as little space as possible. If we started including every possible construction in every language we would soon have a compiler that would not allow you to have much more than a 30 or 40 line program resident with it. This was clearly not our design objective!

Our design philosophy was that if it were possible to emulate constructions found in other languages with existing PL/9 facilities we would not include them in the product. Adhering to this philosophy not only allows you to have a very large program resident in memory with PL/9, but also enables the compiler to run faster than you might think possible for a product that produces such efficient code.

SO before you jump to the conclusion that PL/9 does not support a particular construction which you would like to use remember that you can probably emulate the function using the basic facilities of PL/9.

Even though some emulations may appear to be clumsy they will, in all probability, require less memory and will execute faster than the construction permitted in the original language!

## 9.03.00 ADVANCED CONTROL TECHNIQUES FOR REAL TIME PROGRAMMING

There are a number of points concerning loops that are worth noting.

As the example in section 9.02.07 shows, loops can be nested. In fact there is no practical limit to how deeply you may nest loops and no restriction upon the nature of that nesting. A WHILE loop can contain a REPEAT, which may in turn contain WHILEs or REPEATs. The only limit, in fact, to the depth of nesting is the stack room available during compilation. As long as you use sensible indenting, as in the example, to indicate the range of each loop, you will run out of space on the line long before PL/9 runs out of stack during compilation.

You may have noticed that REPEAT...UNTIL performs the same kind of statement bracketing that BEGIN...END did for the WHILE loop earlier. In general, BEGIN...END are used after IF...THEN or WHILE statements to make a group of statements appear as one. REPEAT works in a slightly different way (it's the only one that performs the test at the bottom of a loop) so it doesn't need the BEGIN...END.

There are also circumstances where you may wish to escape from a loop at a particular point or return to a previous point when a specified condition is met. BREAK and GOTO, which will be explained in a moment, have been provided to facilitate this type of control.

There are also circumstances where you may wish to terminate a procedure used as a subroutine early, for example when a high priority signal is detected and you are in the middle of a three hour shutdown sequence. One of the forms of RETURN, which will also be explained in a moment, has been provided to facilitate this type of control.

The following sections introduce BREAK, GOTO and RETURN. These three keywords give PL/9 the capability of handling all but the most critical real-time applications without recourse to hardware interrupts.

Although hardware interrupts provide the programmer with a powerful tool for handling real-time events their use can be difficult, particularly when things go wrong due to programming errors. The very nature of hardware interrupt handling makes tracking down problems a very daunting task to the uninitiated.

Many programmers prefer to avoid using interrupts in order to 'keep things simple' or reserve their use for specific tasks such as system timekeeping, task selection or synchronization to external events.

PL/9 provides the programmer with all of the essential tools to construct 'polling' oriented software with an inherent ability to rapidly respond to real time events that rivals that of interrupt driven software in practical situations. Generally speaking building up a control package based on software polling will be far easier to troubleshoot when things don't go according to plan.

The main trick to building up a software package that uses polling and is expected to respond rapidly to external events is never write a routine that 'grabs' the processor for any length of time, i.e. long delay loops unless a poll of higher priority events is included in the routine.

A complete example of software polling is included later in this guide.

## 9.03.01 BREAK

There are times when it is necessary to be able to break out of a loop prematurely, before the test condition is satisfied. There are a number of ways of doing this, for example by doing more than one test using the logical operators .AND and .OR. Another way is to use the BREAK command, as in this example;

```
0001 INCLUDE IOSUBS;
0002
0003 PROCEDURE BREAK_DEMO: BYTE CHAR;
0004
0005    REPEAT
0006       CHAR=GETCHAR;
0007       IF CHAR=$1B THEN BREAK;
0008    FOREVER;
```

Using BREAK to escape from REPEAT...UNTIL or WHILE.. construction nested within one another requires a bit of thought and planning on the part of the programmer. This is due to the fact that the compiler is not clairvoyant! For example:

```
COUNT4=0;
REPEAT

   COUNT3=0;
   REPEAT

      COUNT2=0;
      REPEAT

         COUNT1=0;
         REPEAT
            IF PORT=1 THEN BREAK;
            COUNT1=COUNT1+1;
         UNTIL COUNT1=999;

      COUNT2=COUNT2+1;
      UNTIL COUNT2=999;

    COUNT3=COUNT3+1;
    UNTIL COUNT3=999;

COUNT4=COUNT4+1;
UNTIL COUNT4=999;
```

Is the BREAK statement in the innermost loop supposed to pass control to the line just past 'UNTIL COUNT1=999' or the line just past 'UNTIL COUNT4=999'?

As far as the compiler is concerned it will ONLY terminate the current loop, i.e. it will pass control to the line just past 'UNTIL COUNT1=999'.

If you wished to escape from the entire nested loop construction when a given event occurs you have two options. The first is to put the construction in a separate procedure and use 'RETURN' to terminate it. The second is to use a flag as a control mechanism to terminate the subsequent loops when a break condition is found in a construction within it.

9.03.01 BREAK (continued)

Where NESTING of loops occurs BREAK in conjunction with a byte variable used as a flag can be used to rapidly terminate a program.

Some of the constructions required for this illustration have not yet been covered in this guide, but will be shortly. If you are not sure what the various constructions are doing read the rest of this users guide before attempting to try this program.

Since this program polls the keyboard for input and the TRACER is polling for keyboard input at the same time this program will not run properly under the PL/9 tracer. For this reason we have declared an ORIGIN of $B000 for the program to be compiled at. If you want to try this example type in the program exactly as it is presented. Then tell the compiler to compile the program to memory thus 'A:M<CR>' . If no errors occur type 'M<CR>' to enter your system monitor. A warning message will be posted to warn you that you have just entered your monitor and how to get back into PL/9. Use your system monitor JUMP command, 'J' in most monitors, to commence execution of code at $B000. This would be typed in as 'J B000'. Some monitors, notably GMXBUG, require that you type RETURN after entering B000.

Once entered this program will display the decimal values of DELAY1, DELAY2 and DELAY3 on the left side of the screen. Since DELAY3 is the deepest in the nest it will decrement most rapidly. When DELAY1, DELAY2 and DELAY3 all reach 0 the program will terminate and return to PL/9. If you hit the ESCAPE key at any time the program will terminate immediately.

```
0001 ORIGIN=$B000;
0002
0003 GLOBAL BYTE DELAY1,DELAY2,DELAY3;
0004
0005 INCLUDE IOSUBS;
0006 INCLUDE REALCON;
0007
0008 PROCEDURE PRNUM (REAL NUMBER): BYTE BUFFER(20);
0009    PRINT(ASCII(NUMBER,.BUFFER));
0010 ENDPROC;
0011
0012 PROCEDURE PRINT_DELAY_COUNTS:INTEGER DELAY;
0013    PUTCHAR($0D);
0014    SPACE(20);
0015    PUTCHAR($0D);
0016    PRNUM(DELAY1);
0017    SPACE(2)
0018    PRNUM(DELAY2);
0019    SPACE(2);
0020    PRNUM(DELAY3);
0021    SPACE(5);
0022    DELAY=4000;
0023    REPEAT
0024       DELAY=DELAY-1;
0025    UNTIL DELAY=0;
0026 ENDPROC;
0027
```

9.03.01 BREAK (continued)

```
0028 PROCEDURE ESCAPE_TEST:BYTE FLAG;
0029    IF GETKEY=$1B /* ESCAPE */
0030       THEN FLAG=1;
0031       ELSE FLAG=0;
0032 ENDPROC FLAG;
0033
0034 PROCEDURE NESTED_BREAK_DEMO:BYTE ESCFLAG;
0035
0036    ESCFLAG=0;
0037
0038    DELAY1=100;
0039    REPEAT
0040
0041       DELAY2=100;
0042       REPEAT
0043
0044          DELAY3=100;
0045          REPEAT
0046             IF ESCAPE_TEST <> 0
0047                THEN BEGIN
0048                     ESCFLAG=1;
0049                     BREAK;
0050                     END;
0051             DELAY3=DELAY3-1;
0052             PRINT_DELAY_COUNTS;
0053          UNTIL DELAY3=0;
0054
0055          IF ESCFLAG <> 0
0056             THEN BREAK;
0057             ELSE BEGIN
0058                  IF ESCAPE_TEST <> 0
0059                     THEN BEGIN
0060                          ESCFLAG=1;
0061                          BREAK;
0062                          END;
0063                  END;
0064          DELAY2=DELAY2-1;
0065       UNTIL DELAY2=0;
0066
0067       IF ESCFLAG <> 0
0068          THEN BREAK;
0069          ELSE BEGIN
0070               IF ESCAPE_TEST <> 0
0071                  THEN BEGIN
0072                       ESCFLAG=1;
0073                       BREAK;
0074                       END;
0075               END;
0076       DELAY1=DELAY1-1;
0077    UNTIL DELAY1=0;
0078
0079    JUMP $0003; /* RETURN TO PL/9 */
```

9.03.01 BREAK (continued)

The previous example introduces GETKEY another useful routine in IOSUBS. GETKEY is similar to GETCHAR in that it gets a key from the keyboard, but the way it works is significantly different. Whenever GETCHAR is called the system will 'hang up' waiting for a key on the keyboard to be pressed, when one is finally received the character input will be automatically echoed back to the video display.

When GETKEY is called it will return to the calling program immediately whether a key has been pressed or not. If a key has been pressed then GETKEY will return with the appropriate code, if no key has been pressed GETKEY will return with a NULL ($00). For this reason GETKEY cannot be used in conjunction with the CONTROL @ key as this key also returns a NULL. GETKEY does not echo the keyboard character back to the video display.

The next example shows you how NOT to construct a priority termination of a SEQUENTIAL group of delay routines. The fact that the testing we are doing in this example is substantially simpler than the previous example will give you the clue as to what we are doing wrong.

When we are in the middle of the first delay loop and it is terminated by the escape key the construction we have used will cause the program to execute the entire body of the subsequent loops before the program terminates.

In the example given executing the code in the main body of the two subsequent delay loops causes a noticable delay between hitting the escape key and termination of the program. Imagine what would happen if these two delay loops had constructions within them that ran to two pages of source code.

See if you can re-write this program so that if the escape key is hit during the first delay the second two delays are terminated before the delay count is decremented.

HINT: The constructions will be very similar to the one used in DELAY3 of the previous example.

```
0001 ORIGIN=$B000;
0002
0003 GLOBAL BYTE DELAY1,DELAY2,DELAY3,ESCFLAG;
0004
0005 INCLUDE IOSUBS;
0006 INCLUDE REALCON;
0007
0008 PROCEDURE PRNUM (REAL NUMBER): BYTE BUFFER(20);
0009    PRINT(ASCII(NUMBER,.BUFFER));
0010 ENDPROC;
0011
```

9.03.01 BREAK (continued)

```
0012 PROCEDURE PRINT_DELAY_COUNTS:INTEGER DELAY;
0013    PUTCHAR($0D);
0014    SPACE(20);
0015    PUTCHAR($0D);
0016    PRNUM(DELAY1);
0017    SPACE(2)
0018    PRNUM(DELAY2);
0019    SPACE(2);
0020    PRNUM(DELAY3);
0021    SPACE(5);
0022    DELAY=4000;
0023    REPEAT
0024       DELAY=DELAY-1;
0025    UNTIL DELAY=0;
0026 ENDPROC;
0027
0028 PROCEDURE ESCAPE_TEST:BYTE FLAG;
0029    IF GETKEY=$1B /* ESCAPE */
0030       THEN BEGIN
0031            FLAG=1;
0032            ESCFLAG=1;
0033            END;
0034       ELSE FLAG=0;
0035 ENDPROC FLAG;
0036
0037 PROCEDURE SEQUENTIAL_BREAK_DEMO;
0038
0039    ESCFLAG=0;
0040
0041    DELAY1=100;
0042    DELAY2=100;
0043    DELAY3=100;
0044
0045    REPEAT
0046       DELAY1=DELAY1-1;
0047       PRINT_DELAY_COUNTS;
0048    UNTIL DELAY1=0 .OR ESCAPE_TEST <> 0;
0049
0050    REPEAT
0051       DELAY2=DELAY2-1;
0052       PRINT_DELAY_COUNTS;
0053    UNTIL DELAY2=0 .OR ESCFLAG <> 0 .OR ESCAPE_TEST <> 0
0054
0055    REPEAT
0056       DELAY3=DELAY3-1;
0057       PRINT_DELAY_COUNTS;
0058    UNTIL DELAY3=0 .OR ESCFLAG <> 0 .OR ESCAPE_TEST <> 0
0059
0060    JUMP $0003; /* RETURN TO PL/9 */
```

9.03.01 BREAK (continued)

The key to rapid termination of a NESTED or SEQUENTIAL group of control arguments or delay routines is to test for the escape condition or conditions frequently. Other points to note are:

1. Test for the escape condition(s) at any place where the program is looping, e.g. within delay loops.

2. Test for the escape conditions at any place in control arguments that are likely to call external user procedures.

3. Use the BREAK command to terminate the current WHILE or REPEAT loop.

4. If more than one one loop or control argument exists within the procedure which is to be terminated by the escape condition(s) use a GLOBAL flag to inform the remainder of the program that the escape condition(s) have been detected by an earlier loop or control argument.

5. Use a FLAG to to prevent the execution of the main body of code within subsequent loops or control arguments.

6. When you need to escape from nested loops you may also consign the nested loop to a separate procedure and use RETURN in lieu of BREAK.

   See 'RETURN' for information on terminating external procedures when escape conditions are recognized.

N O T E

As mentioned in the Language Reference Manual only ten breaks may be active within any given procedure at any one time. This does not mean that you can only have a total of ten BREAKs in a procedure it means that only ten can be ACTIVE while the procedure is being compiled.

This is due to the fact that when loops are being nested PL/9 will keep pushing things onto the stack during compilation until it finds the deepest portion of the nest. If BREAKS are being used even more information is pushed onto the stack. This process continues until the compiler knows where control is to be passed to when the break occurs. As the extra information required for BREAK is fairly substantial we had to set a limit otherwise you might easily exceed the limits of the available stack space.

This limitation has never posed any problems for the author (or anyone else to our knowledge). It is unlikely that you will ever see the error message 'TOO MANY BREAKS' if you structure your programs well.

## 9.03.02 GOTO

Just as BREAK can be used to terminate a loop early, in effect to branch forward in a program GOTO can be used to branch to a previous point in a particular procedure in order to repeat a particular step or sequence of steps. For example:

```
0001 INCLUDE IOSUBS;
0002
0003 PROCEDURE GOTO_DEMO:BYTE CHAR;
0004
0005    PUTCHAR($20);  /* space */
0006 CANCEL:
0007    PUTCHAR($08);  /* back-space */
0008    PUTCHAR($20);
0009    PUTCHAR($08);
0010
0011    CHAR=GETCHAR;
0012    IF CHAR <> '1 .AND CHAR <> '0
0013       THEN GOTO CANCEL;
0014       ELSE PRINT(" is O.K.");
```

In this example the name 'CANCEL' is being used as a label for the subsequent GOTO. Note that 'CANCEL' is terminated with a colon NOT a semicolon. The purpose of the label CANCEL is tell the compiler 'take note of this address; I might want you to come back here later'. Just because you insert a label does not mean you have to use it, although it would be somewhat pointless if you did so.

What in the world is this program doing I hear you ask? The IOSUBS library supplied uses a standard FLEX I/O interface called 'GETCHR' at $CD15. This routine echos any displayable ASCII character back to the video display before it returns to the calling program. This means that if you type the letter 'A' it will be put up on the screen even if you don't want it there.

The sequence of events before and after the label CANCEL serve the following purpose: First we send a dummy space to the screen, then we send a back-space character to the screen which moves the cursor back to the position where we just put the space. We then move forward one space and back-space once more. This bit of gyration has the effect of removing any character put up on the screen (assuming that your system display supports back-spacing as most do). This technique should be restricted to the left hand side of the screen as some display devices act strangely if you are in the last available column of the display.

Having just made a 'dummy' pass through CANCEL we now enter the main part of the program. First we get a character into the variable CHAR via GETCHAR. We then examine CHAR and see if it is an ASCII 1 or an ASCII 0. If it is either of these values the previously echoed character will remain on the screen and a message ' is accepted' will be printed after it.

If CHAR is neither of these values the 'THEN GOTO CANCEL' forces the program to start executing at the point right after the label CANCEL. In this case we are going to remove the character from the screen as described previously and re-enter the main program again. The process of cancelling the character entered and getting another one for comparision will continue until the character entered passes the test, i.e. it is a zero or a one.

GOTO cannot be used to go to a label declared AFTER the GOTO instruction nor can it be used to branch to a location outside of the current procedure.

### 9.03.03 RETURN

The use of RETURN in procedures used as functions has already been covered in detail in section 7.01.11 and will be covered again in section 9.06.01.

RETURN can also be used as part of a statement that means 'if this condition is met terminate this entire procedure right here and now'.

When RETURN is used to prematurely terminate a procedure, regardless of the level of nesting or number of local variables, the stack will be completely tidied up just though the procedure was terminated in the normal manner by the ENDPROC.

RETURN is commonly used in control applications where a lengthy process has just been entered and there is the possibility of some higher priority event taking place elsewhere in the system. This is an ideal application for interrupt driven software. In spite of its advantages in situations like this many programmers prefer to avoid using interrupts unless they are absolutely necessary or wish to reserve interrupts for other tasks in their software. RETURN has been provided to facilitate development of software that can respond to real time events without recourse to interrupts.

In the following example (it works) the procedure named 'THREE_HOUR_TEST' should be imagined to be a complex series of valve openings and closings with varying degrees of delay between them. In this example we are going to simply enter a delay routine that starts counting at 999999 and continues to count down to -999999 at approximately one count per second. If you hit the RETURN key on the keyboard the procedure will terminate immediately and print the message 'EMERGENCY!' and return to PL/9.

Some of the constructions required for this illustration have not yet been covered in this guide, but will be shortly. If you are not sure what the various constructions are doing read the rest of this users guide before attempting to try this program.

Since this program uses the keyboard for input and the TRACER is looking for keyboard input at the same time this program will not run properly under the PL/9 tracer. For this reason we have declared an ORIGIN of $B000 for the program to be compiled at. If you want to try this example type in the program exactly as it is presented then tell the compiler to compile the program to memory thus 'A:M<CR>' . If no errors occur type 'M<CR>' to enter your system monitor. A warning message will be posted to warn you that you have just entered your monitor and how to get back into PL/9. Use your system monitor JUMP command, 'J' in most monitors, to commence execution of code at $B000. This would be typed in as 'J B000'. Some monitors, notably GMXBUG, require that you type RETURN after entering B000.

The program will begin execution in approximately 2 seconds and will display the number 999999 on the left side of the screen. The number will begin decrementing at approximately one count per second. Hitting the RETURN key on your keyboard will terminate the program IMMEDIATELY, and we mean immediately. There will be absolutely no noticable delay between hitting RETURN and seeing the 'EMERGENCY!' message....try it.

```
0001 ORIGIN=$B000;
0002 GLOBAL BYTE EFLAG;
0003
0004 INCLUDE IOSUBS;
0005 INCLUDE REALCON;
0006
```

9.03.03 RETURN (continued)

```
0007 CONSTANT EMERGENCY=0,SAFE=1,ESCAPE=$1B,BACK=$08,_SPACE=$20,
0008          MASK=$04,_RETURN=$0D;
0009
0010 PROCEDURE PRNUM (REAL NUMBER): BYTE BUFFER(20);  /* see section 9.05.08 */
0011    PRINT(ASCII(NUMBER,.BUFFER));
0012 ENDPROC;
0013
0014 PROCEDURE EMERGENCY_CHECK;
0015    CALL $CD4E; /* FLEX 'STAT' */
0016    IF CCR AND MASK = 0
0017       THEN IF GETCHAR=_RETURN
0018               THEN EFLAG=EMERGENCY;
0019               ELSE BEGIN
0020                    PUTCHAR(BACK);
0021                    PUTCHAR(_SPACE);
0022                    PUTCHAR(BACK);
0023                    END;
0024 ENDPROC;
0025
0026 PROCEDURE THREE_HOUR_TEST:REAL DCOUNT:BYTE DELAY1,DELAY2;
0027    EFLAG=SAFE;
0028    DCOUNT=999999;
0029    CRLF;
0030    CRLF;
0031    REPEAT
0032       EMERGENCY_CHECK;
0033       IF EFLAG=EMERGENCY THEN RETURN;
0034       DELAY1=100;
0035       DELAY2=100;
0036       REPEAT
0037
0038          REPEAT
0039             EMERGENCY_CHECK;
0040             IF EFLAG=EMERGENCY THEN RETURN;
0041             DELAY2=DELAY2-1;
0042          UNTIL DELAY2=0;
0043
0044          EMERGENCY_CHECK;
0045          IF EFLAG=EMERGENCY THEN RETURN;
0046          DELAY1=DELAY1-1;
0047       UNTIL DELAY1=0;
0048       PUTCHAR(_RETURN);
0049       SPACE(6);
0050       PUTCHAR(_RETURN);
0051       PRNUM(DCOUNT);
0052       DCOUNT=DCOUNT-1;
0053    UNTIL DCOUNT=-999999;
0054 ENDPROC;
0055
0056 PROCEDURE MAIN;
0057    REPEAT
0058       THREE_HOUR_TEST;
0059       IF EFLAG=EMERGENCY THEN BREAK;
0060    FOREVER;
0061    PRINT(" \N\NEMERGENCY!\N\N");
0062    JUMP $0003; /* BACK INTO PL/9 */
```

## 9.04.00 GETTING AT THE OUTSIDE WORLD

Most industrial (as opposed to business) uses of microprocessors have a fair amount of hardware in the system used to communicate with the 'real world'. The hardware can be as simple as an MC6821 PIA reading the status of a set of switches on the 'A' port and sending signals to relays on the 'B' port, or a complicated mass of PIA's, ACIA's, A-D converters, D-A converters, etc. Writing programs that can not only initialize the hardware devices but can also communicate with them simply is one of the major failings of programming languages like BASIC and Pascal.

It is not that it is impossible to write such programs in these languages but that the gyrations that one must go through to get around the restrictions of the language not only produces a source file that is often impossible to decipher at a later date but also produces a large amount of code for what should be simple operations. Quite often the only realistic solution to getting around the restrictions of the language is to write all the I/O handling software in assembly language. This is fine if you are a good assembly language programmer, but what if you are not?

PL/9 for all its 'failings' in supporting exotic data types and control structures makes I/O handling and working with external programs written in other languages simplicity itself. PL/9 was DESIGNED with just this type of application in mind!

## 9.04.01 AT

This keyword is used to tell PL/9 where some particular element of hardware is located or where RAM variables are stored in terms of 'hard' (absolute) memory locations. For example supposing you wished to tell PL/9 about a PIA that uses the 'A' side for data input and the 'B' side for data output. A declaration such as the following is all that is required:

```
AT $E040: BYTE ADATA, ACTRL, BDATA, BCTRL;
```

This declaration tells PL/9 that 'ADATA' is at $E040, 'ACTRL' is at $E041, 'BDATA' is at $E042 and 'BCTRL' is at $E043. Once PL/9 knows what to call the PIA and where its elements are the names given can be used just as any other local or global variable. For example supposing we wanted to initialize the PIA for all inputs on the 'A' side and all outputs on the 'B' side. The PL/9 program to do this would look something like:

```
PROCEDURE PIA_INIT;
   ACTRL=0;    /* SELECT DATA DIRECTION REGISTER */
   ADATA=0;    /* ALL INPUTS */
   ACTRL=4;    /* SELECT OUTPUT REGISTER */

   BCTRL=0;    /* SELECT DATA DIRECTION REGISTER */
   BDATA=$FF;  /* ALL OUTPUTS */
   BCTRL=4;    /* SELECT OUTPUT REGISTER */
   BDATA=0;    /* SET ALL OUTPUTS TO ZERO */
ENDPROC;
```

Once initialized the PIA data ports can be treated as bytes of memory. Viz: 'BDATA=ADATA;' would simply echo all input bits to the corresponding output bit.

9.04.01 AT (continued)

AT can also be 'fooled' into calling a particular address by more than one name, viz:

```
AT $E004: BYTE ACIACTRL(0), ACIASTAT, ACIADATA;
```

By declaring 'ACIACTRL' as a vector (we'll get to vectors later in this document) of nil size 'ACIACTRL' and 'ACIASTAT' will both generate a reference to $E004. Of course you could also declare the AT statement as follows to accomplish the same thing:

```
AT $E004: BYTE ACIACTRL;
AT $E004: BYTE ACIASTAT, ACIADATA;
```

AT also has a special purpose when declaring locations of vectors to be used by the JUMP or CALL statements. This use is described in the following section.

Another use of AT is to define fixed memory locations as a mechanism for passing variables to PL/9 from external subroutines or to pass LOCAL or GLOBAL variables in PL/9 to external subroutines.

AT is also useful when several programmers are working on individual programs that will form part on one large control package. At certain points in each of the various programs each programmer may need to pass variables to, or get variables from, procedures written by the other programmers. The existence of variables at fixed addresses makes it very easy for the programmers to test their own routines by simulating the data they expect to receive from the other programmers programs and to verify that the data they are passing back to other programs is correct.

AT, in effect, allocates a permanent pseudo GLOBAL variable at some fixed location other than the stack. Since the variable is permanent excessive use of AT can consume large amounts of memory if used to assign what would normally be LOCAL variables, loop counters for example. A good choice for locating 'AT' RAM variables is just above the point you would normally assign the stack.

AT can also be used to make life easier if you are not inclined to use the PL/9 tracer. We would be the first to recognize that old habits die hard. If you find that working with your system monitor rather than the PL/9 tracer gives you more confidence or faster results then by all means carry on with what you are most comfortable with!

The difficulty in working at the system monitor level with variables that are allocated storage space on the stack can be a daunting one however. By using AT to declare 'hard' addresses for every variable you use, i.e. not using GLOBAL or LOCAL variables at all, this difficulty can be overcome. With each variable used by the program allocated a fixed and permanent location in memory that is KNOWN to you it will be much easier to simulate inputs and examine the results than would otherwise be possible at the system monitor level.

The foregoing was not intended to encourage you not to use the PL/9 tracer. The TRACER can be a very powerful debugging tool as it allows you to examine variables at each step of a program, and to write small programs that can simulate a variety of input data in order to test the program under various operating conditions. The power of the PL/9 debugger/tracer as a debugging tool will only become apparent after you have used it for some time.

### 9.04.02 CALL AND JUMP

These two PL/9 keywords provide a simple interface to assembly language subroutines resident in the computer. JUMP generates a jump instruction and CALL a subroutine call; they are otherwise identical.

The normal use is a statement such as "JUMP $CD03", which would cause FLEX to be re-entered at its warm start address. The operand can also be a CONSTANT (described later in this document) as in:

```
CONSTANT WARMS=$CD03, PUTCHR=$CD18;
.
.
ACCA='*;
CALL PUTCHR;
JUMP WARMS;
```

which prints an asterisk then jumps back into FLEX. ACCA, which will be described in the next section, instructs the compiler to put the character into the 6809's A accumulator, where PUTCHR expects to find it, rather than into a variable in memory.

When a CONSTANT is used to define the address of an external procedure or the address is furnished in the form $XXXX PL/9 will generate the code for EXTENDED addressing: i.e. BD XXXX for CALL, and 7E XXXX for JUMP.

Often addresses of external subroutines are arranged in vector tables. A typical vector table would be that of the GIMIX, SSB, SWTP and WINDRUSH system monitors. The vector table starts at $F800 and contain a series of addresses as follows:

```
F800    RESET       RESET THE SYSTEM MONITOR
F802    CNTRL       WARM START THE SYSTEM MONITOR
F804    INCHNE      INPUT CHARACTER, NEVER ECHO
F806    INCHAE      INPUT CHARACTER, ALWAYS ECHO
F808    INCHECK     CHECK KEYBOARD STATUS
F80A    OUTCH       OUTPUT A CHARACTER
F80C    PDATA       OUTPUT STRING POINTED TO BY 'X'
F80E    CRLF        OUTPUT CARRIAGE RETURN - LINE FEED
F810    PSTRNG      OUTPUT CRLF FOLLOWED BY STRING POINTED TO BY 'X'
F812    LRA         LOAD REAL ADDRESS
```

Since each of the above memory locations contains the address of the desired routine it is intended to be used with INDIRECT addressing, i.e. JSR [$F804], OR INDEXED addressing, i.e. LDX $F800; JSR 0,X.

Vector tables cannot be used with EXTENDED addressing, i.e. JSR $F804. If you attempt to use the standard forms of CALL or JUMP to access routines in a vector table you stand an excellent chance of crashing your system!

9.04.02 CALL AND JUMP (continued)

PL/9 provides a simple mechanism to gain access to external subroutines whose addresses are stored as a vector or in a vector table. Instead of defining the address of the external routine using CONSTANT or CALL $XXXX you must define the address using 'AT', for example:

```
AT $F800: INTEGER RESET, CNTRL, INCHNE, INCHAE, INCHECK, OUTCH, PDATA, CRLF,
                  PSTRNG, LRA;
```

Would define the entire vector table for a standard 6809 system monitor.

To access the various routines you use the normal forms of CALL and JUMP, viz:

```
CALL INCHAE;
CALL OUTCH;
JUMP RESET;
```

This time PL/9 will generate the code required for INDEXED addressing. The first example, 'CALL INCHAE' would be coded as:

```
FC F806     LDD $F806      Get the contents of $F806 into the 'D' accumulator
1F 01       TFR D,X        Transfer 'D' into 'X'
AD 84       JSR 0,X        Jump to subroutine pointed to by 'X'.
```

The last example, 'JUMP RESET' would be coded as:

```
FC F800     LDD $F800      Get the contents of $F800 into the 'D' accumulator
1F 01       TFR D,X        Transfer 'D' into 'X'
6E 84       JMP 0,X        Jump to address pointed to by 'X'
```

Obviously you can't use accumulator 'D' (see section 9.04.03) to pass a variable out to the procedure if you use this construction.

Once a vector table (we will be describing vectors in greater detail later) has been declared using the 'AT' statement as illustrated below all sorts of control possibilities are opened up as the following examples attempt to illustrate:

```
AT $F800: INTEGER MONITOR_VECTOR(10);

/*
   0 RESET          |
   1 CNTRL          |
   2 INCHNE         |
   3 INCHAE         |
   4 INCHECK        | THIS ENTIRE BLOCK IS TREATED AS A SINGLE COMMENT!
   5 OUTCH          |
   6 PDATA          |
   7 CRLF           |
   8 PSTRNG         |
   9 LRA            |
*/
```

9.04.02 CALL AND JUMP (continued)

```
CALL MONITOR_VECTOR(3);
CALL MONITOR_VECTOR(5);
JUMP MONITOR_VECTOR(0);
```

This does exactly the same job as the previous example but is considerably less readable. It does however provide the program with the ability to decide which vector in a table is to be used by assigning the appropriate number to an index variable INDEX, as in:

```
CALL MONITOR_VECTOR(INDEX);
```

which allows a degree of flexibility that is useful in some cases, such as deciding which I/O subroutine to call when data on an input matches the data in a table. We will be covering vectors and data tables in great length in a subsequent section.

C A U T I O N

A word of caution is in order when calling external subroutines. The only registers that may be altered at will by the external routines are the 'D' accumulator, the 'X' register, the 'U' register and the Condition Code Register.

THE 'DP' REGISTER MUST BE PRESERVED IF YOU USE THE 'DPAGE' DIRECTIVE AND THE 'Y' REGISTER MUST BE PRESERVED IF YOU ALLOCATE 'GLOBAL' VARIABLES.

This can be accomplished by pushing all of the registers you intend to use onto the stack before you leave PL/9 via 'GEN' statements and pulling them off the stack via 'GEN' statements when you return. Alternative you can do the same thing in your external procedure.

This caution is particularly important when using the PL/9 tracer. Failure to ensure that all registers other than 'D', 'X', 'U' and the 'CCR' are preserved can result in a complete crash of the system that can cost you the entire contents of a disk.

If you know MURPHY's law like we do this crash will occur the only time you forget to open the door to the disk drive that holds the one and only copy of the 42K program you have been working on for the last year and you are only doing a quick touchup job on a text message that is part of a program that has to be delivered to your customer who is due to show up in five minutes or you will loose a 500 million pound contract! ... GET THE MESSAGE?

SUMMARY

CALL and JUMP should be used with AT variable declarations, when accessing routines pointed to by vector tables, i.e. the address specified by the AT declaration contains the address of the routine you wish to enter.

CALL and JUMP should be used with CONSTANTS to access the routine via extended addressing, i.e. the address specified by the constant is the address at which the procedure may be entered or contains a JMP, BRA or LBRA instruction which vectors the processor to the location where the procedure is.

## 9.04.03 ACCA, ACCB, ACCD, XREG, STACK and CCR

These pseudo variables represent the corresponding MC6809 registers as follows:

a. ACCA is the 'A' accumulator.
b. ACCB is the 'B' accumulator.
c. ACCD is the 'A' and 'B' accumulators concatenated with 'A' most significant.
d. XREG is the 'X' register.
e. STACK is the 'SP' (hardware stack pointer).
f. CCR is the condition code register.

These keywords are provided primarily to interface with external assembly language routines. They can be used to pass LOCAL or GLOBAL variables out to the assembly language program or the assign LOCAL or GLOBAL variables to the result of the external program. These keywords can be treated just as any other variable in PL/9.

Supposing that you want to use the FLEX 'STAT' routine at $CD4E to poll the keyboard for an incoming character and if one is found use the FLEX 'GETCHR' routine at $CD15 to get the character and echo it back to the video display.

The FLEX 'STAT' routine will return with the 6809 condition code register (CCR) 'Z' bit (bit b2) cleared if a key has been hit or set if a key has not been hit.

The FLEX 'GETCHR' routine will 'loop' on the keyboard input port until a key is hit, and will return with the keyboard character in the 6809 A accumulator after echoing the character out to the video display. If this routine is called BEFORE a key is hit the processor can hang up on the key board indefinately. In many circumstances this can be undesirable.

The following example shows how to build a routine that will ALWAYS return whether a key has been hit or not. The ENDPROC ACCA statement simply transfers the contents of ACCA to ACCB as this is where PL/9 constructions expect to find it. The procedure will return with ACCB equal to zero if no key was hit (or if CONTROL @ is hit) or will return with ACCB equal to code of the key that was hit after echoing the character if one has been hit:

```
PROCEDURE GET_KEY_AND_ECHO_IT;
   CALL $CD4E; /* STAT */
   IF CCR AND $04 <> 0   /* this order is MOST important! */
      THEN ACCA=0;
      ELSE CALL $CD15; /* PUTCHR */
ENDPROC ACCA;
```

Two elements of the above construction have not been discussed yet. The first is the bitwise AND of 'CCR and $04'. The second is that ENDPROC is returning a value, 'ACCA'. Both of these will be discussed in detail in subsequent sections. If you don't understand what is going on in this example finish reading the rest of the manual and then return to this section.

### W A R N I N G

If you wish to use more than one of these pseudo variables the order of use is most important. See section 7.01.23 for further details.

### 9.04.04 GEN

This keyword allows you to embed 6809 machine code in line with a PL/9 program. PL/9 does not make any assumptions about GEN statements, IT ASSUMES THAT YOU KNOW WHAT YOU ARE DOING. Therefore we only recommend that GEN be used if you are reasonably conversant with 6809 assembly language programming.

One of our other products is a 6809 assembler called MACE. MACE has a built in facility to convert an assembly language source program into a series of GEN statements. If you find that you need to frequently embed 6809 machine code in PL/9 programs you might consider using MACE to do the work for you.

As mentioned in section 7.01.21 GEN can be used to generate the code necessary to preserve the 6809 registers before you call an external procedure if you are not sure what registers are going to be affected. For example:

```
GEN $34,$28;    /* PSHS DP,Y */
CALL $XXXX;
GEN $35,$28;    /* PULS DP,Y */
```

will preserve the two registers that PL/9 assumes will be preserved. The 'D', 'X', 'U' registers need not be preserved as far as PL/9 code is concerned. One last point ... you should take steps to preserve the state of the 'I' and 'F' flags in the CCR if your PL/9 program is using interupts.

A previous section also described how AT can be used to define a vector and then subsequent references to the variable name via CALL and JUMP will generate INDEXED addressing. In some circumstances the extra code generated by PL/9 to perform the INDEXED addressing may cause problems of one sort or another. The mains ones being that the 'D' accumulator and the 'X' register are used and are therefore not free to be used to pass variables to the subroutine being called.

GEN can also be used to generate INDIRECT addressing, i.e. JSR [$XXXX]. One very useful FLEX I/O routine that very few people know about (it is only documented in the 'DRIVERLESS FLEX MANUAL') is a routine called 'INCHNE', input a character but never echo it. The normal routine, GETCHR, at $CD15, automatically echos the keyboard character before returning. The address of the INCHNE routine is stored in the FLEX vector table at $D3E5. To construct the equivalent of JSR [$D3E5] you would enter:

```
GEN $AD,$9F,$D3,$E5;
```

If you want to make a clean entry into most system monitors (GMX, SSB, SWTP, and WINDRUSH), i.e. the equivalent of JMP [$F802], you would type the following:

```
GEN $6E,$9F,$F8,$02;
```

Alternatively you could make the entry into your system monitor via the 'MONITR' vector in FLEX (i.e. the vector the 'MON' command uses), i.e. the equivalent of JMP [$D3F3], by typing the following:

```
GEN $6E,$9F,$D3,$F3;
```

You may also use BYTE constants with GEN statements, viz:

```
CONSTANT JMP=$6E, IND=$9F, MON_HI=$D3, MON_LO=$F3;

GEN JMP,IND,MON_HI,MON_LO;
```

There are many other uses of GEN. Most will be obvious to assembly language programmers. All will be dangerous if you are not! Enough said!

## 9.04.05 ASMPROC

ASMPROC is a special keyword that allows you to really get carried away with GEN statements!

There are many things that PL/9 cannot handle directly, like stack and register manipulations, and there are many tasks that PL/9 will be less than brilliant at at producing code for.

ASMPROC serves as a label in a block of GEN statements. No ENDPROC is required, in fact the compiler will reject it. The ASMPROC statement with the accompanying name generates no code thereby preserving the assembled relationship between one part of the GEN code and another. You may allocate local variables with ASMPROCs and therefore pass variables to them. It is up to you to know what position the variables will be on the stack when the ASMPROC is called. Refer to the Reference Manual for further details.

ASMPROC allows you to generate a complete assembly language program via GEN statements and embed it amongst other PL/9 procedures. The structure of an ASMPROC is much like that of any other PL/9 procedure, i.e. then can receive data on the stack, and return data in the MC6809's 'D' and 'X' registers. e.g.

```
ASMPROC _ASMPROC_DEMO(BYTE):BYTE;
   GEN ....
   GEN ....
   GEN ....
   GEN $39;
```

is passed one BYTE variable on the stack and will be expected to return a BYTE variable in 'B'.

The only thing that differentiates an ASMPROC from a PROCEDURE is that PL/9 expects all procedures, except the last, to end in an ENDPROC. ASMPROCs do not have this requirement and will therefore not generate the RTS instruction as will PROCEDUREs.

When using ASMPROC It is up to the programmer to ensure the following:

1. The ASMPROC is written in position independent code if you expect the main PL/9 program to be position independent OR wish to use the PL/9 tracer.

2. The ASMPROC preserves the 'DP' and 'Y' registers if 'DPAGE' and 'GLOBAL', respectively, are being used in the main program.

3. The ASMPROC terminates in an RTS, or equivalent, OR Jumps to a known fixed address that is not part of the PL/9 program that includes the ASMPROC, OR branches to a known POSITION in the PL/9 program. These last two options are to ensure position independance.

There are several examples of ASMPROCs in the FLEX I/O library which is covered in a separate section.

Our assembler product 'MACE' has built-in facilities for simplifying the generation of ASMPROCs. If you find that you are coding a lot of procedures as ASMPROCs 'MACE' can save you a bit of work.

## 9.05.00 ARITHMETIC IN PL/9

PL/9 is basically a control language. Although it possesses a good repertoire of numeric abilities, it is not likely to be suitable for the sort of arithmetic found in a wages package or a stock control program, where financial sums running to thousands of pounds (or dollars) must be held to sufficient accuracy to avoid losing the odd pennies. Its numeric abilities are more suited to the world of control systems, where absolute accuracy is less important than speed and reliability. The way arithmetic is handled in PL/9 is therefore designed to allow the programmer to specify exactly how the sums are to be done, in what order the operands are to be accessed and whether a calculation should be done in fixed or floating point.

## 9.05.01 NUMERIC QUANTITIES

PL/9 is a language designed to be used to control things in the real world. The numeric quantities it uses are therefore closely related to the data types the 6809 itself handles. Three sizes of data can be dealt with:

BYTEs are 8-bit quantities, having values ranging from -128 to +127 and are processed in the 6809's B accumulator. Although they are normally regarded as signed quantities, there are times (when performing bit-wise logical operations for example) when they should instead be regarded as unsigned values between 0 and 255. A mechanism is provided in PL/9 for making this distinction; this is described later.

INTEGERs are 16-bit quantities, having values ranging from -32768 to +32767 and are processed in the 6809's D accumulator. They too may be regarded as unsigned numbers, this time in the range 0 to 65535, by the same mechanism as for BYTEs.

REALs are 32-bit floating-point values, stored in a four-byte binary form where the first byte contains the exponent and the remaining three the mantissa. The range covers approximately +-1.0E-38 to +-1.0E37 and there are usually six (decimal) digits of precision. They are processed on the stack and passed about in D and X, with D holding the most significant half.

Further information on this subject can be found in section 9.01.00 of this Guide and the and section 7.03.XX of the Language Reference Manual.

## 9.05.02 VARIABLES AND DATA TYPES

A Variable is a place at which something can be stored. Because the item can be removed and something else put in its place, the contents of the store are variable, hence the name. A Constant, on the other hand, is something eternal and unchanging, something you might put in a variable, for example.

Variables in PL/9, like numeric quantities, come in three sizes; BYTE, INTEGER and REAL. The names you give to your variables do not have to give any clue as to their type; you are free to call a variable anything from X to X_MARKS_THE_SPOT_WHERE_THE_BUTLER_DID_IT. The type of a variable is determined at the point it is declared, and from that point on it is always that type.

In the following example, only local variables (variables private to the procedure in which they are declared) will be declared:

```
PROCEDURE VARIABLE_TYPES:
   BYTE BYTE1, BYTE2, BYTE3:
   INTEGER INTEGER1, INTEGER2, INTEGER3:
   REAL REAL1, REAL2, REAL3;
```

This choice of names is not recommended for use generally - it is to be hoped that you can devise names more suited to the job the program is doing. They just serve here to help distinguish the various types. Note the colons separating the parts of the declaration, which may itself be crammed into or expanded out to as many lines as you wish. The order of the variable declarations is also unimportant; you can declare whichever you like first, or declare some of the BYTEs, then some INTEGERs and REALs, then the rest of the BYTEs. It's entirely up to you. The only point to note is that the variables declared earliest use a shorter form of addressing to access them, so declaring frequently used variables first keeps down the size of your program and makes it run a little faster.

Note also that a colon is used to separate the different data types REAL, INTEGER, and BYTE and that a comma is used to separate the different variables within each type group.

## 9.05.03 LOCAL VARIABLES

The examples in the preceeding sections of this guide have used what are known as LOCAL variables, which are declared as part of a PROCEDURE declaration. Local variables are held on the system stack, with space being allocated for them when they are declared and released when the procedure ends. The name of a local variable is private to the procedure in which it is declared. Any number of different procedures may use the same local variable names; in each case it is a completely different variable.

Parameters passed to procedures (covered in a subsequent section) are also local, the only difference being that they are placed on the stack before the procedure is called and are removed after the procedure exits.

## 9.05.04 GLOBAL VARIABLES

Where a number of procedures are to use the same variables it is wasteful of code to pass the variables as parameters from one procedure to the next, especially if there are many variables. PL/9 makes provision in two ways for variables to be shared. The first mechanism is the Global variable. The GLOBAL statement causes room to be made on the system stack for the declared variables, then the 6809's 'Y' index register is set to point to the base of the space reserved. Any procedure can access the global variables by means of an offset from the 'Y' register. Only one GLOBAL statement can be used in a program and its format is much the same as the variable declaration part of a PROCEDURE declaration. Global variables are stored in memory at a place that depends on where the stack has been allocated. See the Language Reference Manual and section 9.10.00 of this Guide for more details.

The second mechanism for declaring variables that are to be universally available locates them at fixed places in the microprocessor's memory space, and is therefore ideally suited to declaring addresses that form part of the system hardware. The AT declaration, described in section 9.04.01, allows you to define 'hard' addresses by name, enabling drivers to be easily written for Input and Output devices or space to be reserved for communication between a PL/9 program and a module written in another language.

## 9.05.05 CONSTANT

Most programs use many numeric constants. In one of the examples given earlier, the statement IN=IN+1 was used to move along a text string. The significance of the "1" in this case is fairly obvious, but this is not always the case. Suppose, for instance, that a program line contained the procedure call PUTCHAR(10); what is the significance of the number 10? It would be more readable if PUTCHAR(LINE_FEED) were used instead; in this case it is obvious what is happening. It might even be better to use IN=IN+INCREMENT in the first example. PL/9 allows you to define numeric constants so that their symbol names can be used later instead of the numbers they represent. Another benefit of this is that if the value of the constant has to be changed in a later revision of the program then there's only one occurrence that has to be altered, usually fairly early on in the program.

Note that there is no way of declaring the type of a constant -

All constants are effectively INTEGERs but are TRUNCATED to BYTE if necessary

This is particularly important to remember when using BYTE constants in comparisons with INTEGERS or bitwise operations on INTEGERS.

## 9.05.05 CONSTANT (continued)

REAL constants are not implemented in this version of PL/9 however there are two techniques you can use to overcome this limitation. The first is to declare a read-only data table (see section 9.05.07) as multi-element vector before you start writing your procedures. Each item in the table is accessed by subscripting the variable name, viz:

```
REAL POWERS_OF_10  1, 10, 100, 1000, 10000, 100000, 1000000;
```

A reference to 'POWERS_OF_TEN(0)' would be a REAL constant of 1, a reference to 'POWERS_OF_TEN(1)' would be a REAL constant of 10, etc.

The second technique is to give each REAL a name which may then be used in data assignments or evaluations just as INTEGER and BYTE CONSTANTS, viz:

```
REAL PI 3.1415926;
REAL PI_SQUARED 6.2831853;
REAL COEFF1 -6.88677461E-9;
```

etc...

## 9.05.06 PROCEDURES AS VARIABLES

PROCEDUREs can also be regarded as variable types by causing them to return specific values and thereby become functions. The procedure call can then take the place of any variable of the appropriate type.

The ENDPROC and RETURN sections of the Language Reference Manual and section 9.06.01 of this guide, which describes custom functions, cover this topic in greater detail.

## 9.05.07 READ ONLY DATA

A program may require data for initialization of tables and other vectors. The AVERAGE example in section 9.06.04 contains an example of a REAL data list; BYTE and INTEGER lists are also allowed, as the Language Reference Manual explains.

One of the most common uses of read only data is in forming text strings. Text strings can be declared either in-line with the program, as in PRINT("HELLO WORLD"), or in BYTE declarations such as BYTE MESSAGE "HELLO WORLD". Data declarations amount to read- only vectors; PL/9 will refuse to allow you to use one on the left-hand side of an assignment.

Read only data, particularly data tables (data lists) are best declared in the early part of the program just after the STACK, and GLOBAL declarations. You may, if you wish, declare data of this type BETWEEN procedures. This can have the benefit of shortening the addressing range of the PL/9 code (thereby making it more compact and faster) if the data is only going to be used by the subsequent procedure. If the data is to be used frequently by several procedures in the program little benefit will be gained by placing the table between procedures in the program. In this situation placing the data in the middle of the procedures would only tend to make the body of the program less readable.

PL/9 will not allow you to declare read only data in the middle of a procedure unless it is being passed to a function or its address is being assigned to a variable, viz:

```
PRINT("THIS IS A MESSAGE"); is permitted in a procedure

STRING="THIS IS A MESSAGE"; is permitted where STRING is an INTEGER variable
```

STRING will now contain the address of the first byte of the message string and is, in effect, a pointer to the string. You may then say:

```
PRINT(STRING);
```

BUT

```
BYTE MESSAGE "THIS IS A MESSAGE";    is NOT permitted in a procedure and may
                                     only be used outside of procedures.
```

## 9.05.08 PRINTING NUMBERS

As with any feature of a language, the best way of learning it is to try it out. You are therefore encouraged to try every example and satisfy yourself that it works. An incomplete understanding of the way PL/9 works can have results ranging from inconvenience to disaster, and in any case the author would like to feel that the time spent writing this user's guide has not been wasted!

In order to help demonstrate PL/9 arithmetic, it's useful to be able to print out numbers. As explained earlier, PL/9 doesn't contain any built-in I/O routines, so a procedure to do this will be needed. Type in the following program:

```
0001 INCLUDE IOSUBS;
0002 INCLUDE REALCON;
0003
0004 PROCEDURE PRNUM (REAL NUMBER): BYTE BUFFER(20);
0005
0006    PRINT(ASCII(NUMBER,.BUFFER));
0007
0008 ENDPROC;
```

The job of this routine is to take a number passed to it, convert it into an ASCII string and print that on the VDU screen. If the number passed is not already REAL it will be automatically converted ("promoted") to REAL. The procedure ASCII is part of the REALCON.LIB library file, which has to be INCLUDEd. The function of ASCII is to take the REAL number passed to it, convert that into a string of ASCII characters in the buffer whose address is supplied, and terminate the string with a null. ASCII therefore requires two parameters to be passed to it; a REAL number and the address of a buffer. The first of these parameters is the number that was passed to PRNUM; it is merely copied for use by ASCII. The second parameter is declared in line 4 as a vector of type BYTE and of size 20, large enough to hold the longest number that ASCII will generate.

The (REAL NUMBER) in line 4 tells PL/9 that one parameter is required by PRNUM and that its type is REAL. Line 6 is an example of a procedure call within another procedure call; the call to ASCII has a return value equal to the second parameter passed to it, which in this case is the address of BUFFER (indicated by the dot), the correct type for PRINT. PL/9 allows you to use either a dot or an ampersand (&) to indicate the address of a variable (otherwise known as a "pointer to" the variable). Which you use depends upon your personal style; the dot is borrowed from PL/M and the ampersand from C. A later section in this guide has more to say about Pointers and how they are used.

Lastly there is an ENDPROC at the end of this procedure; it's going to be called from elsewhere so it has to be able to return.

Once you have typed in the program and checked that it compiles without errors, type "S=PRNUM.LIB". This writes it to your working disc, ready for use as a library file.

## 9.05.09 "INTEGER" VERSUS "REAL" ARITHMETIC

PL/9 handles BYTE and INTEGER quantities differently from REAL numbers when evaluating arithmetic expressions. The effect may be the same, but a different part of the compiler is used, and different code is generated. PL/9 will usually convert from one type to another if it can, but beware when intermixing REALs with other types - the results may not always be what you expected, as will shortly be explained. For the rest of this manual, the term "Integer Expression" will be used to represent an expression where the result is of type BYTE or INTEGER, while "Real Expression" will represent an expression where the result is REAL.

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                                                                           *
*  It is VERY important to remember that PL/9 sign extends BYTES to form   *
*  INTEGERS when a BYTE is being evaluated in an expression OR when a       *
*  BYTE is being assigned to an INTEGER except when instructed otherwise    *
*  by the programmer.                                                       *
*                                                                           *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

This may sound a bit strange but remember PL/9 assumes that you are working with SIGNED numbers. This technique has been adopted to facilitate comparisons and assignments between SIGNED BYTES and SIGNED INTEGERS which are very common in most control work.

PL/9 has been provided with two special facilities that prevent the sign extension from taking place when you wish to evaluate a BYTE as an unsigned number.

The first is that the compiler handles CONSTANTS declared as HEX numbers and HEX numbers used in comparisons or assignments in a special manner. In these circumstances it assumes that you wish to work with unsigned numbers for if you mean -128 why would you state $80? This distinction is used to greatly simplify constructions that are designed to operate with I/O devices, etc.

The second facility is the exclamation mark (!). This facility simply tells the compiler NOT to sign extend the BYTE variable(s) in the expression before a comparison is made. This facility is also of use when you wish to perform unsigned addition and/or subtraction of binary numbers.

The 'INTEGER' function has been provided to facilitate assigning an unsigned BYTE to an unsigned INTEGER.

Working with unsigned BYTEs and INTEGERs are discussed in detail in section 7.03.02 in the Reference Manual and section 9.05.13 of this guide.

## 9.05.10 ARITHMETIC OPERATORS AND EVALUATORS

The arithmetic operators recognized by PL/9 are as follows:

A + B Add the signed value of B to the signed value of A.

A - B Subtract the signed value of B from the signed value of A.

A * B Multiply the signed value of A by the signed value of B.

A / B Divide the signed value of A by the signed value of B.

A \ B Return the remainder (modulus) of the signed value of A divided by the signed value of B. (only BYTEs and INTEGERs may be used)

A = -B Negation (unary minus).

The arithmetic evaluators recognized by PL/9 are as follows:

= EQUAL

<> NOT EQUAL

<> 0 NOT EQUAL TO ZERO is implicit and may be omitted to shorten code

> GREATER THAN

>= GREATER THAN or EQUAL (=> is NOT acceptable)

< LESS THAN

<= LESS THAN or EQUAL (=< is NOT acceptable)

Other special characters recognized by PL/9 are as follows:

= Assign the value on the left of the equal sign the value or result of the expression on the right.

"strng" Assign the address of the first byte in the enclosed string to the variable. The string will be terminated with a null.

. or & Assign the address of the variable not the value of the variable, i.e. assign a pointer to the variable (or procedure); e.g. VAR1 = .VAR2;

. or & Define a variable as a pointer to REAL, INTEGER or BYTE sized data; e.g. GLOBAL REAL .RPOINTER:INTEGER .IPOINTER:BYTE .BPOINTER;

$ Evaluate or assign a HEX number. These numbers will be treated as unsigned numbers in simple evaluation or assignment expression.

! Do not perform sign extension when evaluating BYTEs or INTEGERs.

' Evaluate or assign an ASCII character.

(n) Subscript a vector or a pointer with a number, a constant, or a variable.

## 9.05.11 ARITHMETIC OPERATOR PRECEDENCE

First of all, an example of arithmetic where all of the operands are simple numbers;

```
0001 INCLUDE IOSUBS;
0002 INCLUDE REALCON;
0003 INCLUDE PRNUM;
0004
0005 PROCEDURE SIMPLE_ARITHMETIC;
0006
0007    PRINT("2+3*5  =  ");
0008    PRNUM(2+3*5);
```

This is the simplest way of demonstrating the way that multiplication takes precedence over addition in PL/9. (For those that can't be bothered to run the example, the answer is 17, not 25.) A precedence table looks like this:

| | |
|---|---|
| Highest Precedence | Negation/unary minus (-) and Functions |
| | Multiplication (*), Division (/), Modulus (\) |
| | Addition (+), Subtraction (-) |
| | AND |
| | OR, EOR/XOR |
| | (=), (<>), (>=), (<=), (>), (<) |
| | .AND |
| Lowest Precedence | .OR, .EOR/.XOR |

"Functions" mean BYTE, INTEGER, SQR etc., which will be described later. The the effect oftable is that unless brackets are used to modify the default order of precedence, negation will be done before multiplication or division, which will in turn be done before addition or subtraction, and so on.

Try modifying the example above to test the effect of other arithmetic expressions. Note that although constants are used in this example, the principle works just as well for variables and elements of vectors.

Note also that although the result of any expression reflects the order of evaluation, the actual arguments (the component numbers) are individually accessed by PL/9 in left to right order. This can be of importance when dealing with real I/O devices rather than simple numbers.

Lastly note that since this is a Real expression (because PRNUM takes a REAL argument), modulus (\) and the bit operators AND, OR and EOR will result in errors if used.

As a further exercise, try using brackets to alter the order of evaluation. (2+3)*5 will of course yield the answer 25, not 17. Brackets can be nested to any required depth.

## 9.05.12 MIXED MODE ARITHMETIC

It was stated earlier that the result of mixing Integer and Real numbers in expressions can cause unexpected results. The last example was a Real Expression because PRNUM expects a REAL operand. This means that each of the terms in the expression must either already be REAL or will be "promoted" to REAL before being included in the calculation. The addition and multiplication will be done in floating point, not integer or byte form. The latter could have been "forced" by the following:

```
0008    PRNUM(FLOAT(2+3*5));
```

FLOAT is a function that takes a BYTE or INTEGER argument and returns the REAL equivalent. In this case the entire expression will be forced to be evaluated as integer, thereby allowing the use of modulus and the logical operators. There is also a FIX operator that performs the opposite function, that of converting a REAL value into INTEGER.

The rule that should always be borne in mind is that the type of an expression is determined by the type of wherever the result is going. An assignment of the form A=B+C thus causes the expression B+C to be evaluated as integer if A is either BYTE or INTEGER, real if A is REAL. In either case, each term of the expression will be converted to the appropriate type before being included in the calculation. This may appear clumsy by comparison with the way other languages handle expressions, but at least it gives the programmer the ability to tightly control the way the compiler does its arithmetic.

In general, mixed mode arithmetic should not cause too many problems as long as you are careful to examine the implications. PL/9 will usually do the necessary type conversions and in cases where this is not possible will report an error.

## 9.05.13 UNSIGNED INTEGER ARITHMETIC (to BITS or not to BITS)

Our sincere apologies to Shakespeare. We hoped that this title has drawn your attention to this section as it is VERY, VERY important that you understand what we are going to tell you here. This section is particularly important if you are an ASSEMBLY language programmer or are converting programs written in SPL/M to PL/9.

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                                                                       *
*  Failing to understand what is said in this section can cause VERY    *
*  undesirable results if you are used to treating BYTES and            *
*  INTEGERS as binary bit patterns rather than as signed numbers.       *
*                                                                       *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

The signed nature of integer arithmetic (that is arithmetic comprising INTEGERs and BYTEs) can cause some unexpected effects if you do not understand the way the compiler sign extends BYTEs into INTEGERs. This topic was covered in detail in section 7.03.02, but will be discussed again here in detail.

To force the compiler into a mode where the entire EVALUATION is to be performed in an unsigned manner the expression is preceeded with an exclamation mark (!), where the exclamation mark means "the following expression, IN ITS ENTIRETY, is unsigned, so don't do any sign extension".

This release of the compiler has attempted to simplify the way the programmer must handle evaluations and assignments of unsigned BYTEs and INTEGERs. The compiler now recognizes CONSTANTS that have been assigned via a HEX number as being distinctly different from CONSTANTS that have been assigned via a signed number. For example $80 and -128 are no longer the same to the compiler. The same set of rules applies to numbers used in simple assignments and evaluations.

It is assumed that when you will be working with signed numbers you will use decimal numbers. HEX numbers are automatically treated as unsigned numbers in simple evaluation or assignment expressions.

For the most part you will not have any problems comparing BYTEs with BYTEs or INTEGERs with INTEGERs but special precautions are necessary when you try to mix the two data sizes in expressions or if you use an evaluation operator other than equal (=) or not equal (<>).

The following are the five primary rules that MUST be obeyed if you wish to work with unsigned numbers successfully.

### R U L E 1

When performing simple evaluations or assignments use CONSTANTS defined by HEX numbers or HEX numbers in the expression.

### R U L E 2

When comparing unsigned INTEGERs with unsigned BYTEs or vice versa use the exclamation mark (!) to tell the compiler not to sign extend the BYTE quantity. This is particularly important when the BYTE quantity is on the left side of the equal sign.

9.05.13 UNSIGNED INTEGER ARITHMETIC continued

R U L E 3

If the greater than (>), greater than or equal to (>=), less than (<), or less than or equal to (<=) evaluators are used in the expression use the exclamation mark (!) to force unsigned evaluation.

R U L E 4

When you wish to perform an unsigned addition or subtraction with a mixture of INTEGERs and BYTEs use the exclamation mark (!) to prevent the compiler from sign extending the BYTE quantities.

R U L E 5

When ASSIGNING an unsigned BYTE to an unsigned INTEGER use the 'INTEGER' function.

The best way to illustrate why these rules exist is to give you some examples. The following declarations are assumed to have been made for the variables and constants used in the examples. It is also assumed that the variables have been initialized as indicated.

```
CONSTANT B=$80, I=$0080;

GLOBAL BYTE B1: INTEGER I1;

PROCEDURE INIT;
   B1=B;
   I1=I;
ENDPROC;
```

RULE 1

The following constructions all produce the results you would expect in unsigned comparisons with CONSTANTS defined by HEX (and only HEX) numbers or comparisons made with HEX (and only HEX) numbers:

```
IF B1 = $80 THEN...          IF $80 = B1 THEN...

IF B = $80 THEN...           IF $80 = B THEN...

IF I1 = $0080 THEN...        IF $0080 = I1 THEN...

IF I = $0080 THEN...         IF $0080 = I THEN...
```

9.05.13 UNSIGNED INTEGER ARITHMETIC continued

RULE 2

The following are also valid but should not be used in order to avoid confusion. If the positions of the INTEGER and BYTE variables are reversed the results will NOT be as expected!

```
IF I = B THEN...          IF I1 = B1 THEN...

IF I = $80 THEN...        IF I1 = B THEN...
```

Here is what happens in circumstances where the BYTE variable is on the left side of the equal sign.

| | |
|---|---|
| IF B = I THEN... | will result in B being sign extended to form $FF80 and will therefore produce an erronious result. |
| IF !B = I THEN... | the (!) prevents B from being sign extended to form $FF80 and therefore gives the expected result. |
| IF B1 = I1 THEN... | will result in B1 being sign extended to form $FF80 and will therefore produce an erronious result. |
| IF !B1 = I1 THEN... | the (!) prevents B1 from being sign extended to form $FF80 and therefore gives the expected result. |
| IF $80 = I THEN... | again produces an error as $80 will be sign extended to form $FF80 before the comparison is made. |
| IF !$80 = I THEN... | the (!) prevents $80 from being sign extended to form $FF80 and therefore gives the expected result. |
| IF B = I1 THEN... | will result in B being sign extended to form $FF80 and will therefore produce an erronious result. |
| IF !B = I1 THEN... | the (!) prevents B from being sign extended to form $FF80 and therefore gives the expected result. |

## 9.05.13 UNSIGNED INTEGER ARITHMETIC continued

### RULE 3

ALL of the following expressions will produce erronious results due to sign extending of any variable 'greater than' $7F.

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

IF B > $7F THEN               IF I > $7F THEN...
```

Simply putting an exclamation mark in front of the first element of the expression tells the compiler not to sign extend any element of the expression and you will therefore get the results you would expect, i.e. all of the following expressions work:

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

IF !B > $7F THEN              IF !I > $7F THEN...
```

### RULE 4

Unsigned addition and subtraction of like sized quantities will never present any problems. The only time this rule comes into force is when you mix BYTEs and INTEGERs in the expression. For example all of the following expressions produce the expected results:

```
IF B+1 = $81 THEN...          IF B-1 = $7F THEN...
IF B1+1 = $81 THEN...         IF B1-1 = $7F THEN...
IF I+1 = $81 THEN...          IF I-1 = $7F THEN...
IF I1+1 = $81 THEN...         IF I1-1 = $7F THEN...

IF I+1 = B+1 THEN...          IF I1-1 = B1-1 THEN...
```

BUT the following will not:

```
IF I1+B = I+B1 THEN...
```

Again, putting the exclamation mark before the expression prevents the sign extension that would have occurred in the previous example and the construction works, viz:

```
IF !I1+B = I+B1 THEN...   will work!
```

### RULE 5

This rule applies when assigning a BYTE variable to an INTEGER variable. For example:

```
I1=B1;
```

would result in I1 being assigned the value of B AFTER it was sign extended, i.e. I1 would be equal to $FF80 after the assignment.

To prevent this from happening you use the INTEGER function, viz.

```
I1=INTEGER(B1);
```

this time I1 would be equal to $0080 as you would expect.

## MIXED EXPRESSIONS ... SIGNED + UNSIGNED = CONFUSION

It is not permitted to mix unsigned and signed evaluations in the same expression. For example if you wanted to compare an unsigned number with the sum of a signed operation you could NOT use an expression like this:

```
IF A + B = !C THEN...
```

the compiler will reject any attempt to put an exclamation mark on the right of an equal sign (=).

The above construction must be performed in two individual steps, first the signed addition:

```
TCHAR = A + B;
```

then the unsigned comparison:

```
IF !TCHAR = C THEN...
```

### N O T E

The exclamation mark can only be used immediately after IF, WHILE, or UNTIL in EVALUATION statements or immediately after an equal sign in an ASSIGNMENT.

Generally speaking you should attempt to keep evaluation and assignments of unsigned numbers as simple as possible, even if this means performing an evaluation over several lines of program source and using intermediate variables for storage. This latter point is particularly important when part of the evaluation is to be performed as signed and another part is to be performed as unsigned.

You are welcome to try to construct complex evaluations or assignments in order to save code. In these situations you should always look at the code that PL/9 produces to make sure that it is doing what you expect. Remember that the compiler will slip into signed operation (i.e. it will start sign extending numbers) with the slightest excuse. If you don't understand the code that PL/9 produces keep the expressions simple and they will work 100% of the time.

The bit operators AND, OR, and EOR may be mixed in with unsigned addition and subtraction operations. As multiplication and division are always signed using these operations in an unsigned expression may produce unexpected results.

Generally speaking the bit operators do not treat BYTEs and INTEGERs as signed values. Special caution must be observed when operating on INTEGERs with BYTE variables or constants and vice versa. The rules for this will be outlined in section 9.07.00.

## 9.05.14 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 and will be discussed in greater detail in a subsequent section.

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 preserves the sign bit (b15 in an INTEGER and b7 in a BYTE) on a right shift and shifts zeros into the least significant bit (b0) on 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.

9.05.14 FUNCTIONS (continued)

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).

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.

NOTE: All functions must form part of a data assignment or evaluation expression in order to be applied. Functions are not commands! i.e.:

```
CHAR1=SHIFT(CHAR2,3);
CHAR=SWAP(CHAR);
INT1=INTEGER(BYTE2);
IF SHIFT(CHAR1,-4) <> SWAP(CHAR) THEN ...
```

are all valid ... whilst:

```
SHIFT(INT3,-5);
SWAP(CHAR3);
INTEGER(BYTE6)
```

are not permitted

## 9.06.00 ADVANCED PROGRAMMING STRUCTURES

The following sections are designed to serve as a guide to the powerful constructions permitted in PL/9.

The topics covered will assume that you already have a good understanding of the topics covered in the preceeding sections.

## 9.06.01 CUSTOM FUNCTIONS (ENDPROC and RETURN)

The built-in set of PL/9 functions described in the preceeding section can be added to at will, by constructing library procedures to perform useful functions. For example, although there is no ABS function in PL/9, one can easily be written - this one takes an INTEGER operand and returns its absolute value:

```
0001 PROCEDURE ABS(INTEGER OPERAND);
0002     IF OPERAND<0 THEN OPERAND=-OPERAND;
0003 ENDPROC OPERAND;
```

Line 3 is the standard construct for creating a function. Any PL/9 procedure can return a value and thereby become a function; all that is needed is to state after the ENDPROC the value or expression that is to be returned as the value of the function.

Another useful function is ROUND, which takes a REAL operand and returns the nearest REAL integer (as distinct from INTEGER):

```
0001 PROCEDURE ROUND(REAL OPERAND);
0002 ENDPROC REAL INT(OPERAND+0.5);
```

Note the use of ENDPROC REAL to tell the compiler what data type is to be returned from the function. If the REAL were omitted then the result would be returned as an INTEGER. PL/9 will choose between BYTE and INTEGER when deciding what type to return; where the returned value is ambiguous it is necessary to specify BYTE, INTEGER or REAL to force the desired type.

The general rule to follow when returning values from a procedure is to ALWAYS specify the data type (BYTE, INTEGER) if the expression is logically or mathematically complicated or involves more than one data type. When REAL numbers are involved it is best to specify REAL in all cases.

In the Language Reference Manual, under GEN, can be found another example of a function, this one being a random number generator.

The SCIPACK.LIB file contains several useful scientific functions that operate in much the same way as SQR; try writing programs that test these functions. See how complex an expression you can write (it can carry on from one line to another) and compare your result with what BASIC or a calculator gives.

## 9.06.01 CUSTOM FUNCTIONS (ENDPROC and RETURN) (continued)

Passed variables MUST always be declared in the same order with a comma separating each item. If a variable is used as a pointer it must be declared as such in the function and assigned as such in the procedure that is calling the function.

For example:

```
PROCEDURE FUNCTION(REAL .POINTER, R1):INTEGER I, BYTE B;
   .
ENDPROC REAL R1;


PROCEDURE FUNCTION_CALL:REAL TABLE(10), ODATA, IDATA:INTEGER COUNT: BYTE CHAR;

   ODATA=FUNCTION(.TABLE(2), IDATA, COUNT, CHAR);

ENDPROC;
```

There are a few points to keep in mind when you are writing function procedures:

(1) All variables are passed to the function on the stack. You may therefore pass as many parameters to the function as your application requires. When the function procedure terminates the stack will be cleaned up by the calling procedure. This facilitates the use of RETURN in function procedures.

(2) The value returned by a function procedure will be returned in 'B' if it is a BYTE, 'D' if it is an INTEGER and in 'D' and 'X' if it is a REAL. Only one variable may be returned in this manner. If you need to return more than one item you must do it via GLOBAL or AT variables.

(3) Since the value returned by a function is returned via registers (i.e. it is NOT on the stack) the value does not have to be used, i.e. it may be ignored.

(4) Multiple function constructions are permitted provided that the function that uses another function is passed the correct data size.

(5) You can force the size of the returned variable in the procedure declaration by declaring the size of the variable you wish to return WITHOUT a name, e.g.

```
PROCEDURE FUNCTION(REAL NUM):INTEGER I:REAL;
                                      ----    This forces the procedure to
                                              return a REAL sized variable.
```

This is particularly useful in recursive procedures.

There are many examples of function procedures throughout this section and in section eight. The Language Reference Manual also describes the use of RETURN in the context of function procedures where it may be used to not only terminate the procedure early but also return a value in much the same way as ENDPROC.

## 9.06.02 VECTORS AND DATA TABLES

The name given to a multiple-element, single dimension (length), read-write variable is a vector. Some programmers prefer to call read only vectors data tables or data lists. Whenever we are referring to 'read-only' data we will use the term 'data table' and whenever we are referring to 'read-write' VARIABLES we will use the term 'vector to help differentiate between the two distinctly' different uses of vectors.

All of the examples so far that have used variables have taken the variable name to represent a single unit of storage, whether it be BYTE, INTEGER or REAL. It is frequently useful to be able to regard a variable name as referring to not one but several units of storage. For example, the word "HELP" could be stored in the four BYTE variables LETTER1 through LETTER4, but this would be very clumsy when it came to printing the word on the terminal. It is much more convenient to use a single variable, say WORD, known to comprise (in this case) four elements each of size BYTE, where the first element contains the letter H, the second E and so on. Likewise, a REAL variable could contain twelve floating point values, each one being a measured value for one of the twelve months in the year.

If you are familiar with BASIC you will no doubt have used Arrays; these are essentially the same thing. Vectors in PL/9 can be of any length but may only have one dimension. This means that there is no direct way of, for example, representing a video display as a single BYTE variable having 24 rows by 80 columns; instead you must visualise it as a long vector of 1920 elements, where each row starts immediately after the one preceeding it. This in fact is the usual distinction between the terms Vector and Array; the latter may have more than one dimension.

Some examples of declaring and using vectors will be presented after the next section, since in PL/9 vectors and pointers are often closely associated. In the mean time, here are some points worth noting about vectors:

1. The elements of a vector are numbered from zero upwards. If you are a BASIC programmer you will no doubt be used to array elements being numbered from one upwards; most BASIC dialects however allow you to use the zeroth element giving you one more element than you DIMensioned. PL/9 vectors are numbered from zero and the total number of elements available is what you specified in the variable declaration.

   Thus the declaration TABLE(3) gives you a three element read-write vector with each element numbered as:

```
TABLE(0)                                  TABLE(1)
TABLE(1)     ..........  NOT  ..........  TABLE(2)
TABLE(2)                                  TABLE(3)
```

2. Expressions involving vectors can be as complicated as you choose. For example, X=A(2-B(C*ABS(D/55))+ROUND(SQR(E*99.23))) is perfectly acceptable to PL/9, if that's what your application needs.

3. The zeroth element of a vector is equivalent to the same variable un-subscripted. For example, VECTOR, VECTOR(0) and VECTOR((((0)))) are all equivalent and will result in the same code being generated.

## 9.06.03 POINTERS

So far, the variables used in the examples have been simply places in which data is kept, with vectors introduced in the last section as multiple-element versions of the same. When a variable is used as a parameter of a procedure, a copy of the variable is passed. The actual location of the original is not known to the procedure called - it has no use for this knowledge anyway.

Suppose however that the procedure has the job of processing a large quantity of data. In this case, not just one value but many may have to be passed to and from the calling program. Since only one value can be directly returned from a procedure, what needs to be passed to the procedure is not the data itself, rather the address of the data, so the procedure can work on it.

There are two ways in which parameters can be passed to procedures, known as Passing by Value and Passing by Address . In the former case, a copy of the original variable, or a constant value, is passed to the procedure, which may then do what it pleases with the parameter without affecting the original. Where a parameter is passed by address, however, the compiler works out the address of the variable (which may be an element of a vector) and passes this address to the procedure. The procedure now has access to the original copy of the variable and may alter it at will. The address that has been passed is termed a "Pointer to" the variable.

A pointer, then, contains the address of some data. Since the 6809 has a 16-bit address bus, all pointers must be 16 bits, the size of an INTEGER. This does not mean that all pointers are INTEGERs, however, as will shortly be explained. It can, however, always be held in an INTEGER variable whether the item to which it points is a single BYTE, INTEGER or REAL variable or a vector of BYTEs, INTEGERs or REALs. Once a pointer has been generated it behaves in the same way as any other INTEGER quantity as far as arithmetic operations ON THE POINTER ITSELF are concerned. Pointers may be added, subtracted, multiplied or divided; PL/9 is not concerned whether the operations make sense.

Consider the function of the PRINT routine in the IOSUBS library. In most of the examples so far, PRINT has been followed by an ASCII string in quotation marks, as in PRINT("HELLO"). What happens here is that the characters H E L L O are placed in memory by the compiler and terminated by a null. The address of the first of these characters (the H) is used as the parameter to be passed. PRINT now knows where the data can be found.

Inside the PRINT procedure, the passed parameter is declared as follows:

```
PROCEDURE PRINT (BYTE .STRING):
```

What the dot in front of STRING tells the compiler is that the passed parameter is a pointer to a sequence of BYTE values. STRING(0) (or just plain STRING) is the first item, the "H". STRING(1) is the "E", and so on. In addition to the original string we have a two-byte pointer residing on the stack, acting as a "window" to the message string but also posessed of some particular proporties of its own. It is the things that you can do with a pointer, as distinct to what you can do with the data it points to, that makes life interesting (and sometimes complicated).

Section 7.03.04 of the Language Reference Manual also covers this topic in detail.

9.06.03 POINTERS (continued)

THE USE OF POINTERS

It's difficult to give a short, concise example of the use of pointers in PL/9. Either the example is trivial or the result unreadable. Over-use of pointers, indeed, can result in a program becoming very confusing to the eye. Pointers should be used where they confer a definite advantage in code size or speed and otherwise omitted. There's no need to get worried about not using a particular language feature; as you become more experienced the applications will become obvious.

It might help at this point to draw a distinction between a pointer to a variable, frequently generated as a parameter of a procedure, and a pointer declared as "BYTE (or INTEGER or REAL) .POINTER". In the former case, we are computing an address, that of the variable that follows the dot. A string such as "HELLO" implies such a computed address. Once the address has been generated, it can be passed to a procedure or assigned to a variable. The latter case, however, creates a two-byte storage element whose job it is to hold the address of some object, such as the computed address just mentioned.

In the particular case of the procedure call PRINT "HELLO", the address of the quoted string is pushed onto the stack prior to the subroutine call to PRINT, thereby creating the necessary two-byte storage element. This storage has a name inside PRINT, namely .STRING. At the point the PRINT procedure is entered, .STRING contains the address of the 'H' in HELLO, but there is nothing immutable about this address and no reason why the programmer should not alter it. Once PRINT has done its job the storage used by .STRING will be recovered and its contents lost.

As an extension to the automatic pointers generated in this way, PL/9 allows you to declare your own pointers. These start life uninitialised, just like any other variables, but unlike simple variables have two values associated with them. The first is the address they contain and the second the value at that address, which might be a BYTE, INTEGER or REAL value. The pointer might be regarded as a "window" to some data. It is important to realise that the type (BYTE, INTEGER or REAL) of the pointer decides how the compiler will interpret what it sees through the window.

To summarise, then, there is no distinction between a pointer declared as a parameter passed to a procedure and one declared in a global or local declaration, except that the former is initialised by the procedure call itself and the latter must be given a value explicitly.

The following program is as much an example of where not to use pointers as where to use them. It's significant that only one pointer is declared in the main program. Because of the way the program data is structured, however, this one pointer is essential to the correct operation of the program.

The requirement is for a single buffer (data table) to hold the names and bank balances of a number of hypothetical customers. In a practical example, further information such as dates of birth, addresses and so forth might be included, but this only adds complication here. Rather than keep separate tables for each of these items it has been decided to hold all the information in one table, with the name of each customer followed by his or her bank balance. Many Pascal and C compilers have the ability to handle this kind of data as records or structures, but PL/9 programmers have to do without such luxuries. The key point is that the data table is a mixture of BYTE strings and REAL variables.

9.06.03 POINTERS (continued)

```
0001 /* POINTER EXAMPLE */
0002
0003 GLOBAL
0004    INTEGER NAMES(10):   /* Pointers to records */
0005    BYTE BUFFER(200);   /* A good-sized buffer */
0006
0007 INCLUDE IOSUBS;
0008 INCLUDE STRSUBS;
0009 INCLUDE REALCON;
0010
0011 PROCEDURE GET_INTEGER(BYTE .PTR): INTEGER N;
0012    N=0;
0013    WHILE PTR>='0 .AND PTR<='9
0014    BEGIN
0015       N=N*10+PTR-'0;
0016       .PTR=.PTR+1;
0017    END;
0018 ENDPROC N;
0019
0020 PROCEDURE POINTER_DEMO:
0021    BYTE SIZE, COUNT:
0022    INTEGER BPTR, NAME:
0023    REAL .AMOUNT:   /* This is the what the fuss is about */
0024    BYTE BUF(81);
0025
0026    BPTR=.BUFFER;
0027    PRINT "\N\NHow many customers? ";
0028    SIZE=GET_INTEGER(INPUT(.BUF,5));
0029
0030    COUNT=0;
0031    REPEAT
0032       PRINT "\NCustomer ";
0033       PRINTINT COUNT+1;
0034       PRINT "'s Name? ";
0035       NAMES(COUNT)=STRCOPY(BPTR,INPUT(.BUF,80));
0036       BPTR=BPTR+STRLEN(BPTR)+5;
0037       COUNT=COUNT+1;
0038    UNTIL COUNT=SIZE;
0039
0040    CRLF;
0041    COUNT=0;
0042    REPEAT
0043       NAME=NAMES(COUNT);
0044       CRLF;
0045       PRINT NAME;
0046       PRINT "'s balance? ";
0047       .AMOUNT=NAME+STRLEN(NAME)+1;
0048       AMOUNT=ASCBIN(INPUT(.BUF,10));
0049       COUNT=COUNT+1;
0050    UNTIL COUNT=SIZE;
0051
```

9.06.03 POINTERS (continued)

```
0052    PRINT "\N\NBalance Summary:\N==============\N";
0053    COUNT=0;
0054    REPEAT
0055       NAME=NAMES(COUNT);
0056       CRLF;
0057       STRCAT(STRCOPY(.BUF,NAME),"                    ");
0058       BUF(20)=0;
0059       PRINT .BUF;
0060       .AMOUNT=NAME+STRLEN(NAME)+1;
0061       PRINT ASCII(AMOUNT,.BUF);
0062       COUNT=COUNT+1;
0063    UNTIL COUNT=SIZE;
```

This program illustrates a number of useful techniques apart from the use of pointers. Starting at the top:

Line 4 declares an array of integers, to contain the addresses of the start of each of the name and bank balance records. (Note that no reference has yet been made to pointers.) Since this is only an example, only ten records have been allowed for. Line 5 declares the buffer that will hold all the records.

Next we have some included procedures, in order to get the functions that will be used by the program.

The procedure GET_INTEGER scans a buffer whose address is given and converts the number in the buffer into an INTEGER, stopping when the first non-decimal digit is found. Note the use of .PTR to scan along the buffer.

The main program, starting at line 20, first declares some working variables, among which is the REAL pointer .AMOUNT, whose job it will be to point to REAL numbers in the data table. The variables BPTR and NAME also hold addresses; they, however, interpret these addresses as INTEGER data.

The program gets under way in line 26 by setting BPTR to the start of the buffer. BPTR marks the end of the part of the buffer currently occupied by data. It could have been declared as BYTE .BPTR, making it a pointer, but since in this program it never has to act as a window to data there's really no need to treat it specially.

Lines 27-28 establish how many records are to be kept. Line 28 is an example of a procedure call within a call; INPUT returns a value of a type appropriate to use as a parameter to GET_INTEGER. (If you have difficulty in working out what such a construct is doing, work from the inside out.)

Lines 30-38 get the customer names. Line 35 is of note, being another nested procedure call. STRCOPY requires two parameters; the first is BPTR, which establishes where the data should go, while the second is the buffer BUF whose address is returned by INPUT after it has been filled with characters. STRCOPY itself returns its first parameter, the address to which the input data has just been copied; this address is placed in the integer array. Line 36 updates BPTR to account for the name entered (not forgetting the terminating null) and four bytes for the as-yet unknown bank balance. If this isn't clear, try running the program and draw a diagram of the memory as you go.

9.06.03 POINTERS (continued)

Lines 40-50 get the bank balance for each of the customers. Now we get to the important bit, being lines 47-48. NAME has already been set (line 43) to contain the address of the customer being dealt with. In line 47 the address of the four bytes reserved for the bank balance is computed and placed in .AMOUNT. Instead of being just four BYTES of memory it is now a REAL number, visible through a window set up for the purpose. Line 48 gets a number and stores it, via the window, into the data table. We have achieved the feat of storing a REAL number in a BYTE vector without PL/9 even noticing!

The rest of the program prints a summary of the stored data, just to prove it's all there. Line 57 copies a name into BUF and tacks 20 spaces on the end; line 58 truncates the result to 20 characters in all, to preserve column alignment. The code in lines 57-59 could have been reduced to a single expression with the help of the LEFT routine in the BASIC string function library; you are left to figure out how. Once again, the .AMOUNT pointer is called to service, in lines 60-61.

## 9.06.03 POINTERS (continued)

### MORE OF THE SAME ...

You should by now have some idea of the applications of pointers, and in particular how to use a vector of INTEGERs in conjunction with pointers. To provide a few more hints, suppose we were to use PL/9 to write a PL/9 compiler. In fact, PL/9 is written in assembly language, for speed and compactness. Most languages of this type are written in themselves these days, but none is efficient enough to do PL/9 in 16k bytes! One of the ways to save memory is to look closely at the way the symbol table is organized. There are six different symbol types in PL/9, viz. constants (and ATs, which share the same table), local variables, global variables, data statements, procedures and labels. Each of these types needs tables for the name strings, the values of the variables and their sizes (BYTE, INTEGER or REAL). Six symbol types times three items make 18 tables in all.

There are a number of ways of maintaining tables, depending whether you are after speed or compactness. (You obviously want both, but life was never as easy as that.) The textbooks recommend you to use hash-coded addressing, for speed, but this method requires fixed-size tables. How big should each table be? If you allow for 400 global variables and 200 constants, someone is bound to want them the other way round; you've then got to patch your compiler to accommodate the change.

The most memory-efficient method is to make each table no bigger than its contents. Let's look at part of our symbol table during compilation, with the addresses of the start of each table:

```
482C    MESSAGE_1                       Data names
        MESSAGE_2
        POWERS_OF_10
        LOG_COEFFICIENTS
485E    8016                            Data addresses
        803F
        814D
        81B7
4866    0           (BYTE)              Data sizes
        0           (BYTE)
        1           (INTEGER)
        2           (REAL)
4A6A    INPUT                           Procedure names
        CRLF
        PRINT
4A7B    937C                            Procedure addresses
        9445
        9458
4A81    1           (INTEGER)           Procedure sizes
        0           (BYTE)
        0           (BYTE)
```

There's no unused space anywhere in this arrangement; each table starts immediately after the previous. If another DATA variable is added to the above table, its name goes in at 485E and all the tables from that point up are moved up (in memory) a distance that depends on the length of the name. Likewise, the address of the DATA goes in at the end of its table and the size at the end of its. Each time new symbols are added, the higher tables are shifted up and as compilation proceeds the symbol table steadily grows.

9.06.03 POINTERS (continued)

How can this structure be coded in PL/9? Since none of the table sizes are known at the start, it's no good declaring 18 tables as global variables. Instead, we'll have to allocate a suitably large area of memory for the largest the combined table will ever grow, as a single vector:

```
GLOBAL BYTE SYMBOL_TABLE(3000);
```

Next we have to have some means of establishing the start of each table at any time. What we need is a BYTE vector that starts at the base of the data names, an INTEGER vector located at the base of the data addresses, and so on. This is where pointers come in. In the example PRINT("HELLO"), the unnamed message string "HELLO" suddenly takes on an identity as a result of being passed to a procedure. In the same way, we can allocate names to particular sections of our symbol table as and when we choose.

To do this, we have to declare pointers for each of the 18 tables needed. Using for simplicity just the six above, the global declaration becomes

```
GLOBAL
   BYTE    .DATA_NAMES:
   INTEGER .DATA_ADDRESSES:
   BYTE    .DATA_SIZES:
   BYTE    .PROCEDURE_NAMES:
   INTEGER .PROCEDURE_ADDRESSES:
   BYTE    .PROCEDURE_SIZES:
   BYTE    SYMBOL_TABLE(3000);
```

Each of the variables with a dot in front of the name is a pointer, and holds an address, whether it is declared as a BYTE or an INTEGER. No values are associated with the pointers yet, so we must first set the pointers to some initial value. Because the tables are all initially empty, all of the pointers can be set to the same value, namely the address of the start of the table itself:

```
.DATA_NAMES=.SYMBOL_TABLE;
.DATA_ADDRESSES=.SYMBOL_TABLE;
.DATA_SIZES=.SYMBOL_TABLE;
.PROCEDURE_NAMES=.SYMBOL_TABLE;
.PROCEDURE_ADDRESSES=.SYMBOL_TABLE;
.PROCEDURE_SIZES=.SYMBOL_TABLE;
```

The dots in front of the pointers are saying "set the pointer to the address given by..." rather than "set the data I'm pointing at to...". The dot in front of SYMBOL_TABLE does not make it too a pointer, it just means "the address of".

Now we have a mixture of BYTE and INTEGER pointers, all pointing to the same place, namely the start of the symbol table. What happens when the first data symbol comes along? As an example, suppose the program counter is at $9A82 when the following line is encountered:

```
BYTE MESSAGE "Hello World\n";
```

We have three pieces of information here for inclusion in the symbol table. Firstly, the data size is BYTE. Secondly, the address of the data is 9A82. Thirdly, its name consists of 8 characters, viz. the string MESSAGE and a terminating null.

9.06.03 POINTERS (continued)

Let's first put the symbol name into the table. We have to make a space eight characters in size for the name, which means all the other tables have to be moved. Moving the data in these tables is half of the operation, and doesn't require any comment from me; the other half is moving the pointers, as follows:

```
.DATA_ADDRESSES=.DATA_ADDRESSES+8;
.DATA_SIZES=.DATA_SIZES+8;
.PROCEDURE_NAMES=.PROCEDURE_NAMES+8;
.PROCEDURE_ADDRESSES=.PROCEDURE_ADDRESSES+8;
.PROCEDURE_SIZES=.PROCEDURE_SIZES+8;
```

Note that once again these operations do not relate to the data in the tables, only to the values of the pointers themselves. Once this operation is done, the characters of the symbol name can be copied into the space created in the symbol table. In the same way, the address of the data and its size can be placed in their respective tables once room has been made for them. Lastly it is a good idea to keep a record of how many symbols of the type have been encountered; we can use variables such as DATA_COUNT and PROCEDURE_COUNT for this.

Once a symbol table has been created, it can be used. When the compiler is scanning a line and comes across the name MESSAGE, it searches its symbol table for the name. A count is kept of how many names it has to skip before MESSAGE is found; this is the index of the symbol. A procedure that does this might be as follows:

```
PROCEDURE SEARCH(BYTE .TABLE,.NAME,MAX);
   INDEX=0;
   WHILE INDEX<MAX
   BEGIN
      IF STRCMP(.TABLE,.NAME)=0 THEN RETURN;
      .TABLE=.TABLE+STRLEN(.TABLE);
      INDEX=INDEX+1;
   END;
   INDEX=-1;
ENDPROC;
```

This procedure requires three parameters, viz. a pointer to the name in question, another pointer to the table to be searched, and the number of entries in the table. It uses two functions from the string library, STRCMP, which compares two strings, and STRLEN, which measures a string. Every time a name is found that is not the one wanted, the pointer TABLE is moved to the start of the next name and the index is bumped. If no name is found that matches, the special value -1 is returned, otherwise the routine returns the index.

Now we have to construct a mechanism for searching up to six different name tables, until the symbol name is found:

```
PROCEDURE LOOK_FOR_DATA_NAME(BYTE .NAME);
   SEARCH(.NAME,.DATA_NAMES,DATA_COUNT);
   IF INDEX=-1 THEN RETURN FALSE;
   ADDRESS=DATA_ADDRESSES(INDEX);
   SIZE=DATA_SIZES(INDEX);
ENDPROC TRUE;
```

9.06.03 POINTERS (continued)

```
PROCEDURE LOOK_FOR_PROCEDURE_NAME(BYTE .NAME);
   SEARCH(.NAME,.PROCEDURE_NAMES,PROCEDURE_COUNT);
   IF INDEX=-1 THEN RETURN FALSE;
   ADDRESS=PROCEDURE_ADDRESSES(INDEX);
   SIZE=PROCEDURE_SIZES(INDEX);
ENDPROC TRUE;
```

Note how the pointers can be indexed just like any vector. This is in fact the only way to implement tables of mixed data sizes; declare the whole lot as BYTE and make part of the table into something else using pointers of appropriate size.

Each of the the above procedures searches a particular table for the name; if it finds it then the corresponding address and size are extracted from their tables and the value TRUE is returned. A simple structure combines them:

```
REPEAT
   IF LOOK_FOR_DATA_NAME THEN BREAK;
   IF LOOK_FOR_PROCEDURE_NAME THEN BREAK;
   ERROR("UNDEFINED SYMBOL");
   BREAK;
FOREVER;
```

The program works its way through the various name tables until it finds the synbol in question, then breaks out of the dummy loop. If no table contains the name an error message is printed. Note that this is a somewhat unusual use of a REPEAT...FOREVER construct; it is however quite an efficient way of doing the job.

The examples and discussion above have tried to convey the concept of a variable whose address can be altered at will and that gives the flexibility to mix data types in the same area of storage. This is the essential property of pointers; only practice can take you on from here.

## 9.06.04 USING VECTORS

The next examples illustrate the use of vectors. The aim of the first program is to prompt for a line of input then print out the line backwards:

```
0001 INCLUDE IOSUBS;
0002
0003 PROCEDURE REVERSE:
0004    BYTE IN, OUT, INBUF(81), OUTBUF(81);
0005
0006    PRINT("TYPE ANY LINE - \N\N");
0007    INPUT(.INBUF,80);
0008
0009    IN=0;
0010    WHILE INBUF(IN)<>0 IN=IN+1;
0011
0012    OUT=0;
0013    WHILE IN>0
0014    BEGIN
0015       IN=IN-1;
0016       OUTBUF(OUT)=INBUF(IN);
0017       OUT=OUT+1;
0018     END;
0019    OUTBUF(OUT)=0;
0020
0021    CRLF;
0022    PRINT(.OUTBUF);
```

In line 4, the variable declaration includes two vectors INBUF and OUTBUF, each of which comprises 81 elements, numbered 0-80. They will be used to hold text strings of up to 80 characters; the extra 81st element is to hold the null that always terminates a string.

Line 6 prompts for input and gives two new-lines. Line 7 is a call to a library routine in IOSUBS.LIB whose job it is to get an edited line of input from the terminal and put it into the buffer whose address is supplied as the first parameter. The second parameter is the maximum length of the string to be accepted; INPUT will refuse to put anything in the buffer beyond this length. The terminating null is assumed not to be part of the string; the maximum string length in this case is therefore 80, not 81. You could of course tell INPUT to accept only 20 or 40 characters, but if you specified 120 then you would stand a risk of causing the program to bomb. PL/9 does not report errors of this kind, it being assumed that you know what you are doing when you specify a vector element beyond its declared range.

Line 7 then is an example of a "pointer to" a variable, indicated by the dot before INBUF. You are free to use either a dot or an ampersand (e.g. &INBUF), whichever you prefer; the dot is borrowed from PL/M and the ampersand from C. In this case, it is the address of INBUF that gets passed to INPUT, allowing the routine to modify the original variable. Passing by value would in this case make no sense; we are not interested in the original contents of INBUF and in any case only one byte would be passed, the contents of INBUF(0).

9.06.04 USING VECTORS (continued)

Lines 9-10 search the typed line for a null, i.e. find the end of the string. This is in fact the job done by STRLEN, one of the routines in STRSUBS.LIB. You could have INCLUDEd this library file then typed IN=STRLEN(.INBUF) in place of lines 9-10, at the cost of a lot of un-needed routines being included in your program. It's up to you to weigh the advantages or otherwise of including library files rather than putting the equivalent code into your programs.

Lines 12-19 copy the contents of INBUF into OUTBUF, starting with the last character if this is non-null. Note that if a null line has been typed, the WHILE statement will ensure that nothing gets copied. OUTBUF is finally itself terminated with a null, then lines 21-22 print the reversed line.

Although the example works perfectly satisfactorily, suppose it was necessary to frequently reverse text strings? You could of course duplicate the body of the routine every time it were needed; alternatively it could be made a function subroutine, as in the next example:

```
0001 INCLUDE IOSUBS;
0002
0003 PROCEDURE REVERSE (BYTE .INBUF, .OUTBUF):
0004    BYTE IN, OUT;
0005
0006    IN=0;
0007    WHILE INBUF(IN)<>0 IN=IN+1;
0008
0009    OUT=0;
0010    WHILE IN>0
0011    BEGIN
0012       IN=IN-1;
0013       OUTBUF(OUT)=INBUF(IN);
0014       OUT=OUT+1;
0015    END;
0016    OUTBUF(OUT)=0;
0017
0018 ENDPROC .OUTBUF;
0019
0020 PROCEDURE TEST: BYTE INBUF(81), OUTBUF(81);
0021
0022    PRINT("TYPE ANY LINE - \N\N");
0023    INPUT(.INBUF,80);
0024    CRLF;
0025    PRINT(REVERSE(.INBUF,.OUTBUF));
```

Lines 3-18 now comprise a function subroutine that requires as parameters "pointers to" (the address of) two buffers. Note that the length of the buffers is not specified since they have been declared elsewhere. The body of the procedure is the same as before, with the two index variables IN and OUT being declared in line 4. Line 18 contains a device for assigning a useful value to the function. There is no need actually to return a value from the function since OUTBUF already contains the effect of reversing whatever was supplied in INBUF. The reason for returning a pointer to OUTBUF is so that the construct in line 25 will work. PRINT requires as its operand the address of (i.e. a pointer to) whatever is to be printed; if REVERSE then returns the appropriate pointer then the call to PRINT can include that to REVERSE, which can in some cases save a good deal of code.

9.06.04 USING VECTORS (continued)

This technique can be taken one stage further; try deleting lines 23-25 and replacing them with

```
PRINT(REVERSE(INPUT(.INBUF,80),.OUTBUF));
```

and see the effect this has on the running of the program. Note that PRINT still only has the one parameter passed to it, but that in this case the call to INPUT is made as the first parameter of REVERSE. The reason why this works is that INPUT returns the address of the buffer it was supplied with, in the same way as REVERSE does.

The next example introduces a slightly different kind of vector:

```
0001 INCLUDE IOSUBS;
0002 INCLUDE REALCON;
0003 INCLUDE PRNUM;
0004
0005 PROCEDURE AVERAGE (REAL .LIST: BYTE ITEMS):
0006    REAL  SUM:  BYTE  N;
0007
0008    SUM=0;  N=0;
0009    REPEAT
0010       SUM=SUM+LIST(N);
0011       N=N+1;
0012    UNTIL N=ITEMS;
0013
0014 ENDPROC REAL SUM/ITEMS;
0015
0016
0017 REAL DATA_LIST
0018    1.543,
0019    102.73,
0020    72.911,
0021    20.005,
0022    6.9876,
0023    39.6821,
0024    16,
0025    83.5,
0026    55.02,
0027    9.9999;
0028
0029 PROCEDURE  TEST;
0030
0031    PRINT("\NTHE AVERAGE IS ");
0032    PRNUM(AVERAGE(.DATA_LIST,10));
```

Lines 1-3 call in the necessary INCLUDEd files (if you've cheated and skipped pages of this guide then you'll have to go back to find out how to create PRNUM.LIB). Lines 5-14 are a function called AVERAGE, that requires two arguments, viz. firstly the address of a list of REAL values and secondly the number of items in the list. The second argument is of type BYTE, implying that there will never be more than 127 items. The function sums the items in the list then returns the total divided by the number of items. Note the expression in Line 10; the term LIST(N) means "the Nth item in the vector LIST".

9.06.04 USING VECTORS (continued)

Lines 17-27 are an example of a data declaration, corresponding roughly to the DATA statement in BASIC. Data in PL/9 can, like much else, be either BYTE, INTEGER or REAL and must always be declared outside procedures. The items specified are converted into the appropriate type and placed into the program. This example is, as its name suggests, a data list of ten items, each one of which is a REAL constant. Data statements can be regarded as declarations of read-only variables, since PL/9 will refuse to compile a construct that attempts to assign a value to a data element.

The main program in the previous example starts at line 29. Line 32 is the meat of the example; it first calls AVERAGE, passing the address of the list of numbers and how many there are. Since AVERAGE returns a REAL value (line 14) this is of the right type to be handed to PRNUM for printing.

Now for a word about a special kind of "pointer to" data. In the examples so far, text strings have been printed using the statement PRINT("STRING"). The examples that follow illustrate more uses of text strings:

```
0001 PROCEDURE DEMO1;
0002    PRINT("TEST STRING\N");

0001 PROCEDURE DEMO2: INTEGER MESSAGE;
0002    MESSAGE="TEST STRING\N";
0003    PRINT(MESSAGE);

0001 BYTE MESSAGE "TEST STRING\N";
0002
0003 PROCEDURE DEMO3;
0004    PRINT(.MESSAGE);

0001 BYTE MESSAGE "TEST STRING\N";
0002
0003 PROCEDURE DEMO4: INTEGER MESSAGE;
0004    MESSAGE=.MESSAGE;
0005    PRINT(MESSAGE);
```

NOTE: In each of the last two examples above the message data string is declared outside of the procedures. This is a requirement of PL/9. All read only data, whether it is a data table, or a message string that will be 'pointed to' MUST be declared outside of all procedures and before the procedures they are used in.

All of the examples do the same job, that is print the message "TEST STRING" followed by a new line, but illustrate four different methods of achieving the same end. Which should be used depends entirely on the program being written and the preferences of the programmer, but as a general rule, if the same text string is to be used more than once in a program then you will save code by declaring the string as a data statement (third and fourth examples) then referring to it via a "pointer to" the data. Declaring an INTEGER variable then using it to hold a "pointer to" data as in the second and fourth examples, may be useful if different strings are to be used in the same way.

If you wish to include double quotes in a string, use the following construct:

```
PRINT("STATE ""MARRIED"" OR ""SINGLE""");
```

Which will print STATE "MARRIED" OR "SINGLE"

## 9.06.05 COMBINING FUNCTIONS, VECTORS, AND POINTERS TO DATA

The following example illustrates how functions, vectors of strings and "pointers to" data can be combined to produce routines that can, for example, generate one of various messages depending on the value of an index. The value of the index in turn is calculated by comparing input data against a table. This type of construction can greatly simplify I/O handling in many control programs as a series of simple and easily modified tables can replace what would otherwise amount to several pages of IF...THEN...ELSE or IF...CASE control arguments.

```
0001 INCLUDE IOSUBS;
0002
0003 CONSTANT ESCAPE=$1B,BELL=$07,_SPACE=$20,_RETURN=$0D;
0004
0005 BYTE INPUTT "ABCDEFG";
0006
0007 BYTE MESSAGE "HELLO                    ",    /* A */
0008              "HELLO AND GOOD MORNING",    /* B */
0009              "GOODNIGHT                ",    /* C */
0010              "GOODBYE                  ",    /* D */
0011              "WHAT TIME IS IT?         ",    /* E */
0012              "I'M TIRED                ",    /* F */
0013              "ISN'T THIS SILLY!        ";    /* G */
0014
0015 PROCEDURE MESSAGE_OUT(BYTE INDEX);
0016    PUTCHAR(_RETURN);
0017    PRINT(.MESSAGE(INDEX*23));
0018    PRINT("\N\N");
0019 ENDPROC;
0020
0021 PROCEDURE LOOK_UP:BYTE INDEX,CHAR;
0022
0023 START:
0024    PRINT("GIVE ME A CHARACTER BETWEEN 'A' and 'G'\N\N?");
0025    CHAR=GETCHAR;
0026    INDEX=0;
0027
0028    REPEAT
0029      IF INPUTT(INDEX)=CHAR
0030         THEN BEGIN
0031              MESSAGE_OUT(INDEX);
0032              BREAK;
0033              END;
0034      INDEX=INDEX+1;
0035    UNTIL INDEX=7;
0036
0037    IF CHAR <> ESCAPE .AND INDEX=7
0038       THEN BEGIN
0039            PRINT("\N\N\BINVALID INPUT!  TRY AGAIN.\N\N");
0040            GOTO START;
0041            END;
0042
0043    IF CHAR <> ESCAPE .AND INDEX < 7
0044       THEN BEGIN
0045            PRINT("\N\NHIT ESCAPE TO EXIT OR ");
0046            GOTO START;
0047            END;
```

## 9.06.05 COMBINING FUNCTIONS, VECTORS AND POINTERS TO DATA (continued)

Line 1 contains the now familiar inclusion of the PL/9 I/O subroutines library. Line 2 uses the keyword CONSTANT to define one keyboard code (ESCAPE) and three video display codes (BELL, SPACE, and RETURN). Note the use of the underscore character before 'SPACE' and 'RETURN'. This is essential because SPACE is also a function in the IOSUBS library and RETURN is a reserved word in PL/9.

If you are ever in doubt as to what procedure names and constant names are used in the INCLUDEd files simply make a dummy file with just the include statements and compile the output to your printer. All of the procedure names will be listed as will the CONSTANTs, GLOBALs and AT declarations. Local variable names are ignored.

Line 5 contains the data table which the incoming data is going to be compared against. Lines 7 through 13 contains another data table, this time it contains a series of strings rather than just a single byte as did the first data table. As the comments indicate the order of the message declarations corresponds to the order of the input data declaration. This is done so that the same index value can be used to access both tables. You will note something special about the way the message strings have been declared. EVERY message is exactly the same length. If the length of the message does not equal the LONGEST message used it is padded with spaces to make it EXACTLY the same length.

THIS PADDING IS ESSENTIAL if you wish to use message strings organized in data tables. As you will see in a moment we must calculate the starting position of subsequent strings based on the value of the index and the length of the string. If all the strings are not EXACTLY the same length the calculations will not produce the desired results.

Lines 15 through 19 form a function called 'MESSAGE_OUT'. The purpose of this function is to print out the message string related to the value INDEX. If you recall the discussion in the previous section (BYTE INDEX) on line 15 tells PL/9 to expect a value to be passed to this procedure from the calling program, in this case it will be the number of the message string to be printed, with zero being the first message, one the second and so on. Line 16 sends a carriage return (but no line feed) out to the video display in order to position the cursor at the beginning of the line. This has the effect of whatever follows being printed over the existing information on the video display. The heart of this function is on line 17 where we calculate the value of the message string pointer based on the value of INDEX. This calculation is '.MESSAGE(INDEX*23)' which simply stated means calculate the pointer to MESSAGE as being the value of the base MESSAGE offset by the value of INDEX multiplied by 23. What is the significance of '23' you may ask. If you count up the number of characters in the message strings you will find that each of them has 22 characters. PL/9 will automatically add a null to the end of each message string making the total 23. Thus we calculate the value of the first message pointer as MESSAGE(0*23) which gives us MESSAGE, the base address, as multiplying a number by zero will yield zero. The second message pointer is calculated as MESSAGE(1*23) or MESSAGE+23 BYTES or 23 BYTES up from the base address. The third message pointer is calculated as MESSAGE(2*23) or MESSAGE+46 BYTES and so on. Once the address of the string has been calculated PRINT can handle it in the usual manner.

Lines 21 to 35 form the body of the main program with lines 37 to 47 being a mechanism to provide an error message or provide a re-start prompt. Line 23 is a label for the subsequent GOTO statement and serves no other purpose. Line 24 is just a prompt to make the program more understandable when you try it out. Line 25 uses GETCHAR in IOSUBS to get a character from the keyboard and assigns the keyboard code to a variable called CHAR. Line 26 initializes our INDEX counter to 0, a very important step.

9.06.05 COMBINING FUNCTIONS, VECTORS AND POINTERS TO DATA (continued)

Lines 28 through 34 are a REPEAT loop that will increment the value of INDEX on each pass until INDEX reaches 7 at which time the loop will automatically terminate. As we have seven items in our table referenced as item zero to item six a value of seven means we have passed the limit of our tables.

Line 29 seeks to match the incoming keyboard character stored in CHAR with the INPUT table, offset (subscripted) by the current value of INDEX. On the first iteration of the loop CHAR will be compared with 'A', on the second 'B', the third 'C' and so on. When a match is found the THEN BEGIN block (lines 30 through 33) will be executed. Line 31 uses the function MESSAGE_OUT to print the message string associated with the current INDEX value. Line 32 tells the program to terminate the REPEAT loop immediately, i.e. begin execution at the line following line 35. This is done to preserve the value of INDEX should a match be found, as well as prevent the continued execution of what could, in reality, be a REPEAT UNTIL INDEX=247 loop. Obviously if we find a match in an early iteration there is no point in continuing to execute the loop because it will be impossible for another match to be found. BREAK provides a simple mechanism for immediate termination of the loop when we find a match.

Lines 37 through 41 handle the cases where a match of the incoming character is not found as indicated by an INDEX value of 7. Lines 43 through 47 handle the cases where a match was found. If the incoming character is equal to the ESCAPE key lines 37 through 47 will be bypassed and the program terminated.

This program will run properly within the PL/9 tracer. Just type it in. Type 'T<CR>' (TRACE). If no error messages are posted type 'G<CR>' (GO) to execute the program.

Several other examples of programs of this type will be presented in subsequent sections. We will be placing heavy emphasis on the use of data tables as an elegant, easy to modify and code efficient mechanism for writing I/O related control programs, primarily because this technique is seldom discussed in books on other languages.

A good understanding of FUNCTIONS, POINTERS TO DATA, and VECTORS will enable you write extremely complex I/O related programs that are not only easy to modify but will execute at incredible speeds.

## 9.06.06 TWO DIMENSIONAL ARRAYS

PL/9 is not a very high-level language, being somewhere below most C implementations as far as complexity is concerned. It makes up for any limitation in this respect by efficient coding and speed of compilation. Since it is quite close to assembly language in allowing you to do most anything you want to, without putting arbitrary barriers in your way, the data types it uses are themselves quite close to those used by assembler programs.

There is no direct mechanism for supporting two or more dimensional arrays. The main reason for this is that to provide such facilities would increase the size of the compiler by a large amount, reducing the size of file it could handle. Fortunately, there are other methods available of handling multi-dimensional data.

The only compound data type allowed by PL/9 is a vector, that is a number of like-sized (BYTE, INTEGER or REAL) items adjacent to each other in memory. This is similar to using a table in an assembler program, where the base address is known and any item can be reached by indexing from that point. PL/9 doesn't actually remember the size of the table and allows you to refer, for example, to the 25th element in a vector of only 21 elements. Whether this makes sense is up to you, just as it is in assembly language.

A two-dimensional array of M rows each having N elements can be represented as follows:

```
Row  0     Element  0 1 2 3 .........N-1
Row  1     Element  0 1 2 3 .........N-1
Row  2     Element  0 1 2 3 .........N-1
.
.
.
Row M-1    Element  0 1 2 3 .........N-1
```

In memory, the second row immediately follows the first, and so on. To access the 4th element of row 7 you must add 7*N+4 to the address of the start of the array. For PL/9 to be able to do this it would have to remember the value of N given in the variable declaration. This would not in itself be difficult but to handle the arrays themselves would add considerably to the complexity of the compiler, without improving the efficiency of the generated code. The calculation above might just as well be done by the application program, using a vector large enough to hold the entire array. To read the data at the 4th element of row 7 you would then use:

```
DATA=ARRAY(7*N+4);
```

and to write to the 9th element of row 15:

```
ARRAY(15*N+9)=DATA;
```

## 9.06.06 TWO DIMENSIONAL ARRAYS (continued)

A technique that can be used to simulate an array of any number of dimensions is to declare two procedures, one for writing to the array and the other for reading from it. Suppose we have a video display, organised as an array of 24 rows by 80 columns:

```
CONSTANT WIDTH=80;

AT $E800: BYTE SCREEN(1920);

PROCEDURE SCREEN_READ(BYTE ROW, COLUMN);
ENDPROC SCREEN(ROW*WIDTH+COLUMN);

PROCEDURE SCREEN_WRITE(BYTE ROW, COLUMN, DATA);
   SCREEN(ROW*WIDTH+COLUMN)=DATA;
ENDPROC;
```

To read the character at the 5th column of the 18th row, use

```
DATA = SCREEN_READ(18,5);
```

and to write an X to the 39th column of the 3rd row, use

```
SCREEN_WRITE(3,39,'X);
```

which is syntactically fairly close to having two-dimensional array capability built into the language.

## 9.07.00 BITWISE OPERATIONS

This section will attempt to cover these ones complement bit functions in considerable depth. As many users of PL/9 may only have had experience with BASIC these bit operators require a fair degree of explanation. Assembly language programmers on the other hand will probably find that only a cursory reading of this section is necessary.

Regardless of your level of programming experience in another language you are encouraged to read this section, particularly the latter part of it where we demonstrate some of the powerful constructions that PL/9 provides through the use of data tables.

The BIT functions AND, OR and EOR (XOR) work interchangeably with BYTEs and INTEGER values. They will not operate on REALs.

The PL/9 bit manipulation functions NOT, SWAP, and SHIFT will also only operate on BYTEs and INTEGERs. SWAP and SHIFT were discussed in section 9.05.14 and will not be discussed here. As NOT is closely related to AND, OR, and EOR we have decided to include it in this section

As far as PL/9 is concerned the numbers being evaluated in a bitwise expression are nothing more than a series of 1's and 0's. The sign and magnitude of the number has no bearing on the results. This means that these functions will find 99% of their application in bit-manipulation of I/O signals.

A few words of caution are in order when working with a mixture of BYTE and INTEGER values:

1. If you perform a bitwise operation of an INTEGER variable with a BYTE variable only the lower 8-bits of the INTEGER variable will be operated on.

2. If you perform a bitwise operation of an INTEGER variable with a BYTE CONSTANT the BYTE will be converted to an INTEGER with the top 8-bits set to zero before the bitwise operation is carried out over the entire 16-bits of the INTEGER variable.

3. If you operate on a BYTE variable with an INTEGER variable or CONSTANT only the lower 8-bits of the INTEGER will be used in the operation as would be expected.

Generally speaking there will seldom be any necessity to perform bitwise operations between a mixture of INTEGERS and BYTEs. These points are being raised primarily to warn you of what is likely to happen if an accidental mix of INTEGERs and BYTEs occurs in a bitwise operation.

The convention we use to define bit positions is the Motorola standard. This standard starts with the least significant bit of a 16-bit variable defined as bit 0 (b0) and the most significant bit defined at bit 15 (b15). This would obviously result in the most significant bit of an 8-bit variable being defined as bit 7 (b7).

DON'T EVER CONSIDER THE LEAST SIGNIFICANT BIT AS BIT 1 AND THE MOST SIGNIFICANT BIT AS BIT 16. PL/9 will produce completely erroneous results if you use these references as subscripts.

### 9.07.01 BITWISE AND

The simplest way to remember what this function does is:

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                                                               *
*  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.      *
*                                                               *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

Thus the bitwise AND is commonly used for two purposes:

1. To clear every bit in a binary input pattern (BYTE or INTEGER) except the one you are interested in to a zero. The BYTE or INTEGER is then tested for an equal or not equal to zero condition to reveal the state of the bit you are interested in. For example:

   ```
   CHAR=$A5;
   IF CHAR AND $01 <> 0
      THEN ...
   ```

   Here we AND a 'copy' of CHAR as part of an IF...THEN expression. The original value of CHAR is preserved so further evaluations may be performed. ANDing $A5 (1010 0101) with $01 (0000 0001) will result in $01 (0000 0001) as bit b0 in the operand (CHAR) was set. Thus the expression '<> 0' will be true in this instance and the THEN statement executed. If we were looking for a logical zero in bit b0 we would have made the expression 'IF CHAR AND $01 = 0".

2. To clear a particular bit in in a binary output pattern (BYTE or INTEGER) to a zero as part of an I/O signal response whilst preserving all other bits in the binary pattern. For example:

   ```
   OUTPORT=OUTPORT AND $FFFE;
   ```

   Where OUTPORT is a 16-bit output port defined by an 'AT' statement earlier in the program. First the output port is read, the binary pattern of the PORT is then ANDed with $FFFE (1111 1111 1111 1110) which will clear bit b0 but leave all the other bits intact. The resulting binary pattern is then written back out to the port.

## 9.07.02 BITWISE OR

The simplest way to remember what this function does is:

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                                                               *
*   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.       *
*                                                               *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

Bitwise OR is commonly used for two purposes:

1. To set every bit in a binary input pattern (BYTE or INTEGER) except the one you are interested in to a one. The BYTE or INTEGER is then tested for an equal or not equal to $FFFF condition to reveal the state of the bit you are interested in. For example:

   ```
   CHAR=$A5;
   IF CHAR OR $FE = $FF
      THEN ...
   ```

   Here we OR a 'copy' of CHAR as part of an IF...THEN expression. The original value of CHAR is preserved so further evaluations may be performed. ORing $A5 (1010 0101) with $FE (1111 1110) will result in $FF (1111 1111) as bit b0 in the operand (CHAR) was set. Thus the expression '= $FF' will be true in this instance and the THEN statement executed. If we were looking for a logical zero in bit b0 we would have made the expression 'IF CHAR OR $FE <> $FF'.

2. To set a particular bit in in a binary output pattern (BYTE or INTEGER) to a one as part of an I/O signal response whilst preserving all other bits in the binary pattern. For example:

   ```
   OUTPORT=OUTPORT OR $0001;
   ```

   First the output port is read, the binary pattern of the PORT is then ORd with $0001 (0000 0000 0000 0001) which will set bit b0 to logical one but leave all the other bits intact. The resulting binary pattern is then written back out to the port.

## 9.07.03 BITWISE EOR (XOR)

The simplest way to remember what this function does is:

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                                                               *
*      EORing two identical bits will yield '0' whilst          *
*      EORing two different bits will yield '1'.                *
*                                                               *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

The most common use of EOR is to compare two bit patterns. If the patterns are different the result will not be equal to zero, if the patterns are identical the result will be equal to zero. For example:

```
CHAR1=$A5F0;
CHAR2=$A5F0;

IF CHAR1 EOR CHAR2 = 0
   THEN...
```

In this example we set two INTEGERS equal to $A5F0 (1010 0101 1111 0000) and performed and EOR operation as part of an IF STATEMENT. The EOR of two identical binary patterns will yield zero and therefore the THEN statement will be executed.

This particular statement could just as easily been stated as:

```
IF CHAR1 = CHAR2
   THEN...
```

If this is so then EOR does not seem to serve much purpose I hear you say. This is true if this is the only comparison we are doing. EOR would be used to greater advantage in a construction such as the following example:

```
TESTCHAR=CHAR1 EOR CHAR2;
IF TESTCHAR <> 0
   THEN ...
```

where the THEN statement begins a series of AND operations on TESTCHAR to find out which bit(s) are ones and takes the appropriate action.

### 9.07.04 BITWISE NOT

This function simply INVERTS every bit in the operand. In other words every bit that was a one will become a zero and every bit that was a zero will become a one. For example:

```
CHAR=$5AF0;

CHAR=NOT(CHAR);
```

would result in CHAR starting off as $5AF0 (0101 1010 1111 0000) and ending up as $A50F (1010 0101 0000 1111).

NOT can also be used as part of a more complicated expression, viz:

```
IF NOT(CHAR1) <> NOT(CHAR2)
   THEN CHAR2=NOT(CHAR4);

IF NOT(CHAR1 AND CHAR2) <> 0
   THEN ...
```

It is important to note that NOT requires the use of parentheses ( ) to enclose the body of the BYTE, INTEGER or expression.

## 9.07.05 BIT ORIENTED TERMINAL I/O

As with most other features of PL/9 one of the best ways to learn how to use the bitwise functions described in the previous section would be to try them out. Since it would be impractical for many people to construct the hardware necessary to provide 16-bit I/O simulation we have developed a couple of small PL/9 programs that enable you to perform bit oriented I/O simulations through your keyboard and video display.

The following are part of the BITIO library which was described in section eight. We have included it in this section as it is a very good example of how table oriented bit operations make a program very compact and efficient.

```
0001 INCLUDE IOSUBS;
0002
0003 CONSTANT ZERO=$30,ONE=$31,_SPACE=$20;
0004
0005 INTEGER MASK $0001,$0002,$0004,$0008,$0010,$0020,$0040,$0080,
0006              $0100,$0200,$0400,$0800,$1000,$2000,$4000,$8000;
0007
0008 /*
0009    THIS PROCEDURE PROMPTS THE OPERATOR FOR A 16-BIT BINARY PATTERN OF
0010    0's  AND  1's AND RETURNS THE VALUE AS AN UNSIGNED INTEGER IN 'D'.
0011    INPUT OF ANY CHARACTER OTHER THAN A ONE OR A ZERO WILL BE IGNORED.
0013 */
0012
0014 PROCEDURE BITSIN:
0015    BYTE COUNT,INCHAR:
0016    INTEGER BITCHAR;
0017
0018    COUNT=16;
0019    BITCHAR=0;
0020    REPEAT
0021       REPEAT
0022         INCHAR=GETCHAR_NOECHO;
0023       UNTIL INCHAR = ZERO  .OR  INCHAR = ONE;
0024
0025       PUTCHAR(INCHAR); /* ECHO 0/1 */
0026          IF INCHAR=ONE THEN BITCHAR=BITCHAR OR MASK(COUNT-1);
0027       PUTCHAR(_SPACE);
0028       COUNT=COUNT-1;
0029   UNTIL COUNT=0;
0030 ENDPROC BITCHAR;
0031
0032 /*
0033   THIS PROCEDURE DOES THE EXACT OPPOSITE OF THE  ABOVE  PROCEDURE.  IT
0034   TAKES  THE  INTEGER  IT  IS PASSED ON THE STACK AND DUMPS THE BINARY
0035   PATTERN OUT TO THE SYSTEM CONSOLE AS A SERIES OF ONES AND ZEROS.
0036 */
0037 PROCEDURE BITSOUT(INTEGER BITCHAR):
0038   BYTE COUNT;
0039
0040   COUNT=16;
0041   REPEAT
0042      IF BITCHAR AND MASK(COUNT-1) = 0
0043         THEN PUTCHAR(ZERO);
0044         ELSE PUTCHAR(ONE);
0045      PUTCHAR(_SPACE);
0046      COUNT=COUNT-1;
0047    UNTIL COUNT=0;
0048 ENDPROC;
```

## 9.07.05 BIT ORIENTED TERMINAL I/O (continued)

The library module just created has two very useful routines that are structured as FUNCTIONS. The first is BITSIN, which, as its name suggests, gets a series of bits from the keyboard, the second is BITSOUT, which, as its name suggests, sends a bit pattern out to the video display. Both routines are structured to provide 16-bit I/O facilities.

If you desire to convert these routines to work with 8-bit data you may do so by restructuring the data table called MASK to be BYTEs rather than INTEGERs and changing the count values in lines 18 and 40 to 8 instead of 16; it's as simple as that.

### BITSIN

Is a procedure that returns a 16-bit variable called BITCHAR. This 16-bit variable is returned in the 6809's 'D' accumulator by the statement ENDPROC BITCHAR. This structure enables you to treat BITCHAR as though it were a variable in the program that wishes to use it. BITSIN starts off by initializing a variable called COUNT to 16 in line 18, and BITCHAR to $0000 in line 19. A repeat loop is then entered at line 20. This repeat loop terminates on line 29 when COUNT reaches 0. Just after we enter the first REPEAT loop we immediately enter another one on line 21. This REPEAT loop terminates on line 23 when a variable called INCHAR is equal to the constant 'ZERO' which was previously declared as being equal to ASCII 0 ($30) or equal to another constant ONE, ASCII 1 ($31). Within this inner REPEAT loop we use the GETCHAR_NOECHO routine in the IOSUBS library to get a character from the keyboard in line 22 and assign it to INCHAR. If you typed in a '1' or a '0' you will escape the loop at line 23. If any other key is pressed the program will return to line 21 and get another key. This process will continue indefinately until you hit a one or a zero.

At line 25 we use PUTCHAR to echo INCHAR back to the video display so you see the 1 or the 0 that you typed. Line 26 is the heart of this routine. At lines 5 and 6 we declared a read-only data table called MASK which comprises 16 INTEGERS. The elements of this data table are subscripted (0) through (15), with 0 being the first item declared. In line 26 we test to see if INCHAR is a one. If it is we proceed to perform the following operation with BITCHAR:

```
BITCHAR=BITCHAR OR MASK(COUNT-1)
```

it will make more sense if we first see what 'MASK(COUNT-1)' is doing. On the first iteration of the outer loop the value of COUNT will be 16. COUNT-1 results in as subscript of 15 being applied to MASK, i.e 'MASK(15)'. Therefore we are accessing item 15 of the data table called MASK which is $8000. Now we can simplify the original expression to:

```
BITCHAR=BITCHAR OR $8000
```

BITCHAR starts off as $0000 (line 19) therefore this operation will OR $0000 with $8000. $8000 is better represented by the bit pattern '1000 0000 0000 0000' at this point. The bitwise OR will result in the top bit of BITCHAR being set to a logical '1', which represents the one you typed in. If you typed in a zero the bitwise OR would not have been performed and therefore this bit would have been left as a zero.

## 9.07.05 BIT ORIENTED TERMINAL I/O (continued)

Line 27 sends a space out to the video display to improve the appearance of the input. Line 28 decrements the count value by one. Line 29 tests to see if the count is zero yet and if not returns the program to line 20 which starts the process all over again. This will keep repeating until you have input a series of 16 ones or zeros.

On each successive pass BITCHAR will be OR'd with the next lower element of the data table 'MASK' if the keyboard character was a one, thus building up a 16-bit INTEGER representation of what you are typing in the variable BITCHAR. When the program is completed, signified by COUNT reaching zero, BITCHAR will be returned to whoever called BITSIN with BITCHAR in the 'D' accumulator for use in either an evaluation expression or a data assignment expression.

### BITSOUT

Operates in a manner virtually the exact opposite to BITSIN. BITSOUT is passed a 16-bit variable on the stack from the calling program and then proceeds to output its bit pattern to the video display as as series of ones and zeros.

Line 40 initializes COUNT to 16. A REPEAT loop is then entered at line 41. The repeat loop will terminate at line 47 when COUNT reaches zero. Line 42 in this procedure is similar to line 26 in BITSIN. Again we are accessing the data table by subscripting MASK with COUNT-1. On the first pass we will again access the last element in the data table, $8000. Thus on the first pass line 42 will be:

```
IF BITCHAR AND $8000 = 0
```

this operation ANDs a copy of BITCHAR with $8000 (1000 0000 0000 0000) thus if the top bit is set(1) the result will be non-zero, if the top bit is clear (0) the result will be zero. If the result is zero line 43 will be executed otherwise line 44 will be executed. Line 43 sends an ASCII 0 ($30) to the video display and line 44 sends an ASCII 1 ($31) to the video display. Line 45 then sends a space, again to improve the screen presentation. COUNT is then decremented and line 47 executed. If COUNT is not yet zero the program will return to line 41 and proceed to execute the loop again. Thus on each iteration of the loop the next lower element of the data table will be accessed in order to acertain the status of a particular bit in the incoming bit pattern represented by BITCHAR. As the status is acertained the appropriate ASCII code will be sent to the video display.

These two routines should prove to be very useful when developing bit oriented I/O routines as they provide a simple mechanism to input bit patterns and display the bit patterns produced by your PL/9 procedures during program testing and debugging.

## 9.07.06 . AND, OR and EOR DEMONSTRATION PROGRAM

Once you have got the BITIO library installed on your working disk you might want to try the following 'tutorial' program which graphically illustrates what happens when bit patterns are ANDed, ORd or EORd.

```
0001 INCLUDE IOSUBS;
0002 INCLUDE BITIO;
0003
0004 CONSTANT BELL=$07,ESCAPE=$1B;
0005
0006 PROCEDURE DELAY:REAL COUNT;
0007    COUNT=2000;
0008    REPEAT COUNT=COUNT-1; UNTIL COUNT=0;
0009 ENDPROC;
0010
```

NOTE: The way we have entered the delay count decrementing routine on line 8. The reason it was not 'nested' in the usual manner is because this has the effect of slowing down a delay routine when inside the PL/9 tracer. If you can get the entire delay count decrementing loop on one line the delay routine will run at almost exactly the same speed whether the program is executing in the PL/9 tracer or executing independently of the PL/9 tracer. This can be a useful 'trick' to know, make a mental note of it.

```
0011 PROCEDURE AND_OR_EOR_DEMO:INTEGER INCHAR1, INCHAR2, OUTCHAR:BYTE OPERATOR;
0012 START:
0013    PRINT("\N\N\NGive me a 16-bit sequence of 1's and 0's\N\N");
0014    INCHAR1=BITSIN;
0015    DELAY;
0016
0017    PRINT("\N\NGive me another 16-bit sequence of 1's and 0's\N\N");
0018    INCHAR2=BITSIN;
0019    DELAY;
0020
0021    PRINT("\N\N(1) AND, (2) OR, (3) EOR:- 1, 2, or 3? ");
0022
0023    REPEAT
0024       OPERATOR=GETCHAR_NOECHO;
0025
0026       IF OPERATOR
0027          CASE '1 THEN BEGIN
0028                       OUTCHAR=INCHAR1 AND INCHAR2;
0029                       BREAK;
0030                       END;
0031
0032          CASE '2 THEN BEGIN
0033                       OUTCHAR=INCHAR1 OR INCHAR2;
0034                       BREAK;
0035                       END;
0036
0037          CASE '3 THEN BEGIN
0038                       OUTCHAR=INCHAR1 EOR INCHAR2;
0039                       BREAK;
0040                       END;
0041    FOREVER;
```

## 9.07.06 AND, OR and EOR DEMONSTRATION PROGRAM (continued)

```
0042
0043    PRINT("\N\NThis number     ");
0044    BITSOUT(INCHAR1);
0045    CRLF;
0046
0047    IF OPERATOR
0048       CASE '1 THEN PRINT("AND'd with      ");
0049       CASE '2 THEN PRINT("OR'd with       ");
0050       CASE '3 THEN PRINT("EOR'd with      ");
0051
0052    BITSOUT(INCHAR2);
0053    PRINT("\NResults in:     ");
0054    BITSOUT(OUTCHAR);
0055    PRINT("\N\N\BHIT ESCAPE TO TERMINATE, ANY OTHER KEY TO CONTINUE! ");
0056    IF GETCHAR_NOECHO <> ESCAPE
0057       THEN GOTO START;
```

This program may be run inside the PL/9 tracer in the usual manner or compiled to disk as a command (A:0=ANDORDEM.CMD).

The program is 'menu' driven and should be self explanatory.

## 9.07.07 OPERATING ON SPECIFIC BITS; AN INDIVIDUAL APPROACH

One of the most frequent tasks to be performed in bit oriented I/O work is the examination of, the setting of, or the clearing of, one or more bits to the exclusion of all others.

The bitwise AND and OR functions are designed to provide the basic tools for this type of work. The use of these bit functions may be alien to many programmers and may therefore take a bit of getting used to. In this and the following section we are going to introduce you to a few simple techniques that will enable you to 'hide' the bit operations in a few procedures. Once the bit operation functions are defined you can get on with the control job at hand and forget about bit operations for the remainder of the program!

For example supposing that you had a PIA initialized with bits b0, b1, b3, b6, and b7 as outputs and bits b2, b4, and b5 as inputs.

This is not a particularly good way to assign data bits but often you have no choice in how the customer wants them or how your predecessor decided to design the hardware. If you have a choice it makes life a little simpler if you keep inputs in one group and outputs in another group.

You can save yourself a lot of confusion in your main program, at the expense of some code efficiency, if you write a series of short procedures to handle each bit as an individual element. It is then possible to assign a name to the procedure that is closely related to the the function of the bit. In this example let us assign the functions of the 8 data bits mentioned earlier as follows:

| BIT | FUNCTION | DATA DIRECTION | SIGNAL DEFINITIONS | |
|---|---|---|---|---|
| b0 | CIRCULATING PUMP | (OUT) | 1=ON, | 0=OFF |
| b1 | FUEL PUMP | (OUT) | 1=ON, | 0=OFF |
| b2 | BOILER TEMPERATURE | (IN ) | 1=HIGH, | 0=NORMAL |
| b3 | BOILER ON | (OUT) | 1=ON, | 0=OFF |
| b4 | INPUT PRESSURE (FUEL) | (IN ) | 1=NORMAL, | 0=LOW |
| b5 | OUTPUT PRESSURE (WATER) | (IN ) | 1=HIGH | 0=NORMAL |
| b6 | MAIN FUEL VALVE | (OUT) | 1=CLOSE, | 0=OPEN |
| b7 | EMERGENCY SHUTDOWN | (OUT) | 1=RUN, | 0=ESD |

One of the more important aspects of the table above is the SIGNAL DEFINITIONS column. This is very crucial as not every system one works with has the logic oriented in the same manner. If you start programming a control job on the ASSUMPTION that '1' means 'ON' and '0' means 'OFF' you might be in for a nasty surprise later in the job.

It is vitally important to obtain this type of information BEFORE you start the main body of any control program as the entire logic of the bitwise AND and OR constructions depend on it. IF IN DOUBT ASK SOMEONE! If it does not form part of your customers specification GET IT IN WRITING!. A wrong assumption in the early stages of program development can cost you dearly in lost time later, as well as a great deal of embarrassment with your customer as you 'really ought to have known better!',

Armed with the knowledge of what each bit on the I/O port is supposed to do you should declare a set of CONSTANTS for the conditions you will be looking for in subsequent sections of your program. This is VERY useful when '1' does not always mean 'ON' or 'NORMAL', or '0' does not always mean 'OFF' or 'ABNORMAL'.

The following example actually works. In order to simulate a PIA at a 'hard' address we have declared DATAPORT to be in the FLEX line buffer.

## 9.07.07 OPERATING ON SPECIFIC BITS; AN INDIVIDUAL APPROACH (continued)

```
0001 AT$C090:BYTE DATAPORT;
0002
0003 CONSTANT CIRCON=1, CIRCOFF=0,  /* CIRCULATION */
0004          FPON=1,   FPOFF=0,    /* FUEL PUMP */
0005          BTOK=0,   BTHI=1,     /* BOILER TEMP */
0006          BLRON=1,  BLROFF=0,   /* BOILER */
0007          IPOK=1,   IPLO=0,     /* IN PRESS */
0008          OPOK=0,   OPHI=1,     /* OUT PRESS */
0009          FVOPN=0,  FVCLS=1,    /* FUEL VALVE */
0010          RUN=1,    ESD=0;      /* ESD */
0011
0012
```

One very important reason for defining the status of each bit via a CONSTANT declaration is that you will not be embedding comparisons with ones and zeros in the main body of your code. If you diligently USE the constant definitions throughout your program a change in signal requirements will only necessitate a quick change to the CONSTANT table rather than searching the length and breadth of your program for hidden references.

The very nature of control programming demands that the programmer prepare himself, AND HIS PROGRAMS, for the inevitable...changes or modifications due to oversights or errors in the original specification or design.

Once armed with a table of constants defining the I/O signals AND the knowledge of which bit represents what, you can then write a series of special purpose functions to handle each bit as a separate entity, viz:

```
0013 PROCEDURE CIRCULATION_PUMP(BYTE STATUS);
0014    IF STATUS  /* IMPLICIT <> 0 */
0015       THEN DATAPORT=DATAPORT OR $01;
0016       ELSE DATAPORT=DATAPORT AND $FE;
0017 ENDPROC;
0018
0019 PROCEDURE FUEL_PUMP(BYTE STATUS);
0020    IF STATUS  /* IMPLICIT <> 0 */
0021       THEN DATAPORT=DATAPORT OR $02;
0022       ELSE DATAPORT=DATAPORT AND $FD;
0023 ENDPROC;
0024
0025 PROCEDURE BOILER_TEMP:BYTE STATUS;
0026    IF DATAPORT AND $04 <> 0
0027       THEN STATUS=1;
0028       ELSE STATUS=0;
0029 ENDPROC STATUS;
0030
0031 PROCEDURE BOILER_POWER(BYTE STATUS);
0032    IF STATUS  /* IMPLICIT <> 0 */
0033       THEN DATAPORT=DATAPORT OR $08;
0034       ELSE DATAPORT=DATAPORT AND $F7;
0035 ENDPROC;
0036
0037 PROCEDURE INPUT_PRESSURE:BYTE STATUS;
0038    IF DATAPORT AND $10 <> 0
0039       THEN STATUS=1;
0040       ELSE STATUS=0;
0041 ENDPROC STATUS;
```

9.07.07 OPERATING ON SPECIFIC BITS; AN INDIVIDUAL APPROACH (continued)

```
0042
0043 PROCEDURE OUTPUT_PRESSURE:BYTE STATUS;
0044    IF DATAPORT AND $20 <> 0
0045       THEN STATUS=1;
0046       ELSE STATUS=0;
0047 ENDPROC STATUS;
0048
0049 PROCEDURE FUEL_VALVE(BYTE STATUS);
0050    IF STATUS  /* IMPLICIT <> 0 */
0051       THEN DATAPORT=DATAPORT OR $40;
0052       ELSE DATAPORT=DATAPORT AND $BF;
0053 ENDPROC;
0054
0055 PROCEDURE EMERGENCY_SHUTDOWN(BYTE STATUS);
0056    IF STATUS  /* IMPLICIT <> 0 */
0057       THEN DATAPORT=DATAPORT OR $80;
0058       ELSE DATAPORT=DATAPORT AND $7F;
0059 ENDPROC;
0060
0061
```

If you have read and understand section 9.06.01 you should have no difficulty in understanding what each of the preceeding and each of the following procedures is doing.

Now with a set of procedures to handle all of the bit oriented side of your I/O you can sit down to write the body of your program and forget about bitwise AND and bitwise OR! Viz:

```
0062 PROCEDURE CONTROL;
0063
0064    IF INPUT_PRESSURE = IPLO
0065       THEN FUEL_PUMP = FPON;
0066       ELSE FUEL_PUMP = FPOFF;
0067
0068    IF OUTPUT_PRESSURE = OPHI
0069       THEN CIRCULATION_PUMP = CIRCOFF;
0070       ELSE CIRCULATION_PUMP = CIRCON;
0071
0072    IF BOILER_TEMP = BTOK
0073       THEN BEGIN
0074            FUEL_VALVE = FVOPN;
0075            BOILER_POWER = BLRON;
0076            EMERGENCY_SHUTDOWN = RUN;
0077            END;
0078       ELSE BEGIN
0079            FUEL_VALVE = FVCLS;
0080            BOILER_POWER = BLROFF;
0081            CIRCULATION_PUMP = CIRCOFF;
0082            EMERGENCY_SHUTDOWN = ESD;
0083            END;
```

Note that we have not used the conventional set of brackets to enclose the data we are passing to the function procedure. This form improves the readability of the program substantially. See section 9.14.03 for further details on permitted variations in syntax.

## 9.07.07 OPERATING ON SPECIFIC BITS; AN INDIVIDUAL APPROACH (continued)

This program may not be a model of code efficiency, as each of the bit operation procedures requires about 30 bytes of memory, BUT it is VERY, VERY readable. If you are working with an application that leaves you with plenty of memory (EPROMs are cheap these days) and the application does not require blistering speed this type of structure will make your life as a control programmer a lot easier.

The following is an adaptation of the previous program sent to us by a PL/9 programmer. If he lets us know who he is we would be happy to include an acknowledgement here. This program uses a much more general purpose input routine (SENSE) and a much more general purpose output routine (SWITCH). As you can tell from the program it is every bit as readable as the preceeding example, but it is significantly more code efficient. The only two elements that require a bit of thought to figure out how they work are 'SENSE' and 'SWITCH'. If you understand what we have outlined in the preceeding section you should not have any trouble understanding them.

```
0001 AT $C000:BYTE DATAPORT;
0002
0003 /* DEFINE THE PIA BITS */
0004
0005 CONSTANT
0006    CIRC_PUMP           = $01,  /* (OUTPUT)  1 = ON, 0 = OFF */
0007    FUEL_PUMP           = $02,  /* (OUTPUT)  1 = ON, 0 = OFF */
0008    BOILER_TEMP_NORMAL  = $04,  /* (INPUT)   1 = HI, 0 = OK  */
0009    BOILER_POWER        = $08,  /* (OUTPUT)  1 = ON, 0 = OFF */
0010    IN_PRESS_LO         = $10,  /* (INPUT)   1 = OK, 0 = LOW */
0011    OUT_PRESS_HI        = $20,  /* (INPUT)   1 = HI, 0 = OK  */
0012    FUEL_VALVE          = $40,  /* (OUTPUT)  1 = ON, 0 = OFF */
0013    EMERG_SHUTDN        = $80,  /* (OUTPUT)  1 = ON, 0 = ESD */
0014
0015    OFF                 = $00,
0016    ON                  = $FF,
0017    OK                  = $FF,
0018
0019 /*
0020    THE STATUS OF THE BITS IN THE FOLLOWING BYTE DEFINES THE ACTIVE
0021    STATE OF THE 'IN' OR 'OUT' BIT:
0022
0023    '1' MEANS THE INPUT OR OUTPUT IS ACTIVE 'LOW'
0024    '0' MEANS THE INPUT OR OUTPUT IS ACTIVE 'HIGH'
0025 */
0026    INVBYTE             = $C4;  /* 1100 0100 */
0027
0028
0029 PROCEDURE SENSE(BYTE DEVICE);
0030    IF (DATAPORT EOR INVBYTE) AND DEVICE = 0
0031       THEN RETURN OFF;
0032 ENDPROC ON;  /* SAME AS 'ELSE RETURN ON' BUT GENERATES LESS CODE */
0033
0034
0035 PROCEDURE SWITCH(BYTE DEVICE, STATUS);
0036    DATAPORT = (DATAPORT AND (NOT(DEVICE)))
0037               OR
0038               (DEVICE AND (STATUS EOR INVBYTE));
0039 ENDPROC;
0040
0041
```

## 9.07.07 OPERATING ON SPECIFIC BITS; AN INDIVIDUAL APPROACH (continued)

```
0042 PROCEDURE CONTROL;
0043    IF SENSE (IN_PRESS_LO) = OK
0044       THEN SWITCH (FUEL_PUMP, ON);
0045       ELSE SWITCH (FUEL_PUMP, OFF);
0046
0047    IF SENSE (OUT_PRESS_HI) = OK
0048       THEN SWITCH (CIRC_PUMP, OFF);
0049       ELSE SWITCH (CIRC_PUMP, ON);
0050
0051    IF SENSE (BOILER_TEMP_NORMAL) = OK
0052       THEN BEGIN
0053            SWITCH (FUEL_VALVE, ON);
0054            SWITCH (BOILER_POWER, ON);
0055            SWITCH (EMERG_SHUTDN, OFF);
0056            END;
0057       ELSE BEGIN
0058            SWITCH (FUEL_VALVE, OFF);
0059            SWITCH (BOILER_POWER, OFF);
0060            SWITCH (EMERG_SHUTDN, ON);
0061            END;
```

## 9.07.08 OPERATING ON SPECIFIC BITS; MORE GENERAL APPROACHES

The preceeding section presented a technique that involves a 30 byte procedure for each of the bits that need to be operated on in an I/O port. This is fine when only a relatively small number of bits is involved in the system I/O. When the system becomes a fair bit larger this approach is still valid providing you have sufficient memory available.

This section will present an alternative solution that uses only two procedures and can perform the bit evaluation or assignment of any INTEGER or BYTE variable. These procedures are part of the BITIO library.

```
0001 AT $C090: INTEGER DATAPORTA;
0002 AT $C092: BYTE    DATAPORTB;
0003
0004 INTEGER BIT_TABLE $0001,$0002,$0004,$0008,$0010,$0020,$0040,$0080,
0005                   $0100,$0200,$0400,$0800,$1000,$2000,$4000,$8000;
0006
0007 PROCEDURE BITIN(INTEGER DATA:BYTE POSITION):BYTE STATUS;
0008    IF DATA AND BIT_TABLE(POSITION) /* IMPLICIT <> 0 */
0009       THEN STATUS=1;
0010       ELSE STATUS=0;
0011 ENDPROC STATUS;
0012
0013 PROCEDURE BITOUT(INTEGER DATA:BYTE POSITION, STATUS);
0014    IF STATUS /* IMPLICIT <> 0 */
0015       THEN DATA=DATA OR BIT_TABLE(POSITION);
0016       ELSE DATA=DATA AND NOT(BIT_TABLE(POSITION));
0017 ENDPROC INTEGER DATA;
0018
0019 PROCEDURE DEMO;
0020    DATAPORTA=$F00E;
0021    DATAPORTB=$FE;
0022
0023    IF BITIN(DATAPORTA,0) = 0
0024       THEN DATAPORTA=BITOUT(DATAPORTA,15,0);
0025
0026    IF BITIN(DATAPORTB,6) = 1
0027       THEN DATAPORTB=DATAPORTB,7,0);
```

The preceeding program outlines the details of two procedures which may be used for general purpose bit manipulation of BYTES and INTEGERS. The first procedure, BITIN is 41 bytes, the second BITOUT is 60 bytes. Both procedures execute very quickly as the majority of the code is divided between two conditional branches in each program.

## 9.07.08 OPERATING ON SPECIFIC BITS; MORE GENERAL APPROACHES (continued)

The last part of the above is a simple demonstration program that illustrates how BITIN and BITOUT are used.

The standard form of BITIN would look like:

```
IF BITIN(DATAPORTA,0)=0
         |         |  |
         |         |  +-- The value of the bit we are looking for.
         |         +----- The position of the bit we are looking at.
         +--------------- The name of the variable we are interested in.
```

BITIN can be used as part of a much more complicated expression if required, viz:

```
IF BITIN(DATAPORT1,3)=0 .AND BITIN(DATAPORT2,7)=1 .AND BITIN(DATAPORT2,3)=0
   THEN ...
```

The standard form of BITOUT would look like:

```
THEN DATAPORTB=BITOUT(DATAPORTB,7,0);
     |                |         | |
     |                |         | +-- The value we wish to assign to the bit.
     |                |         +---- The position of the bit of interest.
     |                +-------------- The name of the READ variable.
     +------------------------------- The name of the WRITE variable.
```

As the above implies the READ variable does not necessarily need to be the same as the WRITE variable. The WRITE variable could just as easily be another data port or variable.

Another technique is also available for output bit manipulation that operates directly on the port/variable of interest. The technique we are about to introduce differs in two respects from the BITOUT procedure just described.

a. The procedure will handle only 8-bit or 16-bit data. If you need to work with both sizes of data you will require two different procedures.

b. The procedure operates directly on the data port or variable. You cannot assign the result of the operation to another variable.

```
0013 PROCEDURE BIT_16_OUT(INTEGER .DATA:BYTE POSITION, STATUS);
0014    IF STATUS  /* IMPLICIT <> 0 */
0015       THEN DATA = DATA OR BIT_TABLE(POSITION);
0016       ELSE DATA = DATA AND NOT(BIT_TABLE(POSITION));
0017 ENDPROC;
0018
0019 PROCEDURE BIT_8_OUT(BYTE .DATA:BYTE POSITION, STATUS);
0020    IF STATUS  /* IMPLICIT <> 0 */
0021       THEN DATA = DATA OR BIT_TABLE(POSITION);
0022       ELSE DATA = DATA AND NOT(BIT_TABLE(POSITION));
0023 ENDPROC;
0024
```

9.07.08 OPERATING ON SPECIFIC BITS; MORE GENERAL APPROACHES (continued)

If you look closely you will hardly notice any difference between the two preceeding procedures other than lines 13 and 19. Since each of these two procedures is being passed a pointer (.DATA) which will contain the address of the variable the procedure must know the SIZE of the data it will be manipulating. Since pointers are always INTEGERS (see the Language Reference Manual) the reference to INTEGER on line 13 and to BYTE on line 19 tells PL/9 the size of the data pointed to. In the first instance the data is 16-bit (INTEGER) and in the second the data is 8-bit (BYTE).

Note also that these two procedures no longer return a variable as structured. They could return the final value of the data if desired by simply stating 'ENDPROC DATA;'. This may be useful in some circumstances.

The remainder of the program demonstrates the new structure required to use the two new BITOUT procedures.

```
0025 PROCEDURE DEMO:BYTE INDEXA, BITA, INDEXB, BITB;
0026
0027    DATAPORT_A = $FOOE;
0028    DATAPORT_B = $FE;
0029
0030    IF BITIN(DATAPORT_A,0) = 0
0031       THEN BIT_16_OUT(.DATAPORT_A,15) = 0;
0032
0033    IF BITIN(DATAPORT_B,0) = 0
0034       THEN BIT_8_OUT(.DATAPORT_B,7) = 0;
0035
0036
0037 /*   O R   S O M E T H I N G   M U C H   M O R E   E L A B O R A T E   */
0038
0039
0040    DATAPORT_A = $FOOE;
0041    DATAPORT_B = $FE;
0042
0043    INDEXA=0;
0044    BITA=1;
0045
0046    IF BITIN(DATAPORT_A,INDEXA) = 0
0047       THEN BIT_16_OUT(.DATAPORT_A,7) = BITA;
0048
0049
0050 /*              O R              */
0051
0052
0053    INDEXB=0;
0054    BITB=1;
0055
0056    IF BITIN(DATAPORT_B,0) = 0
0057       THEN BIT_8_OUT(.DATAPORT_B,INDEXB,BITB);
```

The main difference between this program and its predecessor is the lack of the "=" in the THEN statement. This is because the new BITOUT procedures operate directly on the data pointed to and do not require a data assignment operator.

## 9.07.08 OPERATING ON SPECIFIC BITS; MORE GENERAL APPROACHES (continued)

These new forms of BITOUT are structured as follows:

```
THEN BIT_8_OUT(.DATAPORTB,7,1);
               ||         | |
               ||         | +-- The value we wish to assign to the bit
               ||         +---- The position of the bit of interest.
               |+-------------- The name of the READ/WRITE variable.
               |
               +--------------- NOTE the '.' to pass the address NOT the value!
```

As mentioned previously this structure requires that YOU, the programmer, ensure that the correct procedure is used to operate on 8-bit and 16-bit data. Although this technique requires more code (there are now two routines instead of one) than the general purpose routine outlined in the preceeding section it will execute MUCH faster.

When the application is simple and is not likely to GROW the techniqes illustrated in this and the preceeding section form a simple method of avoiding the complexity of embedding bitwise operations into the main body of your program.

Where the control program is likely to remain small or is fairly straightforward the use of data tables (discussed in the following sections) should be avoided. Unless a data table oriented program is heavily documented by comments they tend to be very difficult to work with for the uninitiated but can, in the right circumstances, produce very compact and FAST programs.

See section 9.14.03 for permitted variations in syntax which may be used to improve the readability of programs when passing variables to function procedures.

## 9.07.09 I/O BIT MANIPULATION THROUGH DATA TABLES

In this and the following sections we are going to introduce and elaborate on working with bit intensive I/O through the use of data tables.

It might seem counter-productive to develop a technique that would, on the surface, appear to cloud the actual I/O operations taking place. This would be true except for the one cardinal rule of working with data tables. This rule is that the data tables must be thoroughly documented in the comments section of a program. Once the data involved in I/O work is placed in hex form in data tables it will take a clever programmer indeed to figure out what is going on six months after he wrote the program. Flow charts can also be very useful in defining the inner workings of a data table oriented program, but they can be difficult to integrate into the program comments section due to the limitations of the ASCII code set. It is far better to integrate two or three pages of comments into the source file of a program than to rely on a separate sheet of paper with a flow chart on it. Murphy's law dictates that this one vital piece of paper will not be found when it comes time to modify the program.

When should data tables be used and when should they not be used? Generally speaking when IF...THEN or IF...CASE statements begin to exceed 6 or 7 arguments is the time to think of structuring a program around data tables. Control arguments much in excess of 15 IF...THEN arguments can start to produce excessive amounts of code and slow the I/O transfer down in the process. Not only do data tables require less memory than the equivalent number of IF...THEN statements they will also execute exceedingly fast in comparison. The code that PL/9 produces in normal IF...THEN arguments is very efficient by most standards, the code PL/9 produces when working with data tables rivals that produced by an assembly language program!

A working example is in order here. Supposing that you have to write a sequence of 16-bit binary patterns out through an output port that will generate the patterns defined on the following page in a step-by-step sequence forever. Add to this that the customer wants a one second delay between patterns, AND to test a switch to see if the routine is to be stopped or continued, AND to test a second switch to determine if the routine is to be terminated.

If you started writing a program like this:

```
PORT=$0000;
DELAY;
TEST_SWITCH1; /* STOPS DISPLAY */
IF SWITCH2=1 THEN RETURN;

PORT=$0001;
DELAY;
TEST_SWITCH1;
IF SWITCH2=1 THEN RETURN;

PORT=$0002;
.
.
.
```

You would probably run out of patience very quickly, but you'd get the job done and it would work but it would not be particularly easy to modify the binary sequence or add other facilities which the customer might come back and ask you for in three months time (and this ALWAYS happens!).

## 9.07.09 I/O BIT MANIPULATION THROUGH DATA TABLES (continued)

The following represents the binary sequence desired by the customer:

| | | |
|---|---|---|
| 1. | 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | ($0000) |
| 2. | 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 | ($0001) |
| 3. | 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 | ($0002) |
| 4. | 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 | ($0004) |
| 5. | 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 | ($0008) |
| 6. | 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 | ($0010) |
| 7. | 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 | ($0020) |
| 8. | 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 | ($0040) |
| 9. | 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 | ($0080) |
| 10. | 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 | ($0100) |
| 11. | 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 | ($0200) |
| 12. | 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 | ($0400) |
| 13. | 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 | ($0800) |
| 14. | 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 | ($1000) |
| 15. | 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 | ($2000) |
| 16. | 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | ($4000) |
| 17. | 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | ($8000) |
| 18. | 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | ($0000) |
| 19. | 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 | ($8001) |
| 20. | 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 | ($4002) |
| 21. | 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0 | ($2004) |
| 22. | 0 0 0 1 0 0 0 0 0 0 0 0 1 0 0 0 | ($1008) |
| 23. | 0 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 | ($0810) |
| 24. | 0 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0 | ($0420) |
| 25. | 0 0 0 0 0 0 1 0 0 1 0 0 0 0 0 0 | ($0240) |
| 26. | 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 | ($0180) |
| 27. | 0 0 0 0 0 0 1 0 0 1 0 0 0 0 0 0 | ($0240) |
| 28. | 0 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0 | ($0420) |
| 29. | 0 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 | ($0810) |
| 30. | 0 0 0 1 0 0 0 0 0 0 0 0 1 0 0 0 | ($1008) |
| 31. | 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0 | ($2004) |
| 32. | 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 | ($4002) |
| 33. | 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 | ($8001) |
| 34. | 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | ($0000) |
| 35. | 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 | ($8001) |
| 36. | 1 1 0 0 0 0 0 0 0 0 0 0 0 0 1 1 | ($C003) |
| 37. | 1 1 1 0 0 0 0 0 0 0 0 0 0 1 1 1 | ($E007) |
| 38. | 1 1 1 1 0 0 0 0 0 0 0 0 1 1 1 1 | ($F00F) |
| 39. | 1 1 1 1 1 0 0 0 0 0 0 1 1 1 1 1 | ($F81F) |
| 40. | 1 1 1 1 1 1 0 0 0 0 1 1 1 1 1 1 | ($FC3F) |
| 41. | 1 1 1 1 1 1 1 0 0 1 1 1 1 1 1 1 | ($FE7F) |
| 42. | 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 | ($FFFF) |
| 43. | 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 | ($7FFE) |
| 44. | 0 0 1 1 1 1 1 1 1 1 1 1 1 1 0 0 | ($3FFC) |
| 43. | 0 0 0 1 1 1 1 1 1 1 1 1 1 0 0 0 | ($1FF8) |
| 46. | 0 0 0 0 1 1 1 1 1 1 1 1 0 0 0 0 | ($0FF0) |
| 47. | 0 0 0 0 0 1 1 1 1 1 1 0 0 0 0 0 | ($07E0) |
| 48. | 0 0 0 0 0 0 1 1 1 1 0 0 0 0 0 0 | ($03C0) |
| 49. | 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 | ($0180) |

The other elements of the 'specification' are:

1. There is to be a one second delay between output bit patterns.
2. The position of a switch is to be tested for an ON-OFF condition. If the switch is ON the sequence is to RUN; if it is OFF the sequence is to STOP.
3. The position of another switch is to be tested. If it is ON the sequence is to be terminated immediately.

## 9.07.09 I/O BIT MANIPULATION THROUGH DATA TABLES (continued)

As a control programmer you probably know that when repetitive sequences of events take place they are best done within a loop. But how do you organize a loop around a changing pattern of input or output data?

The program below illustrates the use of a data table being used to solve the problem of supplying changing data to a loop. This demonstration program is one of many possible solutions to the problem posed by our 'customer'.

Due to the keyboard polling technique used this program will not operate properly within the tracer. It should be assembled to disk as a command (A:0=DEMO1.CMD<CR>) and then called from FLEX (+++DEMO1<CR>)

Once the program is loaded and running holding down the space bar will allow the output display to run, releasing it will stop the display. Hitting ESCAPE at any time will terminate the display and return to FLEX.

```
0001 GLOBAL BYTE ENDFLAG;
0002
0003 INCLUDE 0.IOSUBS;
0004 INCLUDE 0.BITIO;
0005
0006 INTEGER OUTDATA $0000,$0001,$0002,$0004,$0008,$0010,$0020,$0040,$0080,
0007                       $0100,$0200,$0400,$0800,$1000,$2000,$4000,$8000,
0008                 $0000,$8001,$4002,$2004,$1008,$0810,$0420,$0240,$0180,
0009                       $0240,$0420,$0810,$1008,$2004,$4002,$8001,$0000,
0010                 $8001,$C003,$E007,$F00F,$F81F,$FC3F,$FE7F,$FFFF,$7FFE,
0011                 $3FFC,$1FF8,$0FF0,$07E0,$03C0,$0180;
0012
0013 PROCEDURE DELAY:
0014    BYTE CHAR,COUNT;
0015    COUNT=5;
0016    REPEAT
0017       CHAR=GETKEY;
0018       IF CHAR=$1B
0019          THEN BEGIN
0020               ENDFLAG=0;
0021               BREAK;
0022               END;
0023       IF CHAR=$20
0024          THEN COUNT=COUNT-1;
0025    UNTIL COUNT=0;
0026 ENDPROC;
0027
0028 PROCEDURE PATTERN_OUT:
0029    BYTE COUNT;
0030    ENDFLAG=1;
0031    COUNT=0;
0032    PRINT("\N\N");
0033    REPEAT
0034       PUTCHAR(CR);
0035       BITSOUT(OUTDATA(COUNT));
0036       PRINT("    ");
0037       DELAY;
0038       IF ENDFLAG=0
0039          THEN BREAK;
0040       COUNT=COUNT+1;
0041    IF COUNT=49 THEN COUNT=0;
0042    FOREVER;
```

## 9.07.09 I/O BIT MANIPULATION THROUGH DATA TABLES (continued)

The use of data tables provides an elegant solution to the requirement to supply different data to the loop on each iteration. Not only can the data be simple write only data like this example, the data can be bit masking tables, vectors to various subroutines, data for comparison with incoming data, etc. Proper use of data tables will not only speed up program development but will also make future modifications and/or additions that much easier!

## 9.07.10 PRIORITIZED I/O HANDLING THROUGH DATA TABLES

The following procedure demonstrates the ability of PL/9 to work with a 16-bit input data table and a vector table of output data rather than use a mass of "IF...THEN...ELSE" or "IF...CASE1...CASE2..." statements

A prime ingredient for successful use of tables is a well defined set of input-output conditions. If the I/O combinations are to be prioritized very careful consideration must be given to the order presentation in the table.

This demonstration will produce one of sixteen messages based upon the binary input pattern. Only one logical '1' is permitted in the input pattern, all others are considered illegal and an error message to this effect will be posted.

The table and program are organized to accept the highest priority input and output the corresponding message. Obviously a binary pattern could be sent to an I/O port in lieu of a message in a practical application.

To try this demonstration program out type it in and run it under the PL/9 tracer.

### I/O ASSIGNMENTS

| BINARY INPUT | MESSAGE |
|---|---|
| 1000000000000000 | TOTAL SHUTDOWN |
| 0100000000000000 | HYDRAULIC SHUTDOWN |
| 0010000000000000 | PNEUMATIC SHUTDOWN |
| 0001000000000000 | ELECTRICAL SHUTDOWN |
| 0000100000000000 | COOLANT SHUTDOWN |
| 0000010000000000 | MAIN FUEL PUMPS OFF |
| 0000001000000000 | AUX FUEL PUMPS OFF |
| 0000000100000000 | MAIN PRODUCT PUMPS OFF |
| 0000000010000000 | AUX PRODUCT PUMPS OFF |
| 0000000001000000 | VALVE 7 OFF |
| 0000000000100000 | VALVE 6 OFF |
| 0000000000010000 | VALVE 5 OFF |
| 0000000000001000 | VALVE 4 OFF |
| 0000000000000100 | VALVE 3 OFF |
| 0000000000000010 | VALVE 2 OFF |
| 0000000000000001 | VALVE 1 OFF |

Any other input pattern is to be considered invalid.

## 9.07.10 PRIORITIZED I/O HANDLING THROUGH DATA TABLES (continued)

```
0001 INCLUDE 0.IOSUBS;
0002 INCLUDE 0.BITIO;
0003
0004 CONSTANT BELL=$07, ESCAPE=$1B;  /* ASCII EQUATES */
0005
0006 INTEGER DATAIN $8000,$4000,$2000,$1000,$0800,$0400,$0200,$0100,
0007               $0080,$0040,$0020,$0010,$0008,$0004,$0002,$0001;
0008
0009 /*
0010    NOTE: EACH MESSAGE MUST BE PADDED TO EQUAL THE LENGTH OF THE
0011          LONGEST MESSAGE OR SOME STRANGE THINGS WILL HAPPEN!
0012 */
0013 BYTE MESSAGEOUT "TOTAL SHUTDOWN         ",
0014                 "HYDRAULIC SHUTDOWN    ",
0015                 "PNEUMATIC SHUTDOWN    ",
0016                 "ELECTRICAL SHUTDOWN   ",
0017                 "COOLANT SHUTDOWN      ",
0018                 "MAIN FUEL PUMPS OFF   ",
0019                 "AUX FUEL PUMPS OFF    ",
0020                 "MAIN PRODUCT PUMPS OFF",
0021                 "AUX PRODUCT PUMPS OFF ",
0022                 "VALVE (7) CLOSED      ",
0023                 "VALVE (6) CLOSED      ",
0024                 "VALVE (5) CLOSED      ",
0025                 "VALVE (4) CLOSED      ",
0026                 "VALVE (3) CLOSED      ",
0027                 "VALVE (2) CLOSED      ",
0028                 "VALVE (1) CLOSED      ";
0029
0030 PROCEDURE BITS_IN_AND_OUT_DEMO:
0031    BYTE COUNT,FLAG,TCOUNT:
0032    INTEGER CHAR;
0033
0034    REPEAT
0035       FLAG=FALSE;
0036       COUNT=0;
0037       PRINT("\N\N INPUT A 16-BIT BINARY PATTERN WITH A SINGLE '1'\N\N");
0038       CHAR=BITSIN;
0039          REPEAT
0040             IF DATAIN(COUNT)=CHAR
0041                THEN BEGIN
0042                   PRINT("\N\N");
0043                   PRINT(.MESSAGEOUT(COUNT*23));
0044                   FLAG=TRUE;
0045                END;
0046             IF FLAG=TRUE THEN BREAK;
0047             COUNT=COUNT+1;
0048             IF COUNT=16
0049                THEN BEGIN
0050                     PRINT("\N\NINVALID INPUT!  TRY AGAIN.");
0051                     PUTCHAR(BELL);
0052                     END;
0053          UNTIL COUNT=16;
0054       PRINT("\N\NHIT ESCAPE TO TERMINATE, ANY OTHER KEY TO CONTINUE\N");
0055       UNTIL GETCHAR=ESCAPE;
```

## 9.07.11 NON-PRIORITIZED I/O HANDLING THROUGH DATA TABLES

This demonstration will produce one of sixteen messages based upon the binary input pattern. As the word 'VALVE' and 'OPEN/CLOSED' are repeated from one message to the next we will 'assemble' the message in software rather than define the entire message in a data table as we did in the last demonstration program.

This technique is only practical where there is a large degree of similarity between several messages and or/the messages are lengthy. There are many applications that the number of messages required will tend to gobble up memory space at an alarming rate if the programmer allocates each message a dedicated string. This program demonstrates one of the techniques that can be used to save a very large amount of memory in such a program. For if this program were not structured the way it is we would have to provide storage for the following messages sixteen times!

VALVE 1 = (|) OPEN
VALVE 1 = (/) CLOSED

This would require considerably more code storage than the structure we illustrate in the following program. Also note that we are taking the effort in the program structure to keep an orderly column presentation as we move from one one digit valve numbers (0 - 9) to two digit valve numbers (10 - 16). This program will work properly in the PL/9 tracer.

### I/O ASSIGNMENTS

| BINARY INPUT | MESSAGE |
|---|---|
| 1000000000000000 | VALVE 16 OPEN IF '1', CLOSED IF '0'. |
| 0100000000000000 | VALVE 15 OPEN IF '1', CLOSED IF '0'. |
| 0010000000000000 | VALVE 14 OPEN IF '1', CLOSED IF '0'. |
| 0001000000000000 | VALVE 13 OPEN IF '1', CLOSED IF '0'. |
| 0000100000000000 | VALVE 12 OPEN IF '1', CLOSED IF '0'. |
| 0000010000000000 | VALVE 11 OPEN IF '1', CLOSED IF '0'. |
| 0000001000000000 | VALVE 10 OPEN IF '1', CLOSED IF '0'. |
| 0000000100000000 | VALVE 9 OPEN IF '1', CLOSED IF '0'. |
| 0000000010000000 | VALVE 8 OPEN IF '1', CLOSED IF '0'. |
| 0000000001000000 | VALVE 7 OPEN IF '1', CLOSED IF '0'. |
| 0000000000100000 | VALVE 6 OPEN IF '1', CLOSED IF '0'. |
| 0000000000010000 | VALVE 5 OPEN IF '1', CLOSED IF '0'. |
| 0000000000001000 | VALVE 4 OPEN IF '1', CLOSED IF '0'. |
| 0000000000000100 | VALVE 3 OPEN IF '1', CLOSED IF '0'. |
| 0000000000000010 | VALVE 2 OPEN IF '1', CLOSED IF '0'. |
| 0000000000000001 | VALVE 1 OPEN IF '1', CLOSED IF '0'. |

## 9.07.11 NON-PRIORITIZED I/O THROUGH DATA TABLES (continued)

```
0001 INCLUDE 0.IOSUBS;
0002 INCLUDE 0.BITIO;
0003
0004 CONSTANT BELL=$07, ESCAPE=$1B;  /* ASCII EQUATES */
0005
0006 CONSTANT OPEN=$FF,CLOSED=0;     /* VALVE STATUS FLAGS */
0007
0008 INTEGER INMASK $8000,$4000,$2000,$1000,$0800,$0400,$0200,$0100,
0009                $0080,$0040,$0020,$0010,$0008,$0004,$0002,$0001;
0010
0011 /*
0012    NOTE: EACH MESSAGE MUST BE PADDED TO EQUAL THE LENGTH OF THE
0013          LONGEST MESSAGE OR SOME STRANGE THINGS WILL HAPPEN!
0014          IN THIS PROGRAM NOTE THE WAY WE HANDLE THE DIFFERENT
0015          MESSAGE LENGTHS IN THE TWO TABLES.
0016 */
0017 BYTE HIVALVENO "16","15","14","13","12","11","10";
0018 BYTE LOVALVENO "9","8","7","6","5","4","3","2","1";
0019
0020 BYTE VALVETABLE "\N\N\N\N\N\N\N\N
0021 INPUT VALVE STATUS AS A SERIES OF ONES (1) AND ZEROS (0)\N\N
0022       0 = VALVE CLOSED ..... 1 = VALVE OPEN\N
0023       ======================================\N\N
0024                1 1 1 1 1 1 1\N
0025 VALVE NUMBER:- 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1\N
0026                | | | | | | | | | | | | | | | |\N
0027                ";
0028
0029 PROCEDURE BITS_IN_AND_OUT_DEMO:
0030    BYTE COUNT:
0031    INTEGER CHAR;
0032
0033    REPEAT
0034       COUNT=0;
0035       PRINT(.VALVETABLE);
0036       CHAR=BITSIN;
0037       CRLF;
0038          REPEAT
0039             PRINT("\NVALVE ");
0040             IF COUNT <= 6
0041                THEN BEGIN          /* TWO DIGIT NUMBER */
0042                     PRINT("(");
0043                     PRINT(.HIVALVENO(COUNT*3));
0044                     END;
0045                ELSE BEGIN          /* ONE DIGIT NUMBER */
0046                     PRINT(" (");
0047                     PRINT(.LOVALVENO((COUNT-7)*2));
0048                     END;
0049             PRINT(") = ");
0050             IF CHAR AND INMASK(COUNT) <> 0
0051                THEN PRINT("(|)  OPEN");
0052                ELSE PRINT("(/)  CLOSED");
0053             COUNT=COUNT+1;
0054          UNTIL COUNT=16;
0055       PRINT("\N\NHIT ESCAPE TO TERMINATE, ANY OTHER KEY TO CONTINUE\N");
0056       UNTIL GETCHAR=ESCAPE;
```

## 9.07.12 EVERYTHING BUT THE KITCHEN SINK THROUGH DATA TABLES

This final sample program is a representation of a process control program and is hoped to give you some insight into how to construct simple but very code efficient programs using the facilities in PL/9 that we have discussed in this and other sections. You are encouraged to type this program in and try it out as it uses many constructions that you should become familiar with.

This program will not run under the PL/9 tracer due to the keyboard polling used and will have to be compiled to disk as a command (A:O=1.DEMO4.CMD<CR>) and executed from FLEX (+++DEMO4<CR>). When the program is terminated it will re-enter FLEX.

Once running the program will prompt you for the initial status of the system as follows:

```
+------------------------------ EMERGENCY SHUTDOWN  (INPUT)
| +---------------------------- FIRE               (INPUT)
| | +-------------------------- PRODUCTION SHUTDOWN (INPUT)
| | | +------------------------ OVER TEMPERATURE    (INPUT)
| | | |
| | | | +---------------------- VALVE 4 RESPONSE    (INPUT)
| | | | | +-------------------- VALVE 3 RESPONSE    (INPUT)
| | | | | | +------------------ VALVE 2 RESPONSE    (INPUT)
| | | | | | | +---------------- VALVE 1 RESPONSE    (INPUT)
| | | | | | | |
| | | | | | | | +-------------- VALVE 4 DEMAND      (OUTPUT)
| | | | | | | | | +------------ VALVE 3 DEMAND      (OUTPUT)
| | | | | | | | | | +---------- VALVE 2 DEMAND      (OUTPUT)
| | | | | | | | | | | +-------- VALVE 1 DEMAND      (OUTPUT)
| | | | | | | | | | | |
| | | | | | | | | | | | +------- VALVE 4 MANUAL      (INPUT)
| | | | | | | | | | | | | +----- VALVE 3 MANUAL      (INPUT)
| | | | | | | | | | | | | | +--- VALVE 2 MANUAL      (INPUT)
| | | | | | | | | | | | | | | +- VALVE 1 MANUAL      (INPUT)
| | | | | | | | | | | | | | | |
E F P O R R R R D D D D M M M M
S I R V V V V V V V V V V V V V
D R D T 4 3 2 1 4 3 2 1 4 3 2 1
```

You then supply a series of 1's and 0's which represent the system status. If any of the top four bits are set the system will take over and provide a preset set of valve openings and closures and then prompt you for the valve response. If the valve response does not match the demand signals then 'FAILED' warnings will be displayed. If the top four bits are all zero the system will be in normal operation where the input from the MANUAL inputs will be echoed to the the DEMAND outputs. The system will again prompt for a valve RESPONSE which is matched against the demand signal, if they do not match the FAILED message will be posted where appropriate.

If we constructed this program on IF...THEN or IF...CASE arguments alone the program would have run to over ten pages and taken five times as much code!

## 9.07.12 EVERYTHING BUT THE KITCHEN SINK THROUGH DATA TABLES (continued)

```
0001 ORIGIN=$B000;
0002
0003 GLOBAL INTEGER INCHAR,OUTCHAR,OPIN,BITCHAR,INBITS,OUTBITS:
0004        BYTE INCOUNT,COUNT;
0005
0006 INCLUDE 0.IOSUBS;
0007 INCLUDE 0.BITIO;
0008
0009 CONSTANT START=0,END=$FF,ESCAPE=$1B;
0010
0011 INTEGER INMASK       $8000,$4000,$2000,$1000;
0012 INTEGER OUT_ONMASK   $0030,$0010,$0000,$0020;
0013 INTEGER OUT_OFFMASK  $FF3F,$FF1F,$FF3F,$FFFF;
0014
0015 BYTE STATUS " EMERGENCY SHUTDOWN ",
0016             "  * * * FIRE * * *  ",
0017             "PRODUCTION EMERGENCY",
0018             "  OVER TEMPERATURE  ",
0019
0020             "  NORMAL OPERATION\N\N
0021 OPERATOR COMMANDS ARE\N
0022 ====================\N";
0023
0024 BYTE SYSTEM_RESPONSE "\N\N
0025 THE SYSTEMS RESPONSE IS\N
0026 ======================";
0027
0028 BYTE VALVE_RESPONSE "\N\N
0029 WHAT IS THE VALVE RESPONSE?\N
0030 ==========================";
0031
0032 BYTE OP_OPEN "OPEN  VALVE (4)",
0033              "OPEN  VALVE (3)",
0034              "OPEN  VALVE (2)",
0035              "OPEN  VALVE (1)";
0036
0037 BYTE OP_CLOSE "CLOSE VALVE (4)",
0038               "CLOSE VALVE (3)",
0039               "CLOSE VALVE (2)",
0040               "CLOSE VALVE (1)";
0041
0042 BYTE OPEN_VALVE "OPENING VALVE (4)",
0043                 "OPENING VALVE (3)",
0044                 "OPENING VALVE (2)",
0045                 "OPENING VALVE (1)";
0046
0047 BYTE CLOSE_VALVE "CLOSING VALVE (4)",
0048                  "CLOSING VALVE (3)",
0049                  "CLOSING VALVE (2)",
0050                  "CLOSING VALVE (1)";
0051
0052 BYTE VALVE_OPEN "VALVE (4) = (|)    OPEN",
0053                 "VALVE (3) = (|)    OPEN",
0054                 "VALVE (2) = (|)    OPEN",
0055                 "VALVE (1) = (|)    OPEN";
0056
```

## 9.07.12 EVERYTHING BUT THE KITCHEN SINK THROUGH DATA TABLES (continued)

```
0057 BYTE VALVE_CLOSED "VALVE (4) = (/)  CLOSED",
0058                   "VALVE (3) = (/)  CLOSED",
0059                   "VALVE (2) = (/)  CLOSED",
0060                   "VALVE (1) = (/)  CLOSED";
0061
0062 BYTE MAIN_PROMPT "
0063 WHAT IS THE SYSTEM STATUS?\N
0064 ========================";
0065
0066 BYTE IO_PROMPT "\N\N
0067 E F P O R R R R D D D D M M M M\N
0068 S I R V V V V V V V V V V V V V\N
0069 D R D T 4 3 2 1 4 3 2 1 4 3 2 1\N
0070 | | | | | | | | | | | | | | | |\N";
0071
0072 PROCEDURE NEW_SCREEN;
0073    COUNT=24;
0074    REPEAT
0075       CRLF;
0076       COUNT=COUNT-1;
0077    UNTIL COUNT=0;
0078 ENDPROC;
0079
0080 PROCEDURE DELAY;
0081    PRINT("\N\NHIT ESCAPE TO CONTINUE");
0082    REPEAT
0083    UNTIL GETKEY=$1B;
0084    PRINT("\N\N");
0085 ENDPROC;
0086
0087 PROCEDURE SYSTEM_STATUS;
0088    INCOUNT=0;
0089    REPEAT
0090       IF !INCHAR AND INMASK(INCOUNT) <> 0
0091          THEN BREAK;
0092       INCOUNT=INCOUNT+1;
0093    UNTIL INCOUNT=4;
0094    CRLF;
0095    PRINT(.STATUS(INCOUNT*21));
0096 ENDPROC;
0097
0098 PROCEDURE OPERATOR_COMMAND_IN;
0099    OPIN=INCHAR;
0100    OPIN=SWAP(OPIN);      /* THIS SAVES US FROM DOING SHIFT(OPIN,12)! */
0101    OPIN=SHIFT(OPIN,4);   /* PUT OPERATOR VALVE DEMANDS INTO TOP 4-BITS */
0102    OPIN=OPIN AND $F000; /* CLEAR THE LOWER 12 BITS */
0103
0104    COUNT=0;
0105    REPEAT
0106       PRINT("\N  ");
0107       IF OPIN AND INMASK(COUNT) <> 0
0108          THEN PRINT(.OP_OPEN(COUNT*16));
0109          ELSE PRINT(.OP_CLOSE(COUNT*16));
0110       COUNT=COUNT+1;
0111    UNTIL COUNT=4;
0112 ENDPROC;
0113
```

## 9.07.12 EVERYTHING BUT THE KITCHEN SINK THROUGH DATA TABLES (continued)

```
0114 PROCEDURE OPERATOR_COMMAND_OUT;
0115    OUTCHAR=INCHAR;
0116    BITCHAR=INCHAR;
0117    BITCHAR=SHIFT(BITCHAR,4);     /* SHIFT OPERATOR INPUT TO COMMAND OUT */
0118    BITCHAR=BITCHAR AND $00F0;    /* CLEAR ALL SURROUNDING BITS TO 0 */
0119    OUTCHAR=OUTCHAR OR BITCHAR;   /* SET OPEN COMMANDS TO 1 */
0120    BITCHAR=BITCHAR OR $FF0F;     /* SET ALL SURROUNDING BITS TO 1 */
0121    OUTCHAR=OUTCHAR AND BITCHAR; /* SET ALL CLOSE COMMANDS TO 0 */
0122 ENDPROC;
0123
0124 PROCEDURE EMERGENCY_COMMAND_OUT;
0125    OUTCHAR=INCHAR;
0126    OUTCHAR=OUTCHAR OR OUT_ONMASK(INCOUNT);   /* SET OPEN COMMANDS TO 1 */
0127    OUTCHAR=OUTCHAR AND OUT_OFFMASK(INCOUNT); /* ALL CLOSE CMDS TO 0 */
0128 ENDPROC;
0129
0130 PROCEDURE VALVE_COMMAND_OUT;
0131    PRINT(.SYSTEM_RESPONSE);
0132    PRINT(.IO_PROMPT);
0133    BITSOUT(OUTCHAR);
0134    CRLF;
0135    BITCHAR=SWAP(OUTCHAR); /* GET COMMAND BITS INTO TOP 4-BITS */
0136    COUNT=0;
0137    REPEAT
0138       CRLF;
0139       IF BITCHAR AND INMASK(COUNT) <> 0
0140          THEN PRINT(.OPEN_VALVE(COUNT*18));
0141          ELSE PRINT(.CLOSE_VALVE(COUNT*18));
0142       COUNT=COUNT+1;
0143    UNTIL COUNT=4;
0144 ENDPROC;
0145
0146 PROCEDURE VALVE_STATUS_IN;
0147    COUNT=0;
0148    BITCHAR=SWAP(INCHAR);
0149    CRLF;
0150    REPEAT
0151       CRLF;
0152       IF BITCHAR AND INMASK(COUNT) <> 0
0153          THEN PRINT(.VALVE_OPEN(COUNT*24));
0154          ELSE PRINT(.VALVE_CLOSED(COUNT*24));
0155       COUNT=COUNT+1;
0156    UNTIL COUNT=4;
0157 ENDPROC;
0158
```

## 9.07.12 EVERYTHING BUT THE KITCHEN SINK THROUGH DATA TABLES (continued)

```
0159 PROCEDURE VALVE_RESPONSE_IN;
0160    PRINT(.VALVE_RESPONSE);
0161    PRINT(.IO_PROMPT);
0162    INCHAR=BITSIN;
0163
0164    OUTBITS=SWAP(OUTCHAR);
0165    INBITS=SHIFT(INCHAR,4);
0166    COUNT=0;
0167    CRLF;
0168    REPEAT
0169       CRLF;
0170       IF INBITS AND INMASK(COUNT) <> 0
0171          THEN PRINT(.VALVE_OPEN(COUNT*24));
0172          ELSE PRINT(.VALVE_CLOSED(COUNT*24));
0173       IF INBITS AND INMASK(COUNT) <> OUTBITS AND INMASK(COUNT)
0174          THEN PRINT("   * FAILED *");
0175       COUNT=COUNT+1;
0176    UNTIL COUNT=4;
0177 ENDPROC;
0178
0179 PROCEDURE NORMAL_OPERATION;
0180    OPERATOR_COMMAND_IN;
0181    DELAY;
0182    OPERATOR_COMMAND_OUT;
0183    VALVE_COMMAND_OUT;
0184    DELAY;
0185    VALVE_RESPONSE_IN;
0186 ENDPROC;
0187
0188 PROCEDURE EMERGENCY_OPERATION;
0189    VALVE_STATUS_IN;
0190    DELAY;
0191    EMERGENCY_COMMAND_OUT;
0192    VALVE_COMMAND_OUT;
0193    DELAY;
0194    VALVE_RESPONSE_IN;
0195 ENDPROC;
0196
0197 PROCEDURE REAL_SYSTEM_DEMO:BYTE ENDFLAG;
0198    REPEAT
0199       INCOUNT=0;
0200       NEW_SCREEN;
0201       PRINT(.MAIN_PROMPT);
0202       PRINT(.IO_PROMPT);
0203       INCHAR=BITSIN;
0204       CRLF;
0205       SYSTEM_STATUS;
0206       IF INCOUNT = 4
0207          THEN NORMAL_OPERATION;
0208          ELSE EMERGENCY_OPERATION;
0209       PRINT("\N\NHIT ESCAPE TO CONTINUE ... ANY OTHER KEY TO EXIT\N");
0210       REPEAT
0211          ENDFLAG=GETKEY;
0212          IF ENDFLAG <> 0 .AND ENDFLAG <> ESCAPE
0213             THEN ENDFLAG=END;
0214      UNTIL ENDFLAG <> 0;
0215    UNTIL !ENDFLAG=END;
```

## 9.08.00 RECURSIVE PROGRAMMING

Recursion is a natural and elegant solution to many programming problems that can otherwise be solved only by far more complicated non-recursive algorithms. Simplicity is a key to reliability, hence faster program development. PL/9 is a stack-oriented language and is therefore able to handle recursive programs, as long as the programmer is aware of the potential pitfalls. To introduce the concept of recursion, consider the following example:

```
0001 INCLUDE IOSUBS;
0002
0003 PROCEDURE PRINTINT(INTEGER NUM:BYTE BASE);
0004
0005    IF NUM<0 THEN
0006    BEGIN
0007       PUTCHAR('-);
0008       NUM=-NUM;
0009    END;
0010
0011    IF NUM >= BASE
0012       THEN PRINTINT(NUM/BASE,BASE);
0013    NUM = NUM\BASE + '0
0014    IF NUM > '9
0015       THEN NUM=NUM + '7;
0016    PUTCHAR(NUM);
0017
0018 ENDPROC;
0019
0020 PROCEDURE TEST: INTEGER N;
0021    N=-1000;
0022    REPEAT
0023       PRINTINT(N,10);
0024       CRLF;
0025       N=N+1;
0026    UNTIL N=1000;
```

The task of the recursive procedure PRINTINT is to print an integer, without leading zeros. To do this it first detects if the supplied number is negative, printing a minus sign and negating if it is (lines 5-9), then if the number is greater than the BASE it calls itself (this is what is meant by recursive programming) with the same number divided by BASE. After this it prints the remainder of the same division (the modulus operator \ performs this function) after it adjusts the number for the specified BASE.

Consider the number 1234 passed to PRINTINT with BASE equal to 10. In line 12, 123 will be passed to a second invocation of PRINTINT, which in turn passes 12 a to third invocation, which passes 1 to a fourth invocation. At last the test in line 11 fails, so the 1 is printed. Returning to level 3 causes the 2 to be printed. 3 gets printed in level 2 and finally 4 is printed in the outer level before the procedure finishes. This process may sound complicated but it is in fact quite simple; the difficulty is only in explaining what is happening. The following table should make things a little clearer:

| LEVEL OF CALL | VALUE OF NUM | DIGIT PRINTED |
|---|---|---|
| 1 | 1234 | (123)4 |
| 2 | 123 | (12)3 |
| 3 | 12 | (1)2 |
| 4 | 1 | 1 |

## 9.08.00 RECURSIVE PROGRAMMING (continued)

This procedure will produce rather strange results (but they are in fact correct) when negative numbers to base 2 or base 16 are passed to it.

Recursion can also be applied to functions, as in the following example:

```
0001 INCLUDE IOSUBS;
0002 INCLUDE REALCON;
0003 INCLUDE PRNUM;
0004
0005 PROCEDURE FACTORIAL(REAL NUM);
0006
0007    IF NUM<=1 THEN RETURN REAL 1;
0008
0009 ENDPROC REAL NUM*FACTORIAL(NUM-1);
0010
0011 PROCEDURE TEST: REAL NUM;
0012
0013    NUM=0;
0014    REPEAT
0015       NUM=NUM+1;
0016       PRINT("\NFACTORIAL "); PRNUM(NUM);
0017       PRINT(" IS "); PRNUM(FACTORIAL(NUM));
0018    UNTIL NUM=20;
```

The function procedure FACTORIAL is passed a REAL parameter NUM, and returns the factorial of that parameter. It does this by subtracting one from the number and calling itself recursively, until the value of the parameter passed as NUM is one or less, at which point the function returns unity. Each invocation of the procedure only performs one of the multiplications needed to get the final result.

A point to note is that since the return type (BYTE, INTEGER or REAL) of a PL/9 procedure is determined by the argument of an ENDPROC or RETURN statement then the recursive call must follow a RETURN otherwise PL/9 does not know what type the function returns (and assumes INTEGER). If there is no RETURN before the recursive call then a dummy must be put in, such as

```
IF FALSE THEN RETURN REAL 0;
```

which fixes the type of the function without causing a return.

An alternative, available in PL/9 version 4.XX onwards, is to declare the size of the returned variable in the procedure declaration thus:

```
PROCEDURE FUNCTION(REAL NUM):INTEGER I:REAL;
                             ----
```

This forces the procedure to return a REAL sized variable.

See section 7.01.11 (PROCEDURE FIVE) for further details.

Recursive programs can eat into stack at a surprising rate, so always be sure to leave a generous amount. The last example, for instance, uses six bytes for each nested call, resulting in 120 bytes being used when calculating the factorial of 20. If any local variables were declared by FACTORIAL (as they might well be in other examples) then these appear in each of the 20 invocations, greatly adding to the space needed.

## 9.09.00 MULTI-TASKING PROGRAMS

One of the more powerful aspects of program architecture 'aided and abetted' by PL/9 is the construction of true multi-tasking programs. Multi-tasking programs appear to be doing several jobs at once, when in reality they are only doing a little bit of each program and then going on to do a little bit of the next program, and so on.

The key to writing multi-tasking programs is in the basic rule...

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                                                       *
*  NEVER  WRITE  A  PROCEDURE  THAT 'GRABS' THE         *
*                                                       *
*  PROCESSOR FOR ANY APPRECIABLE PERIOD OF TIME         *
*                                                       *
* * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

This means that you can NEVER use routines like GETCHAR in the IOSUBS library as part of a multi-tasking program. The GETKEY routine on the other hand is ideally suited for multi-tasking programs. Examine the differences between these two routines and you should be able to tell why.

This rule is also very important where delay routines are concerned. This seems to be a contradiction in terms doesn't it? How do you write a delay routine that doesn't delay? The answer lies in how you construct the delay routine. Take the following example:

```
0001 GLOBAL INTEGER DCOUNT1, DCOUNT2;
0002
0003 INCLUDE 0.IOSUBS;
0004
0005 PROCEDURE DELAY1;
0006    IF DCOUNT1=0
0007       THEN DCOUNT1=231; /* RESET THE DELAY COUNTER */
0008       ELSE DCOUNT1=DCOUNT1-1;
0009 ENDPROC;
0010
0011 PROCEDURE DELAY2;
0012    IF DCOUNT2=0
0013       THEN DCOUNT2=148;
0014       ELSE DCOUNT2=DCOUNT2-1;
0015 ENDPROC DCOUNT2;
0016
0017
0018 PROCEDURE MAIN;
0019    DCOUNT1=1;
0020    DCOUNT2=1;
0021
0022    REPEAT
0023       DELAY1;
0024       IF DCOUNT1=0
0025          THEN PRINT("\NHELLO");
0026
0027       IF DELAY2=0
0028          THEN PRINT("   GOODBYE");
0029    FOREVER;
```

## 9.09.00 MULTI-TASKING PROGRAMS (continued)

In the example note that all of the delay counts are declared as GLOBAL variables. Also note that we have written the two delay routines slightly differently. One is written as a normal procedure with no local variables and the other is written as a function.

The difference is only to illustrate that there is more than one way to accomplish the same end. In this example the structure of 'DELAY2' results in a slightly more readable structure in the main program.

Look at the way DELAY1 is structured.

```
0005 PROCEDURE DELAY1;
0006    IF DCOUNT1=0
0007       THEN DCOUNT1=231; /* RESET THE DELAY COUNTER */
0008       ELSE DCOUNT1=DCOUNT1-1;
0009 ENDPROC;
```

Note that there is no way for this routine to 'lock up' and steal the processor. On line 6 we simply test whether the global variable DCOUNT1 has reached 0 yet. If it has we reset it to the delay time constant via line 7. If it is not 0 we simply decrement DCOUNT in line 8. Since this procedure does not pass any value back to who called it the main program must also refer to the global variable DCOUNT1.

Now look at the way we structured DELAY2. Again we are using a global variable for the delay counter DCOUNT2, but this time we pass a 'copy' of it back to whoever called it. This allows us to effectively call DELAY2 and find out what it did to DCOUNT2 as one single statement.

```
0011 PROCEDURE DELAY2;
0012    IF DCOUNT2=0
0013       THEN DCOUNT2=148;
0014       ELSE DCOUNT2=DCOUNT2-1;
0015 ENDPROC DCOUNT2;
```

The differences in structure between DELAY1 and DELAY2 are highlighted in the main program.

```
0022    REPEAT
0023       DELAY1;
0024       IF DCOUNT1=0
0025          THEN PRINT("\NHELLO");
0026
0027       IF DELAY2=0
0028          THEN PRINT("   GOODBYE");
0029    FOREVER;
```

To use DELAY1 we have to first 'CALL' it in line 23. Then find out what DELAY1 did to DCOUNT1 we have to evaluate DCOUNT1 as illustrated in line 24.

9.09.00 MULTI-TASKING PROGRAMS (continued)

The structure of DELAY2 is much more suited to the job we wish to accomplish in this example as line 27 illustrates. Here we simply treat DELAY2 as though it were a variable. This has the effect of greatly simplifying the code as well as improving the readability of the program.

The structure of DELAY1 does have its advantages in some circumstances such as those where you wish to put all of the delay counters into one single routine in order to simplify the control flow of a program.

Another important point to note is that all delay counters MUST be declared as global variables. If they are declared as local variables they will be lost when the procedure terminates. Generally speaking multi-tasking programs will tend to rely heavily on GLOBAL variables for two reasons. First so that they are not lost between procedures. Second so that they may be accessed by all procedures.

The sample program presented actually works. You are encouraged to type it in and run it under the PL/9 tracer. Try fiddling with the counter reset values in lines 7 and 13 and see what effect it has on the ratio and speed of presentation of the two messages.

The program presented on the following pages is a much more elaborate demonstration of how to write a multi-tasking program. It also works but will have to be compiled to memory or disk and run directly as it will not operate properly in the PL/9 tracer due to the keyboard polling technique used.

This demonstration program runs FIVE tasks simultaneously. The tasks will appear on the video display as follows:

```
HIT ESCAPE TO TERMINATE PROGRAM     <---- this message will flash on and off.

|---------------------------|       <---- is a 30 column buffer where you may
                                           type in characters and back-space.

99 <---------------------------------- is a fast counter moving between +/- 99
99 <---------------------------------- is a slightly slower counter.
99 <---------------------------------- is an even slower counter.
```

This program will work best with a fast video display. If you use it with a terminal we suggest that the baud rate be no less than 9600 baud.

Four constants in the program must also be set to the values recognized by your terminal or video display for:

non-destructive cursor right
non-destructive cursor left (usually back-space)
non-destructive cursor up
non-destructive cursor down

We will present a full explanation of each section of the program following the source listing.

9.09.00 MULTI-TASKING PROGRAMS (continued)

A FIVE TASK MULTI-TASKING PROGRAM

```
0001 GLOBAL INTEGER DNUMBER(3), DCOUNT(3), PASSES, FLASH_COUNT:
0002        BYTE COL_COUNT, INCHAR, FLASH_TOGGLE, TOGGLE(3), DISPLAY_DELAY,
0003             CURSOR_MOVE_FLAG;
0004
0005 INCLUDE 0.IOSUBS;
0006 INCLUDE 0.REALCON;
0007
0008 /* DISPLAY DEPENDANT EQUATES */
0009
0010 CONSTANT NON_DESTRUCTIVE_CURSOR_UP=$0B,
0011          NON_DESTRUCTIVE_CURSOR_DN=$0E,
0012          NON_DESTRUCTIVE_CURSOR_RT=$0C,
0013          NON_DESTRUCTIVE_BACKSPACE=$08;
0014
0015 CONSTANT ESCAPE=$1B, _SPACE=$20, BELL=$07, _RETURN=$0D,
0016          LOLIMIT=$20, HILIMIT=$7D, COL_LIMIT=30, UP=0, DN=1,
0017          CDELAY=20, FDELAY=30, PCOUNT=200;
0018
0019 PROCEDURE PRNUM(REAL NUMBER):BYTE BUFFER(20);
0020    PRINT(ASCII(NUMBER,.BUFFER));
0021 ENDPROC;
0022
0023 PROCEDURE CURSOR_UP(BYTE NUMBER);
0024    REPEAT
0025       PUTCHAR(NON_DESTRUCTIVE_CURSOR_UP);
0026       NUMBER=NUMBER-1;
0027    UNTIL NUMBER=0;
0028 ENDPROC;
0029
0030 PROCEDURE CURSOR_DN(BYTE NUMBER);
0031    REPEAT
0032       PUTCHAR(NON_DESTRUCTIVE_CURSOR_DN);
0033       NUMBER=NUMBER-1;
0034    UNTIL NUMBER=0;
0035 ENDPROC;
0036
0037 PROCEDURE CURSOR_RT(BYTE NUMBER);
0038    NUMBER=NUMBER-1;
0039    IF NUMBER=0
0040       THEN RETURN; /* DON'T MOVE */
0041    REPEAT
0042       PUTCHAR(NON_DESTRUCTIVE_CURSOR_RT);
0043       NUMBER=NUMBER-1;
0044    UNTIL NUMBER=0;
0045 ENDPROC;
0046
0047 PROCEDURE DELAY_TIMERS:BYTE TIMER;
0048    TIMER=0;
0049    REPEAT
0050       IF DCOUNT(TIMER)=0
0051          THEN DCOUNT(TIMER)=CDELAY*(TIMER+1);
0052          ELSE DCOUNT(TIMER)=DCOUNT(TIMER)-1;
0053       TIMER=TIMER+1;
0054    UNTIL TIMER=3;
0055 ENDPROC;
0056
```

9.09.00 MULTI-TASKING PROGRAMS (continued)

```
0057 PROCEDURE COUNT:BYTE COUNTER;
0058    COUNTER=0;
0059    DELAY_TIMERS;
0060
0061    REPEAT
0062       IF DCOUNT(COUNTER)=0
0063          THEN IF TOGGLE(COUNTER)=UP
0064                  THEN BEGIN
0065                       DNUMBER(COUNTER)=DNUMBER(COUNTER)+1;
0066                       IF DNUMBER(COUNTER)=99
0067                          THEN TOGGLE(COUNTER)=DN;
0068                       END;
0069                  ELSE BEGIN
0070                       DNUMBER(COUNTER)=DNUMBER(COUNTER)-1;
0071                       IF DNUMBER(COUNTER)=-99
0072                          THEN TOGGLE(COUNTER)=UP;
0073                       END;
0074       COUNTER=COUNTER+1;
0075    UNTIL COUNTER=3;
0076 ENDPROC;
0077
0078 PROCEDURE NEW_LINE(BYTE LINES);
0079    CURSOR_MOVE_FLAG=TRUE;
0080    CURSOR_DN(LINES);
0081    PUTCHAR(_RETURN);
0082    SPACE(4);
0083    PUTCHAR(_RETURN);
0084 ENDPROC;
0085
0086 PROCEDURE DISPLAY_COUNTS:BYTE COUNTER;
0087    COUNTER=0;
0088
0089    REPEAT
0090       IF DCOUNT(COUNTER)=0
0091          THEN BEGIN
0092               NEW_LINE(COUNTER+2);
0093               PRNUM(DNUMBER(COUNTER));
0094               CURSOR_UP(COUNTER+2);
0095               END;
0096       COUNTER=COUNTER+1;
0097    UNTIL COUNTER=3;
0098 ENDPROC;
0099
0100 PROCEDURE FLASH_MESSAGE;
0101    IF FLASH_COUNT=0
0102       THEN BEGIN
0103            CURSOR_UP(2);
0104            CURSOR_MOVE_FLAG=TRUE;
0105            PUTCHAR(_RETURN);
0106            FLASH_COUNT=FDELAY;
0107            FLASH_TOGGLE=NOT(FLASH_TOGGLE);
0108            IF FLASH_TOGGLE=TRUE
0109               THEN PRINT("HIT ESCAPE TO TERMINATE PROGRAM");
0110               ELSE SPACE(31);
0111            CURSOR_DN(2);
0112            END;
0113    FLASH_COUNT=FLASH_COUNT-1;
0114 ENDPROC;
```

## 9.09.00 MULTI-TASKING PROGRAMS (continued)

```
0115
0116 PROCEDURE OP_INPUT;
0117    PASSES=0;
0118    IF CURSOR_MOVE_FLAG=TRUE
0119       THEN BEGIN
0120            CURSOR_MOVE_FLAG=FALSE;
0121            PUTCHAR(_RETURN);
0122            CURSOR_RT(COL_COUNT);
0123            END;
0124
0125    REPEAT
0126       INCHAR=GETKEY;
0127       IF INCHAR >= LOLIMIT .AND INCHAR <= HILIMIT
0128          THEN BEGIN
0129               IF COL_COUNT < COL_LIMIT
0130                  THEN BEGIN
0131                       PUTCHAR(INCHAR);
0132                       COL_COUNT=COL_COUNT+1;
0133                       END;
0134                   ELSE PUTCHAR(BELL);
0135               END;
0136
0137       IF INCHAR = NON_DESTRUCTIVE_BACKSPACE
0138          THEN IF COL_COUNT <> 1
0139               THEN BEGIN
0140                    PUTCHAR(NON_DESTRUCTIVE_BACKSPACE);
0141                    PUTCHAR(_SPACE);
0142                    PUTCHAR(NON_DESTRUCTIVE_BACKSPACE);
0143                    COL_COUNT=COL_COUNT-1;
0144                    END;
0145          ELSE PUTCHAR(BELL);
0146       PASSES=PASSES+1;
0147       IF INCHAR=ESCAPE .OR INCHAR=_RETURN
0148          THEN BREAK;
0149    UNTIL PASSES=PCOUNT;
0150 ENDPROC;
0151
0152 INTEGER NUMBRINIT 100,100,100;
0153 INTEGER DELAYINIT 1,1,1;
0154 BYTE    TOGGLINIT DN,DN,DN;
0155
0156 PROCEDURE MAIN:BYTE INDEX;
0157    CURSOR_MOVE_FLAG=TRUE;
0158    FLASH_COUNT=FDELAY;
0159    FLASH_TOGGLE=TRUE;
0160
0161    INDEX=0;
0162    REPEAT
0163       DNUMBER(INDEX)=NUMBRINIT(INDEX);
0164       DCOUNT(INDEX)=DELAYINIT(INDEX);
0165       TOGGLE(INDEX)=TOGGLINIT(INDEX);
0166       INDEX=INDEX+1
0167    UNTIL INDEX=3;
0168
0169    PRINT("\N\N\N\N\N\N\N\N\N\N\N");
0170    CURSOR_UP(4);
0171
```

## 9.09.00 MULTI-TASKING PROGRAMS (continued)

```
0172 OP_INPUT_NEW_LINE:
0173
0174    PUTCHAR(_RETURN);
0175    PRINT("  * * TYPE SOMETHING HERE * * ");  /* EQUAL TO DISPLAY WIDTH */
0176    COL_COUNT=1;
0177    PUTCHAR(_RETURN);
0178
0179    REPEAT
0180       FLASH_MESSAGE;
0181       DISPLAY_COUNTS;
0182       COUNT;
0183       OP_INPUT;
0184
0185       IF INCHAR = _RETURN
0186          THEN GOTO OP_INPUT_NEW_LINE;
0187
0188       IF INCHAR = ESCAPE
0189          THEN BREAK;
0190    FOREVER;
0191
0192    CURSOR_DN(4);
```

9.09.00 MULTI-TASKING PROGRAMS (continued)

Lines 1, 2 and 3 are the GLOBAL variable declaration. Note that it is the first declaration in the procedure. Of particular interest is they way we have declared DNUMBER, DCOUNT, and TOGGLE. Each of these has been declared as a 3 element vector (0, 1, and 2). The reasons for this will become clearer later in this discussion.

Lines 5 and 6 are the now familiar library inclusions.

Lines 10 through 13 are a series of constants that tell the remainder of the program what codes to use for the functions indicated. Declaring system dependant variables like this at the top of a program as constants will not only make the program more portable than would be the case if we had imbedded the codes into the body of the program but will also clearly identify system dependant variables.

Lines 15 through 17 are another group of CONSTANTS placed in a prominent position in the program for similar reasons to those stated above. You are allowed to make as many CONSTANT declarations in a program as you wish, and position them wherever you wish.

Lines 19 through 21 are the PRNUM procedure discussed in section 8.05.08.

Lines 23 through 28 are a function that is passed a single byte called NUMBER. One each iteration of the REPEAT loop between lines 24 and 27 the code declared in the CONSTANT 'NON-DESTRUCTIVE_CURSOR_UP' will be transmitted to the video display via the PUTCHAR routine in IOSUBS on line 25. NUMBER will then be decremented on line 26 and tested for a zero condition on line 27. If NUMBER is not zero the loop will be executed over and over again until NUMBER finally equals zero. This procedure allows us to move the display cursor up a specified number of lines, the minimum is 1 the maximum is 255.

The procedure between lines 30 - 35 performs a similar function but moves the cursor down instead of up.

The procedure between lines 37 through is also similar but it decrements the value of number immediately upon entry at line 38. If NUMBER is equal to zero at this point the procedure will be terminated via the RETURN statement in line 40. If NUMBER is not zero at this point a REPEAT loop similar to the one just discussed will be entered. This program has obviously been structured to reject the value of 1 being assigned to NUMBER. The reason for this will become obvious shortly.

The next procedure between lines 47 and 55 is somewhat more complicated. Again a REPEAT...UNTIL loop is present so we must be doing something several times, but what? Upon entry we set a local variable called TIMER to zero. In this procedure TIMER will be used to subscript vectors. On the first iteration of the REPEAT...UNTIL loop line 50 can be read as: 'IF DCOUNT(0)=0' which points to the first element of DCOUNT and compares it with zero. If it is zero the THEN statement on line 51 will be executed. On the first iteration of the loop this line can be read as: 'THEN DCOUNT(0)=DCOUNT * (1)' CDELAY is a constant which will be multiplied by one and assigned to DCOUNT(0). This will reset the delay counter to its upper limit. If the comparison with zero on line 50 were not true then line 52 would be executed. On the first iteration this line could be read as 'DCOUNT(0)=DCOUNT(0)-1', a simple decrement of the first element in the vector. Line 53 is then executed which bumps the value of TIMER to 1. Line 54 looks to see if TIMER is 3 which it won't be until two more iterations are completed.

9.09.00 MULTI-TASKING PROGRAMS (continued)

On the second and third (final) iterations the only line that varies significantly is line 51. On the second iteration CDELAY will be multiplied by 2 and on the third by 3. By the end of the third iteration we will have either reset every delay counter, i.e. DCOUNT(0), DCOUNT(1) and DCOUNT(2) to its reset count value or will have decremented it by one. The structure of this delay counter routine is a very efficient expansion of the single delay counter routine discussed earlier in this section. Instead of assigning a reset count based on the multiplication of CDELAY and TIMER we could just as well taken the data from a READ-ONLY data table OR another set of vectors. In fact to illustrate this point we will be doing just this later in the program.

Lines 57 through 76 form another procedure that accesses a series of vectors via an incrementing index. Line 58 initializes the index COUNTER to zero. Line 59 calls the routine DELAY_TIMERS we just discussed. Note that the key word CALL is not used. The structure of PL/9 is such that simply entering a procedure name implies a call to that procedure. Perhaps if we had called DELAY_TIMERS 'DECREMENT_THE_3_DELAY_TIMERS_IF_THEY_ARE_NOT_ZERO_OR_RESET_THEM_IF_THEY_ARE'... this single entry would have made more sense. There is nothing wrong with calling a procedure by a name such as this, and it does not cause PL/9 to generate any extra code. Long procedure names only have two drawbacks...they are a pain in the neck to type in if the procedure is used several times in the program AND they can interfere with the presentation when nesting the control arguments.

Lines 61 through 75 form the main body of this program. On the first iteration of the REPEAT...UNTIL loop line 62 would read 'IF DCOUNT(0) = 0'. What we are doing here is testing to see if the first delay counter DCOUNT(0) has timed out yet. If the time delay is completed, as indicated by DCOUNT(0) being equal to zero the program will proceed into the body of the program between lines 63 to 73. If DCOUNT(0) is not zero the main body of the program will be completely bypassed and line 74 executed. Assuming that DCOUNT(0) is 0 line 63 will then test TOGGLE(0) and see if it is set to 'UP'. UP is a constant equal to 0. UP simply serves as a convenient label to tell us what we are actually looking for and to improve the readability of the program. If we had said 'IF TOGGLE(0) =0 THEN' what would be the significance of '0'. The use of constants rather than embedding cryptic references to 0 and 1 (or any other states or counter values) greatly improves program readability AT NO EXPENSE IN CODE EFFICIENCY OR EXECUTION SPEED.

If TOGGLE(0) was equal to UP lines 64 through 68 will be executed. If TOGGLE(0) was not equal to UP lines 69 through 73 would be executed. Assume that TOGGLE(0) was equal to UP for a moment. Line 65 would be executed and may be read as DNUMBER(0)=DNUMBER(0)+1, a simple increment. Then line 66 is executed which tests to see if DNUMBER(0) has reached 99 yet. If it has line 67 will then be executed as TOGGLE(0)=DN. Where DN is another constant declared as being 1. Control is now passed to line 74.

Lines 69 through 73 form the alternative path which handle the case where the counter is counting down toward -99. When -99 is reached the TOGGLE is then set to start the counter counting up toward +99.

Line 74 bumps COUNTER to the next value, which, after the first iteration, will make COUNTER equal to 1. Line 75 tests COUNTER and compares it with the termination value of 3. Obviously 1 is not equal to 3 so the main body of the program commencing at line 61 will be executed over again, this time with an index of 1. The second iteration will do exactly the same as the first but will access the second element of the vectors DCOUNT, DNUMBER, and DTOGGLE as DCOUNT(1), DNUMBER(1), and DTOGGLE(1);

9.09.00 MULTI-TASKING PROGRAMS (continued)

The use of vectors in this application CONSIDERABLY improves not only the code efficiency of the program but also the execution speed. If we had not used vectors we have had to duplicate the main body of the DELAY_TIMERS procedure three times and the main body of the COUNT procedure three times. This would have in turn necessitated three calls to the different delay routines and three calls to the different COUNT routines. Virtually any program that requires the same or similar function to be repeated several times will be best done through vectors and data tables. Once vectors are fully understood the procedures that use them will be just as readable as the procedures that don't.

Lines 78 through 84 form a function called NEW_LINE. Line 79 sets the global variable CURSOR_MOVE_FLAG to true in order to inform another part of the program that the cursor has been moved. The cursor is then moved down by calling the CURSOR_DN routine and passing along the incoming variable LINES which was passed to NEW_LINE by whoever called it. Line 81 returns the cursor to the left side of the screen by sending a carriage return (but no line feed!) to the video display via the PUTCHAR routine in IOSUBS. The next line sends 4 spaces out to the video display via another routine in IOSUBS, and the following line sends a another carriage return out to the video display. This bit of gyration is used to cancel the effect of the decrementing displays starting out as 99, a two digit number, decrementing into single digit numbers and then finally decrementing into negative two digit numbers. This procedure simply erases the contents of the line of interest before the latest count value is displayed by another routine.

Lines 86 through 98 form yet another vector oriented REPEAT...UNTIL procedure called DISPLAY_COUNTS. As the name suggests this is the routine that will put the counter values on the video display. Again we initialize an index called COUNTER to zero on line 87. The main body of the procedure is contained between line 89 and line 97. Line 90 can be read as 'IF DCOUNT(0) = 0' on the first iteration of the REPEAT...UNTIL loop. This means if delay counter DCOUNT(0) is zero we are to execute lines 91 through 95, if it is not control will be passed directly to line 96. Lets take the case where DCOUNT(0) is in fact 0. If you remember the previous procedure would have decremented DNUMBER(0) if DCOUNT(0) was zero. What this procedure does is display the number ONLY if it has been decremented.

Line 92 will position the cursor two lines down PLUS the value of COUNTER. On the first iteration this would move the cursor down two lines, on the second three lines, etc. NEW_LINE clear off the last count value as previously described. PRNUM will then print 'DNUMBER(0)' on the video display. Control will then be passed to line 96 which will increment our index COUNTER. Line 97 looks for a termination value of 3, and, as in the two previous examples causes the main body of the loop to be executed two more times to access the next two elements of the vectors.

It is important to note the trouble we are going to to prevent the cursor from being moved EXCEPT when we actually need to position it in order to display data. This is a very important consideration for serial I/O.

Lines 100 through 114 form a procedure called FLASH_MESSAGE that, believe it or not, does NOT use vectors! This procedure, as its name implies, flashes a message on the video display. On entry line 101 tests to see if the flash delay counter FLASH_COUNT has reached zero yet. If it has lines 102 through 112 will be executed. Otherwise control will be passed to line 113 which will simply decrement FLASH_COUNT and terminate the procedure at line 114. Again we have created a procedure that produces a delay without actually seizing the processor, a mandatory requirement of a multi-tasking program that is not interrupt driven.

9.09.00 MULTI-TASKING PROGRAMS (continued)

Lines 102 through 112 form the main body of the FLASH_MESSAGE procedure. Lets take it apart and see what makes it tick. First we move the cursor up two lines via the function call to CURSOR_UP on line 103. (We'll be explaining what all the cursor movements are for a bit later in this somewhat protracted treatis) Then we then set the CURSOR_MOVE_FLAG to indicate that we have moved the cursor (these long names are really nice aren't they!). We then send a carriage return (no line feed) to the video display via PUTCHAR on line 105. Next we reset the flash delay counter FLASH_COUNT to the value of the constant FDELAY.

Remember what we said earlier in this discussion about declaring time constants, delays, terminal I/O characters, I/O addresses, etc., as constants at the BEGINNING of a program. Imagine what a nuisance it would be to have to dig through the body of this program to change the time constants we have used so far. With numbers of this nature declared near the start of the program the 'fine tuning' of program delay counters, or the reconfiguration of the system hardware will be greatly simplified at a later date. One of the main rules that we, and others involved in industrial control work, have learned the hard way is that...

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                                                                         *
*  Control programs should be STRUCTURED to accommodate INEVITABLE changes! *
*                                                                         *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

ANY control program that is written on the basis of 'THE SPECIFICATION' without any thought as to providing the 'hooks' to facilitate the inevitable 'small change' is setting a course for disaster and probably a very dissatisfied client at the end of the day.

It the job of a good control programmer to structure his programs so that when changes must be incorporated it will not require a complete re-write of the program. PL/9 has built in capabilities to assist the programmer along these lines but a great deal of the responsibility for program structure lies in the hands of the programmer. If, for example, you wrote this program as one big MAIN procedure without any of the smaller subroutine structures illustrated, you would produce a program that would be very difficult indeed to maintain. When you define separate elements of the main program as individual procedures not only is the program easier to troubleshoot it is also easy to add extra bits here and there when then are needed without major surgery!

Where were we? Oh yes, line 107. This line serves to invert the variable FLASH_TOGGLE. The state of FLASH_TOGGLE will determine if we are going to put the message up on the display or if we are going to send a blank line to the display. Each time FLASH_COUNT reaches zero we will be toggling the state of FLASH_TOGGLE back and forth to give a 50% duty cycle on the flash rate. Line 108 then tests the state of FLASH_TOGGLE to determine if line 109 or line 110 is to be executed. Line 109 will print the message on the screen and line 110 will remove the message from the screen.

## 9.09.00 MULTI-TASKING PROGRAMS (continued)

Lines 116 through 150 form the procedure OP_INPUT which controls the operator input line. Line 116 initializes a pass counter to zero. Line 116 sends a carriage return to the display. Line 118 finds out if the cursor has been moved by somebody else and if it has begins executing lines 119 through 123 which restore it to the position the operator left it in. Line 120 resets the CURSOR_MOVE_FLAG. Line 121 puts the cursor at the left margin and line 122 moves the cursor to the right by the current value of COL_COUNT. This serves to put the cursor back to the place the operator last had it when this task is selected.

Lines 125 through 149 form the main body of this procedure. If you look closely at line 149 and see what value we have assigned to PCOUNT (200) it would appear that we are breaking one of our own rules. This procedure will hang on to the processor for 199 iterations of the loop between lines 125 and 149. Why? I hear you ask! PRIORITY I answer! This procedure is one of several tasks which manipulate the display cursor up and down the screen AND this task is expecting input from an operator. Since the operator needs to see the cursor in order to manipulate his data line we must take steps to ensure that the cursor remains on the operator input line the majority of the time. If we did not assign this task a high priority by making the processor execute it 199 times the program would still work but the operator would not be able to see the cursor. Try changing the constant PCOUNT=200 to PCOUNT=1 and see what happens.

Line 126 uses the IOSUBS procedure GETKEY to poll the keyboard. GETKEY, unlike GETCHAR, does not hang onto the processor. It will return with with a keyboard code if a key was hit or a NULL ($00) if one wasn't. Line 121 then compares the incoming character with the CONSTANTS LOLIMIT and HILIMIT to determine whether the code is acceptable or not. If the incoming character passes the limit test lines 128 through 135 will be executed, otherwise control will be passed to line 137. Line 129 compares the variable COL_COUNT, which represents the current column the cursor is in, with the constant COL_LIMIT. Once again we used constants to define three potentially variable items: LOLIMIT, HILIMIT and COL_LIMIT to make our program more adaptable. If COL_COUNT is less than COL_LIMIT the operator still has room on his display to enter characters so lines 130 through 133 will be executed. Line 131 echos the incoming character back to the video display. Line 132 bumps the column counter. If line 129 found that the operator was already at the right column limit (entering characters moves the cursor to the right!) line 134 would send a BELL code to the video display, which, if it supports the BELL code $07, will 'beep' the operator to warn him that he is at his display limit. All he can do at this point is to back-space or keep hitting characters and getting a 'beep' for his efforts.

Lines 137 through 145 handle the situation when the incoming character is a backspace. The structure of line 121 assumes that backspace will not be within the range of LOLIMIT to HILIMIT. If there was any chance of the backspace code falling into this range we could simply add another argument as follows:

```
IF INCHAR >= LOLIMIT .AND INCHAR <= HILIMIT
   .AND INCHAR <> NON_DESTRUCTIVE_BACKSPACE
   THEN BEGIN
```

9.09.00 MULTI-TASKING PROGRAMS (continued)

If the incoming character is verified to be a back-space on line 137, line 138 then tests to see if COL_COUNT is at the left margin which is signified by COL_COUNT being equal to one. If this is the case line 145 will be executed and another 'beep' sent to the operator. If the COL_COUNT indicates that the cursor is anywhere else on the display lines 140 through 144 will be executed. Lines 140 through 142 are identical to the 'rub-out' routine discussed in the section on 'GOTO' (9.03.02) and serve to back up the cursor and rub-out the character just back-spaced over. Line 143 then decrements the column counter to keep track of the back spaces.

Line 146 bumps the pass counter PASSES. Line 147 tests to see if the incoming character is equal to the constant ESCAPE or the constant _RETURN. If it is the OP_INPUT procedure will be terminated immediately via the BREAK statement on line 148. Line 149 completes the loop and keeps passing control back to line 125 until PASSES is equal to the constant PCOUNT thus holding onto the processor for several passes in order to establish this event as a high priority task.

Now that we have defined the various procedures in this program lets see how they fit together to give us our five concurrent tasks:

1. We have DELAY_TIMERS which either resets or decrements the three delays used to control the rate that the three counters are decremented/incremented.

2. We have COUNT which either increments or decrements the three counters depending on the state of the delay count and the state of the toggle flag.

3. We have DISPLAY_COUNTS which will display each of the counter values AS it is decremented. If the count value has not changed since the last screen update the information on the screen will be left alone.

4. We have FLASH_MESSAGE which either sends a message or a blank line to the display at the end of each delay interval which it handles internally.

5. We have the high priority task OP_INPUT which controls an edited line of information on the screen.

Armed with all of these procedures (which can be tested one at a time if problems arise) we now enter the realm of the main procedure, appropriately named MAIN. The first section of MAIN between lines 157 and 167 deals with the initialization of all of the variables used in the program, a very important step! Lines 157 through 159 should be pretty obvious.

As promised earlier we are now going to initialize all of the READ/WRITE vectors from a set of READ-ONLY data tables. The data tables are declared at lines 152 through 154. They could just as easily been declared in the early part of the program, say at line 18, just after the constants. In practice this is where they really should have been declared if one suspects that they may require alteration or fiddling at a later date. The use of data tables to initialize every element of a vector to the same value is a bit of overkill to say the least BUT it does give one the opportunity to vary the value assigned to each element of the vector should the need arise. Just before we enter the REPEAT...UNTIL loop between lines 162 and 167 we initialize the index counter INDEX to zero at line 161. Once inside the REPEAT...UNTIL loop we have the following simplification on the first iteration:

```
DNUMBER(0)=NUMBRINIT(0);
DCOUNT(0)=DELAYINIT(0);
TOGGLE(0)=TOGGLINIT(0);
```

## 9.09.00 MULTI-TASKING PROGRAMS (continued)

This structure simply plucks the data out of the data table and installs it the appropriate vector. The second and third iterations are just a repetition of the process for the remaining elements of the vectors.

Line 169 performs 10 carriage return - line feeds to clear an area on the video display for the information to be presented by the program. Line 170 immediately moves the cursor up 4 lines. This position (the operator input line) will be the base position for all subsequent cursor movements.

We now enter the meat of the program. Simple isn't it? Generally speaking providing that you use used STRUCTURED PROGRAMMING throughout your software development jobs most, if not all, MAIN procedures will look as simple as this! All of the complexity should be placed into individual procedures which may be run on their own, and, perhaps more importantly, DEBUGGED on their own. The LAST thing you want to do is write a single large procedure and TRY to debug it!

Lines 179 through 190 embody the main program which as REPEAT...FOREVER suggests is designed to run forever. Just before we enter this part of the program note that a label 'OP_INPUT_NEW_LINE:' has been declared at line 172. This will be used for a subsequent GOTO instruction. Line 174 returns the cursor to the left margin. Line 175 then prints a message out to the display on the operator input line (remember we left the cursor on this line). Line 176 resets the cursor column counter to indicate that the cursor is in the left column. Line 177 then puts the cursor where we just said it was.

We now enter the body of the REPEAT...FOREVER loop. The only method of escaping from the program is provided in lines 185 and 188. If the RETURN key is hit (it will be detected during OP_INPUT and assigned to the global variable INCHAR) line 185 will be true and therefore line 186 will be executed. Line 186 passes control back to the label 'OP_INPUT_NEW_LINE' where the current line is replaced by as indicated in line 175. The main body of the program is then re-entered.

If you hit the ESCAPE key (it is also detected within OP_INPUT) line 188 will be true the program will be terminated by the BREAK statement in line 189. Line 175 then moves the cursor down 4 lines. The omission of the last ENDPROC will cause PL/9 to JUMP to the FLEX warm entry point ($CD03). If you put an ENDPROC at line 176 the program would CRASH when the escape key is hit! If you put an ENDPROC at the end of a program that can be terminated, i.e. you have provided an escape from the REPEAT FOREVER loop, YOU, the programmer must direct the program to some sensible place, back into FLEX, the System Monitor, etc;

Now to define the various cursor movements as promised earlier. As we mentioned the base position for the video display cursor is on the operator input data line of the display. When FLASH_MESSAGE is called in line 180 it will move the cursor up two lines if, and only if, the message line is to be updated. Once the message line is updated the cursor is moved back to the operator input line. At this point the cursor may not be in the appropriate column but since FLASH_MESSAGE has set a flag to tell OP_INPUT it moved the cursor OP_INPUT will be able to restore it to the correct column. When DISPLAY_COUNTS is called in line 181 it to will move the cursor only when required. It will also restore the cursor to the operator input line and set the global CURSOR_MOVE_FLAG. When OP_INPUT is finally called in line 183 it will restore the cursor to the appropriate column if it finds the CURSOR_MOVE_FLAG set.

We hope this rather long dissertation has given you some insights into the structures required to write multi-tasking software without recourse to hardware interrupts. The two areas where hardware interrupts are mandatory are when your polling loop is too slow to capture a real-time event or when you must input or output data at a steady rate.

## 9.09.01 A MULTI-TASKING KERNEL IN PL/9

If this section looks familiar it is because it appeared in the May 1984 issue of '68 Micro Journal. This section is not identical to the article, however, as it uses the STACK key word that is now available in PL/9.

Have you ever written a program that does two or more jobs at the same time? If you have, then you've probably run across the problem of resource allocation, going something like this:

The main program is cycling round, looking at the keyboard to see whether you want it to do anything. When you hit a key it goes into a flurry of activity while it sorts out your request. If this activity only means updating some variables or doing a few calculations, then the program will get back to the keyboard before your finger has even released the key. But suppose that you've asked it to set in motion some complex I/O function, such as ramping up the voltage on a D/A converter, which requires the program to change a value every time a timer overflows? In a single-task program, your keyboard will go dead until that task has finished. The problem is compounded if there are several D/A converters, each ramping at different rates.

OK, so it's possible to write a single-task program, like the program described in the preceeding section, that can handle this sort of job. The problem then is that each of the tasks gets thoroughly bound up with each of the others, to the extent that maintaining the program becomes a real pain. What would be really nice would be to write each of the tasks separately and tell the CPU to execute all of them in parallel.

Now I know that OS-9 is out there just waiting to be used for this sort of job. OS-9 is great if the drivers already exist for the I/O device you have to handle, but the job of writing special interface drivers is just too much for a lot of people. Furthermore, you don't actually know when OS-9 is going to suspend a task and start up another, which makes interlocking global variables a bit tricky. However, if you can handle OS-9 and if OS-9 can handle your application, and you don't mind having to integrate a copy of the OS-9 kernal (with the associated license fees) in each and every product you sell, use it by all means.

What we are going to describe in this section is a simple technique for writing programs that are divided into separate tasks, all operating concurrently, with as much or as little interaction between them as you wish, and all without the use of interrupts of any kind!.

The heart of the technique is a routine called SWITCH, whose job it is to change from one task to another. Suppose that three tasks are set up; a main program and two secondary jobs called TASK1 and TASK2. Each must contain sufficient calls to SWITCH to avoid hogging the CPU. When SWITCH is called, if the main program is running then it is suspended and TASK1 is resumed; if TASK1 is running then TASK2 is resumed and if TASK2 is running then the main program is resumed. Each task can run for as long as it likes, then a call to SWITCH gives the next task some CPU time in a round-robin fashion.

The advantage of this scheme is that unlike one that uses timer interrupts there is no need to guard against one process interrupting another and interfering with a variable that is being worked with. If you want to mess with a variable you do it, only handing control over to the next task when you're good and ready.

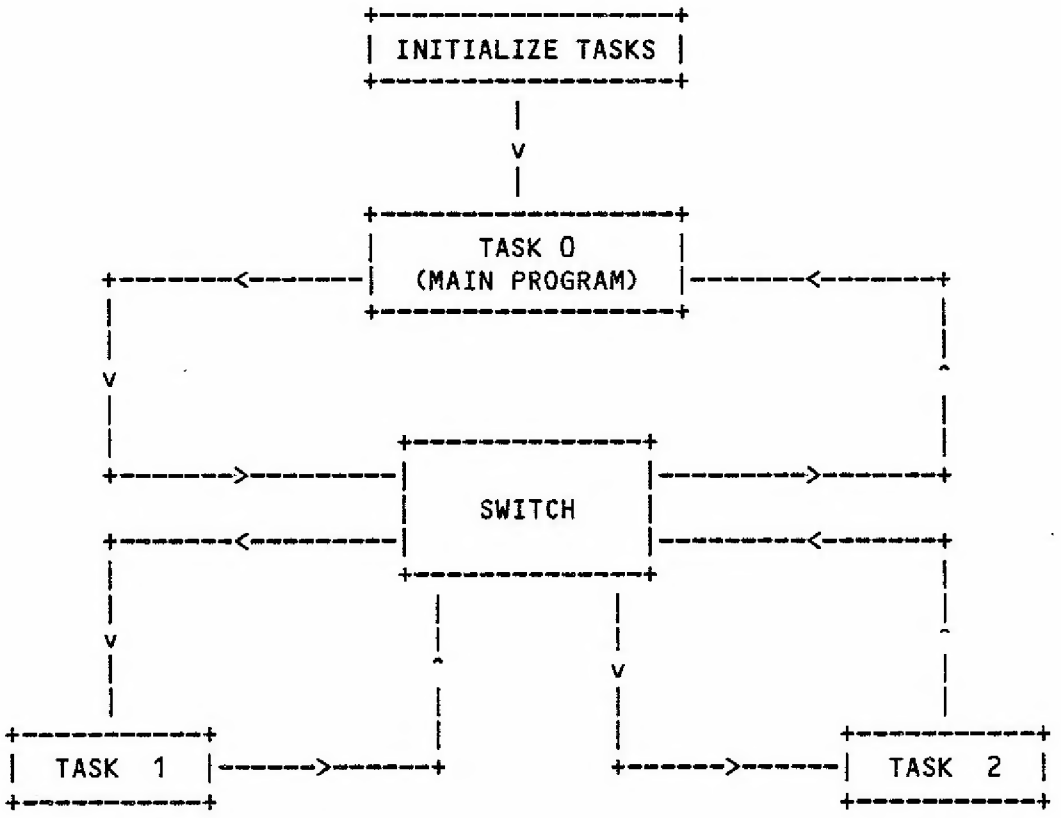
It is important to note that any program loop that grabs the processor (like a repeat ... until loop) should include a call to SWITCH unless you actually want to suspend all other tasks while the current program loop is being executed.

## 9.09.01 A MULTI-TASKING KERNEL IN PL/9 (continued)

The way the system works is quite simple. As part of your program's scratch space you allocate a stack for each of the secondary tasks. The amount of space you allocate to each stack is determined by the subroutine nest level, local variable allocation and whether there are any recursive procedures being used. It is a good idea to be generous with stack allocation as if one tasks stack collides with another the result will be disaster! The starting address of each secondary task is placed at the very top of its reserved stack area. A two-byte pointer to this stored address is also kept for each task and one is reserved for the main program. Lastly, a byte is reserved that records which task is currently active. This byte is intially set to zero, the main program.

Having done this initialisation, the main program can now start. At some point it is necessary to start up the multi-tasking system. A good place to put a call to SWITCH is in the program loop that polls the keyboard to see if a key has been pressed. This ensures that as long as no key is operated a steady supply of calls to SWITCH will be generated.

The first call to SWITCH causes the stack pointer to be saved in the two-byte pointer reserved for the main program. It is then re-loaded with the contents of task 1's pointer and a return-from-subroutine is executed. This causes the address of task 1 to be put into the program counter, which starts it executing. At some point it too calls SWITCH, which in the same way saves the current status and starts up task 2. When task 2 calls SWITCH the main program is resumed at the point it left off. This structure is illustrated in the following diagram:




Note the most important feature of the diagram, that once the multi-tasking has begun it goes on forever. None of the tasks can be allowed to finish or the system will stop. In practice, this restriction can be overcome by allocating flags that tell SWITCH which tasks are currently active. In this way, new tasks can be introduced when they are needed. The alternative is for an inactive task to stay in an endless loop, generating calls to SWITCH.

9.09.01 A MULTI-TASKING KERNEL IN PL/9 (continued)

The following PL/9 program should illustrate the use of this technique. Firstly, the necessary scratch variables. These are declared here as GLOBAL variables but they could equally well be declared using the AT statement. Since this program will actually run, I'll put ORIGIN and STACK statements first:

```
ORIGIN = $8000; STACK = *;   /* I.E. STACK = $8000 */

CONSTANT NO_OF_TASKS=3;    /* TASKS 0, 1 AND 2 */

GLOBAL

   BYTE DUMMY1(100):      /* LENGTH OF STACK FOR TASK 1 */
   INTEGER STACK1:

   BYTE DUMMY2(100):      /* LENGTH OF STACK FOR TASK 2 */
   INTEGER STACK2:

   INTEGER TASK_TABLE(NO_OF_TASKS):
   BYTE TASK;
```

The user's own variables can be included anywhere in this list. Note that each of the secondary tasks has a stack area (declared using dummy variables), and space for the return address (variables stack1 and stack2). The length of each stack (100 bytes) is completely arbitrary. The amount of stack is determined by how much procedure nesting takes place, the amount of local variables being used in the various procedures and if any recursive procedures are used. If you don't allocate sufficient space you will find out soon enough as the program will crash as soon as the stacks collide with one another. The main program has a pointer table (task_table) and a byte variable (task) that records which task is currently active.

The SWITCH procedure must be near the start of the program, since it's called by other procedures It looks like this:

```
PROCEDURE SWITCH;
   TASK_TABLE(TASK) = STACK;
   TASK = TASK + 1;
   IF TASK = NO_OF_TASKS THEN TASK = 0;
   STACK = TASK_TABLE(TASK);
ENDPROC;
```

Now comes the body of the program, including the secondary tasks. The first of these is called TASK_1 (but you can use any name you wish). Its job is to cause the terminal bell to ring at regular intervals. Note the call to SWITCH in the inner loop:

```
PROCEDURE TASK_1: INTEGER COUNT;
   REPEAT
      COUNT = 1000;
      REPEAT
         SWITCH;
         COUNT = COUNT - 1;
      UNTIL COUNT = 0;
      ACCA = 7;          /* ASCII "BELL" */
      CALL $CD18;        /* FLEX "PUTCHR" */
   FOREVER;
ENDPROC;
```

## 9.09.01 A MULTI-TASKING KERNEL IN PL/9 (continued)

The second task, called task_2, sends to the terminal the complete ASCII character set from $20 through $7D, over and over again:

```
PROCEDURE TASK_2: BYTE CHAR;
   REPEAT
      CHAR = $20;
      REPEAT
         SWITCH;
         ACCA = CHAR;
         CALL $CD18;
         CHAR = CHAR + 1;
      UNTIL CHAR = $7E;
   FOREVER;
ENDPROC;
```

The next routine forms part of the main program. It echoes to the printer every key that is typed. It is so designed as to generate calls to SWITCH while it is waiting for a key:

```
PROCEDURE ECHO_TO_PRINTER: BYTE CHAR;
   CALL $CCCO;                  /* INITIALISE PRINTER */
   REPEAT
      REPEAT
         SWITCH;
         CALL $CD4E;            /* FLEX "STAT" */
      UNTIL (CCR AND 4) = 0;
      CALL $CD15;               /* FLEX "GETCHR" */
      CHAR = ACCA;
      CALL $CCE4;               /* OUTPUT TO PRINTER */
   UNTIL CHAR = $1B;            /* WAIT FOR <ESCAPE> */
ENDPROC;
```

Lastly we have the main program, which as its first job must set up the multi-tasking system:

```
PROCEDURE MAIN;

   TASK=0;                      /* TASK 0 (MAIN PROGRAM) ACTIVE */

   TASK_TABLE(1)=.STACK1;       /* ADDRESS OF STACK1 -> TABLE */
   STACK1=.TASK_1;              /* ADDRESS OF TASK 1 -> STACK1 */

   TASK_TABLE(2)=.STACK2;       /* ADDRESS OF STACK2 -> TABLE */
   STACK2=.TASK_2;              /* ADDRESS OF TASK 2 -> STACK2 */

   ECHO_TO_PRINTER;
```

9.09.01 A MULTI-TASKING KERNEL IN PL/9 (continued)

All that "main" does after initialising the multi-tasking system is to call "echo_to_printer" to run the primary task. In this example we have placed the primary task in a 'repeat ... until' loop which gives us an oportunity to make a smooth return to FLEX whenever the ESCAPE key is hit. In a practical dedicated program the primary task is usually within a 'repeat ... forever' (endless) loop. Since the object of this section was to present a program that the majority of FLEX users could type in, compile and run we restricted it to calls through 'FLEX'. The only assumption that we have made about the readers hardware is that he or she has a printer!

A multi-tasking system of this kind is designed to cope with real-time applications where actions are taken dependant upon some outside event, and it's rather difficult to produce a realistic example when the only outside event is the keyboard and the only I/O device is a printer and terminal!

NOTE 1: You should turn off the TTYSET PAUSE and load your printer drivers before starting the program.

NOTE 2: Your version of FLEX must preserve the registers as stated in the programmers manual or the program is likely to crash.

If you are used to working with multi-tasking software or interrupt driven software you are probably saying to yourself that the basic structure described in this section should also work with interrupts. YOU ARE CORRECT! There are, however, a few additional rules to follow. We will present these in the next section.

## 9.09.02 MULTI-TASKING WITH INTERRUPTS (SWI)

Before we dive into the discussion of this topic we are going to simplify the example program used in the previous section so that it will be easier to define the differences between the various structures required to implement interrupt driven multi-tasking programs.

If you refer to the listing on the following two pages you will see the simplification referred to above (Multi-Tasking Demonstration Program 2). This multitasking program has three tasks, defined as 'TASK_0', 'TASK_1' and 'TASK_2' instead of the original 'ECHO_TO_PRINTER', 'TASK_1' and 'TASK_2'. The tasks themselves have also been greatly simplified: 'TASK_0' outputs the ASCII character 'A', 'TASK_1' outputs the ASCII character 'B' and 'TASK_2' outputs the ASCII character 'C'. Not particularly clever but all we want is an illustration of the structures involved.

Start-up is achieved in much the same manner as the original program; the return addresses of the second and third tasks are placed at the top of their respective stacks and the 'TASK_TABLE' initialized to point to the stacks for the second and third tasks. 'TASK_0' is then called and the muti-tasking operation begins. As each task ends it makes a call to switch which 'retires' the task being executed, saving its return address and stack pointer in the process. The next task is then started up, when it ends it is retired and the next task initiated. And on it goes in round-robin fashion. The key to successful operation is:

A. Each task must be within a 'REPEAT ... FOREVER' loop.
B. Each task must not grab the processor for any appreciable length of time.

This type of multi-tasking structure has one great advantage:

YOU ARE IN COMPLETE CONTROL OF HOW LONG A TASK RUNS

This means that there is never any problem of interlocking global variables.

Multi-Tasking Demonstration Program 3, four pages along, is the exact same program but instead of having a normal procedure called 'SWITCH' we are using the reserved procedure name 'SWI' (Software Interrupt) to do the same job. Notice that the contents of 'SWI' and 'SWITCH' are identical. The only difference between program '3' and program '2' is the way the stacks and the

stack pointers are initialized. Lets examine the stack initialization closely as it is important that you understand the fundamental differences required when you intend to multi-task with the use of interrupts. The same basic stack initialization is used regardless of the source of the interrupt: SWI, SWI2 or SWI3 (user generated) or IRQ, FIRQ or NMI (hardware generated).

Line 69 saves the stack pointer associated with the stack for TASK_0. Line 71 puts the address of TASK_1 at the top of the stack associated with TASK_1. Line 73 preloads the stack and moves the stack pointer down by the appropriate number of bytes. Line 74 saves the the stack pointer in the TASK_TABLE. The major difference between this initialization sequence and the initialization sequence associated with program 3 is the way we manipulate the stack pointer BEFORE we save its value in the TASK_TABLE. WHY? In program 2 only an RTS (PULS PC) instruction is executed at the end of SWITCH and hence only the return address need be on the stack. In program 2 an RTI (PULS CC,D,DP,X,Y,U,PC) instruction is executed at the end of SWI and hence a complete set of register values must be present on the stack before they are pulled. Since we are not using hardware interrupts there is no need to alter the state of the 'CC' before we pre-load the stack, nor is there any need to preserve the 'Y' register (global pointer) as you are in complete control of the call to 'NEXT_TASK'.

## 9.09.02 MULTI-TASKING WITH INTERRUPTS (SWI) (continued)

```
0001 /* Multi-Tasking Demonstration Program 2 */
0002
0003 ORIGIN = $8000;
0004 STACK = *;
0005
0006 CONSTANT NO_OF_TASKS = 3;    /* Tasks 0, 1 and 2 */
0007
0008 GLOBAL
0009    BYTE DUMMY1(100):     /* Length of stack for task 1 */
0010    INTEGER STACK1:
0011
0012    BYTE DUMMY2(100):     /* Length of stack for task 2 */
0013    INTEGER STACK2:
0014
0015    INTEGER TASK_TABLE(NO_OF_TASKS):
0016    BYTE TASK;
0017
0018
0019 PROCEDURE SWITCH;
0020    TASK_TABLE(TASK) = STACK;
0021    TASK = TASK + 1;
0022    IF TASK = NO_OF_TASKS
0023       THEN TASK = 0;
0024    STACK = TASK_TABLE(TASK);
0025 ENDPROC;
0026
0027
0028 PROCEDURE OUTCHAR(BYTE CHAR);
0029    ACCA = CHAR;
0030    CALL $CD18;
0031 ENDPROC;
0032
0033
0034 PROCEDURE TASK_0;
0035    REPEAT
0036       OUTCHAR('A);
0037       SWITCH;
0038    FOREVER;
0039 ENDPROC;
0040
0041
0042 PROCEDURE TASK_1;
0043    REPEAT
0044       OUTCHAR('B);
0045       SWITCH;
0046    FOREVER;
0047 ENDPROC;
0048
0049
0050 PROCEDURE TASK_2;
0051    REPEAT
0052       OUTCHAR('C);
0053       SWITCH;
0054    FOREVER;
0055 ENDPROC;
```

## 9.09.02 MULTI-TASKING WITH INTERRUPTS (SWI) (continued)

```
0056
0057
0058
0059 PROCEDURE MAIN;
0060
0061    TASK = 0;                  /* Task 0 (main program) active */
0062
0063    STACK1 = .TASK_1;          /* put address of task 1 on top of stack       */
0064    TASK_TABLE(1) = .STACK1; /* save base of stack in task table */
0065
0066    STACK2 = .TASK_2;          /* put address of task 2 on top of stack       */
0067    TASK_TABLE(2) = .STACK2; /* save base of stack in task table */
0068
0069
0070    TASK_0;
```

NOTE: Ensure that you set up the TTYSET 'WD' to the width of your terminal screen (e.g. WD=80) and turn off the PAUSE function (PS=N) before you run any of the programs in this and the following sections.

## 9.09.02 MULTI-TASKING WITH INTERRUPTS (SWI) (continued)

```
0001 /* Multi-Tasking Demonstration Program 3 */
0002
0003 ORIGIN = $8000;
0004 STACK = *;
0005
0006 CONSTANT NO_OF_TASKS = 3;    /* Tasks 0, 1 and 2 */
0007
0008 GLOBAL
0009    BYTE DUMMY1(100):     /* Length of stack for task 1 */
0010    INTEGER STACK1:
0011
0012    BYTE DUMMY2(100):     /* Length of stack for task 2 */
0013    INTEGER STACK2:
0014
0015    INTEGER TASK_TABLE(NO_OF_TASKS):
0016    BYTE TASK:
0017    INTEGER STACK_SAVE;
0018
0019
0020 PROCEDURE SWI;
0021    TASK_TABLE(TASK) = STACK;
0022    TASK = TASK + 1;
0023    IF TASK = NO_OF_TASKS
0024       THEN TASK = 0;
0025    STACK = TASK_TABLE(TASK);
0026 ENDPROC;
0027
0028
0029 PROCEDURE OUTCHAR(BYTE CHAR);
0030    ACCA = CHAR;
0031    CALL $CD18;
0032 ENDPROC;
0033
0034
0035 PROCEDURE NEXT_TASK;
0036   GEN $3F;   /* SWI */
0037 ENDPROC;
0038
0039
0040 PROCEDURE TASK_0;
0041    REPEAT
0042       OUTCHAR('A);
0043       NEXT_TASK;
0044    FOREVER;
0045 ENDPROC;
0046
0047
0048 PROCEDURE TASK_1;
0049    REPEAT
0050       OUTCHAR('B);
0051       NEXT_TASK;
0052    FOREVER;
0053 ENDPROC;
0054
0055
```

## 9.09.02 MULTI-TASKING WITH INTERRUPTS (SWI) (continued)

```
0056 PROCEDURE TASK_2;
0057    REPEAT
0058       OUTCHAR('C);
0059       NEXT_TASK;
0060    FOREVER;
0061 ENDPROC;
0062
0063
0064
0065 PROCEDURE MAIN;
0066
0067    TASK = 0;              /* Task 0 (main program) active */
0068
0069    STACK_SAVE = STACK;    /* save main program stack pointer */
0070
0071    STACK1 = .TASK_1;      /* put address of task 1 on top of stack */
0072    STACK = .STACK1;       /* aim stack at stack area for task 1 */
0073    GEN $34,$7F;           /* pre-load stack 1 (PSHS CC,D,DP,X,Y,U) */
0074    TASK_TABLE(1) = STACK; /* save base of stack in task table */
0075
0076    STACK2 = .TASK_2;      /* put address of task 2 on top of stack */
0077    STACK = .STACK2;       /* aim stack at stack area for task 2 */
0078    GEN $34,$7F;           /* pre-load stack 2 (PSHS CC,D,DP,X,Y,U) */
0079    TASK_TABLE(2) = STACK; /* save base of stack in task table */
0080
0081    STACK = STACK_SAVE;    /* re-aim stack back to task 0 stack area *      /
0082
0083    TASK_0;
```

## 9.09.03 MULTI-TASKING WITH INTERRUPTS (IRQ)

Multi-tasking Demonstration Program 4, two pages along, is a progression of program 3. The former uses 'IRQ' as the task switching mechanism whilst the latter used 'SWI'. Obviously the major difference is going to be the inclusion of a hardware device that we can program to generate interrupts. The Motorola MC6840 is ideally suited to this purpose as the programmer has a degree of control over the rate at which interrupts are generated that is not possible with other devices with the possible exception of the Rockwell R6522 VIA.

Aside from the hardware content of this program (between lines 24 and 38) the major difference between program 3 and this one is once again in the initialization sequence. This time we save the 'Y' register (global pointer) in line 93. This is necessary as you will not have any idea of what code the processor is executing, and hence the state of the 'Y' register, when an interrupt occurs. If you wish to access global variables from within an interrupt procedure you must save and restore 'Y'. See section 9.11.00 for futher details.

Line 96 clears the IRQ flag of the CCR. Since we have not initialized the interrupt generating device at this stage this will not have any effect on the execution of the program. Lines 100 through 103 are identical in function to lines 71 through 74 in program 3. Line 112 starts the IRQ generator and line 114 gets the ball rolling long before the first IRQ occurs.

Other points to note are line 42 which restores the 'Y' register to point at the base of the global variables. Likewise if you have used 'DPAGE' to aim the 'DP' register at some area of 'AT' variables you should restore the 'DP' within the IRQ service routine if you are going to make any reference to them during the IRQ service interval. Line 50 services the MC6840 and forces it to release the IRQ line before we terminate the IRQ service routine.

One danger when working in a multi-tasking environment driven by interrupts is that you must interlock all global variables. Since PL/9 procedures that use local variables and/or are passed/return variables are inherently re-entrant there is never any danger in this area. The danger exists in interrupting a procedure that is working on a global variable (including 'AT' variables which obviously includes I/O devices as well) that will be used by another procedure or manipulated by another procedure. Suppose, for example, that TASK_0's job was to read a 16 channel A/D converter and update a global variable data table and that TASK_1's job was to analyze the data in the data table and take one of several courses of action depending on what it finds there. Clearly if TASK_0 is interrupted before its job is done the result could be disaster. One way of preventing this is to mask the IRQ flag in the CCR whilst critical global variables are being manipulated or decisions that are based on thier contents are taken. Another approach is to set a flag whilst a critical global access is taking place. This flag is interrogated by the interrupt service routine and if set simply refuses to switch to the next task until it is cleared. As soon as the procedure that is manipulating the global variables is finished it clears the flag and normal operation resumes.

The two approaches are equally valid. The former will result in the period between interrupts varying which means that the interrupt 'tick' will not occur on a regular basis. The latter approach ensures that the interrupt 'tick' will occur at a steady rate which may be useful if the interrupt 'tick' is being used for system time-keeping.

We mask and unmask the IRQ flag between lines 54 and 58. This enables you to use the <ESCAPE> key to stop output and prevents the FLEX 'PUTCHR' routine, which is not re-entrant, from being interrupted.

## 9.09.03 MULTI-TASKING WITH INTERRUPTS (IRQ) (continued)

Note how we suspend each of the tasks (lines 70, 78 and 86). Each of the tasks will execute until it hits 'SUSPEND_TASK'. When 'SUSPEND_TASK' is called and the 'CWAI' instruction is encountered by the MC6809 it will simply stop dead in its tracks and wait for the IRQ to occur. At this point the IRQ service routine will switch to the next task.

THIS IS VERY WASTEFUL OF PROCESSING TIME

BUT

IT ENSURES THAT THE INTERRUPT 'TICK' OCCURS AT A STEADY RATE

Obviously it would be advantageous to have better control of the termination of a task and its priority in relation to other tasks. This will be discussed in the analysis of Program 5 which is in the next section.

## 9.09.03 MULTI-TASKING WITH INTERRUPTS (IRQ) (continued)

```
0001 /* Multi-Tasking Demonstration Program 4 */
0002
0003 ORIGIN = $8000;
0004 STACK = $7FFE;  /* 'Y' will be saved at $7FFE/F */
0005
0006
0007 AT $E030:BYTE TIMER;          /* base address of MC6840 timer */
0008
0009 CONSTANT NO_OF_TASKS = 3;     /* Tasks 0, 1 and 2 */
0010
0011
0012 GLOBAL
0013    BYTE DUMMY1(100):          /* Length of stack for task 1 */
0014    INTEGER STACK1:
0015
0016    BYTE DUMMY2(100):          /* Length of stack for task 2 */
0017    INTEGER STACK2:
0018
0019    INTEGER TASK_TABLE(NO_OF_TASKS):
0020    BYTE TASK:
0021    INTEGER STACK_SAVE;
0022
0023
0024 PROCEDURE INITIALIZE_TIMER;
0025    TIMER(1) = 0;     /* select control register # 3 */
0026    TIMER(0) = $42;   /* enable IRQ, use '/E' for clock source */
0027    TIMER(6) = $4E;
0028    TIMER(7) = $20;   /* ($4E20 = 20000) 100 Hertz w/2 MHz CPU */
0029    TIMER(1) = 1;     /* select control register # 1 */
0030    TIMER(0) = 1;     /* reset timer */
0031    TIMER(0) = 0;     /* start IRQ's */
0032 ENDPROC;
0033
0034
0035 PROCEDURE SERVICE_TIMER;
0036    ACCB = TIMER(1);
0037    ACCB = TIMER(6);  /* clear IRQ */
0038 ENDPROC;
0039
0040
0041 PROCEDURE IRQ;
0042    GEN $10,$BE,$7F,$FE;  /* LDY $7FFE (recover global pointer) */
0043                          /* also recover 'DP' if you use DPAGE! */
0044
0045    TASK_TABLE(TASK) = STACK;
0046    TASK = TASK + 1;
0047    IF TASK = NO_OF_TASKS
0048       THEN TASK = 0;
0049    STACK = TASK_TABLE(TASK);
0050    SERVICE_TIMER;
0051 ENDPROC;
0052
0053
0054 PROCEDURE OUTCHAR(BYTE CHAR);
0055    CCR = CCR OR $10;   /* kill IRQ */
0056    ACCA = CHAR;
0057    CALL $CD18;
0058    CCR = CCR AND $EF;  /* enable IRQ */
0059 ENDPROC;
```

9.09.03 MULTI-TASKING WITH INTERRUPTS (IRQ) (continued)

```
0060
0061
0062 PROCEDURE SUSPEND_TASK;
0063    GEN $3C,$EF;   /* CWAI #$EF */
0064 ENDPROC;
0065
0066
0067 PROCEDURE TASK_0;
0068    REPEAT
0069       OUTCHAR('A);
0070       SUSPEND_TASK;
0071    FOREVER;
0072 ENDPROC;
0073
0074
0075 PROCEDURE TASK_1;
0076    REPEAT
0077       OUTCHAR('B);
0078       SUSPEND_TASK;
0079    FOREVER;
0080 ENDPROC;
0081
0082
0083 PROCEDURE TASK_2;
0084    REPEAT
0085       OUTCHAR('C);
0086       SUSPEND_TASK;
0087    FOREVER;
0088 ENDPROC;
0089
0090
0091 PROCEDURE MAIN;
0092
0093    GEN $10,$BF,$7F,$FE;   /* STY $7FFE (just above stack) */
0094
0095    TASK = 0;              /* Task 0 active */
0096    CCR = CCR AND $EF;     /* enable IRQ in pre-load CCR */
0097
0098    STACK_SAVE = STACK;    /* save task 0 program stack pointer */
0099
0100    STACK1 = .TASK_1;      /* put address of task 1 on top of stack */
0101    STACK = .STACK1;       /* aim stack at stack area for task 1 */
0102    GEN $34,$7F;           /* pre-load stack 1 (PSHS CC,D,DP,X,Y,U) */
0103    TASK_TABLE(1) = STACK; /* save base of stack in task table */
0104
0105    STACK2 = .TASK_2;      /* put address of task 2 on top of stack */
0106    STACK = .STACK2;       /* aim stack at stack area for task 2 */
0107    GEN $34,$7F;           /* pre-load stack 2 (PSHS CC,D,DP,X,Y,U) */
0108    TASK_TABLE(2) = STACK; /* save base of stack in task table */
0109
0110    STACK = STACK_SAVE;    /* re-aim stack back to task 0 stack area *      /
0111
0112    INITIALIZE_TIMER;
0113
0114    TASK_0;
```

## 9.09.04 BETTER CONTROL OF AN IRQ DRIVEN MULTI-TASKING PROGRAM

Program 5 introduces some additional control over task switching in the form of giving each task a pre-determined number of IRQ time slices. These are directly related to the tasks priority within the system, the higher the priority the more time is allocated to it.

The major differences between this program and program 4 is the way the IRQ service routine is structured and the absense of any task terminating function (CWAI).

Each time the IRQ service routine is entered, by interrupting the current task via the hardware timer, the state of the 'TASK_DURATION_COUNTER' will be evaluated. If the counter is zero the next task will be selected and the counter loaded with the count (priority) associated with that task. If the counter is not zero the counter is decremented by one and control is returned to the interupted program.

It should be obvious that a running program can terminate itself by setting the 'TASK_DURATION_COUNTER' to zero.

This gives us a bit more control but this may not be enough for many applications. The final program, discussed in the following section, will illustrate several additional control mechanisms.

## 9.09.04 BETTER CONTROL OF AN IRQ DRIVEN MULTI-TASKING PROGRAM (continued)

```
0001 /* Multi-Tasking Demonstration Program 5 */
0002
0003 ORIGIN = $8000;
0004 STACK = $7FFE;  /* 'Y' will be saved at $7FFE/F */
0005
0006
0007 AT $E030:BYTE TIMER;          /* base address of MC6840 timer */
0008
0009 CONSTANT NO_OF_TASKS = 3,       /* Tasks 0, 1 and 2 */
0010          TASK0_PRIORITY = 10,  /* 10 IRQ TIME SLICES */
0011          TASK1_PRIORITY = 20,  /* 20 IRQ TIME SLICES */
0012          TASK2_PRIORITY = 30;  /* 30 IRQ TIME SLICES */
0013
0014
0015 GLOBAL
0016    BYTE DUMMY1(100):          /* Length of stack for task 1 */
0017    INTEGER STACK1:
0018
0019    BYTE DUMMY2(100):          /* Length of stack for task 2 */
0020    INTEGER STACK2:
0021
0022    INTEGER TASK_TABLE(NO_OF_TASKS):
0023    BYTE TASK_PRIORITY_TABLE(NO_OF_TASKS):
0024    BYTE TASK, TASK_DURATION_COUNTER, CURRENT_TASK:
0025    INTEGER STACK_SAVE;
0026
0027
0028 PROCEDURE INITIALIZE_TIMER;
0029    TIMER(1) = 0;    /* select control register # 3 */
0030    TIMER(0) = $42;  /* enable IRQ, use '/E' for clock source */
0031    TIMER(6) = $4E;
0032    TIMER(7) = $20;  /* ($4E20 = 20000) 100 Hertz w/2 MHz CPU */
0033    TIMER(1) = 1;    /* select control register # 1 */
0034    TIMER(0) = 1;    /* reset timer */
0035    TIMER(0) = 0;    /* start IRQ's */
0036 ENDPROC;
0037
0038
0039 PROCEDURE SERVICE_TIMER;
0040    ACCB = TIMER(1);
0041    ACCB = TIMER(6);  /* clear IRQ */
0042 ENDPROC;
0043
0044
```

9.09.04 BETTER CONTROL OF AN IRQ DRIVEN MULTI-TASKING PROGRAM (continued)

```
0045 PROCEDURE IRQ;
0046    GEN $10,$BE,$7F,$FE;  /* LDY $7FFE (recover global pointer) */
0047                          /* also recover 'DP' if you use DPAGE! */
0048
0049    IF TASK_DURATION_COUNTER <> 0
0050       THEN TASK_DURATION_COUNTER = TASK_DURATION_COUNTER - 1;
0051
0052    IF TASK_DURATION_COUNTER = 0
0053       THEN BEGIN
0054            TASK_TABLE(TASK) = STACK;
0055            TASK = TASK + 1;
0056            IF TASK = NO_OF_TASKS
0057               THEN TASK = 0;
0058            TASK_DURATION_COUNTER = TASK_PRIORITY_TABLE(TASK);
0059            STACK = TASK_TABLE(TASK);
0060            END;
0061
0062    SERVICE_TIMER;
0063 ENDPROC;
0064
0065
0066 PROCEDURE OUTCHAR(BYTE CHAR);
0067    CCR = CCR OR $10;   /* kill IRQ */
0068    ACCA = CHAR;
0069    CALL $CD18;
0070    CCR = CCR AND $EF;  /* enable IRQ */
0071 ENDPROC;
0072
0073
0074 PROCEDURE TASK_0;
0075    REPEAT
0076       OUTCHAR('A);
0077    FOREVER;
0078 ENDPROC;
0079
0080
0081 PROCEDURE TASK_1;
0082    REPEAT
0083       OUTCHAR('B);
0084    FOREVER;
0085 ENDPROC;
0086
0087
0088 PROCEDURE TASK_2;
0089    REPEAT
0090       OUTCHAR('C);
0091    FOREVER;
0092 ENDPROC;
0093
0094
```

## 9.09.04 BETTER CONTROL OF AN IRQ DRIVEN MULTI-TASKING PROGRAM (continued)

```
0095 PROCEDURE MAIN;
0096
0097    GEN $10,$BF,$7F,$FE;     /* STY $7FFE (just above stack) */
0098
0099    TASK = 0;                /* Task 0 active */
0100
0101    TASK_DURATION_COUNTER  = TASK0_PRIORITY;
0102
0103    TASK_PRIORITY_TABLE(0) = TASK0_PRIORITY;
0104    TASK_PRIORITY_TABLE(1) = TASK1_PRIORITY;
0105    TASK_PRIORITY_TABLE(2) = TASK2_PRIORITY;
0106
0107    CCR = CCR AND $EF;       /* enable IRQ in pre-load CCR */
0108
0109    STACK_SAVE = STACK;      /* save task 0 program stack pointer */
0110
0111    STACK1 = .TASK_1;        /* put address of task 1 on top of stack */
0112    STACK = .STACK1;         /* aim stack at stack area for task 1 */
0113    GEN $34,$7F;             /* pre-load stack 1 (PSHS CC,D,DP,X,Y,U) */
0114    TASK_TABLE(1) = STACK;   /* save base of stack in task table */
0115
0116    STACK2 = .TASK_2;        /* put address of task 2 on top of stack */
0117    STACK = .STACK2;         /* aim stack at stack area for task 2 */
0118    GEN $34,$7F;             /* pre-load stack 2 (PSHS CC,D,DP,X,Y,U) */
0119    TASK_TABLE(2) = STACK;   /* save base of stack in task table */
0120
0121    STACK = STACK_SAVE;      /* re-aim stack back to task 0 stack area */
0122
0123    INITIALIZE_TIMER;
0124
0125    TASK_0;
```

## 9.09.04 BETTER CONTROL OF AN IRQ DRIVEN MULTI-TASKING PROGRAM (continued)

Program 6 introduces several control mechanisms and tidies up some of the undesirable features of the earlier programs.

The first thing to note is that we have now put all of the global variables right after the 'GLOBAL' declaration to shorten the addressing range required to access them. This results in a significant code saving, even in a program of this size. Next note that we must now explicity declare the stack associated with 'TASK_0'.

We have two additional timer control routines between lines 48 and 57. They function as their names suggest.

The 'ASMPROC' named 'NEW_TASK' at line 66 forces an eary IRQ service by simulating the register push that will occur when a real IRQ takes place. Control is then passed into the normal IRQ service routine at line 70.

The IRQ service routine is very similar to that of program 5 except for the additional control code between lines 79 and 93. The purpose of this control code is to determine if a task has been put to sleep (line 84) and if it has to move on to the next task (line 85). Obviously if there is any possiblity that all of the tasks may accidentally get put to sleep you must insert some additional arguments after line 84 otherwise the program will loop between lines 80 and 85 forever.

Lines 88 through 92 handle the situation where the previous task has been terminated. These lines ensure that the next task is allocated a full IRQ time slice rather than just the remainder of the previous task's time slice.

Lines 98 through 103 provide the primary mechanism by which a task may terminate itself early.

Lines 106 through 106 provide the mechanism by which a task may put itself to sleep. The syntax for using this procedure is: 'SLEEP_TASK(N)' where 'N' is any valid task number. Once a task is put to sleep it will not be serviced again until another task wakes it up.

Lines 112 through 114 provide the mechansim to wake up a sleeping task. The syntax is: 'WAKE_TASK(N)' where 'N' is any valid task number.

There is nothing in principle to prevent you from dynamically altering the priorities of the various tasks whist the program is running. The restraint is that an interrupt must not occur whilst you are manipulating the global data involved!

If you run this program as presented you will notice something very strange if you monitor the IRQ line with an oscilloscope. THERE ARE NO IRQ'S BEING GENERATED!!! This is because each task lasts less than the IRQ time slice. When each task terminates itself it forces the processor to switch to the next task and initiate a new time slice which will do exactly the same thing.

Obviously if you want to use the IRQ for system time-keeping some other approach must be used. Try changing lines 128, 136, and 144 to 'TASK_DURATION_COUNTER = 0' and see what happens.

If you intend to adopt multi-tasking as a programming approach we suggest that you spend a reasonable amount of time trying out your programming ideas with simple programs such as this one. This way should something go wrong you stand a fair chance of finding the problem whilst the structure of your program is very primative.

## 9.09.04 BETTER CONTROL OF AN IRQ DRIVEN MULTI-TASKING PROGRAM (continued)

```
0001 /* Multi-Tasking Demonstration Program 6 */
0002
0003 ORIGIN = $8000;
0004 STACK = $7FFE;  /* 'Y' will be saved at $7FFE/F */
0005
0006
0007 AT $E030:BYTE TIMER;          /* base address of MC6840 timer */
0008
0009 CONSTANT NO_OF_TASKS = 3,       /* Tasks 0, 1 and 2 */
0010          STOPPED = -1,
0011          RUNNING = 0,
0012          TERMINATED = 1,
0013          ACTIVE = -1,
0014          SLEEP  = 0,
0015          TASK0_PRIORITY = 10,  /* 10 IRQ TIME SLICES */
0016          TASK1_PRIORITY = 20,  /* 20 IRQ TIME SLICES */
0017          TASK2_PRIORITY = 30;  /* 30 IRQ TIME SLICES */
0018
0019
0020 GLOBAL
0021    INTEGER TASK_TABLE(NO_OF_TASKS):
0022    BYTE TASK_PRIORITY_TABLE(NO_OF_TASKS):
0023    BYTE TASK_STATUS_TABLE(NO_OF_TASKS):
0024    BYTE TASK, TASK_DURATION_COUNTER, CURRENT_TASK, TIMER_STATUS:
0025    INTEGER STACK_SAVE:
0026
0027    BYTE DUMMY0(100):          /* Length of stack for task 0 */
0028    INTEGER STACK0:
0029
0030    BYTE DUMMY1(100):          /* Length of stack for task 1 */
0031    INTEGER STACK1:
0032
0033    BYTE DUMMY2(100):          /* Length of stack for task 2 */
0034    INTEGER STACK2;
0035
0036
0037 PROCEDURE INITIALIZE_TIMER;
0038    TIMER(1) = 0;     /* select control register # 3 */
0039    TIMER(0) = $42;   /* enable IRQ, use '/E' for clock source */
0040    TIMER(6) = $4E;
0041    TIMER(7) = $20;   /* ($4E20 = 20000) 100 Hertz w/2 MHz CPU */
0042    TIMER(1) = 1;     /* select control register # 1 */
0043    TIMER(0) = 1;     /* hold timer in preset state */
0044    TIMER_STATUS = STOPPED;
0045 ENDPROC;
0046
0047
0048 PROCEDURE TIMER_ON;
0049    TIMER(0) = 0;
0050    TIMER_STATUS = RUNNING;
0051 ENDPROC;
0052
0053
0054 PROCEDURE TIMER_OFF;
0055    TIMER(0) = 1;
0056    TIMER_STATUS = STOPPED;
0057 ENDPROC;
```

9.09.04 BETTER CONTROL OF AN IRQ DRIVEN MULTI-TASKING PROGRAM (continued)

```
0058
0059
0060 PROCEDURE SERVICE_TIMER;
0061    ACCB = TIMER(1);
0062    ACCB = TIMER(6);  /* clear IRQ */
0063 ENDPROC;
0064
0065
0066 ASMPROC NEW_TASK;
0067    GEN $34,$7F;   /* PSHS CC,D,DP,X,Y,U  (SIMULATE IRQ EXCEPT FOR 'PC') */
0068
0069
0070 PROCEDURE IRQ;
0071    GEN $10,$BE,$7F,$FE;  /* LDY $7FFE (recover global pointer) */
0072                          /* also recover 'DP' if you use DPAGE! */
0073
0074    IF TASK_DURATION_COUNTER <> 0
0075       THEN TASK_DURATION_COUNTER = TASK_DURATION_COUNTER - 1;
0076
0077    IF TASK_DURATION_COUNTER = 0
0078       THEN BEGIN
0079            TASK_TABLE(TASK) = STACK;
0080       TRY_ANOTHER:
0081            TASK = TASK + 1;
0082            IF TASK = NO_OF_TASKS
0083               THEN TASK = 0;
0084            IF TASK_STATUS_TABLE(TASK) = SLEEP
0085               THEN GOTO TRY_ANOTHER;
0086            TASK_DURATION_COUNTER = TASK_PRIORITY_TABLE(TASK);
0087            STACK = TASK_TABLE(TASK);
0088            IF CURRENT_TASK = TERMINATED
0089               THEN BEGIN
0090                    CURRENT_TASK = ACTIVE;
0091                    TIMER_ON;  /* initiate fresh time-slice */
0092                    END;
0093            END;
0094    SERVICE_TIMER;
0095 ENDPROC;
0096
0097
0098 PROCEDURE TERMINATE_TASK;
0099    TASK_DURATION_COUNTER = 0;
0100    CURRENT_TASK = TERMINATED;
0101    TIMER_OFF;  /* ensures next task gets full time-slice */
0102    NEW_TASK;
0103 ENDPROC;
0104
0105
0106 PROCEDURE SLEEP_TASK(BYTE N);
0107    TASK_STATUS_TABLE(N) = SLEEP;
0108    TERMINATE_TASK;
0109 ENDPROC;
0110
```

9.09.04 BETTER CONTROL OF AN IRQ DRIVEN MULTI-TASKING PROGRAM (continued)

```
0111
0112 PROCEDURE WAKE_TASK(BYTE N);
0113    TASK_STATUS_TABLE(N) = ACTIVE;
0114 ENDPROC;
0115
0116
0117 PROCEDURE OUTCHAR(BYTE CHAR);
0118    CCR = CCR OR $10;   /* kill IRQ */
0119    ACCA = CHAR;
0120    CALL $CD18;
0121    CCR = CCR AND $EF;  /* enable IRQ */
0122 ENDPROC;
0123
0124
0125 PROCEDURE TASK_0;
0126    REPEAT
0127       OUTCHAR('A);
0128       TERMINATE_TASK;
0129    FOREVER;
0130 ENDPROC;
0131
0132
0133 PROCEDURE TASK_1;
0134    REPEAT
0135       OUTCHAR('B);
0136       TERMINATE_TASK;
0137    FOREVER;
0138 ENDPROC;
0139
0140
0141 PROCEDURE TASK_2;
0142    REPEAT
0143       OUTCHAR('C);
0144       TERMINATE_TASK;
0145    FOREVER;
0146 ENDPROC;
0147
0148
```

9.09.04 BETTER CONTROL OF AN IRQ DRIVEN MULTI-TASKING PROGRAM (continued)

```
0149 PROCEDURE MAIN;
0150
0151    GEN $10,$BF,$7F,$FE;     /* STY $7FFE (just above stack) */
0152
0153    TASK = 0;                /* Task 0 active */
0154
0155    CURRENT_TASK = ACTIVE;
0156    TASK_DURATION_COUNTER = TASK0_PRIORITY;
0157
0158    TASK_PRIORITY_TABLE(0) = TASK0_PRIORITY;
0159    TASK_PRIORITY_TABLE(1) = TASK1_PRIORITY;
0160    TASK_PRIORITY_TABLE(2) = TASK2_PRIORITY;
0161
0162    TASK_STATUS_TABLE(0) = ACTIVE;
0163    TASK_STATUS_TABLE(1) = ACTIVE;
0164    TASK_STATUS_TABLE(2) = ACTIVE;
0165
0166    CCR = CCR AND $EF;       /* enable IRQ in pre-load CCR */
0167
0168    STACK2 = .TASK_2;        /* put address of task 2 on top of stack */
0169    STACK = .STACK2;         /* aim stack at stack area for task 2 */
0170    GEN $34,$7F;             /* pre-load stack 2 (PSHS CC,D,DP,X,Y,U) */
0171    TASK_TABLE(2) = STACK;   /* save base of stack in task table */
0172
0173    STACK1 = .TASK_1;        /* put address of task 1 on top of stack */
0174    STACK = .STACK1;         /* aim stack at stack area for task 1 */
0175    GEN $34,$7F;             /* pre-load stack 1 (PSHS CC,D,DP,X,Y,U) */
0176    TASK_TABLE(1) = STACK;   /* save base of stack in task table */
0177
0178    STACK = .STACK0;         /* aim stack at task 0 area */
0179
0180    INITIALIZE_TIMER;
0181    TIMER_ON;
0182
0183    TASK_0;
```

## 9.09.04 BETTER CONTROL OF AN IRQ DRIVEN MULTI-TASKING PROGRAM (continued)

You may have noted that we have declared a GLOBAL variable called 'TIMER_STATUS' but have not used it anywhere other than between lines 37 and 57. This variable would normally be used in a supervisory program to determine if the IRQ timer was running or not.

We hope this section has given you some insight into how simple multi-tasking kernals really are. Once you understand the basic principles involved you should be able to write a multi-tasking program that runs under interrupts with the same ease you would write normal PL/9 programs.

The main rules when writing multi-tasking programs are:

1. Make sure you allocate sufficient stack space to each task. Be generous if you are in doubt. If any recursive programs are going to be used make sure there is PLENTY of stack available.

2. Don't forget to service the interrupting device in the interrupt service routine. Failing to do so will generally result in the IRQ line remaining low which will cause the service routine to be re-entered the instant the RTI is executed ... this will completely lock up the system.

3. Be careful with global variables that are used by several procedures. Ensure that the next task is not selected until the current task is completely finished manipulating/analyzing them.

4. Even though the examples we have presented have not used local variables in any of the tasks or in the interrupt service procedure there is nothing to prevent you from using local variables in multi-tasking programs in the same manner you would with conventional PL/9 programs.

5. If you write any ASMPROCS that are to be used by a multi-tasking program that runs under hardware interrupts make sure it is re-entrant. This means that all variable storage must be on the stack. If the ASMPROC manipulates GLOBAL data, including 'AT' data which includes I/O devices the program is NOT re-entrant. If you must do this within a procedure ensure that you either mask the interrupt until you have completed manipulating the data or set a flag to let other procedures that will use the data know that it is still being processed.

## 9.10.00 PROGRAM ENTRY AND EXIT

Most PL/9 programs will be self-contained and only called from FLEX, another PL/9 program, or will be the ONLY program in a dedicated control system. When they finish (if ever) they return to FLEX or to wherever the programmer specifies.

There are occasions, however, when a program must behave in its entirety as a subroutine; it may, for example, be called from BASIC or another language to perform some task not suited to the latter.

Suppose that you have written a PL/9 program that is going to interface to the system hardware, say to read a number of samples in real time from an analog-to-digital converter. Your program will be called from BASIC, which is unable to do that kind of job itself but which is perfectly capable of doing the subsequent analysis, especially if the data is to be presented in tabular manner.

Assuming that you have a compiled PL/9 program called 'SAMPLE.BIN', the BASIC program line EXEC "GET SAMPLE" will load the sampling program into memory for use with a 'USR' call. See your BASIC manual for further information on how to use the 'USR' function.

The following sections will provide guidelines on how to construct PL/9 programs such that they will not interfere with the memory allocations of the 'parent' program and will return to the 'parent' program with ALL of the MC6809 registers intact.

The information in the following sections will be presented in the usual order they MUST be declared in if they are used in a PL/9 program.

## 9.10.01 ORIGIN

A PL/9 program has a single entry point, this being the address specified by the first ORIGIN in the program. This keyword has only been mentioned superficially so far in the User's Guide; its purpose is to allow you to locate your program wherever you choose in memory. Without an ORIGIN, all programs start at address $0000.

This may be fine for many applications but it conflicts with BASIC as well as many other languages. In these cases you will have to choose somewhere else to locate it. Exactly where you decide to put it depends on your system hardware configuration, what the main program is doing etc. As there are too many variations it is impossible to suggest a suitable location. Generally speaking if you allocate your programs to the area of memory just below FLEX, say between $B000 and $BFFF you will avoid 'knocking' into anything else. The FLEX transient command area between $C100 and $C6FF is another good location if your program is not going to make use of any of the FLEX disk resident commands.

Since the tracer ignores ORIGIN statements locating your program in the FLEX TCA will not cause any conflicts.

You are allowed to declare as many ORIGINs as you wish in a PL/9 program. They may also be declared in any order, i.e. you can have the first origin declared as $8000, the second as $0100, the third as $E000, etc. PL/9 assumes that you know what you are doing in these instances so you must be careful not to declare an origin that overlaps the code produced by the procedures based at another origin.

We also remind you to observe the caution detailed in section 7.01.03 concerning declaring the first procedure or read-only data, even if it means declaring a dummy procedure or read-only BYTE, before you declare the second ORIGIN.

ORIGIN may be declared as:

```
ORIGIN=$C100;
```

Or it may be assigned via a CONSTANT, e.g.

```
CONSTANT FLEX_TCA=$C100;

ORIGIN=FLEX_TCA;
```

One very important point to note about PL/9 program structures that use the STACK, GLOBAL, or DPAGE declarations is that you MUST always enter the PL/9 program at the FIRST declared ORIGIN. This is because all of the code for the STACK, GLOBAL, and DPAGE assignments is placed immediately after the first ORIGIN. After the code for the above facilities you will always find a long branch to the last procedure in the file.

If STACK, GLOBAL and DPAGE are not used in a file containing a series of PL/9 subroutines, a library file for example, the above rule does not apply. In this instance you may enter each of the PL/9 procedures at the specific address you decide to locate them at either by multiple ORIGIN statements or by looking at the code PL/9 produces to ascertain the starting address of each procedure. When STACK, GLOBAL, and DPAGE are not used only a long branch to the last procedure will be present at the first ORIGIN.

### 9.10.02 STACK

Here's a keyword that you can use to wreck havoc on your system if you are not careful with it! This keyword, if used, MUST be declared immediately after the first ORIGIN and in the absence of an origin as the first line of a program. Comment lines may appear before both the ORIGIN and the STACK declarations or in between them for that matter, but nothing else may!

This keyword initializes the PL/9 stack to be wherever you want it. Most programs called from FLEX can leave the stack pointer alone as the FLEX stack area between $C000 and $C080 should be adequate for most programs. If this does not provide enough space remember that the stack will simply grow downwards. If you locate your program in a sensible area of low memory the chances of the stack growing into it are remote.

A suitable location for the STACK is left to the discretion of the programmer. If you want to put it right on top of FLEX, go ahead...PL/9 doesn't mind!!!

It is quite permissible to relocate the stack if the PL/9 program is going to return to FLEX as entering the FLEX warm start address at $CD03 will reinitialize the stack pointer to the position FLEX wants it in.

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                                                                     *
*    If you are going to call the PL/9 procedure from BASIC,          *
*    another language or even another PL/9 procedure you              *
*    should NEVER re-assign the stack unless you take special         *
*    precautions to preserve the existing stack pointer.              *
*                                                                     *
*  Refer to the section on ENDPROC END for further information.       *
*                                                                     *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

If, on the other hand, the PL/9 program is to form the main (or only) program in a self starting environment the stack MUST ALWAYS be assigned.

Like the ORIGIN assignment STACK may be assigned directly or via a constant.

NOTE: 'STACK' is also a pseudo register keyword when it appears within a procedure.

### 9.10.03 GLOBAL

This keyword, if used, must appear after the ORIGIN statement, if used, and after the STACK statement, if used, and before the DPAGE statement, if used.

It's purpose is to allocate permanent storage on the stack in order to assign variables that will be known by all procedures and accessible by all procedures. GLOBAL storage is permanent in that it is not lost between procedures as local variables are.

Using the GLOBAL statement also causes PL/9 to generate the code required to push the entire register set onto the stack before the stack pointer is shifted to make room for the global data storage. If you are writing a PL/9 procedure that must behave as a subroutine, AND you require that the subroutine preserves all of the registers (as is the case of a subroutine called from many BASICs) using the GLOBAL statement, even if you are not using any global variables, is an easy way of accomplishing this end. In this instance you would simply declare a dummy global variable 'GLOBAL BYTE DUMMY;'. Also see the section on ENDPROC END for information on how to terminate a PL/9 program that uses global storage.

It is important to note that the variables declared immediately after the GLOBAL statement will reside at the base of the stack (the stack will grow down from the location of the first global variable). The closer a variable is to the base of the stack the shorter will be the code required to address it.

Frequently used global variables should therefore be declared first as this can significantly reduce the amount of code PL/9 will generate if global variables are frequently used in the subsequent procedures.

GLOBAL may be assigned directly or via a constant.

### 9.10.04 DPAGE

DPAGE, if used, must be declared after the STACK statement, if used, and after the GLOBAL statement, if used. This keyword allows you to set the MC6809 direct page register to any location in memory. Normally the direct page register is set to $00 by reset. Using DPAGE to set the direct page register to the top 8-bits of the address of your 'AT' variables or your I/O addresses will enable PL/9 to generate more efficient code when accessing 'hard' memory locations over a range of 256 bytes.

For example in a WINDRUSH or a GIMIX system stating 'DPAGE=$E0' will improve the code to access any variable from $E000 to $E0FF which is the entire SS-30 I/O section.

If you use DPAGE in a PL/9 program that is to be called from BASIC you should also use GLOBAL before it (even if this means declaring a dummy global variable) and ENDPROC END at the end of the main PL/9 program. This will ensure that the MC6809 direct page register is returned to the calling program intact.

Altering the direct page register in a subroutine and returning to the calling program without re-instating the original value of the direct page register is one of the niftiest way of generating the strangest crashes you have ever seen.

If you use the 'DPAGE' directive any you wish to access 'AT' variables that reside on the direct page within a hardware interrupt procedure (IRQ, NMI, FIRQ and RESET) you MUST re-initialize the 'DP' within the interrupt procedure.

DPAGE may be assigned directly or set via a constant.

### 9.10.05 ENDPROC END

Earlier in this Guide it was stated that the main procedure in a PL/9 program does not need an ENDPROC statement if you are calling the PL/9 program from FLEX and expect it to return there.

When you are trying to build a PL/9 program as a subroutine for another language, or another PL/9 program for that matter, the main procedure WILL require an ENDPROC.

There are two distinct cases to consider, viz. where global variables have not been used and where they have.

Without global variables, an ENDPROC (or an ENDPROC END) after the last procedure will generate a Return-From-Subroutine ($39) and therefore return you tidily to the calling program. In this instance any or all of the MC6809 registers may have been altered. The stack pointer, however, will be just where it was when the PL/9 program was entered.

If a GLOBAL declaration is used the compiler generates code to save all registers, then makes some room on the system stack and sets the Y index register to point to the base of the variables just created. At the very end of the program it is therefore necessary to release this reserved space and pop all of the saved registers; a simple Return_From_Subroutine is not enough. PL/9 recognizes the statement ENDPROC END as meaning "that was the last procedure of the program so now clean up the stack".

This simple construct is all that is necessary to make even the largest PL/9 program a complete subroutine that can called from anywhere and return with not only the stack pointer preserved but also the entire register set as well.

The above implies that if you wish to preserve the entire register set you should make a GLOBAL declaration even if you are not going to use global variables in the subsequent procedures. In this case you would simply make a dummy declaration as 'GLOBAL BYTE DUMMY;' and terminate the program with an 'ENDPROC END;'.

Note, of course, that wherever the PL/9 program is called from must provide enough room on the stack for any global or local variables that will be declared in the subroutine as well as any growth downward created by nested subroutine calls.

There is no automatic provision for saving the stack pointer on entry, assigning a temporary stack area, and restoring the original stack before exiting.

If this ever becomes necessary you can structure the PL/9 program as indicated on the following page. Here you must break one of the primary rules governing the entry of PL/9 programs; you must enter it at an address other than the first declared ORIGIN. The use of multiple ORIGIN statements in the example precludes the necessity of having to look at the code PL/9 produces in order to determine where to enter the program and where the final procedure must jump to to terminate it.

This type of program structure does have one distinct disadvantage:

IT IS NOT POSITION INDEPENDENT

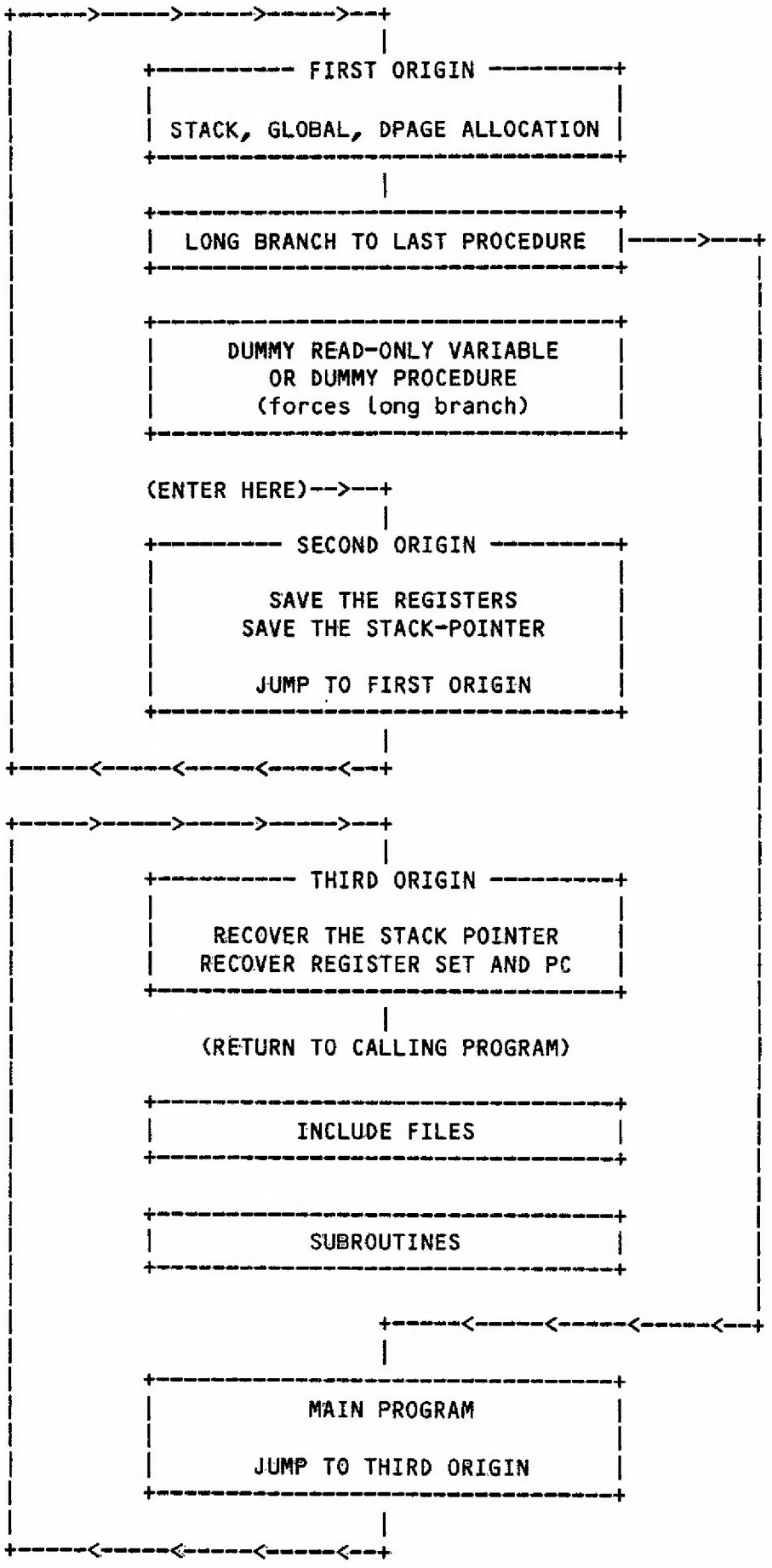
## 9.10.05 ENDPROC END (continued)

```
0001 CONSTANT PL9PROG = $8000,
0002          ENTER_HERE = $8020,
0003          EXIT_HERE = $8030,
0004
0005          STACK_INIT = $7FFD,
0006
0007          STACK_SAVE_HI = $7F,
0008          STACK_SAVE_LO = $FE;
0009
0010
0011 ORIGIN = PL9PROG;
0012 STACK = STACK_INIT;
0013 GLOBAL INTEGER I1,I2;
0014 DPAGE = $EF;
0015
0016 BYTE  DUMMY $12;  /* See section 7.01.03 if you don't understand this as
0017                      it is critical to the operation of the compiler!  */
0018
0019 ORIGIN = ENTER_HERE;
0020
0021 PROCEDURE ENTER_THE_PL9_PROGRAM_HERE;
0022    GEN $34,$7F;                                  /* PSHS  CC,D,DP,X,Y,U */
0023    GEN $10,$FF,STACK_SAVE_HI,STACK_SAVE_LO;      /* STS STACK_SAVE */
0024    JUMP PL9PROG;
0025 ENDPROC;
0026
0027
0028
0029 ORIGIN = EXIT_HERE;
0030
0031 PROCEDURE HEAD_FOR_HOME;
0032    GEN $10,$FE,STACK_SAVE_HI,STACK_SAVE_LO;      /* LDS STACK_SAVE */
0033    GEN $35,$FF;                                  /* PULS  CC,D,DP,X,Y,U,PC */
0034 ENDPROC;
0035
0036
0037 INCLUDE 0.IOSUBS;
0038
0039
0040 /*
0041   * * * * * * * * * * * * * * * * * * * *
0042   *  PUT ANY SUBROUTINE PROCEDURES HERE *
0043   * * * * * * * * * * * * * * * * * * * *
0044 */
0045
0046
0047 PROCEDURE MAIN;
0048 /*
0049   * * * * * * * * * * * * * * * * * *
0050   *  DO WHATEVER YOU LIKE HERE!   *
0051   * * * * * * * * * * * * * * * * * *
0052 */
0053    PRINT("THIS WAS A QUICK TRIP");
0054
0055    JUMP EXIT_HERE;  /* LEAVE PROGRAM VIA THIRD ORIGIN */
```

Under NO CIRCUMSTANCES let the MAIN procedure terminate with an ENDPROC as this will POSITIVELY crash the system!

9.10.05 ENDPROC END (continued)

The following block diagram should clarify what the program structure on the previous page is doing.



## 9.11.00 HANDLING INTERRUPTS (SWI, SWI2, SWI3, NMI, FIRQ, and IRQ)

Interrupts provide a convenient way of getting your computer to effectively do two or more things at once. Most high level languages can only handle interrupts with a great deal of difficulty and assembly language patches.

The first point to note about interrupts is that they are SYSTEM DEPENDANT in that when an interrupt occurs, program execution is suspended and the processor jumps to an address dependant on what it finds at the very top of memory. In most 6809 development systems the System Monitor traps the interrupt and decides what to do with it. PL/9 assumes that your system monitor, following the usual convention, maintains a RAM vector for each interrupt. The address of your interrupt handler is then placed in the RAM vector. Thus when an interrupt occurs control will be passed to your routine.

The configuration program SETPL9 (see section three) enables you to install the correct interrupt vector addresses in your copy of PL/9. The addresses you give SETPL9 however, depend on the environment in which your application program will run. Normally you should supply SETPL9 with the addresses of your system monitor RAM vectors. This will allow you to compile and test programs within your development system. When you wish to compile the program for use in a stand-alone or self-starting system all you need do is specify the 'R' (ROM) option when you compile the program, e.g. 'A:0=MYPROG.BIN,R'.

Because of the hardware dependant nature of interrupt generating devices, it is very difficult t give you examples which you can try out, since they will vary so much from one system to another. Some guidelines are in order however:

1. An interrupt procedure is very much like any other procedure in PL/9 , except that you can only pass parameters to it via GLOBAL and AT variables. Three procedure names are reserved for the software interrupts, viz. SWI, SWI2, and SWI3. Three procedure names are also reserved for the hardware interrupts, viz. NMI, FIRQ, and IRQ (RESET will be covered in the next section). These names tell PL/9 to set up the interrupt vectors specified when you ran SETPL9. Local variables can be declared within an interrupt procedure. At the end of the procedure PL/9 generates a Return-From-Interrupt (RTI) instead of the usual Return-From-Subroutine (RTS).

2. Global variables, and 'AT' variables in the direct page (as defined oy the 'DPAGE' directive) can only be accessed with caution and require a conscious effort, on the part of the programmer, to preserve and restore the 'Y' index register if GLOBALS are being accessed and the 'DP' register if AT variables in the direct page are being accessed. If you recall PL/9 uses the 'Y' index register to point to the base of the global variables. When an interrupt occurs there is no telling what will be in either of these two registers. If, for example, the interrupt occurred whilst the processor was executing a FLEX subroutine the contents of the 'Y' and/or 'DP' register(S) would positively be questionable. This means that if you wish to access GLOBAL and AT variables from within an interrupt procedure you must take steps to recover the 'Y' and 'DP' registers before any such accesses take place. This latter point is particularly important if the interrupt procedure is going to use any of the other PL/9 procedures as subroutines. In these circumstances you have to be very careful NOT to use any procedures that use GLOBAL or AT variables. In large programs it may be difficult to be absolutely certain that subroutines called from the interrupt procedure do not make some reference to GLOBAL or AT variables.

   What follows is a demonstration of a simple, but effective, technique for preserving the contents of the 'Y' index register at the start of a program and recovering it from within the interrupt procedure.

9.11.00 HANDLING INTERRUPTS (SWI, SWI2, SWI3, NMI, FIRQ, and IRQ) (continued)

This technique requires that you have access to two bytes of memory at some absolute address. This should not pose any problems 99.997% of the time. The technique we have adopted is to assign the stack to a location two bytes further down in memory than we normally would. For example our dedicated systems usually have 1K of RAM between $E400 and $E7FF. Instead of assigning the stack to $E5FF, for example, we assign it to $E5FD. The exact memory location you chose to store the 'Y' register is not particulary important but it must not be used by anything else in the system.

The following program illustrates the technique:

```
0001 CONSTANT STACK_INIT = $E7FE,
0002          YSAVE_HI   = $E7,
0003          YSAVE_LO   = $FE;
0004
0005
0006 STACK=STACK_INIT;
0007
0008 GLOBAL I1;
0009
0010 INCLUDE 0.IOSUBS;
0011
0012
0013 PROCEDURE SAVE_GLOBAL_POINTER;
0014     GEN $10,$BF,YSAVE_HI,YSAVE_LO;  /* STY YSAVE */
0015 ENDPROC;
0016
0017
0018 PROCEDURE RESTORE_GLOBAL_POINTER;
0019     GEN $10,$BE,YSAVE_HI,YSAVE_LO;  /* LDY YSAVE */
0020 ENDPROC;
0021
0022
0023 PROCEDURE ONE;
0024 ENDPROC;
0025
0026 .
0027 .
0028 .
0029 .
0030 .
0031
0032 PROCEDURE TEN;
0033 ENDPROC;
0034
0035
0036 PROCEDURE NMI;
0037     RESTORE_GLOBAL_POINTER;
0038     I1=I1+1;
0039 ENDPROC;
0040
0041
0042 PROCEDURE MAIN;
0042     SAVE_GLOBAL_POINTER;
0044     .
0045     .
0045     .
0047 ENDPROC END;
```

9.11.00 HANDLING INTERRUPTS (SWI, SWI2, SWI3, NMI, FIRQ, and IRQ) (continued)

The main points to note are that the global pointer is saved as one of the first actions in the MAIN procedure and is recovered as one of the first actions of the interrupt procedure. For an example of how to recover the 'DP' register see the PL/9 'MINI-MONITOR' program in section 9.12.02.

3. It is not normally possible to trace interrupt procedures using PL/9's built-in trace/debugger. The tracer, which is described in section six, works by placing extra code in the object program that allows it to retain control over the program while it is running. An internal subroutine in PL/9 is called once for every source line; this subroutine can stop the program, if necessary, to examine variables. Because the tracer is not re-entrant, however, an interrupt will destroy information relevant to the procedure that was being executed when the interrupt occurred.

   In order to ensure that a program using interrupts can be traced you MUST ensure that that the interrupt procedures can't. To do this, write them to a file and INCLUDE them at the appropriate point in the program. Since INCLUDEd procedures are never traced (they are assumed to already work) the interrupt will never be seen by the tracer, which will continue to operate on the main program.

   A warning must be issued here: If you write a program that generates timer interrupts and includes an interrupt procedure to handle them, the timer will continue to run and generate interrupts after you have finished tracing and returned to the PL/9 editor command level or even FLEX. Re-compiling the program may result in the new version of the interrupt routine at a different place to the previous version. Since PL/9 writes the interrupt vectors last of all, an interrupt occurring during compilation will probably cause the system to crash. It is therefore advisable to ensure that sources of interrupts are disabled before exiting the tracer. Hitting hardware reset and then entering PL/9 at its warm start address ($0003) is a crude, but effective way of killing interrupts from most devices.

4. The MC6809 Fast Interrupt (FIRQ) can be handled by a procedure of the same name. Since only the return address (PC) and condition code register (CCR) are pushed onto the stack when an FIRQ occurs this interrupt would be of little use to a PL/9 procedure. This is because PL/9 procedures, as a minimum, will use the 'B' accumulator, and will, more often than not use 'A' and 'X' as well.

   In order to enable the programmer to make some use of the FIRQ in PL/9 programs the compiler generates the code necessary to push the remaining registers (DP, D, X, Y, and U) onto the stack at the start of the procedure ($34 $7E) and to pull them off the stack at the end of the procedure ($35 $7E) just before the RTI ($3B). This effectively makes FIRQ the same as IRQ but, due to the extra register push-pull operations, also makes it SLOWER than IRQ.

5. You can declare and use local variables within an interrupt procedure just as you would for any other PL/9 procedure. Obviously it is not possible to pass an interrupt procedure a variable on the stack or return one in one of the registers. You may, of course, pass and return variables via GLOBAL or AT variables providing you restore the 'Y' index register and the 'DP' register as appropriate.

## 9.12.00 STARTING A PL/9 PROGRAM FROM POWER-UP (RESET)

Unless you are an assembly language programmer this is an area where virtually every high level language, including Pascal, C, and Basic pose very awkward problems.

More often than not with these languages the mechanism to provide an interrupt vector table to handle system startup MUST be coded in assembly language. This can be a bit daunting if you are NOT an assembly language programmer.

This section will outline the PL/9 program structures used to generate code which can be simply placed in an EPROM and used to start a program up from a COLD start with no intervention whatsoever. As the same techniques are applicable to the MC6809 hardware and software interrupts they will also be covered in this section.

PL/9 has seven reserved procedure names, viz. SWI, SWI2, SWI3, NMI, FIRQ, IRQ and RESET. When you configure your copy of PL/9 with the SETPL9 program (see section three) you will be asked to specify the system RAM vector addresses for all of these routines except for RESET. Generally speaking RESET cannot be in RAM unless there is hardware protected non-volatile memory in the system. RESET is therefore always assumed to be $FFFE/F.

When you are developing your programs you might want to use the system RAM vectors to assist you in testing your programs in the FLEX environment rather than having to program an EPROM every time you want to test something. Once the majority of the program is tested and debugged in a RAM environment you can then revert to using the MC6809 hardware vector table for the final ROM tests by using the 'R' (ROM) option during program compilation, e.g. 'A:0=MYPROG.BIN,R'.

If the code you are producing is intended for use in a dedicated system where it will be placed in ROM and used to start the system it is recommended that you declare ALL interrupt procedures, whether you are using them or not. This will ensure that each of the MC6809 interrupt vectors point to a dummy RTI instruction so that should, for example, an accidental NMI occur due to a hardware fault of some sort, the system will not crash.

The following section presents two programs that represent what must be a record for a 'minimum' program that most users an compile and test in their own systems providing they have an EPROM programmer.

## 9.12.01 DUMB-BUG ... A SELF STARTING PROGRAM

Well what would you call a program that just spins in circles?

The two pages which follow present a self-starting program whose sole task in life s to echo all characters sent by a terminal back to the terminal. The only 'intelligence' imparted to the program is the ability to recognize the <RETURN> key on your keyboard and send a CARRIAGE-RETURN ... LINE-FEED sequence back to the terminal whenever it sees it.

The result is that you can convert your terminal and computer into what amounts to a typewriter ... clever stuff folks!

Two versions of the program are presented. The first is for systems that use a Motorola MC6850 ACIA to communicate with the terminal. The second is for systems that use a Rockwell R6551 ACIA to communicate with the terminal. In both cases a 2K EPROM, such as a 2716, is assumed. If your system uses a larger device just alter the 'PROM_BASE' statement on line 4 to match the base address of the EPROM in your system.

The MC6850 program assumes that the baud clock fed to the MC6850 is at the 'X16' rate that is common in GIMIX, SWTP and WINDRUSH systems. SSB systems use the 'X64' rate so the constant in line 25 must be changed to '$16'.

The R6551 program assumes the terminal can communicate at 9600 baud. If this is not the case you will have to alter the '$9E' in line 25 to match the capabilities of your terminal. Consult the R6551 data sheet for details.

Both programs assume that your terminal an communicate using 8-data bits, no start bits, 2 stop bits and no parity. Most will.

To run this program follow these steps:

1. Type in the program that applies to your system.
2. Save it to disk: S=1.DUMBUG.PL9<CR>
3. Compile it to disk using the 'R' option: A:0=1.DUMBUG.BIN,R<CR>
4. Return to FLEX and invoke your EPROM programmer_software.
5. Clear out the EPROM buffer area (fill it with $FF).
6. Load the file 'DUMBUG.BIN' into the EPROM buffer area.
7. The program code should be at the start of the buffer. The buffer address that corresponds to $FFFE/F in the EPROM should contain the address of the 'RESET' procedure ($F807).
8. Program the EPROM.
9. Turn the power to your computer system off.
10. Remove your existing SYSTEM MONITOR EPROM and install the one just programmed.
11. Turn the power to your computer system back on.
12. Start typing on your terminal keyboard ... everything you type should appear on the screen of the terminal.

## 9.12.01 DUMB-BUG ... A SELF STARTING PROGRAM (continued)

```
0000  0001 /* DUMB-BUG FOR MC6850 ACIA'S */
0000  0002
0000  0003
0000  0004 CONSTANT PROM_BASE  = $F800,
0000  0005          STACK_INIT = $E7FF,
0000  0006
0000  0007          RDR_FULL   = $01,
0000  0008          TDR_FULL   = $02;
0000  0009
0000  0010 AT $E004: BYTE ACIA_CONTROL(0), ACIA_STATUS, ACIA_DATA;
0000  0011
0000  0012
0000  0013 ORIGIN = PROM_BASE;
F800  0014
F800  0015 STACK = STACK_INIT;
F804  0016
F804  0017
F804  0018 PROCEDURE RESET;
F807  0019    JUMP PROM_BASE;
F80A  0020 ENDPROC;
F80B  0021
F80B  0022
F80B  0023 PROCEDURE INIT;
F80B  0024    ACIA_STATUS = $03; /* RESET */
F810  0025    ACIA_CONTROL = $11; /* 8 DATA, 2 STOP, NO PARITY */
F815  0026 ENDPROC;
F816  0027
F816  0028
F816  0029
F816  0030 PROCEDURE GETCHAR:BYTE INCHAR;
F818  0031    REPEAT UNTIL ACIA_STATUS AND RDR_FULL; /* IMPLICIT <> 0 */
F820  0032    INCHAR = ACIA_DATA AND $7F;            /* STRIP PARITY */
F827  0033 ENDPROC INCHAR;
F82C  0034
F82C  0035
F82C  0036 PROCEDURE PUTCHAR(BYTE OUTCHAR);
F82C  0037    REPEAT UNTIL ACIA_STATUS AND TDR_FULL; /* IMPLICIT <> 0 */
F834  0038    ACIA_DATA = OUTCHAR;
F839  0039 ENDPROC;
F83A  0040
F83A  0041
F83A  0042 PROCEDURE CRLF;
F83A  0043    PUTCHAR($0D);
F842  0044    PUTCHAR($0A);
F84A  0045 ENDPROC;
F84B  0046
F84B  0047
F84B  0048 PROCEDURE MAIN:BYTE CHAR;
F84D  0049    INIT;
F84F  0050    REPEAT
F84F  0051       CHAR = GETCHAR;
F853  0052       IF CHAR = $0D
F855  0053          THEN CRLF;
F85D  0054          ELSE PUTCHAR(CHAR);
F868  0055    FOREVER;
```

9.12.01 DUMB-BUG ... A SELF STARTING PROGRAM (continued)

```
0000  0001 /* DUMB-BUG FOR R6551 ACIA'S */
0000  0002
0000  0003
0000  0004 CONSTANT PROM_BASE  = $F800,
0000  0005          STACK_INIT = $E7FF,
0000  0006
0000  0007          RDR_FULL   = $08,
0000  0008          TDR_FULL   = $10;
0000  0009
0000  0010 AT $E300: BYTE ACIA_DATA, ACIA_STATUS, ACIA_COMMAND, ACIA_CONTROL;
0000  0011
0000  0012
0000  0013 ORIGIN = PROM_BASE;
F800  0014
F800  0015 STACK = STACK_INIT;
F804  0016
F804  0017
F804  0018 PROCEDURE RESET;
F807  0019    JUMP PROM_BASE;
F80A  0020 ENDPROC;
F80B  0021
F80B  0022
F80B  0023 PROCEDURE INIT;
F80B  0024    ACIA_STATUS = $FF; /* RESET */
F810  0025    ACIA_CONTROL = $9E; /* 9600 BAUD, 8 DATA, 2 STOP */
F815  0026    ACIA_COMMAND = $0B; /* NO PARITY, FULL DUPLEX, NO INTERRUPTS */
F81A  0027 ENDPROC;
F81B  0028
F81B  0029
F81B  0030 PROCEDURE GETCHAR:BYTE INCHAR;
F81D  0031    REPEAT UNTIL ACIA_STATUS AND RDR_FULL; /* IMPLICIT <> 0 */
F825  0032    INCHAR = ACIA_DATA AND $7F;            /* STRIP PARITY */
F82C  0033 ENDPROC INCHAR;
F831  0034
F831  0035
F831  0036 PROCEDURE PUTCHAR(BYTE OUTCHAR);
F831  0037    REPEAT UNTIL ACIA_STATUS AND TDR_FULL; /* IMPLICIT <> 0 */
F839  0038    ACIA_DATA = OUTCHAR;
F83E  0039 ENDPROC;
F83F  0040
F83F  0041
F83F  0042 PROCEDURE CRLF;
F83F  0043    PUTCHAR($0D);
F847  0044    PUTCHAR($0A);
F84F  0045 ENDPROC;
F850  0046
F850  0047
F850  0048 PROCEDURE MAIN:BYTE CHAR;
F852  0049    INIT;
F854  0050    REPEAT
F854  0051       CHAR = GETCHAR;
F858  0052       IF CHAR = $0D
F85A  0053          THEN CRLF;
F862  0054          ELSE PUTCHAR(CHAR);
F86D  0055    FOREVER;
```

## 9.12.02 A PL/9 MINI-MONITOR FOR THE MC6809

What follows is a complete program designed to illustrate the structures required to build PL/9 programs that start up automatically on RESET.

The program illustrated is a scaled down system monitor with all of the basic functions. The program is presented as a guide to constructing a self starting program NOT as an exercise in writing a system monitor. The point to note is this:

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                                                                     *
*  IF YOU CAN WRITE A PROGRAM THAT HAS THE HARDWARE INTENSITY         *
*  OF A SYSTEM MONITOR IN PL/9 YOU CAN  WRITE  VIRTUALLY  ANY         *
*  OTHER HARDWARE ORIENTED PROGRAM AS WELL!                           *
*                                                                     *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

PL/9 is not the ideal choice of languages (assembler is) to handle the stack manipulation required of a system monitor, but just you TRY to do it in any other language. As a result there are several GEN statements required to achieve the desired results. The program is just under 2K in size and provides the following facilities:

MEMORY EXAMINE AND CHANGE

This facility has got to be the PRIME requirement of a System Monitor. It allows you to examine or change any memory location. Entering 'M' will result in the prompt: ADDRESS? The desired address is then entered as four HEX characters, e.g. E400. Back-spacing is not allowed and entry of the last digit signifies completion. The address just entered is then displayed on the following line with the contents of the memory location in HEX adjacent to it. You then have five options:

1. (+) which moves to NEXT memory location and displays its address and contents.

2. (-) which moves to the PREVIOUS memory location and displays its address and contents.

3. (/) which re-reads the CURRENT location. Very handy for looking at I/O ports.

4. Entering any non-HEX character, e.g. <CR>, will return you to the main menu.

5. Entering two HEX characters will result in the information being written to the current memory location. The memory location will then be read again and the write/read data compared. If they are the same the address and contents of the NEXT memory location will be displayed. If the values are not the same a '?' will be displayed and a bell code sent to the system console. The memory location will then be re-read and displayed again.

## 9.12.02 A PL/9 MINI-MONITOR FOR THE MC6809 (continued)

MEMORY EXAMINE AND CHANGE (continued)

This basic facility will allow you to enter simple machine code programs and test them. The program should be terminated in an RTS ($39) instruction to provide a clean re-entry back into the system monitor. You can also debug a machine code program by inserting a break-point in the form of an SWI ($3F) instruction in the middle of your code. You will have to make a note of the the existing byte in order to re-instate it later.

JUMP TO USER PROGRAM

this command enables you to JUMP to a specified point in memory with the MC6809 registers loaded with pre-determined values in order to enter and begin execution of a program. Entering 'J' will result in the prompt: ADDRESS? being issued. Any character other than 4 HEX digits will return you to the main menu. Once the jump address is entered the register pre-load table will be displayed and you will be prompted: O.K.?. A 'Y' response will result in the jump being executed. Any response other than 'Y' will result in you being prompted to change each of the pre-load registers one at a time. As each register name is displayed you have the choice of entering HEX digits to alter it or any non-HEX character to skip it. Once you have been prompted for each of the pre-load values the pre_load values will again be displayed with the prompt: JUMP?. A 'Y' response will result in the jump being executed. Any other response will return you to the main menu.

The program should be terminated in an RTS or an SWI instruction.

DUMP REGISTERS ON RTS

When an RTS instruction is encountered in a running program the current register values will be dumped to the display and the main menu re-entered. Since the program counter value is not valid in this instance (there is no way of knowing where the code was that pulled the program counter off the stack) it will be displayed as FFFF as a reminder.

BREAK-POINT PROCESSOR

When an SWI instruction is encountered in a program being executed the contents of the registers, the position of the stack pointer, and the value of the program counter will be dumped to the system console and the main menu re-entered.

DUMP REGISTERS ON NMI

When an NMI is received the contents of all the registers will be displayed as for the break point processor. In this instance, after the dump takes place, the program will be re-entered and execution will continue. This facility is very handy to sort out routines that 'lock up'.

## 9.12.02 A PL/9 MINI-MONITOR FOR THE MC6809 (continued)

The register dump mentioned in the preceeding discussion looks like the following in all cases:

```
AA BB DP XXXX YYYY UUUU PCPC  CC  EFHI NZVC
00 00 00 0000 0000 0000 0000  D0  1101 0000
```

The register values will obviously vary from one dump to the next.

FORMATTED MEMORY DUMP

This facility enables you to dump 256 byte blocks out to the terminal in HEX/ASCII format. It is invoked by typing 'D'. You will then be prompted: 'ADDRESS?'. The desired starting address of the dump is entered as 4 HEX digits. The dump will then commence. When 256 bytes have been dumped a prompt 'M0RE?' will be issued. Hitting any key other than 'N' or 'n' will result in the next 256 bytes being displayed. Hitting 'N' or 'n' will return you to command level.

The MINI-MONITOR also has a two additional facilities which were included to prove the point of how easy it is to access GLOBAL variables from within interrupt procedures. Whenever an IRQ is received the message 'IRQ! 0000 0000' will be displayed. The bit pattern represents the binary value of a GLOBAL integer variable which will be incremented each time an interrupt occurs. FIRQ responds in a similar manner.

The following entry points may be of interest if you want to hand enter a couple of test programs:

GETCHAR is at $F827 and returns with the keyboard character in 'A'.

PUTCHAR is at $F81B and expects the display character in 'A'.

Both of these routines alter the contents of 'D' and the CCR.

Once you have the 'MINI-MONITOR' up and running try entering the following code at $E400 using the 'M' command:

```
E400 BD
E401 F8
E402 27    (JSR $F827 ... GETCHAR)
E403 BD
E404 F8
E405 1B    (JSR $F81B ... PUTCHAR)
E406 7E
E407 E4
E408 00    (JMP $E400 ... LOOP FOREVER)
```

Now use the 'J' command to start program execution at $E400.

You should get the same 'typewriter' effect of 'DUMB-BUG'.

## 9.12.02 A PL/9 MINI-MONITOR FOR THE MC6809 (continued)

The following, alterable, assumptions have been made about the hardware environment the monitor has been designed to run in:

(1) It assumes that it is communicating through a serial port comprising an MC6850 addressed at $E004 with X16 baud rate inputs. Data is transmitted as: 8 data bits, two stop bits and no parity.

(2) It assumes that there is RAM at $E400 through $E7FF.

(3) It assumes that the system monitor ROM is based at $F800 through $FFFF. i.e. a standard 2516 (single volt 2716).

All of these assumptions are met by any GIMIX or WINDRUSH 6809 processor board and SS-50 system. If you own a Windrush EUROCARD system that uses the Rockwell R6551 ACIA the R6551 drivers in DUMB-BUG must be substituted for lines 61 through 77. Don't forget to modify lines 8 and 21 to match the address of the R6551 and don't forget to add the constants required by the procedures to the 'CONSTANT' declaration at the start of the program.

The program source listing is presented on the following pages. After the listing we will describe the program in block-diagram form. We will then describe any elements of the program that are not covered in other sections of this manual.

Several of the I/O procedures used are nothing more than extractions or modified extractions from the IOSUBS, HEXIO and BITIO libraries described in section eight.

## 9.12.02 A PL/9 MINI-MONITOR FOR THE MC6809 (continued)

```
0000  0001 /* PL9 MINI-MONITOR PROGRAM VERSION 4.0 */
0000  0002
0000  0003 CONSTANT PROM_BASE  = $F800,
0000  0004          STACK_INIT = $E7FD,
0000  0005          YSAVE      = $E7FE,
0000  0006          YSAVE_HI   = $E7,
0000  0007          YSAVE_LO   = $FE,
0000  0008          IO_BASE    = $E0,    /* TOP 8-bits OF I/O ADDRESS AREA */
0000  0009
0000  0010          _8DATA_2STOP_NOPARITY = $11, ACIA_RESET = $03,
0000  0011          RX_DATA_FULL = $01, PARITY_STRIPPER = $7F,
0000  0012          TX_DATA_EMPTY = $02,
0000  0013
0000  0014          CR = $0D, LF = $0A, SP = $20, BEL = $07,
0000  0015
0000  0016          INT_ON_MASK = $AF,
0000  0017
0000  0018          TRUE = -1, FALSE = 0, ZERO = $30, ONE = $31;
0000  0019
0000  0020
0000  0021 AT $E004:BYTE ACIA_CTRL(0), ACIA_STAT, ACIA_DATA;
0000  0022
0000  0023
0000  0024 /*
0000  0025   * * * * * * * * * * * * * * * * * * * * * * * * *
0000  0026   *  THE MAIN BODY OF THE PROGRAM CODE STARTS HERE  *
0000  0027   * * * * * * * * * * * * * * * * * * * * * * * * *
0000  0028 */
0000  0029
0000  0030 ORIGIN = PROM_BASE;
F800  0031
F800  0032 STACK  = STACK_INIT;
F804  0033
F804  0034 /*
F804  0035    * * * * * * * * * * * * * * * * * * * * * * * * *
F804  0036    * THE FIRST ITEM DECLARED WILL BE AT THE BASE OF  *
F804  0037    * THE STACK. i.e. THE STACK WILL GROW DOWN FROM   *
F804  0038    * THIS POINT.                                     *
F804  0039    * * * * * * * * * * * * * * * * * * * * * * * * *
F804  0040 */
F804  0041
F804  0042 GLOBAL BYTE    JUMP_REGISTERS(0), CONDITION_CR, A_ACCUMULATOR,
F804  0043                B_ACCUMULATOR, DIRECT_PAGE:
F804  0044
F804  0045        INTEGER X_REGISTER, Y_REGISTER, U_REGISTER, PROGRAM_CTR,
F804  0046                RTS_VECTOR:
F804  0047
F804  0048        BYTE    ERFLAG, KEYCHAR, .B_POINTER:
F804  0049
F804  0050        INTEGER FIRQ_COUNT, IRQ_COUNT;
F80B  0051
F80B  0052
F80B  0053 DPAGE = IO_BASE;  /* SHORTENS CODE WHEN ADDRESSING 'AT' VARIABLES */
F80F  0054
F80F  0055
```

9.12.02 A PL/9 MINI-MONITOR FOR THE MC6809 (continued)

```
F80F  0056 /*
F80F  0057   * * * * * * * * * * *
F80F  0058   *  I/O SUBROUTINES  *
F80F  0059   * * * * * * * * * * *
F80F  0060 */
F80F  0061 PROCEDURE INITIALIZE_CONSOLE_ACIA;
F812  0062    ACIA_CTRL = ACIA_RESET;
F816  0063    ACIA_CTRL = _8DATA_2STOP_NOPARITY;
F81A  0064 ENDPROC;
F81B  0065
F81B  0066
F81B  0067 PROCEDURE PUTCHAR(BYTE OUTCHAR);
F81B  0068    REPEAT UNTIL ACIA_STAT AND TX_DATA_EMPTY;  /* IMPLICIT <> 0 */
F822  0069    ACIA_DATA = OUTCHAR;
F826  0070 ENDPROC;
F827  0071
F827  0072
F827  0073 PROCEDURE GETCHAR:BYTE INCHAR;
F829  0074    REPEAT UNTIL ACIA_STAT AND RX_DATA_FULL;  /* IMPLICIT <> 0 */
F830  0075    INCHAR = ACIA_DATA AND PARITY_STRIPPER;
F836  0076    PUTCHAR(INCHAR);
F83E  0077 ENDPROC INCHAR;
F843  0078
F843  0079
F843  0080 PROCEDURE MACHINE_PUT; /* USE THIS ONE FROM MACHINE CODE ROUTINES */
F843  0081    PUTCHAR(ACCA);
F84B  0082 ENDPROC;
F84C  0083
F84C  0084
F84C  0085 PROCEDURE MACHINE_GET; /* USE THIS ONE FROM MACHINE CODE ROUTINES */
F84C  0086    ACCA = GETCHAR;
F850  0087 ENDPROC;
F851  0088
F851  0089
F851  0090 PROCEDURE GET_UPPER_CASE:BYTE INCHAR;
F853  0091    INCHAR = GETCHAR;
F857  0092    IF INCHAR >= 'a .AND INCHAR <= 'z
F865  0093       THEN INCHAR = INCHAR - $20;
F879  0094 ENDPROC INCHAR;
F87E  0095
F87E  0096
F87E  0097 PROCEDURE CRLF;
F87E  0098    PUTCHAR(CR);
F886  0099    PUTCHAR(LF);
F88E  0100 ENDPROC;
F88F  0101
F88F  0102
F88F  0103 PROCEDURE PRINT(BYTE .STRING);
F88F  0104    WHILE STRING                    /* IMPLICIT <> 0 */
F88F  0105       BEGIN;
F897  0106          IF STRING = '\
F89A  0107             THEN BEGIN;
F8A0  0108                  .STRING = .STRING + 1;
F8A7  0109                  IF STRING
F8A7  0110                     CASE 'N THEN CRLF;
F8B3  0111                     CASE 'B THEN PUTCHAR(BEL);
F8C5  0112                  END;
```

9.12.02 A PL/9 MINI-MONITOR FOR THE MC6809 (continued)

```
F8C5  0113              ELSE PUTCHAR(STRING);
F8D2  0114              .STRING = .STRING + 1;
F8D9  0115         END;
F8D9  0116 ENDPROC;
F8DC  0117
F8DC  0118
F8DC  0119
F8DC  0120 /*
F8DC  0121   * * * * * * * * * * *
F8DC  0122   *  BIT DUMP ROUTINE *
F8DC  0123   * * * * * * * * * * *
F8DC  0124 */
F8DC  0125 BYTE MASK $01,$02,$04,$08,$10,$20,$40,$80;
F8E4  0126
F8E4  0127
F8E4  0128 PROCEDURE BITSOUT(BYTE BITCHAR):
F8E4  0129    BYTE COUNT;
F8E6  0130    COUNT = 8;
F8EA  0131    REPEAT
F8EA  0132       IF BITCHAR AND MASK(COUNT-1) = 0
F900  0133          THEN PUTCHAR(ZERO);
F910  0134          ELSE PUTCHAR(ONE);
F91C  0135       COUNT = COUNT - 1;
F91E  0136       IF COUNT = 4
F920  0137          THEN PUTCHAR(SP);
F92F  0138    UNTIL COUNT = 0;
F935  0139 ENDPROC;
F938  0140
F938  0141
F938  0142
F938  0143 /*
F938  0144   * * * * * * * * * * *
F938  0145   *  HEX I/O ROUTINES *
F938  0146   * * * * * * * * * * *
F938  0147 */
F938  0148 PROCEDURE GET_HEX_NIBBLE:BYTE INCHAR;
F93A  0149    INCHAR = GET_UPPER_CASE;
F93F  0150    KEYCHAR = INCHAR;
F943  0151    ERFLAG = TRUE;
F947  0152    IF INCHAR >= '0 .AND INCHAR <= '9
F955  0153       THEN BEGIN
F963  0154            INCHAR = INCHAR - '0;
F969  0155            ERFLAG = FALSE;
F96B  0156            END;
F96B  0157       ELSE IF INCHAR >= 'A .AND INCHAR <= 'F
F97C  0158               THEN BEGIN
F98A  0159                    INCHAR = INCHAR - '7;
F990  0160                    ERFLAG = FALSE;
F992  0161                    END;
F992  0162 ENDPROC INCHAR;
F997  0163
F997  0164
F997  0165 PROCEDURE GET_HEX_BYTE:BYTE INCHAR;
F999  0166       INCHAR = SHIFT(GET_HEX_NIBBLE,4);
F9A1  0167       IF ERFLAG = TRUE
F9A3  0168          THEN RETURN;
F9AC  0169       INCHAR = INCHAR OR GET_HEX_NIBBLE;
F9BA  0170 ENDPROC INCHAR;
```

## 9.12.02 A PL/9 MINI-MONITOR FOR THE MC6809 (continued)

```
F9BF  0171
F9BF  0172
F9BF  0173 PROCEDURE GET_HEX_ADDRESS:INTEGER INCHAR;
F9C1  0174       INCHAR = SWAP(INTEGER(GET_HEX_BYTE));
F9C8  0175       IF ERFLAG = TRUE
F9CA  0176          THEN RETURN;
F9D3  0177       INCHAR = INCHAR OR INTEGER(GET_HEX_BYTE);
F9E0  0178 ENDPROC INCHAR;
F9E5  0179
F9E5  0180
F9E5  0181 PROCEDURE PUT_HEX_NIBBLE(BYTE OUTCHAR);
F9E5  0182    OUTCHAR = (OUTCHAR AND $0F)  +  '0; /* STRIP TOP 4-bits & MAKE ASCII */
F9ED  0183    IF OUTCHAR > '9
F9EF  0184       THEN OUTCHAR = OUTCHAR + 7;      /* A-F OFFSET */
F9FB  0185    PUTCHAR(OUTCHAR);
FA04  0186 ENDPROC;
FA05  0187
FA05  0188
FA05  0189 PROCEDURE PUT_HEX_BYTE(BYTE OUTCHAR);
FA05  0190    PUT_HEX_NIBBLE(SHIFT(OUTCHAR,-4)); /* FIRST DIGIT */
FA11  0191    PUT_HEX_NIBBLE(OUTCHAR);           /* LAST DIGIT */
FA19  0192 ENDPROC;
FA1A  0193
FA1A  0194
FA1A  0195 PROCEDURE PUT_HEX_ADDRESS(INTEGER OUTCHAR);
FA1A  0196    PUT_HEX_BYTE(SWAP(OUTCHAR)); /* FIRST TWO DIGITS */
FA24  0197    PUT_HEX_BYTE(BYTE(OUTCHAR)); /* LAST TWO DIGITS */
FA2C  0198 ENDPROC;
FA2D  0199
FA2D  0200
FA2D  0201 PROCEDURE PUT_ASCII_BYTE(BYTE CHAR);
FA2D  0202    IF CHAR < $20 .OR CHAR > $7D
FA3B  0203       THEN PUTCHAR('.);
FA52  0204       ELSE PUTCHAR(CHAR);
FA5E  0205 ENDPROC;
FA5F  0206
FA5F  0207
FA5F  0208
FA5F  0209 /*
FA5F  0210   * * * * * * * * * * * * * * * *
FA5F  0211   *  GLOBAL POINTER ROUTINES  *
FA5F  0212   * * * * * * * * * * * * * * * *
FA5F  0213 */
FA5F  0214 PROCEDURE SAVE_GLOBAL_POINTER;
FA5F  0215    GEN $10,$BF,YSAVE_HI,YSAVE_LO; /* STY YSAVE */
FA63  0216 ENDPROC;
FA64  0217
FA64  0218
FA64  0219 PROCEDURE RESTORE_GLOBAL_POINTER;
FA64  0220    GEN $10,$BE,YSAVE_HI,YSAVE_LO; /* LDY YSAVE */
FA68  0221    ACCB = IO_BASE;                /* RESTORE THE DIRECT PAGE */
FA6A  0222    GEN $1F, $9B;                  /* TFR B, DP */
FA6C  0223 ENDPROC;
FA6D  0224
FA6D  0225
FA6D  0226
```

9.12.02 A PL/9 MINI-MONITOR FOR THE MC6809 (continued)

```
FA6D  0227 /*
FA6D  0228   * * * * * * * * * * * * * * * * *
FA6D  0229   *  DUMP THE STACKED REGISTERS *
FA6D  0230   * * * * * * * * * * * * * * * * *
FA6D  0231 */
FA6D  0232 PROCEDURE ONE_SPACE; /* NICE AND CLOSE TO MINIMIZE ADDRESS RANGE! */
FA6D  0233    PUTCHAR(SP);
FA76  0234 ENDPROC;
FA77  0235
FA77  0236
FA77  0237 PROCEDURE TWO_SPACES;
FA77  0238    ONE_SPACE;
FA79  0239    ONE_SPACE;
FA7B  0240 ENDPROC;
FA7C  0241
FA7C  0242
FA7C  0243 PROCEDURE REGISTER_DUMP(INTEGER STACKBASE):BYTE COUNT, CHAR, .B_POINTER;
FA7E  0244
FA7E  0245    PRINT("\N\NAA BB DP XXXX YYYY UUUU PCPC SPSP  CC  EFHI NZVC\N");
FAC2  0246
FAC2  0247    .B_POINTER = STACKBASE + 1;     /* POINT AT 'A' */
FAC9  0248    REPEAT
FAC9  0249       PUT_HEX_BYTE(B_POINTER);
FAD3  0250       IF .B_POINTER > STACKBASE + 3
FAD9  0251          THEN BEGIN
FAE6  0252               .B_POINTER = .B_POINTER + 1;
FAED  0253               PUT_HEX_BYTE(B_POINTER);
FAF7  0254               END;
FAF7  0255       ONE_SPACE;
FAFA  0256       .B_POINTER = .B_POINTER + 1;
FB01  0257    UNTIL .B_POINTER = STACKBASE + 12;
FB12  0258
FB12  0259    PUT_HEX_ADDRESS(.B_POINTER);      /* STACK POINTER */
FB1B  0260
FB1B  0261    TWO_SPACES;
FB1E  0262    .B_POINTER = STACKBASE;
FB22  0263    PUT_HEX_BYTE(B_POINTER);                  /* CCR IN HEX  */
FB2C  0264    TWO_SPACES;
FB2F  0265    BITSOUT(B_POINTER);                       /* CCR IN BITS */
FB39  0266    CRLF;
FB3C  0267    CRLF;
FB3F  0268 ENDPROC;
FB42  0269
FB42  0270
FB42  0271
FB42  0272 /*
FB42  0273   * * * * * * * * * * * * * * * * * * * * * * *
FB42  0274   *  SOFTWARE INTERRUPT SERVICE ROUTINES  *
FB42  0275   * * * * * * * * * * * * * * * * * * * * * * *
FB42  0276 */
FB42  0277 PROCEDURE SWI;                    /* USED AS A BREAK-POINT PROCESSOR */
FB42  0278    GEN $AE,$6A;  /* LDX 10,S */
FB44  0279    GEN $30,$1F;  /* LEAX -1,X */
FB46  0280    GEN $AF,$6A;  /* STX 10,S   (-1 TO POINT TO THE SWI INSTRUCTION) */
FB48  0281
FB48  0282    RESTORE_GLOBAL_POINTER;
FB4B  0283    REGISTER_DUMP(STACK);
FB54  0284    JUMP PROM_BASE;  /* A HARD RESET OF THE MONITOR */
FB57  0285 ENDPROC;
```

9.12.02 A PL/9 MINI-MONITOR FOR THE MC6809 (continued)

```
FB58  0286
FB58  0287
FB58  0288 PROCEDURE SWI2; /* DUMMY RTI INSTRUCTION ONLY */
FB58  0289 ENDPROC;
FB59  0290
FB59  0291
FB59  0292 PROCEDURE SWI3; /* DUMMY RTI INSTRUCTION ONLY */
FB59  0293 ENDPROC;
FB5A  0294
FB5A  0295
FB5A  0296
FB5A  0297 /*
FB5A  0298   * * * * * * * * * * * * * * * * * * * * *
FB5A  0299   *  HARDWARE INTERRUPT SERVICE ROUTINES  *
FB5A  0300   * * * * * * * * * * * * * * * * * * * * *
FB5A  0301 */
FB5A  0302 PROCEDURE NMI;
FB5A  0303    RESTORE_GLOBAL_POINTER;
FB5D  0304    REGISTER_DUMP(STACK);
FB66  0305 ENDPROC;
FB67  0306
FB67  0307
FB67  0308 PROCEDURE FIRQ;
FB69  0309    RESTORE_GLOBAL_POINTER;
FB6C  0310    PRINT("\NFIRQ!  \B");
FB84  0311    FIRQ_COUNT = FIRQ_COUNT + 1;
FB8D  0312    BITSOUT(FIRQ_COUNT);
FB97  0313    CRLF;
FB9A  0314 ENDPROC;
FB9D  0315
FB9D  0316
FB9D  0317 PROCEDURE IRQ;
FB9D  0318    RESTORE_GLOBAL_POINTER;
FBA0  0319    PRINT("\NIRQ!  \B");
FBB8  0320    IRQ_COUNT = IRQ_COUNT + 1;
FBC1  0321    BITSOUT(IRQ_COUNT);
FBCB  0322    CRLF;
FBCE  0323 ENDPROC;
FBCF  0324
FBCF  0325
FBCF  0326 PROCEDURE RESET;
FBCF  0327    JUMP PROM_BASE; /* YOU MUST GO TO THE FIRST 'ORIGIN' */
FBD2  0328 ENDPROC;
FBD3  0329
FBD3  0330
```

9.12.02 A PL/9 MINI-MONITOR FOR THE MC6809 (continued)

```
FBD3  0331
FBD3  0332 /*
FBD3  0333   * * * * * * * * * * * * * * * * * * * * * * * * * * * *
FBD3  0334   *  HANDLE ARBITRARY PROGRAM COUNTER PULL FROM STACK *
FBD3  0335   * * * * * * * * * * * * * * * * * * * * * * * * * * * *
FBD3  0336 */
FBD3  0337 PROCEDURE RETURN_FROM_SUBROUTINE;
FBD3  0338
FBD3  0339    GEN $34,$FF;          /* PSHS CC,A,B,DP,X,Y,U,PC */
FBD5  0340    GEN $8E,$FF,$FF;      /* LDX  #$FFFF */
FBD8  0341    GEN $AF,$6A;           /* STX  10, S ... (PC IS NOW $FFFF) */
FBDA  0342
FBDA  0343    RESTORE_GLOBAL_POINTER;
FBDD  0344    REGISTER_DUMP(STACK);
FBE6  0345    JUMP PROM_BASE; /* A HARD RESET OF THE MONITOR */
FBE9  0346 ENDPROC;
FBEA  0347
FBEA  0348
FBEA  0349
FBEA  0350 /*
FBEA  0351   * * * * * * * * * * * * * * * * *
FBEA  0352   *  MEMORY EXAMINE AND CHANGE  *
FBEA  0353   * * * * * * * * * * * * * * * * *
FBEA  0354 */
FBEA  0355 PROCEDURE ADDRESS_PROMPT;
FBEA  0356    PRINT("\NADDRESS? ");
FC03  0357 ENDPROC GET_HEX_ADDRESS;
FC07  0358
FC07  0359
FC07  0360 PROCEDURE MEMORY_EXAMINE_AND_CHANGE:
FC07  0361    BYTE READ_BYTE, WRITE_BYTE, .ADDRESS;
FC09  0362
FC09  0363    .ADDRESS = ADDRESS_PROMPT;
FC0D  0364    IF ERFLAG = TRUE
FC0F  0365       THEN RETURN;
FC18  0366
FC18  0367    PRINT("\N\N(+) next\N
FC27  0368 (-) prev\N
FC2F  0369 (/) again\N");
FC49  0370
FC49  0371 TRY_AGAIN:
FC49  0372    CRLF;
FC4C  0373 READ_AGAIN:
FC4C  0374    PUTCHAR(CR);
FC55  0375    PUT_HEX_ADDRESS(.ADDRESS);
FC5E  0376    ONE_SPACE;
FC61  0377    PUT_HEX_BYTE(ADDRESS);
FC6B  0378    ONE_SPACE;
FC6E  0379
FC6E  0380    WRITE_BYTE = GET_HEX_BYTE;
FC73  0381    IF ERFLAG = TRUE
FC75  0382       THEN IF KEYCHAR = '- .OR KEYCHAR = '+  .OR KEYCHAR = '/
FC97  0383               THEN BEGIN;
FCA5  0384                    IF KEYCHAR
FCA5  0385                       CASE '- THEN .ADDRESS = .ADDRESS-1;
FCB5  0386                       CASE '+  THEN .ADDRESS = .ADDRESS+1;
FCC5  0387                       CASE '/ THEN GOTO READ_AGAIN;
FCD1  0388                    GOTO TRY_AGAIN;
FCD4  0389                    END;
```

9.12.02 A PL/9 MINI-MONITOR FOR THE MC6809 (continued)

```
FCD4  0390                ELSE RETURN;
FCDA  0391
FCDA  0392     ADDRESS = WRITE_BYTE;
FCDF  0393     IF WRITE_BYTE <> ADDRESS
FCE1  0394        THEN BEGIN;
FCE8  0395             PRINT (" ?\B");
FCFA  0396             GOTO TRY_AGAIN;
FCFD  0397             END;
FCFD  0398     .ADDRESS = .ADDRESS + 1;
FD04  0399     GOTO TRY_AGAIN;
FD07  0400 ENDPROC;
FD0A  0401
FD0A  0402
FD0A  0403
FD0A  0404 /*
FD0A  0405   * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
FD0A  0406   * JUMP TO ADDRESS SPECIFIED BY USER WITH REGISTERS PRE-LOADED *
FD0A  0407   * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
FD0A  0408 */
FD0A  0409 BYTE PROMPT "CC",
FD0D  0410              " A",
FD10  0411              " B",
FD13  0412              "DP",
FD16  0413              " X",
FD19  0414              " Y",
FD1C  0415              " U";
FD1F  0416
FD1F  0417
FD1F  0418 BYTE PRE_LOAD "\N\NPRE-LOAD VALUES";
FD33  0419
FD33  0420
FD33  0421 PROCEDURE JUMP_TO_USER_PROGRAM:
FD33  0422     INTEGER JUMP_ADDRESS, IN_ADDRESS:
FD33  0423     BYTE    COUNT, IN_BYTE;
FD35  0424
FD35  0425 TRY_AGAIN:
FD35  0426     IN_ADDRESS = ADDRESS_PROMPT;
FD3A  0427     IF ERFLAG = TRUE
FD3C  0428        THEN RETURN;
FD45  0429     PROGRAM_CTR = IN_ADDRESS;
FD49  0430
FD49  0431     PRINT(.PRE_LOAD);
FD53  0432
FD53  0433     REGISTER_DUMP(.JUMP_REGISTERS);
FD5C  0434
FD5C  0435 WRONG_ANSWER:
FD5C  0436     PRINT("\NO.K.? (Y/N) ");
FD78  0437
FD78  0438     IN_BYTE = GET_UPPER_CASE;
FD7D  0439     IF IN_BYTE = 'N
FD7F  0440        THEN BEGIN;
FD85  0441             COUNT = 0;
FD87  0442             CRLF;
FD8A  0443             PRINT("\N<CR> to skip\NHEX  to alter\N\N");
FDB9  0444
```

9.12.02 A PL/9 MINI-MONITOR FOR THE MC6809 (continued)

```
FDB9  0445            REPEAT
FDB9  0446               PRINT(.PROMPT(COUNT*3));
FDF1  0447               PRINT("? ");
FE01  0448               IF COUNT < 4
FE03  0449                  THEN BEGIN;
FE09  0450                       IN_BYTE = GET_HEX_BYTE;
FE0E  0451                       IF ERFLAG = FALSE
FE10  0452                          THEN JUMP_REGISTERS(COUNT) = IN_BYTE;
FE21  0453                       END;
FE21  0454                  ELSE BEGIN;
FE24  0455                       IN_ADDRESS = GET_HEX_ADDRESS;
FE29  0456                       IF ERFLAG = FALSE
FE2B  0457                          THEN X_REGISTER(COUNT-4) = IN_ADDRESS;
FE40  0458                       END;
FE40  0459               CRLF;
FE43  0460               COUNT = COUNT + 1;
FE45  0461            UNTIL COUNT = 7;
FE4D  0462
FE4D  0463            PRINT(.PRE_LOAD);
FE58  0464            REGISTER_DUMP(.JUMP_REGISTERS);
FE61  0465            PRINT("\NJUMP? (Y/N) ");
FE7D  0466            IF GET_UPPER_CASE <> 'Y
FE80  0467               THEN RETURN;
FE89  0468            END;
FE89  0469         ELSE IF IN_BYTE <> 'Y
FE8E  0470               THEN GOTO WRONG_ANSWER;
FE97  0471
FE97  0472      STACK = .JUMP_REGISTERS;
FE9D  0473      GEN $35,$FF; /* PULS CC,D,DP,X,Y,U,PC ... LOAD EM UP AND GO! */
FE9F  0474   ENDPROC;
FEA2  0475
FEA2  0476
FEA2  0477
FEA2  0478   /*
FEA2  0479     ******************
FEA2  0480     * HEX/ASCII DUMP *
FEA2  0481     ******************
FEA2  0482   */
FEA2  0483   PROCEDURE HEX_DUMP:BYTE .ADDRESS, COUNT, PASSES;
FEA4  0484      .ADDRESS = ADDRESS_PROMPT;
FEA9  0485      CRLF;
FEAC  0486   DUMP_AGAIN:
FEAC  0487      PASSES = 0;
FEAE  0488      REPEAT
FEAE  0489         COUNT = 0;
FEB0  0490         CRLF;
FEB3  0491         PUT_HEX_ADDRESS(.ADDRESS);
FEBC  0492         TWO_SPACES;
FEBF  0493         REPEAT
FEBF  0494            PUT_HEX_BYTE(ADDRESS(COUNT));
FECF  0495            ONE_SPACE;
FED2  0496            COUNT = COUNT + 1;
FED4  0497         UNTIL COUNT = 16;
FEDA  0498         TWO_SPACES;
FEDD  0499         COUNT = 0;
```

## 9.12.02 A PL/9 MINI-MONITOR FOR THE MC6809 (continued)

```
FEDF  0500        REPEAT
FEDF  0501           PUT_ASCII_BYTE(ADDRESS(COUNT));
FEEF  0502           COUNT = COUNT + 1;
FEF1  0503        UNTIL COUNT = 16;
FEF7  0504        .ADDRESS = .ADDRESS + 16;
FEFE  0505        PASSES = PASSES + 1;
FF00  0506     UNTIL PASSES = 16;
FF06  0507     PRINT(" MORE?\B");
FF1C  0508     IF GET_UPPER_CASE <> 'N
FF1F  0509        THEN GOTO DUMP_AGAIN;
FF27  0510  ENDPROC;
FF2A  0511
FF2A  0512
FF2A  0513
FF2A  0514  /*
FF2A  0515    * * * * * * * * * * * * * * * * * * *
FF2A  0516    *  PL/9 MINI-MONITOR MAIN PROGRAM *
FF2A  0517    * * * * * * * * * * * * * * * * * * *
FF2A  0518  */
FF2A  0519  PROCEDURE MINI_MONITOR:BYTE COUNT;
FF2C  0520
FF2C  0521     SAVE_GLOBAL_POINTER;   /* A VERY IMPORTANT STEP IF YOU WANT TO ACCESS
FF2F  0522                               GLOBAL VARIABLE FROM INTERRUPT ROUTINES    */
FF2F  0523     FIRQ_COUNT = 0;
FF35  0524     IRQ_COUNT = 0;
FF3B  0525
FF3B  0526     INITIALIZE_CONSOLE_ACIA;
FF3E  0527
FF3E  0528     CONDITION_CR = $D0; /* INITIALIZE JUMP REGISTERS */
FF42  0529     COUNT = 1;
FF46  0530     REPEAT
FF46  0531        JUMP_REGISTERS(COUNT) = 0;
FF4F  0532        COUNT = COUNT + 1;
FF51  0533     UNTIL COUNT = 12;
FF57  0534
FF57  0535     RTS_VECTOR = .RETURN_FROM_SUBROUTINE;
FF5F  0536
FF5F  0537     CCR = CCR AND INT_ON_MASK;  /* ENABLE FIRQ AND IRQ */
FF65  0538
FF65  0539     CRLF;
FF68  0540
FF68  0541     REPEAT
FF68  0542        PRINT("PL/9 MINI-MONITOR V:4.0\N> ");
FF91  0543
FF91  0544        IF GET_UPPER_CASE
FF94  0545           CASE 'D THEN HEX_DUMP;
FF9E  0546           CASE 'J THEN JUMP_TO_USER_PROGRAM;
FFAA  0547           CASE 'M THEN MEMORY_EXAMINE_AND_CHANGE;
FFB6  0548           ELSE PRINT (" WHAT?\B");
FFCF  0549
FFCF  0550        CRLF;
FFD2  0551        CRLF;
FFD5  0552     FOREVER;
```

9.12.02 A PL/9 MINI-MONITOR FOR THE MC6809 (continued)

```
PROCEDURES:
INITIALIZE_CONSOLE_ACIA F812
PUTCHAR                 F81B
GETCHAR                 F827   BYTE
MACHINE_PUT             F843
MACHINE_GET             F84C
GET_UPPER_CASE          F851   BYTE
CRLF                    F87E
PRINT                   F88F
BITSOUT                 F8E4
GET_HEX_NIBBLE          F938   BYTE
GET_HEX_BYTE            F997   BYTE
GET_HEX_ADDRESS         F9BF   INTEGER
PUT_HEX_NIBBLE          F9E5
PUT_HEX_BYTE            FA05
PUT_HEX_ADDRESS         FA1A
PUT_ASCII_BYTE          FA2D
SAVE_GLOBAL_POINTER     FA5F
RESTORE_GLOBAL_POINTER  FA64
ONE_SPACE               FA6D
TWO_SPACES              FA77
REGISTER_DUMP           FA7C
SWI                     FB42
SWI2                    FB58
SWI3                    FB59
NMI                     FB5A
FIRQ                    FB67
IRQ                     FB9D
RESET                   FBCF
RETURN_FROM_SUBROUTINE  FBD3
ADDRESS_PROMPT          FBEA   INTEGER
MEMORY_EXAMINE_AND_CHANGE FC07
JUMP_TO_USER_PROGRAM    FD33
HEX_DUMP                FEA2
MINI_MONITOR            FF2A

DATA:
MASK                    F8DC   BYTE
PROMPT                  FD0A   BYTE
PRE_LOAD                FD1F   BYTE

EXTERNALS:
PROM_BASE               F800
STACK_INIT              E7FD
YSAVE                   E7FE
YSAVE_HI                00E7
YSAVE_LO                00FE
IO_BASE                 00E0
_8DATA_2STOP_NOPARITY   0011
ACIA_RESET              0003
RX_DATA_FULL            0001
PARITY_STRIPPER         007F
TX_DATA_EMPTY           0002
CR                      000D
```

## 9.12.02 A PL/9 MINI-MONITOR FOR THE MC6809 (continued)

```
LF                      000A
SP                      0020
BEL                     0007
INT_ON_MASK             00AF
TRUE                    FFFF
FALSE                   0000
ZERO                    0030
ONE                     0031
ACIA_CTRL               E004    BYTE
ACIA_STAT               E004    BYTE
ACIA_DATA               E005    BYTE

GLOBALS:
JUMP_REGISTERS          0000    BYTE
CONDITION_CR            0000    BYTE
A_ACCUMULATOR           0001    BYTE
B_ACCUMULATOR           0002    BYTE
DIRECT_PAGE             0003    BYTE
X_REGISTER              0004    INTEGER
Y_REGISTER              0006    INTEGER
U_REGISTER              0008    INTEGER
PROGRAM_CTR             000A    INTEGER
RTS_VECTOR              000C    INTEGER
ERFLAG                  000E    BYTE
KEYCHAR                 000F    BYTE
B_POINTER               0010    BYTE
FIRQ_COUNT              0012    INTEGER
IRQ_COUNT               0014    INTEGER

 FIRQ at $FB67
  IRQ at $FB9D
  NMI at $FB5A
  SWI at $FB42
 SWI2 at $FB58
 SWI3 at $FB59
RESET at $FBCF
```

## 9.12.02 A PL/9 MINI-MONITOR FOR THE MC6809 (continued)

This section will describe the main points of the preceeding program to give you a better understanding of the constructions required to to make a self starting program.

Before we start explaining the body of the program let's look at the block diagram on the following page and compare it with the text of the program it represents:

(1) lines 2 through 18, with line 1 being the program title.
(2) line 21
(3) line 30
(4) line 32
(5) lines 42 through 50
(6) line 53
(7) lines 61 through 205
(8) lines 214 through 323
(9) lines 326 through 328
(10) lines 337 through 510
(11) lines 519 through 552

Upon power up, or hardware RESET, the MC6809 processor will be vectored to the start of the procedure named RESET at which point it will begin to execute code and will ultimately transfer control to the first declared origin (12).

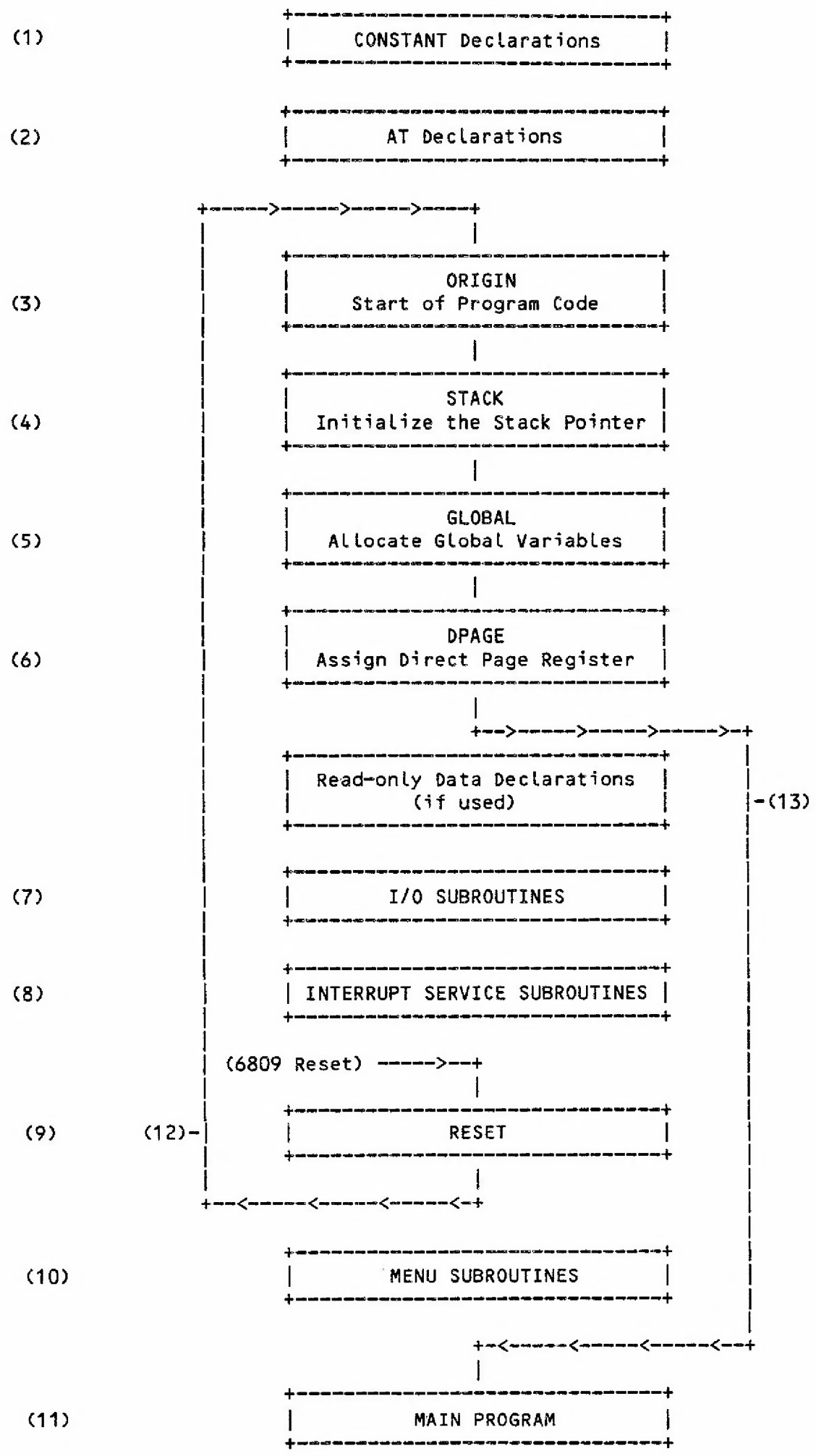
```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                                                         *
*  The RESET routine MUST, repeat MUST always end in      *
*  a JUMP to the FIRST declared ORIGIN of the program.    *
*                                                         *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

This point is very important. PL/9 will always generate the code for the stack, global and direct page assignments just AFTER the first declared origin and then transfer control to the last procedure in the program (13). The only reason we have not forced RESET to point to the first ORIGIN and have provided a procedure named RESET is to allow experienced assembly language programmers some extra flexibility in what goes on before the main PL/9 program is started.

In this example we have stated JUMP PROM_BASE as being the only code in the RESET procedure. Using a constant to define PROM_BASE greatly simplifies the changes to the program as it is also used to set the origin, viz ORIGIN=PROM_BASE. Thus all we have to do is alter the value declared for PROM_BASE and we will automatically relocate the program as well as the point that the RESET procedure will jump to.

```
+ + + + + + + + + + + + + + + + + + + + + + + + + + + + + +
+ MORE OFTEN THAN NOT A SIMPLE JUMP TO THE ORIGIN IS      +
+ ALL THAT WILL BE PRESENT IN THE RESET PROCEDURE.        +
+ + + + + + + + + + + + + + + + + + + + + + + + + + + + + +
```

Constants are an ideal mechanism to help simplify changes to PL/9 programs. Any item that is likely to change should be declared as a constant or part of a read-only data table near the start of the program. This will circumvent the drudgery of having to dig through the body of the code to look for hidden references when the inevitable changes are to be incorporated.


(1)
CONSTANT Declarations
(2)
AT Declarations
(3)
ORIGIN
Start of Program Code
(4)
STACK
Initialize the Stack Pointer
(5)
GLOBAL
Allocate Global Variables
(6)
DPAGE
Assign Direct Page Register
Read-only Data Declarations
(if used)
(13)
(7)
I/O SUBROUTINES
(8)
INTERRUPT SERVICE SUBROUTINES
(6809 Reset)
(9)
(12)
RESET
(10)
MENU SUBROUTINES
(11)
MAIN PROGRAM

9.12.02 A PL/9 MINI-MONITOR PROGRAM FOR THE MC6809 (continued)

The sequence of events at power-up or hardware reset is illustrated by the dashed lines on the block diagram.

On reset the MC6809 will be vectored to the start of the RESET procedure (9). The RESET procedure may then do other things (usually written in assembly language using only the registers and branch/jump instructions as the stack is not initialized at this point) but MUST end with a jump to the first declared ORIGIN. This jump is represented by (12).

The PL/9 program is now entered at (3) where the STACK is initialized (4), the GLOBAL variables allocated (5), and the direct page assigned (6). When the compiler has determined that the program header has ended, as signified by a read-only data declaration OR a procedure declaration it will insert a long branch to the last procedure in the file. In this case the long branch will be immediately after the DPAGE declaration and is represented by (13).

THIS LAST POINT IS CRITICAL if you use multiple ORIGIN statements. If, for example, we had a second ORIGIN statement between lines 54 and 55 the program positively and unequivocally would NOT work!

The reason is the way the compiler senses the end of the program header. The compiler is looking for one of two events to take place to signify the end of the program header:

(1) A read-only data declaration.
(2) A procedure declaration.

When one of either of these two events takes place the compiler makes a note of the current address and reserves three bytes for the long branch to the last procedure in the file before it continues to compile the program.

If you make a second ORIGIN statement before one of these two events takes place (remember that PL/9 is a single pass compiler) the compiler will simply adjust the program counter value accordingly. This has the effect of leaving a gap between the code just generated, in this example the DPAGE assignment, and the forward branch to the last procedure. This gap in executable code will cause the program to crash in no uncertain terms. See section 7.01.03 for further details.

Once the program header has been executed and the forward branch successfully navigated the MAIN program will then be started up. It is very important that the MAIN program include (or have calls to) all of the hardware and variable initialization routines BEFORE the body of the MAIN procedure, usually a REPEAT...FOREVER loop, is entered.

Another important point to note is that even if you are not intending to use SWI, SWI2, SWI3, NMI, FIRQ, or IRQ you should declare the procedures and an ENDPROC as follows:

```
PROCEDURE SWI;  ENDPROC;
PROCEDURE SWI2; ENDPROC;
PROCEDURE SWI3; ENDPROC;
PROCEDURE NMI;  ENDPROC;
PROCEDURE FIRQ; ENDPROC;
PROCEDURE IRQ;  ENDPROC;
```

This will ensure that the MC6809 vector table points to dummy RTI instructions in all cases. This will prevent a system crash should an accidental interrupt occur. This bit of security only costs you six bytes (six RTI instructions) and a very small overhead in the text file.

## 9.12.02 A PL/9 MINI-MONITOR PROGRAM FOR THE MC6809 (continued)

### A MINI-MONITOR FOR THE MC6809

The following sections contain an outline of the procedures used to develop the system monitor program detailed on the preceeding pages.

### PROGRAM HEADER

The program header, lines 1 through 53 should be self explanatory at this stage and will not be covered. If what we are doing is not obvious we suggest that you go back and read the appropriate section of the Language Reference Manual or the Users Guide.

### I/O SUBROUTINES

Lines 61 through 77 deal with initializing and communicating through an MC6850 ACIA, some understanding of an MC6850 is therefore required to make any sense of these routines.

Lines 80 through 87 are included to facilitate writing machine code routines that communicate through the system console.

Lines 90 through 116 are extracted from the IOSUBS library, lines 125 through 139 are from the BITIO library, lines 148 through 198 are from the HEXIO library. All of these procedures are covered in section nine.

### INTERRUPT SERVICE SUBROUTINES

Lines 219 through 245 are procedures to save and restore the global variable pointer 'Y' and the 'DP' register. Routines similar to this MUST be used if you wish to access GLOBAL variables including AT variables on the direct page established by 'DPAGE' from within interrupt procedures. This topic was discussed in detail in section 9.11.00.

Lines 232 through 268 form three procedures involved with displaying the contents of the MC6809 register set that has been pushed onto the hardware stack.

Upon entry the procedure 'REGISTER_DUMP' is passed an integer variable (STACKBASE) containing the address of the base of the stack. The way we are going to use this integer dictates that it is not defined as a pointer, i.e. '.STACKBASE'. Line 245 prints a heading on the video display in order to define the information contained on the next line. The line of register data built up by the program between line 248 to line 265.

First A, B, and DP are displayed in HEX, then X, Y, U, and the Program Counter are displayed in HEX by the loop between line 248 and line 257. Then the Stack pointer in HEX, followed by the CCR in HEX followed by the bit pattern of the CCR are displayed by the program between line 259 and line 265.

## 9.12.02 A PL/9 MINI-MONITOR PROGRAM FOR THE MC6809 (continued)

As mentioned previously STACKBASE will contain the address of the base of the saved registers. Since the registers are saved one on top of the other the data may be considered to be a vector table of BYTE elements organized as follows:

```
Highest address ---> PC-LO
      |              PC-HI
      |              U-LO
      |              U-HI
      |              Y-LO
      |              Y-HI
      |              X-LO
      |              X-HI
      |              DP
      |              B
      |              A   <----- STACKBASE + 1.
Lowest address --->  CCR <------ STACKBASE contains the address of this variable.
```

The mechanism we use to output the contents of the data present on the hardware stack is a good example of a practical use of pointers.

Before entering the loop between lines 248 and 257 the contents of STACKBASE (plus one) are assigned to a BYTE pointer called '.B_POINTER'. This pointer now contains the address of the byte associated with the 'A' accumulator.

On the first three iterations of the loop between lines 248 and 257 '.B_POINTER' will be less than STACKBASE + 3 so the code between lines 251 and 254 will not be executed. This results in BYTE sized variables being printed for the first three iterations ('A', 'B' and 'DP'). On subsequent iterations the code between lines 251 and 254 will be executed. This results in INTEGER sized variables being printed for the last four iterations ('X', 'Y', 'U' and 'PC').

Line 259 prints the address held in '.B_POINTER' (NOT the byte pointed to by '.B_POINTER') which at this stage is the address where the hardware stack pointer ('SP') will be after the register set is pulled.

Line 262 aims '.B_POINTER' at the 'CCR' value on the stack.

Line 263 prints the 'CCR' in HEX. Line 265 prints the 'CCR' as a binary bit pattern.

The function procedures 'PUT_HEX_BYTE' and 'BITSOUT' expect a BYTE sized variable to be passed to them. The function procedure 'PUT_HEX_ADDRESS' expects an INTEGER sized variable to be passed to it.

Note how we handle '.B_POINTER' differently in lines 259 and line 263. Line 259 can be read as: "Print the HEX address contained in the variable called '.B_POINTER'". Line 263 can be read as: "Print the HEX byte pointed to by the address held in '.B_POINTER'".

The only difference between the two lines is that line 259 contains the 'dot' in front of the pointers name (which means treat me like any other variable and use my CONTENTS as data) and line 263 does not include the 'dot' in front of the pointers name (which means treat me as a 'window' and use my contents to tell you where the data you want is stored).

Pointers are discussed in detail in their own section.

## 9.12.02 A PL/9 MINI-MONITOR PROGRAM FOR THE MC6809 (continued)

Lines 277 through 328 inclusive are the routines that will be executed in response to software or hardware interrupts being presented to the MC6809.

When a software interrupt occurs the entire register set, including the program counter, will be pushed onto the stack and the SWI interrupt service routine will be entered. When an RTI, return from interrupt, instruction is encountered, the saved register set, including the program counter, will be pulled of the stack causing execution of the interrupted program to continue as though the interrupt had never occurred.

Lines 277 through 285 are the SWI service routine which is being used as a simple break-point processor. The purpose of the break-point processor is to dump the contents of the MC6809 registers and return to system monitor command level.

The GEN statements on lines 278 through 304 takes a copy of the saved program counter off the stack, subtracts one from it and returns it to the stack. This is done to compensate for the fact that the program counter would have been incremented by one when the SWI instruction was executed. Subtracting one from the saved program counter ensures that the displayed value for 'PC' will point to the location of the SWI instruction NOT one byte past it.

Line 282 recovers the 'DP' and 'Y' registers so we can access GLOBAL variables and AT variables, which in this case include the ACIA I/O device. Line 283 then passes the value of the hardware stack pointer to REGISTER_DUMP. The stacked registers will then be displayed as described previously.

Line 284 forces a complete RE-START of the system monitor by vectoring back to the first ORIGIN statement. This is a crude, but effective, way of avoiding the RTI instruction that will be generated by line 285.

Lines 288 through 293 are dummy interrupt procedures that simply ensure that the MC6809 interrupt vector table points to RTI instructions.

The NMI interrupt service routine is virtually identical to the SWI routine except for two points. The first is that the PC value is not altered. The second is that the routine terminates in an RTI. This routine will simply dump the register set whenever an NMI occurs and then return to the interrupted program as though nothing had happened.

You will note that line 303 restores the GLOBAL pointer. Why should this be necessary when the NMI routine does not access any GLOBAL variables? The reason is simple as well as fundamental. Whenever an interrupt procedure calls other PL/9 procedures (that may in turn call other PL/9 procedures, that may in turn call other PL/9 procedures.....) you should assume that somewhere down the chain a global variable may be used. The four bytes of code required to restore the global pointer can give you a lot of piece of mind in these cases.

If the entire interrupt procedure is self contained and does not make any references to global variables and you are 100% certain that none of the procedures that are used by the interrupt procedure make use of GLOBAL variables restoration of the 'Y' index register is not required.

The same thing applies to the 'DP' register if you are intending to access any of the AT variables on the direct page which was established by 'DPAGE'. Since 'DPAGE' is most often used to point to the direct page where the I/O devices are and interrupt procedures, more often than not, access I/O devices and store the results in GLOBAL variables, it is good practice to restore 'DP' and 'Y' within all interrupt procedures as a general practice in your programming.

## 9.12.02 A PL/9 MINI-MONITOR PROGRAM FOR THE MC6809 (continued)

The FIRQ interrupt service routine is a simple demonstration of how easy it is to access global variables from within an interrupt procedure. In this instance we are simply going to increment a global variable called FIRQ_COUNT and display its bit pattern each time a FIRQ occurs.

There is one important point to note about the way PL/9 treats FIRQ. If you know about FIRQ you will remember that FIRQ only saves the CCR and the Program Counter before being vectored to the interrupt service routine. Thus a normal FIRQ routine cannot make any use of the registers and is generally restricted to the TST, INC, DEC, and CLR instructions in assembly language. Since PL/9 procedures require the use of the 'A', 'B' and 'X' registers the FIRQ interrupt is not of much use to PL/9 procedures as it stands. Add to this that PL/9 will require you to restore the 'DP' if AT variables on the direct page are being accessed and the 'Y' register if GLOBAL variables are being used and you can see that a normal FIRQ is pretty useless to PL/9 procedures.

To make the FIRQ usable PL/9 will produce the code ($34 $7E) to push the remaining registers (D, DP, X, Y, and U) onto the stack at the start of the FIRQ procedure. At the end of the FIRQ interrupt procedure PL/9 will generate the code ($35 $7E) to pull these registers off the stack before executing the RTI. All of this is invisible to the PL/9 programmer. FIRQ can be treated just the same as IRQ but because of the code required for the extra push/pull operations it will be SLOWER than the normal IRQ routine.

Obviously there is nothing to stop you from producing an assembly language FIRQ service routine as an ASMPROC which ends in an RTI. In this case you will have to manually insert the ASMPROC's starting address in the FIRQ interrupt vector.

If the vector is in RAM your program can automatically install the vector by using the following statement:

```
AT $E742:INTEGER FIRQVEC;
```

Then, assuming that your ASMPROC for FIRQ is called 'FIRQ_SERVE' you would put the following code at the start of your main program before you enable the FIRQ flag in the CCR:

```
FIRQVEC = .FIRQ_SERVE;     /* note the 'dot' */
```

If the FIRQ vector is in ROM the problem is a bit more difficult. Either you must manually insert the address of FIRQ_SERVE into the vector at $FFF6 at the time you program the EPROM or you must fiddle the compiler with multiple ORIGIN statements and generate a READ-ONLY vector at $FFF6 just after your ASMPROC:

```
ASMPROC FIRQ_SERVE;
   GEN $3B;  /* RTI */

ORIGIN = $FFF6
INTEGER FIRQ_VEC .FIRQ_SERVE;

ORIGIN = $xxxx  /* one byte past the RTI instruction ($3B) as determined
                   by A:T,C compile option. */
```

The IRQ interrupt service routine is virtually identical to the FIRQ routine.

The RESET routine simply performs the MANDATORY jump to the first ORIGIN.

## 9.12.02 A PL/9 MINI-MONITOR PROGRAM FOR THE MC6809 (continued)

### MENU SUBROUTINES

### RETURN FROM SUBROUTINE

Lines 337 through 346 handle the eventuality of random program counter pull from the stack. This routine is used in conjunction with the MINI-MONITOR 'J' command to provide a simple technique for terminating simple subroutines entered into memory in machine code. The way we have organized the GLOBAL variables is very important to the operation of the system monitor and bears looking at in detail at this juncture.

```
Highest address ---> RTS_VECTOR (LO)
      |              RTS_VECTOR (HI) <--- SP points here after program entered
      |              PROGRAM_CTR (LO)
      |              PROGRAM_CTR (HI)
      |              U_REGISTER (LO)
      |              U_REGISTER (HI)
      |              Y_REGISTER (LO)
      |              Y_REGISTER (HI)
      |              X_REGISTER (LO)
      |              X_REGISTER (HI)
      |              DIRECT_PAGE
      |              B_ACCUMULATOR
      |              A_ACCUMULATOR
Lowest address  -->  CONDITION_CR   <--- SP points here before 'J' is executed
```

The address of CONDITION_CR is also occupied by the variable 'JUMP_REGISTERS' which has been declared as a vector of nil size. This is a handy technique when you want to call the same location in memory by more than one name.

The first thing you should notice about the order of the variables is that it is identical to the natural order the MC6809 pushes and pulls registers on the hardware stack. This is no coincidence! The one extra item in the list is 'RTS_VECTOR'. If you understand the way the MC6809 works you will know that when an RTS instruction is encountered the MC6809 simply pulls the next two bytes off the stack into the program counter and commences executing code at that address.

The MINI-MONITOR (as well as the Windrush GT-BUG System Monitor) take advantage of this fact by providing a sensible address at the memory location that will be pulled into the program counter when an RTS is encountered. After the 'J' command (discussed shortly) is executed it will leave the stack pointer pointing to 'RTS_VECTOR'. When, and if, the users machine code program terminates in an RTS the address stored in 'RTS_VECTOR' will be pulled into the program counter and execution will commence at that address. The MAIN procedure initializes RTS_VECTOR to contain the address of the procedure 'RETURN_FROM_SUBROUTINE'.

This routine is located between line 337 and line 346. This routine is designed to dump the register set and return to the MINI-MONITOR. Since an RTS instruction does not save any registers the first thing that must be done is to save them. This is done in line 339. Since the value of the PC on the stack is meaningless lines 340 through 341 set it to $FFFF. Line 344 passes the value of the hardware stack pointer to REGISTER_DUMP which performs the register dump already discussed. The procedure is terminated by a JUMP back to the first ORIGIN which will re-start the MONITOR.

9.12.02 A PL/9 MINI-MONITOR PROGRAM FOR THE MC6809 (continued)

MEMORY EXAMINE AND CHANGE

This procedure is contained between lines 360 and 400. The small procedure between lines 355 and 357 sends a message to the system console which prompts 'ADDRESS?' and then waits for 4 HEX bytes to be entered. The implication of it being designed as a small subroutine is that it will be used more than once.

The main memory examine and change procedure is located between lines 360 and 400. Line 365 'calls' the message subroutine 'ADDRESS_PROMPT' and places the returned 4 digit HEX number in a BYTE sized pointer called '.ADDRESS'. Line 364 tests to see if the HEX address returned contains an error i.e. a non HEX digit was entered. If an error is detected the procedure is terminated on the spot by line 365.

If no errors were detected another set of prompts is issued, a CRLF, then a CR sent to the system console. You will note that there are two labels in the program. One at line 371 the other at line 373. These are going to be used for subsequent GOTO instructions.

Line 375 prints the address contained in the pointer '.ADDRESS'. Line 377 prints the contents of the BYTE pointed to by the contents of '.ADDRESS'.

Line 380 assigns the value returned by GET_HEX_BYTE to a local variable called WRITE_BYTE. Line 381 tests the global error flag ERFLAG if it indicates an error further tests are carried out on line 382. A global variable called KEYCHAR will contain the last key hit before GET_HEX_BYTE terminated. If the key was a (-) or a (+) or a (/) lines 383 through 389 will be entered. If it was none of these keys the procedure is terminated by the statement in line 390.

Lines 385 through 387 seek to identify which key has been hit and adjust the value of '.ADDRESS' as required. If the key was a (+) or a (-) line 388 will be executed and control passed to the label TRY_AGAIN. If the key was a (/) then control will be passed to the label READ_AGAIN. The only difference between these two labels is the CRLF between them but it is most important in terms of the display presentation.

When either of the local labels is re-entered the value of '.ADDRESS' is again displayed along with the memory contents of the address. If the (+) key was hit the next address will be displayed on the next line. If the (-) key was hit the previous address will be displayed on the next line. If the (/) key was hit the same address will be displayed on the same line. In the latter case the address will be the same but the data may not be.

If line 381 did not find error flag true then control will be passed to line 392. Here we take the HEX byte supplied by the operator, which will be in WRITE_BYTE, and insert it into the memory location pointed to by the contents of '.ADDRESS'.

Line 393 then reads the memory location just written to and compares the value that was written with the value just read. If they are not the same lines 394 through 397 will be executed. If they are the same line 397 will increment the address pointer and return control to the local label indicated.

If the read and write data differ line 395 will send a 'beep' and a '?' to the system console adjacent to the data previously entered by the operator and then pass control back to the local label indicated which will re-display the data on the following line.

## 9.12.02 A PL/9 MINI-MONITOR PROGRAM FOR THE MC6809 (continued)

### JUMP TO USER PROGRAM

Lines 418 through 474 contain one of the more complex routines in the MINI-MONITOR. Lines 409 through 418 are a read-only data tables. Wait a minute! Haven't we been telling you to declare read-only data near the start of the program? Then why are we breaking our own 'rules'?

We have been telling you to put data that is likely to change near the start of the program. We have also recommended that message strings be declared as read-only data near the beginning of the program. What we have not told you is that the closer read only data is to the procedure that uses it the shorter will be the addressing range. The shorter the addressing ranges require less code.

In most programs this will only save 10 or 20 bytes or so. In this program those 10 extra bytes would have pushed us beyond 2K. What you see here is an effort to reduce code at the expense of maintainability. Normally we would have declared virtually every single message string used in a program near the start of it as we did in the example in section 9.07.12.

Shortening the addressing range will substantially reduce the code generated if the data is being accessed as an element of a vector table as it will be in this program.

Line 425 contains a label for a GOTO statement. Line 426 uses the address prompting routine at lines 355-357 and assigns the value returned to a local variable called IN_ADDRESS. Line 427 tests the error flag and if found to be TRUE the procedure is terminated immediately via line 428.

Line 429 assigns the value of IN_ADDRESS to the variable PROGRAM_CTR which, if you recall the section on the RTS routine is a variable on our pseudo MC6809 stack.

Line 431 prints the read-only data string on line 418. Line 433 performs a dump of the values stacked up above the variable 'JUMP_REGISTERS' (note we are passing the address of JUMP_REGISTERS as signified by the '.' preceeding it). If you recall what we said earlier JUMP_REGISTERS is the same memory location as CONDITION_CR. Thus we are passing the base address of our pseudo MC6809 stack to the dump register routine which will display all of the values.

The MAIN routine initializes all of these values to zero except the CCR which is initialized to $D0, a standard CCR condition to enter subroutines with. Once all of the pseudo stack variables are displayed line 436 prompts the operator whether the values are O.K. The operators response is assigned to a local variable called IN_BYTE in line 438. Line 439 then compares the response with an ASCII 'N' if IN_BYTE is an 'N' then lines 440 through 468 will be executed. Otherwise control will pass to line 469. If the operators response was not a 'Y' then control will pass to the local label indicated and the prompt reissued. If the response was a 'Y' line 472 will assign the address of the base of our pseudo stack to the real hardware stack pointer. Line 473 then pulls the pseudo stack into the appropriate registers, including the PC. The PC will now contain the address that was originally supplied by the operator hence program execution will begin at this point with the MC6809 registers pre-loaded with the values contained in the pseudo stack. Simple once you know what's going on!

## 9.12.02 A PL/9 MINI-MONITOR PROGRAM FOR THE MC6809 (continued)

Back to line 440 in the event the operator said that the pre-load values were not acceptable. Line 441 initializes a local variable that will be used as a loop counter. Line 443 sends a prompt to the operator giving him further instructions. The main prompting routine is contained within the REPEAT...UNTIL loop between lines 445 and 461. The purpose of this loop is to prompt the operator for a change of pre-load data for each of the MC6809 registers. On the first iteration line 446 and 447 will prompt 'CC?', on the second ' A?', on the third ' B?', etc. On the first four iterations (COUNT = 0, 1, 2 & 3) BYTE quantities (CCR, A, B & DP) will be dealt with so lines 449 through 453 will be executed. For all subsequent iterations (COUNT = 4, 5 & 6) INTEGER quantities (X, Y & U) will be dealt with so lines 454 through 458 will be entered.

Lets take the first iteration. Line 450 will get a HEX byte from the operator and assign it to a local variable called IN_BYTE. Line 451 tests for a non HEX character being entered, and, if it finds one, control will be passed to line 459. If a valid HEX byte was entered it will be assigned to our pseudo stack by treating it as a vector table in line 452 with JUMP_REGISTERS being used as the base address for all offsets.

Now lets take the last iteration. Line 455 will get a HEX address from the operator and assign it to a local variable called IN_ADDRESS. Line 456 test for a non HEX character being entered, and, if it finds one, control will be passed to line 459. If a valid HEX address was entered it will be assigned to our pseudo stack, again by treating it as a vector table, in line 457. This time X_REGISTER is used as the base address in the calculations and four is subtracted from the current value of COUNT to compensate for the new base address. The reason we shifted from using JUMP_REGISTERS as our base address is that JUMP_REGISTERS is defined as BYTE sized data. Therefore any references to a vector offset from JUMP_REGISTERS will be treated as BYTE sized by the compiler and the code it generates. Since X_REGISTER is defined as an INTEGER sized variable the compiler will treat vector indices as INTEGER sized variables.

Thus the operator is presented with a prompt to change each of the pre-load values one by one. When he has entered the last value the REPEAT...UNTIL loop will terminate and control will pass to line 463.

Once again the pre-load table will be presented to the operator. He will then be prompted 'JUMP?' any response other than 'Y' will terminate the procedure via line 467. If the response is 'Y' control will pass to line 472 and the jump executed as described previously.

## 9.12.02 A PL/9 MINI-MONITOR PROGRAM FOR THE MC6809 (continued)

### FORMATTED HEX/ASCII DUMP

This routine occupies lines 483 through 510. Line 484 prompts for a starting address and assigns the returned value to a BYTE sized pointer called '.ADDRESS'. Line 487 initializes the variable called PASSES to zero. Lines 488 through 506 are a REPEAT ... UNTIL loop with two REPEAT ... UNTIL loops inside it. Line 489 initializes the variable called COUNT to zero. Line 491 prints the current contents of the pointer called '.ADDRESS'. At this juncture it will be the same address as was supplied by the operator.

When we enter the REPEAT ... UNTIL loop between lines 493 and 497. Line 494 outputs the HEX byte pointed to by the address held in '.ADDRESS' subscripted by COUNT. On the first iteration this line will output the BYTE at the address supplied by the operator, on the second iteration the next higher BYTE will be printed, and so on. This REPEAT ... UNTIL loop will output 16 BYTES in hexadecimal form. Control will then pass to another REPEAT ... UNTIL loop between lines 500 and 503.

This loop is very similar to the last one except that the procedure 'PUT_ASCII_BYTE' is used instead of 'PUT_HEX_BYTE'. The former will output the ASCII code of any BYTE passed to it provided it falls in the range of $20 (space) to $7D (}). Any BYTE outside of this range will be printed as a period (.). This prevents the control codes from $00 to $1F and the meta codes from $80 through $FF from being transmitted to the terminal. The codes $7E (tilde '~') and $7F (delete) are not transmitted as many terminals exhibit strange behaviour whenever they see one of these codes. The same 16 BYTEs previously output in hexadecimal form will now be output with their ASCII equivalents if they are displayable or a period if they are not.

Line 504 bumps the memory pointer '.ADDRESS' by 16 to adjust it for the 16 bytes just printed. The PASSES counter is bumped by one in line 505. Line 506 forces control to return to line 488 until 16 passes have been completed. When 16 passes have been completed (i.e. 256 BYTES have been dumped) control will pass to line 507 which issues the prompt: 'MORE?'. Hitting any key other than 'N' will result in control passing to line 486 which will result in another 256 BYTEs being dumped. If you hit 'N' the routine will terminate.

### THE MAIN PROCEDURE

Lines 519 through 552 enclose the body of the main procedure that will be entered via a long branch just after the 'DPAGE' assignment. The first step is to save the global pointer. This is accomplished in line 521. Then the system variables are initialized in line 523 through line 524. Then the console ACIA is initialized in line 526. The pseudo stack is initialized between lines 428 and 535. The FIRQ and IRQ interrupts enabled in line 528.

The MAIN procedure is typically small (typical if you STRUCTURE your programs). The heart of the program is enclosed within the REPEAT...FOREVER loop between lines 541 and 552. What is going on should be self explanatory at this stage.

## 9.13.00 LARGE PROGRAMS

There comes a time in the development of larger programs when the message "MEMORY FULL" or "NOT ENOUGH ROOM FOR STACK" appears, or when the source file simply gets too large to handle conveniently.

PL/9 provides a simple mechanism by which the size of the file being handled can be kept to a minimum, this being the INCLUDE directive. If used sensibly, INCLUDE can free large amounts of memory and at the same time cut down on wasteful listings.

The best way to use include is to identify a procedure or a group of procedures that have been fully debugged. These procedures may then be written out to a disk file. The procedures are then deleted from the program and replaced with INCLUDE FILENAME at the appropriate point.

A problem arises, however, in that the extracted procedures may use the same global variables as the main program. Without a GLOBAL declaration being provided in the INCLUDE file it will not compile without "UNDEFINED SYMBOL" errors appearing. If you put in the necessary GLOBAL declaration you will get a "ONLY ONE GLOBAL DECLARATION" error when you try to INCLUDE the file in your main program.

You may of course reason that since the procedures have already been debugged there's no need to worry whether the file they are in compiles separately. This reasoning is fine until the inevitable happens, much later, when you make a fundamental change in the logic of the program and realize that you have to go back and change one of the library modules. The chance of making even a simple change to a complex program and have the result work is small. It is therefore highly desirable to have each library module separately compilable for testing.

The easy way out of this dilemma is to adopt the following strategy. Your program has a number of global variables and constants that are used by procedures throughout the program. Take the GLOBAL declaration and the CONSTANT and the AT statements and consign them to a file, called say VARIABLE.LIB. Now every module should have the line "INCLUDE VARIABLE;" near its start, together with INCLUDEs for any other modules that are needed for compilation. The main program can also have INCLUDEs for each of the modules thus created. PL/9 always ignores an INCLUDE within an INCLUDEd file, so only one set of the declarations gets seen by the compiler.

## 9.14.00 PROGRAMMING HINTS

This just about completes the technical side of the PL/9 Users Guide. If you have tried all, or at least most, of the examples in this section you should have a reasonable understanding of how structured control programs are written in PL/9.

The Reference Manuals for the Editor, the Compiler, the Tracer and the Language contain more information on the points outlined in this guide and on other points that have been left out.

This section is concerned with hints on programming style and some of the 'freedom of expression' permitted by the syntax of PL/9.

As we have mentioned previously PL/9 is similar in many ways to Pascal but allows you to get considerably closer to your hardware than Pascal will generally permit. The similarities are sufficient for any book on Pascal to contain many useful hints on programming style.

## 9.14.01 SPACING

Readers who may until now have only used BASIC or assembly language will probably already noticed the author's attempt in this manual to give a clean and consistant structure to each of the examples by indenting each line according to what group of lines it belongs with, while blank lines have been used to separate blocks of code that have specific functions.

Inserting 'plenty of white stuff' in the form of extra spaces and blank lines does not cause any extra code to be generated nor does it increase the size of the text file by any significant amount. What it does do is very significantly improve the READABILITY of the program. This is a very, very important consideration if the software must have errors corrected or additions at a later date, or is to be read by someone other than the author, OR is to be read by the author himself several months after the program was written.

## 9.14.02 INDENTING

The method you choose to highlight program flow and nesting is very much a matter of personal preference. You SHOULD adopt some form of indenting your source lines to indicate how the program is nesting the control arguments. For example both of the examples below generate EXACTLY the same code. Which one provides the clearest indication of what is going on AND what conditions are required to print the message: 'HELLO EVERYBODY'.

```
IF A=B .AND C=D THEN BEGIN PRINT("HELLO");IF B=D THEN PRINT(" THERE"); ELSE
PRINT(" EVERYBODY");END;ELSE PRINT("GOODBYE");
```

```
IF A=B .AND C=D
   THEN BEGIN
        PRINT("HELLO");
        IF B=D
           THEN PRINT(" THERE");
           ELSE PRINT(" EVERYBODY");
        END;

   ELSE PRINT("GOODBYE");
```

9.14.02 INDENTING (continued)

The precise style you adopt and the way you break up constructions is very much a matter of personal choice. Most people, however, find that indenting by multiples of three spaces is a reasonable compromise. The SETPL9 program (see section three) allows you define a key, usually TAB, that, when pressed, will cause three spaces to be generated.

Please note that there is no overhead in using multiple spaces and blank lines as far as program storage is concerned; both PL/9 and FLEX internally represent multiple spaces as single characters (a technique known as space compression is used). A badly laid out program, on the other hand, can be very difficult to maintain, even by the author.

9.14.03 SYNTAX

There is a certain amount of freedom of expression permitted in PL/9 when constructing expressions that pass variables to function procedures. This has been permitted primarily to help improve the readability of programs.

PASSING STRING POINTERS

The following forms are permitted:

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

or

```
PRINT(.POINTER);
PRINT .POINTER;
PRINT=.POINTER;
```

PASSING VARIABLES

The following forms are permitted:

```
PUTCHAR(CHAR);
PUTCHAR CHAR;
PUTCHAR=CHAR;
```

or

```
PRINTINT(NUMBER,BASE);
PRINTINT NUMBER,BASE;
PRINTINT=NUMBER,BASE;
```

You are also permitted to place extra spaces between a variable and its subscript, e.g. VARIABLE(COUNT) and VARIABLE (COUNT) are both acceptable.

And last, but by no means least, you can write your program in upper or lower case or a combination of upper and lower case.

### 9.14.04 SAVING CODE

There are several techniques you can use to reduce the amount of code PL/9 generates without recourse to patching your program with GEN statements. These techniques, when used properly, can reduce the generated code by about 5%.

This may not sound like much but when you are going to try to get a program into 2K or 4K of ROM for a dedicated application this code saving can make all the difference in the world.

### POSITION OF PROCEDURES

A procedure used as a subroutine by another procedure should be located as close to it as possible in order to reduce the addressing range.

### POSITION OF READ-ONLY VARIABLES

The same rule applies here. ANY text string used more than once should be declared as a read-only byte as close to the procedure that uses it as is practical and pointers to the message passed to the appropriate procedure(s).

### POSITION OF VARIABLES

Frequently used global or local variables should be declared first in order to reduce the addressing range. Generally speaking accessing a local variable generates less code than accessing a global variable. This can be an asset if the local variable is used several times in a procedure. If the variable is only used once the extra code required to offset and recover the stack pointer can cancel any benefits that might be realized by using local variables in lieu of global variables.

### TOGGLES AND FLAGS

Whenever possible the states of variables used as toggles and flags should have one of the states equal to zero. Ideally the two states should complement each other. i.e. a BYTE flag would be $00 or $FF (0 or -1) NOT 0 or 1. If the state of the variable is managed in this manner the 'NOT' function can be used to toggle the variable from one state to another. e.g. VAR=NOT(VAR). With one of the logical states being zero you can then use the next construction...

### IMPLICIT NOT EQUAL TO ZERO

This construction can save a VERY considerable amount of code. If you construct IF...THEN, REPEAT...UNTIL, and WHILE... arguments to look for a <> zero condition rather than some specific value you may omit the '<> 0' statement. PL/9 interprets this omission as an implicit not equal to zero statement and generates far less code. e.g. IF VARIABLE <> 0 THEN... will generate much more code than IF VARIABLE THEN...

Until you get used to this construction it might be wise to comment any lines using it with /* IMPLICIT <> 0 */ as a reminder of what you are doing.

## 9.14.05 WHEN THINGS DON'T GO ACCORDING TO PLAN

There will come a time, yes it happens to the best of us, when the program we have written refuses to work the way we have designed it to but compiles without any errors being reported.

In these circumstances there are several options available to you to assist in debugging the construction of your program.

The PL/9 tracer should be your first recourse as it has many facilities to help locate the area in the program where things are going wrong.

You can also include the IOSUBS, BITIO and/or HEXIO libraries to assist in printing messages/data out to the system console. These same routines may also be used to get sample test data for your program from the system console.

The routines that will be of most use in this respect would be PRINT to facilitate printing messages when a particular branch is taken, BITSIN and BITSOUT for bit oriented I/O, and the entire HEXIO library if you wish to input/output data in HEX numbers.

The four most common problems that people have experienced with PL/9 programs are:

(1) Failing to assign a large enough area for a buffer This can produce VERY strange effects if the buffer grows into other variables. The problem can be absolutely catastrophic if the buffer is declared as a local variable in a subroutine and the data written to the buffer extends into the return address on the stack!

(2) Failing to size a vector table properly or accessing an element of the vector table that does not exist. For example if you declared the following: 'GLOBAL BYTE VECTOR(3),COUNT;' and then made a statement like VECTOR(3)=99 this would overwrite the data in the variable COUNT. You must remember that PL/9 subscripts the elements of a vector starting from zero so the highest number a subscript may have is ONE LESS than the declared number of elements. In this instance the cause of the problem should be fairly obvious. This type of error can be difficult to trace down when the vector table is being accessed by a subscript, e.g. VECTOR(COUNT), and for one reason or another the subscript, in this case COUNT, is allowed to be greater than the range of the vector.

(3) Failing to understand the signed nature of PL/9 arithmetic and evaluations. This area generally only causes people problems when they are working with I/O devices, PIA's for example, and wish to evaluate the port as a 'bit pattern' rather than as a signed number. The problems only start to occur when a BYTE is 'greater' than $7F or an INTEGER is 'greater' than $7FFF. To PL/9 these numbers are NEGATIVE. A special mechanism (the exclamation mark) and two functions (BYTE and INTEGER) have been provided to simplify working with unsigned BYTEs and INTEGERs, they do, however, take a bit of getting used to, particulary if you are an assembly language programmer.

(4) Incorrectly sized pointers can also cause some very strange problems. If you define a pointer as 'REAL .RPTR' and then use it on INTEGER or BYTE sized data the problems caused will be very obscure and equally difficult to locate! The PL/9 tracer is good at locating faults of this nature only if they do not cause crashes.

THIS PAGE INTENTIONALLY LEFT BLANK

## 10.00.00 SAMPLE PROGRAMS IN PL/9

This section includes several working programs to assist you in developing other programs. We have not supplied these programs on disk in order to provide you with some exercises in entering and compiling programs. We can supply all of these programs on disk for $20.00 including AIR MAIL postage. To order it write to us and ask for the 'PL9 MANUAL PROGRAM PACK' and enclose an INTERNATIONAL MONEY ORDER for this amount. We will supply the software on 5" disk unless you specifically request 8".

## 10.00.01 SIEVE OF ERATOSTHENES

This familiar benchmark coded in PL/9.

## 10.00.02 DAMPED SINE WAVE DEMONSTRATION

This program plots a damped sine wave on your terminal using its X-Y cursor addressing facilities (defined in TERMSUBS.LIB) and demonstrates the speed of the floating point arithmetic and scientific functions in PL/9.

## 10.00.03 LUNAR LANDER

This program is a PL/9 adaptation of a BASIC program written by John Holdsworth. This program also requires a terminal with X-Y cursor addressing facilites and that the TERMSUBS library be configured for it. It is a real-time simulation of an orbiting body attempting to deaccelerate and land in a precise position. As you will discover when you run this program it's not easy! Its amazing that the APOLLO lunar missions succeeded when you consider all of the physical forces involved in landing.

## 10.00.04 UPPER-CASE TO LOWER-CASE CONVERSION PROGRAM

This program converts a PL/9 text file entered in UPPER-CASE letters to a text file with lower case letters. All text within comments '/*...*/' and within double quotes (strings) will not be altered. The program runs in the FLEX TCA ($C100 - $C6FF). A '.PL9' file extension is assumed.

Compile the source to a file called 'LCASE.CMD' and call it from FLEX:

+++LCASE,1.INFILE.PL9<CR>

## 10.00.05 LOWER-CASE TO UPPER-CASE CONVERSION PROGRAM

This program does the exact opposite of the above program. It converts a program written in lower-case letters to a program in UPPER-CASE letters. The FLEX command line syntax is identical.

### 10.00.06 SORTED DISK DIRECTORY

This program produces a sorted directory listing of a FLEX disk containing up to 300 files. The program is optimized for an 80 column VDU. This program was originally published as a Pascal program in '68 Micro Journal. This program runs just under MEMEND ($BFFF).

### 10.00.07 MOVE UTILITY

This program runs in the FLEX TCA and can be used to move object code about in the absolute memory map. It provides a working example of how to work with pointers and unsigned numbers.

### 10.00.08 INTEL HEX FORMAT DUMP

This program runs in the FLEX TCA and will dump a FLEX binary file through the resident printer driver in INTEL HEX format. This program is useful when you need to download a FLEX binary file to another system, EPROM programmer or emulator.

### 10.00.09 MOTOROLA HEX FORMAT DUMP

This program is identical to the one above except that it dumps the file in the MOTOROLA HEX format.

## 10.01.01 SIEVE OF ERATOSTHENES

Page 1: ERASTOSTHENES PRIME NUMBER PROGRAM IN PL/9 June 1 1984

```
0000  0001 /* ERASTOSTHENES PRIME NUMBER PROGRAM IN PL/9 */
0000  0002
0000  0003
0000  0004 origin=$8000; stack=*;
8003  0005
8003  0006 /*  FIRST SOME I/O ROUTINES  */
8003  0007
8003  0008 constant cr=13,lf=10,sp=32,false=0,true=-1;
8003  0009
8003  0010 procedure putchar(byte char);
8006  0011    acca=char;
800A  0012    call $cd18;
800D  0013 endproc;
800E  0014
800E  0015 procedure crlf;
800E  0016    putchar(cr); putchar(lf);
801E  0017 endproc;
801F  0018
801F  0019 procedure print(byte .string):
801F  0020    byte char: integer pos;
8021  0021    pos=0;
8026  0022    while string(pos)  /* IMPLICIT <> 0 */
802E  0023       begin
8033  0024          char=string(pos); pos=pos+1;
8044  0025          if char='\ then
804C  0026             begin
804C  0027                char=string(pos); pos=pos+1;
805D  0028                if char
805D  0029                case 'N then crlf;
8068  0030                case 'E then putchar(27);
8079  0031                case '0 then putchar(0);
808B  0032                else putchar(char);
8097  0033             end;
8097  0034          else putchar(char);
80A3  0035       end;
80A3  0036 endproc;
80A8  0037
80A8  0038 integer bbcd 10000,1000,100,10,1;
80B2  0039
80B2  0040 procedure prnum(integer n): byte i,flag,buffer(6);
80B4  0041    i=0;
80B6  0042    repeat
80B6  0043       buffer(i)=n/bbcd(i);
8130  0044       n=n-n/bbcd(i)*bbcd(i);
8181  0045       i=i+1;
8183  0046    until i=5;
818B  0047    buffer(5)=0;
818D  0048    i=0;
818F  0049    flag=false;
```

10.01.01 SIEVE OF ERATOSTHENES (continued)

Page 2: ERASTOSTHENES PRIME NUMBER PROGRAM IN PL/9 June 1 1984

```
8191 0050    repeat
8191 0051       if buffer(i) .or flag then
81B2 0052       begin
81B2 0053          buffer(i)=buffer(i)+'0;
81BF 0054          flag=true;
81C3 0055       end;
81C3 0056       else buffer(i)=sp;
81D1 0057       i=i+1;
81D3 0058    until i=5;
81D9 0059    print(.buffer);
81E2 0060 endproc;
81E5 0061
81E5 0062 /*  NOW THE SIEVE PROGRAM ITSELF  */
81E5 0063
81E5 0064 constant size=8190;
81E5 0065
81E5 0066 procedure sieve:
81E5 0067    integer i,prime,k,count:
81E5 0068    byte iter,flags(8191);
81E9 0069    print("\N10 ITERATIONS\N");
8208 0070    iter=0;
820A 0071       repeat
820A 0072          count=0;
820F 0073          i=0;
8214 0074          repeat
8214 0075             flags(i)=true;
821E 0076             i=i+1;
8225 0077          until i>size;
822C 0078          i=0;
8231 0079          repeat
8231 0080             if flags(i) then
823E 0081             begin
823E 0082                prime=i+i+3;
8247 0083                k=i+prime;
824D 0084                while k<=size
824F 0085                begin
8256 0086                   flags(k)=false;
825E 0087                   k=k+prime;
8264 0088                end;
8264 0089                count=count+1;
826D 0090             end;
826D 0091             i=i+1;
8274 0092          until i>size;
827B 0093          iter=iter+1;
827D 0094       until iter=10;
8283 0095    prnum(count);
828C 0096    print(" PRIMES\N");
```

10.01.01 SIEVE OF ERATOSTHENES (continued)

Page 3: ERASTOSTHENES PRIME NUMBER PROGRAM IN PL/9 June 1 1984

```
PROCEDURES:
putchar             8006
crlf                800E
print               801F
prnum               80B2
sieve               81E5

DATA:
bbcd                80A8    INTEGER

EXTERNALS:
cr                  000D
lf                  000A
sp                  0020
false               0000
true                FFFF
size                1FFE

GLOBALS:
```

## 10.01.02 DAMPED SINE WAVE DEMONSTRATION

You must configure the 'CURSOR', 'HOME' and 'ERASE_EOP' commands in the TERMSUBS library before this program will operate properly.

```
Page  1:  DAMPED SINE WAVE DEMONSTRATION PROGRAM                June 1 1984

 0000  0001 /* DAMPED SINE WAVE DEMONSTRATION PROGRAM
 0000  0002
 0000  0003 origin=$9000; stack=*;
 9003  0004
 9003  0005 include 0.iosubs.lib;
 92EB  0006 include 0.termsubs.lib;
 9395  0007 include 0.scipack.lib;
 9CE0  0008 include 0.realcon.lib;
 A2A1  0009
 A2A1  0010 byte axes
 A2A1  0011 " 1.0 |\N
 A2A5  0012       |                                        1.6*SIN(2X)\N
 A2E1  0013       |                              Plot of   ----------\N
 A31D  0014       |                                        (X+1)\N
 A355  0015       |\N
 A35D  0016  0.5  |\N
 A365  0017       |\N
 A36D  0018       |\N
 A375  0019       |\N
 A37D  0020       |\N
 A385  0021  0.0  |-------------------------------------------------------
 A3C5  0022 ----------\N
 A3D1  0023       0         1         2         3         4         5         6
 A415  0024           7\N
 A421  0025       |\N
 A429  0026       |\N
 A431  0027       |\N
 A439  0028 -0.5  |\N
 A441  0029       |\N
 A449  0030       |\N
 A451  0031       |\N
 A459  0032       |\N
 A461  0033 -1.0  |\N
 A469  0034     Points Plotted\N";
 A47F  0035
 A47F  0036 procedure demo: integer count: real i: byte buffer(20);
 A482  0037    home; erase_eop;
 A488  0038    print(.axes);
 A493  0039    i=0;
 A4A0  0040    count=0;
 A4A5  0041    repeat
 A4A5  0042       cursor(fix(i*10)+5,10-fix(sin(i*2)*16/(i+1)));
 A507  0043       putchar('*);
 A510  0044       count=count+1;
 A517  0045       cursor(0,21);
 A524  0046       print(ascii(count,.buffer));
 A539  0047       i=i+0.035;
 A54E  0048    until i>7;
```

10.01.02 DAMPED SINE WAVE DEMONSTRATION (continued)

Page 2: DAMPED SINE WAVE DEMONSTRATION PROGRAM June 1 1984

PROCEDURES:

| | | |
|---|---|---|
| monitor | 9006 | |
| warms | 900B | |
| getchar | 900F | BYTE |
| getchar_noecho | 9015 | BYTE |
| getkey | 901C | BYTE |
| convert_lc | 9034 | BYTE |
| get_uc | 9059 | BYTE |
| get_uc_noecho | 9066 | BYTE |
| putchar | 9073 | |
| printint | 907B | |
| remove_char | 9120 | |
| input | 913C | INTEGER |
| crlf | 91E0 | |
| print | 91F3 | |
| space | 92CF | |
| nulls | 92EB | |
| erase_eol | 9308 | |
| erase_eop | 931D | |
| cursor | 9332 | |
| home | 935D | |
| home_n_clr | 936A | |
| attr_on | 936F | |
| attr_off | 9382 | |
| _poly | 93A1 | REAL |
| ln | 9826 | REAL |
| log | 988B | REAL |
| exp | 98C4 | REAL |
| alog | 999D | REAL |
| xtoy | 99B7 | REAL |
| sin | 9A2B | REAL |
| cos | 9B06 | REAL |
| tan | 9BDC | REAL |
| atn | 9C1E | REAL |
| binary | 9CE0 | REAL |
| ascbin | 9EBD | REAL |
| ascii | 9ED5 | INTEGER |
| demo | A47F | |

DATA:

| | | |
|---|---|---|
| _pio2 | 9395 | REAL |
| _e | 9399 | REAL |
| _log2 | 939D | REAL |
| log_coeff | 9802 | REAL |
| exp_coeff | 98A4 | REAL |
| sin_coeff | 9A13 | REAL |
| cos_coeff | 9AEE | REAL |
| atn_coeff | 9BFA | REAL |
| axes | A2A1 | BYTE |

10.01.02 DAMPED SINE WAVE DEMONSTRATION (continued)

Page 3: DAMPED SINE WAVE DEMONSTRATION PROGRAM June 1 1984

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

GLOBALS:
```

## 10.01.03 LUNAR LANDER

You must configure the 'CURSOR', 'HOME' and 'ERASE_EOP' commands in the TERMSUBS library before this program will operate properly.

```
Page  1:  LUNAR LANDER                                          June 1 1984

0000  0001 /* LUNAR LANDER */
0000  0002
0000  0003 origin=$1000; stack=*;
1003  0004
1003  0005 global
1003  0006    real mass, fuel_rate, fuel_no, height, angle,
1003  0007         distance, time,
1003  0008         v_acc, h_acc, v_vel, h_vel;
100A  0009
100A  0010 include 0.iosubs.lib;
12F2  0011 include 0.termsubs.lib;
139C  0012 include 0.scipack.lib;
1CE7  0013 include 0.realcon.lib;
22A8  0014
22A8  0015 procedure abs(real n);
22A8  0016    if n<0 then n=-n;
22C9  0017 endproc n;
22D2  0018
22D2  0019 procedure putf(real n): byte buffer(20);
22D5  0020    print ascii(int(n),.buffer);
22EE  0021 endproc;
22F2  0022
22F2  0023 procedure fprint(byte x,y: real n);
22F2  0024    cursor x,y; print "          ";
2317  0025    if n>=0 then x=x+1;
232D  0026    cursor x,y;
233A  0027    putf n;
2343  0028 endproc;
2344  0029
2344  0030 byte text
2344  0031 "                                  Lunar Lander\n
236C  0032                                  ============\n
2398  0033 \n
2398  0034           You are the pilot of a small lander module orbiting\n
23D8  0035           the  moon.  Your  task is to land on or as close as\n
2418  0036           possible to a predetermined  site  by  the  use  of\n
2458  0037           instruments alone.\n
2474  0038 \n
2478  0039           You have a display showing your altitude and speed,\n
24B4  0040           the distance to your target, the amount of fuel you\n
24F4  0041           have  left,  etc. The only controls you can use are\n
2534  0042           those that control the attitude jets and  the  fuel\n
2574  0043           rate valve.\n
2588  0044 \n
258C  0045           You will earn a score that depends on how close you\n
25CC  0046           land to the target and how much fuel you have left.\n
2608  0047           Your  touch-down  speed must be less than 5 ft/s in\n
2648  0048           both vertical and horizontal directions.\n
267C  0049 \n
2680  0050           Press any key to start...\n
26A4  0051 \n
26A4  0052                                              Good luck!\n
26D8  0053 ";
26DD  0054
```

10.01.03 LUNAR LANDER (continued)



```
26DD  0055 procedure explain;
26DD  0056    home; erase_eop;
26E3  0057    print .text;
26EE  0058    getchar_noecho;
26F1  0059 endproc;
26F2  0060
26F2  0061 procedure border: byte row;
26F4  0062    home; erase_eop;
26FA  0063    cursor 11,0;
2707  0064    print "************ Lander Status Report ************";
2743  0065    row=0;
2745  0066    repeat
2745  0067       cursor 11,row; putchar '*;
275B  0068       cursor 57,row; putchar '*;
2771  0069       row=row+1;
2773  0070    until row=19;
2779  0071    cursor 11,18;
2786  0072    print "************************************************";
27C2  0073 endproc;
27C5  0074
27C5  0075 procedure setup;
27C5  0076    h_vel=5000;
27D3  0077    height=50000;
27E0  0078    mass=20000;
27ED  0079    distance=1e6;
27FB  0080    fuel_rate=0;
2808  0081    fuel_no=0;
2815  0082    time=0;
2823  0083    angle=0;
2831  0084    v_acc=0;
283F  0085    h_acc=0;
284D  0086    v_vel=0;
285B  0087
285B  0088    border;
285E  0089    cursor 15,2;
286B  0090    print "Height                    Feet";
289B  0091    cursor 15,4;
28A8  0092    print "Distance to go            Feet";
28D8  0093    cursor 15,6;
28E5  0094    print "Horizontal speed          Feet/sec";
2919  0095    cursor 15,8;
2926  0096    print "Vertical speed            Feet/sec";
295A  0097    cursor 15,10;
2967  0098    print "Fuel left                 Pounds";
2999  0099    cursor 15,12;
29A6  0100    print "Fuel rate                 Pounds/sec";
29DC  0101    cursor 15,14;
29E9  0102    print "Lander attitude           Degrees";
2A1C  0103    fprint 35,14,angle;
2A2F  0104    cursor 15,16;
2A3C  0105    print "Elapsed time              Seconds";
2A6F  0106    cursor 20,20;
2A7C  0107    print " A  = Rotate Anticlockwise";
2AA4  0108    cursor 20,21;
2AB1  0109    print " C  = Rotate Clockwise";
2AD5  0110    cursor 20,22;
```

10.01.03 LUNAR LANDER (continued)

Page 3: LUNAR LANDER June 1 1984

```
2AE2  0111     print "0-9 = Proportional Fuel Rate";
2B0C  0112 endproc;
2B0D  0113
2B0D  0114 procedure update_display;
2B0D  0115     fprint 35,2,height;
2B1F  0116     fprint 35,4,distance;
2B32  0117     fprint 35,6,h_vel;
2B45  0118     fprint 35,8,v_vel;
2B58  0119     fprint 35,10,mass-5000;
2B75  0120     fprint 35,12,fuel_rate;
2B87  0121     fprint 35,16,time;
2B9A  0122     cursor 0,0;
2BA7  0123 endproc;
2BA8  0124
2BA8  0125 procedure score(real n);
2BA8  0126     cursor 15,21;
2BB5  0127     print "\bYour score is ";
2BD3  0128     putf n; putchar '.;
2BE6  0129 endproc;
2BE7  0130
2BE7  0131 procedure control:
2BE7  0132     byte key: integer count;
2BE9  0133     count=12000;
2BEE  0134     repeat
2BEE  0135        key=getkey;
2BF3  0136        if key then
2BFA  0137        begin
2BFA  0138           if key='A .or key='a then
2C16  0139           begin
2C16  0140              angle=angle+10;
2C2D  0141              if angle>180 then angle=angle-360;
2C59  0142              fprint 35,14,angle;
2C6C  0143           end;
2C6C  0144           else if key='C .or key='c then
2C8B  0145           begin
2C8B  0146              angle=angle-10;
2CA2  0147              if angle<-180 then angle=360+angle;
2CD1  0148              fprint 35,14,angle;
2CE4  0149           end;
2CE4  0150           else if key>='0 .and key<='9 then
2D03  0151           begin
2D03  0152              fuel_no=key-'0;
2D17  0153              fuel_rate=fuel_no*(mass-3500)/1500;
2D3F  0154              fprint 35,12,fuel_rate;
2D51  0155           end;
2D51  0156        end;
2D51  0157        count=count-1;
2D58  0158     until count=0;
2D61  0159 endproc;
2D64  0160
2D64  0161 procedure approach;
2D64  0162     setup;
2D67  0163     repeat
2D67  0164        update_display;
2D6A  0165        control;
```

10.01.03 LUNAR LANDER (continued)

Page 4: LUNAR LANDER June 1 1984

```
2D6D  0166        if mass<=5000 then
2D81  0167        begin
2D81  0168           fuel_rate=0;
2D8E  0169           fuel_no=0;
2D9B  0170        end;
2D9B  0171        if height=0 then return;
2DB0  0172        mass=mass-fuel_rate;
2DC2  0173        if mass<5000 then mass=5000;
2DE3  0174        v_acc=h_vel*h_vel/(4.1e6+height)
2E05  0175             +fuel_rate*6400/mass
2E15  0176             *cos(angle*3.14159265/180)-6;
2E57  0177        v_vel=v_vel+v_acc;
2E6C  0178        h_acc=-fuel_rate*6400/mass
2E7F  0179             *sin(angle*3.14159265/180);
2EB3  0180        h_vel=h_vel+h_acc;
2EC8  0181        height=height+v_vel;
2EDB  0182        if height<0 then height=0;
2EFC  0183        distance=distance-h_vel;
2F11  0184        if distance>1.6e7 then distance=3.2e7-distance;
2F3D  0185        if distance<-1.6e7 then distance=3.2e7+distance;
2F6C  0186        fuel_rate=fuel_no*(mass-3500)/1500;
2F94  0187        time=time+1;
2FAB  0188     forever;
2FAB  0189  endproc;
2FAF  0190
2FAF  0191  procedure lander: byte char;
2FB1  0192     explain;
2FB4  0193  LOOP:
2FB4  0194     approach;
2FB7  0195     cursor 0,20; erase_eop;
2FC7  0196     cursor 15,20;
2FD4  0197     if abs(h_vel)>9 .or abs(v_vel)>9 then
3004  0198     begin
3004  0199        print "You Crashed!";
301E  0200        score=0;
302B  0201     end;
302B  0202     else if abs(h_vel)>4 .or abs(v_vel)>4 then
305E  0203     begin
305E  0204        print "Bad landing - some damage.";
3086  0205        score=(mass-6000+(1e6-abs(distance))/1000)/10;
30CD  0206     end;
30CD  0207     else
30D0  0208     begin
30D0  0209        print "Congratulations on a safe landing!";
3100  0210        score=(mass-5000+(1e6-abs(distance))/1000)/10;
3147  0211     end;
3147  0212     cursor 60,22;
3154  0213     print "Try again? ";
316D  0214     repeat
316D  0215        char=getchar_noecho;
3172  0216        if char>='a .and char<='z then char=char-$20;
3194  0217     until char='Y .or char='N;
31AE  0218     if char='Y then goto loop;
```

10.01.03 LUNAR LANDER (continued)

Page 5: LUNAR LANDER June 1 1984

PROCEDURES:

| | | |
|---|---|---|
| monitor | 100D | |
| warms | 1012 | |
| getchar | 1016 | BYTE |
| getchar_noecho | 101C | BYTE |
| getkey | 1023 | BYTE |
| convert_lc | 103B | BYTE |
| get_uc | 1060 | BYTE |
| get_uc_noecho | 106D | BYTE |
| putchar | 107A | |
| printint | 1082 | |
| remove_char | 1127 | |
| input | 1143 | INTEGER |
| crlf | 11E7 | |
| print | 11FA | |
| space | 12D6 | |
| nulls | 12F2 | |
| erase_eol | 130F | |
| erase_eop | 1324 | |
| cursor | 1339 | |
| home | 1364 | |
| home_n_clr | 1371 | |
| attr_on | 1376 | |
| attr_off | 1389 | |
| _poly | 13A8 | REAL |
| ln | 182D | REAL |
| log | 1892 | REAL |
| exp | 18CB | REAL |
| alog | 19A4 | REAL |
| xtoy | 19BE | REAL |
| sin | 1A32 | REAL |
| cos | 1B0D | REAL |
| tan | 1BE3 | REAL |
| atn | 1C25 | REAL |
| binary | 1CE7 | REAL |
| ascbin | 1EC4 | REAL |
| ascii | 1EDC | INTEGER |
| abs | 22A8 | INTEGER |
| putf | 22D2 | |
| fprint | 22F2 | |
| explain | 26DD | |
| border | 26F2 | |
| setup | 27C5 | |
| update_display | 2B0D | |
| score | 2BA8 | |
| control | 2BE7 | |
| approach | 2D64 | |
| lander | 2FAF | |

10.01.03 LUNAR LANDER (continued)



```
DATA:
_pio2                   139C    REAL
_e                      13A0    REAL
_log2                   13A4    REAL
log_coeff               1809    REAL
exp_coeff               18AB    REAL
sin_coeff               1A1A    REAL
cos_coeff               1AF5    REAL
atn_coeff               1C01    REAL
text                    2344    BYTE

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

GLOBALS:
mass                    0000    REAL
fuel_rate               0004    REAL
fuel_no                 0008    REAL
height                  000C    REAL
angle                   0010    REAL
distance                0014    REAL
time                    0018    REAL
v_acc                   001C    REAL
h_acc                   0020    REAL
v_vel                   0024    REAL
h_vel                   0028    REAL
```

## 10.01.04 UCASE TO LCASE CONVERSION PROGRAM

Page 1: Lower to Upper Case Conversion June 1 1984

```
0000  0001 /* Lower to Upper Case Conversion */
0000  0002
0000  0003 at $c840: byte fcb, error(319);
0000  0004 at $cc14: integer line_pointer;
0000  0005
0000  0006 at 0:
0000  0007    integer pointer, last:
0000  0008    byte name(20), buffer(10000);
0000  0009
0000  0010 origin=$C100;
C100  0011
C100  0012 /* THE FOLLOWING HAVE BEEN EXTRACTED FROM IOSUBS.LIB */
C100  0013
C100  0014 constant nul = $00,
C100  0015          abt = $03,
C100  0016          bel = $07,
C100  0017           bs = $08,
C100  0018           lf = $0a,
C100  0019           cr = $0d,
C100  0020          can = $18,
C100  0021          esc = $1b,
C100  0022           sp = $20;
C100  0023
C100  0024
C100  0025 procedure putchar(byte char);
C103  0026    acca = char;
C107  0027    call $cd18; /* FLEX 'PUTCHR'  (HONOURS 'TTYSET' PARAMETERS) */
C10A  0028 endproc;
C10B  0029
C10B  0030
C10B  0031 include 0.flex.lib;
C199  0032
C199  0033 procedure abort;
C199  0034    report_error(.fcb);
C1A2  0035    flex;
C1A5  0036 endproc;
C1A6  0037
C1A6  0038 procedure output(byte char);
C1A6  0039    putchar(char);
C1AF  0040    if char=cr then putchar(lf);
C1C0  0041    write(.fcb,char);
C1CE  0042    if error then abort;
C1D8  0043 endproc;
C1D9  0044
C1D9  0045 procedure skip_comment;
C1D9  0046    output('/);
C1E1  0047    output('*);
C1E9  0048    pointer=pointer+2;
C1F0  0049    repeat
C1F0  0050       output(buffer(pointer));
C1FF  0051       pointer=pointer+1;
C206  0052    until buffer(pointer-2)='* .and buffer(pointer-1)='/;
C234  0053 endproc;
C235  0054
```

## 10.01.04 UCASE TO LCASE CONVERSION PROGRAM (continued)

Page 2: Lower to Upper Case Conversion June 1 1984

```
C235  0055 procedure skip_string;
C235  0056    output('"');
C23E  0057    pointer=pointer+1;
C245  0058    repeat
C245  0059       if buffer(pointer)='" .and buffer(pointer+1)='" then
C272  0060       begin
C272  0061          output('");
C27B  0062          pointer=pointer+1;
C282  0063       end;
C282  0064       output(buffer(pointer));
C292  0065       pointer=pointer+1;
C299  0066    until buffer(pointer)='" .and buffer(pointer+1)<>'";
C2C4  0067    output('");
C2CD  0068    pointer=pointer+1;
C2D4  0069 endproc;
C2D5  0070
C2D5  0071 procedure ulc: byte count, extension(3);
C2D7  0072    get_filename(.fcb);
C2E1  0073    if error then abort;
C2EC  0074    if fcb(12)=0 then
C2F5  0075    begin
C2F5  0076       fcb(12)='P;
C2FA  0077       fcb(13)='L;
C2FF  0078       fcb(14)='9;
C304  0079    end;
C304  0080    open_for_read(.fcb);
C30E  0081    if error then abort;
C319  0082
C319  0083    pointer=0;
C31E  0084    repeat
C31E  0085       buffer(pointer)=read(.fcb);
C333  0086       if error then
C33B  0087          if error=8 then break else abort;
C34D  0088       pointer=pointer+1;
C354  0089    forever;
C354  0090    last=pointer;
C35A  0091    close_file(.fcb);
C364  0092
C364  0093    extension(0)=fcb(12);
C369  0094    extension(1)=fcb(13);
C36E  0095    extension(2)=fcb(14);
C373  0096
C373  0097    fcb(12)='T;
C378  0098    fcb(13)='M;
C37D  0099    fcb(14)='P;
C382  0100
C382  0101    open_for_write(.fcb);
C38C  0102    if error then abort;
C397  0103
C397  0104    pointer=0;
C39C  0105    repeat
C39C  0106       if buffer(pointer)='/ .and buffer(pointer+1)='*
C3BB  0107          then skip_comment;
C3CC  0108       else if buffer(pointer)='" then skip_string;
C3E1  0109       else if buffer(pointer)='' then
```

10.01.04 UCASE TO LCASE CONVERSION PROGRAM (continued)

Page 3: Lower to Upper Case Conversion June 1 1984

```
C3F3  0110        begin
C3F3  0111           output('');
C3FC  0112           output(buffer(pointer+1));
C40F  0113           pointer=pointer+2;
C416  0114        end;
C416  0115        else
C419  0116        begin
C419  0117           if buffer(pointer)>='a .and buffer(pointer)<='z
C435  0118              then buffer(pointer)=buffer(pointer)-$20;
C450  0119           output(buffer(pointer));
C460  0120           pointer=pointer+1;
C467  0121        end;
C467  0122     until pointer>=last;
C46F  0123     close_file(.fcb);
C479  0124     if error then abort;
C484  0125
C484  0126     fcb(12)=extension(0);
C489  0127     fcb(13)=extension(1);
C48E  0128     fcb(14)=extension(2);
C493  0129
C493  0130     delete_file(.fcb);
C49D  0131     if error then abort;
C4A8  0132
C4A8  0133     fcb(12)='T;
C4AD  0134     fcb(13)='M;
C4B2  0135     fcb(14)='P;
C4B7  0136
C4B7  0137     count=0;
C4B9  0138     repeat
C4B9  0139        fcb(count+53)=fcb(count+4);
C4D3  0140        count=count+1;
C4D5  0141     until count=8;
C4DB  0142
C4DB  0143     fcb(61)=extension(0);
C4E0  0144     fcb(62)=extension(1);
C4E5  0145     fcb(63)=extension(2);
C4EA  0146
C4EA  0147     rename_file(.fcb);
C4F4  0148     if error then abort;
```

## 10.01.04 UCASE TO LCASE CONVERSION PROGRAM (continued)

Page 4: Lower to Upper Case Conversion June 1 1984

```
PROCEDURES:
putchar             C103
flex                C10B
get_filename        C10E
set_extension       C118
report_error        C120
open_for_read       C125
read                C12E   BYTE
open_for_write      C139
write               C142
read_sector         C149
write_sector        C15D
set_binary          C171
close_file          C179
delete_file         C182
rename_file         C190
abort               C199
output              C1A6
skip_comment        C1D9
skip_string         C235
ulc                 C2D5

DATA:

EXTERNALS:
fcb                 C840   BYTE
error               C841   BYTE
line_pointer        CC14   INTEGER
pointer             0000   INTEGER
last                0002   INTEGER
name                0004   BYTE
buffer              0018   BYTE
nul                 0000
abt                 0003
bel                 0007
bs                  0008
lf                  000A
cr                  000D
can                 0018
esc                 001B
sp                  0020

GLOBALS:
```

## 10.01.05 LCASE TO UCASE CONVERSION PROGRAM

Page 1: Upper to Lower Case Conversion June 1 1984

```
0000  0001 /* Upper to Lower Case Conversion */
0000  0002
0000  0003 at $c840: byte fcb, error(319);
0000  0004 at $cc14: integer line_pointer;
0000  0005
0000  0006 at 0:
0000  0007    integer pointer, last:
0000  0008    byte name(20), buffer(10000);
0000  0009
0000  0010 origin=$C100;
C100  0011
C100  0012 /* THE FOLLOWING HAVE BEEN EXTRACTED FROM IOSUBS.LIB */
C100  0013
C100  0014 constant nul = $00,
C100  0015          abt = $03,
C100  0016          bel = $07,
C100  0017           bs = $08,
C100  0018           lf = $0a,
C100  0019           cr = $0d,
C100  0020          can = $18,
C100  0021          esc = $1b,
C100  0022           sp = $20;
C100  0023
C100  0024
C100  0025 procedure putchar(byte char);
C103  0026    acca = char;
C107  0027    call $cd18; /* FLEX 'PUTCHR'  (HONOURS 'TTYSET' PARAMETERS) */
C10A  0028 endproc;
C10B  0029
C10B  0030 include 0.flex.lib;
C199  0031
C199  0032 procedure abort;
C199  0033    report_error(.fcb);
C1A2  0034    flex;
C1A5  0035 endproc;
C1A6  0036
C1A6  0037 procedure output(byte char);
C1A6  0038    putchar(char);
C1AF  0039    if char=cr then putchar(lf);
C1C0  0040    write(.fcb,char);
C1CE  0041    if error then abort;
C1D8  0042 endproc;
C1D9  0043
C1D9  0044 procedure skip_comment;
C1D9  0045    output('/);
C1E1  0046    output('*);
C1E9  0047    pointer=pointer+2;
C1F0  0048    repeat
C1F0  0049       output(buffer(pointer));
C1FF  0050       pointer=pointer+1;
C206  0051    until buffer(pointer-2)='* .and buffer(pointer-1)='/;
C234  0052 endproc;
C235  0053
C235  0054 procedure skip_string;
C235  0055    output('");
```

## 10.01.05 LCASE TO UCASE CONVERSION PROGRAM (continued)



```
C23E  0056     pointer=pointer+1;
C245  0057     repeat
C245  0058        if buffer(pointer)='" .and buffer(pointer+1)='" then
C272  0059        begin
C272  0060           output('");
C27B  0061           pointer=pointer+1;
C282  0062        end;
C282  0063        output(buffer(pointer));
C292  0064        pointer=pointer+1;
C299  0065     until buffer(pointer)='" .and buffer(pointer+1)<>'";
C2C4  0066     output('");
C2CD  0067     pointer=pointer+1;
C2D4  0068  endproc;
C2D5  0069
C2D5  0070  procedure ulc: byte count, extension(3);
C2D7  0071     get_filename(.fcb);
C2E1  0072     if error then abort;
C2EC  0073     if fcb(12)=0 then
C2F5  0074     begin
C2F5  0075        fcb(12)='P;
C2FA  0076        fcb(13)='L;
C2FF  0077        fcb(14)='9;
C304  0078     end;
C304  0079     open_for_read(.fcb);
C30E  0080     if error then abort;
C319  0081
C319  0082     pointer=0;
C31E  0083     repeat
C31E  0084        buffer(pointer)=read(.fcb);
C333  0085        if error then
C33B  0086           if error=8 then break else abort;
C34D  0087        pointer=pointer+1;
C354  0088     forever;
C354  0089     last=pointer;
C35A  0090     close_file(.fcb);
C364  0091
C364  0092     extension(0)=fcb(12);
C369  0093     extension(1)=fcb(13);
C36E  0094     extension(2)=fcb(14);
C373  0095
C373  0096     fcb(12)='T;
C378  0097     fcb(13)='M;
C37D  0098     fcb(14)='P;
C382  0099
C382  0100     open_for_write(.fcb);
C38C  0101     if error then abort;
C397  0102
C397  0103     pointer=0;
C39C  0104     repeat
C39C  0105        if buffer(pointer)='/ .and buffer(pointer+1)='*
C3BB  0106           then skip_comment;
C3CC  0107        else if buffer(pointer)='" then skip_string;
C3E1  0108        else if buffer(pointer)='' then
C3F3  0109        begin
C3F3  0110           output('');
```

10.01.05 LCASE TO UCASE CONVERSION PROGRAM (continued)

Page 3: Upper to Lower Case Conversion June 1 1984

```
C3FC  0111          output(buffer(pointer+1));
C40F  0112          pointer=pointer+2;
C416  0113       end;
C416  0114       else
C419  0115       begin
C419  0116          if buffer(pointer)>='A .and buffer(pointer)<='Z
C435  0117             then buffer(pointer)=buffer(pointer)+$20;
C450  0118          output(buffer(pointer));
C460  0119          pointer=pointer+1;
C467  0120       end;
C467  0121    until pointer>=last;
C46F  0122    close_file(.fcb);
C479  0123    if error then abort;
C484  0124
C484  0125    fcb(12)=extension(0);
C489  0126    fcb(13)=extension(1);
C48E  0127    fcb(14)=extension(2);
C493  0128
C493  0129    delete_file(.fcb);
C49D  0130    if error then abort;
C4A8  0131
C4A8  0132    fcb(12)='T;
C4AD  0133    fcb(13)='M;
C4B2  0134    fcb(14)='P;
C4B7  0135
C4B7  0136    count=0;
C4B9  0137    repeat
C4B9  0138       fcb(count+53)=fcb(count+4);
C4D3  0139       count=count+1;
C4D5  0140    until count=8;
C4DB  0141
C4DB  0142    fcb(61)=extension(0);
C4E0  0143    fcb(62)=extension(1);
C4E5  0144    fcb(63)=extension(2);
C4EA  0145
C4EA  0146    rename_file(.fcb);
C4F4  0147    if error then abort;
```

## 10.01.05 LCASE TO UCASE CONVERSION PROGRAM (continued)

Page 4: Upper to Lower Case Conversion June 1 1984

```
PROCEDURES:
putchar             C103
flex                C10B
get_filename        C10E
set_extension       C118
report_error        C120
open_for_read       C125
read                C12E    BYTE
open_for_write      C139
write               C142
read_sector         C149
write_sector        C15D
set_binary          C171
close_file          C179
delete_file         C182
rename_file         C190
abort               C199
output              C1A6
skip_comment        C1D9
skip_string         C235
ulc                 C2D5

DATA:

EXTERNALS:
fcb                 C840    BYTE
error               C841    BYTE
line_pointer        CC14    INTEGER
pointer             0000    INTEGER
last                0002    INTEGER
name                0004    BYTE
buffer              0018    BYTE
nul                 0000
abt                 0003
bel                 0007
bs                  0008
lf                  000A
cr                  000D
can                 0018
esc                 001B
sp                  0020

GLOBALS:
```

## 10.01.06 SORTED DISK DIRECTORY PROGRAM

Page 1: SORTED DIRECTORY OF FLEX DISK June 1 1984

```
0000  0001 /* SORTED DIRECTORY OF FLEX DISK */
0000  0002
0000  0003 /* THIS IS AN ADAPTATION OF A PROGRAM WRITTEN IN PASCAL */
0000  0004 /* THAT WAS PUBLISHED IN '68 MICRO JOURNAL. IT WILL     */
0000  0005 /* SORT A DISK WITH UP TO 300 DIRECTORY ENTRIES AS IT   */
0000  0006 /* STANDS.  IF YOUR DISK HAS MORE FILES ON IT THAN THIS */
0000  0007 /* ALTER THE GLOBAL VALUES ACCORDINGLY.                */
0000  0008
0000  0009 /* SYNTAX IS:  +++SORT-DIR,<drive number> <CR> */
0000  0010
0000  0011
0000  0012 /* This program uses quite a lot of stack space for the */
0000  0013 /* vectors it will be sorting.  The program runs just   */
0000  0014 /* under 'MEMEND' in a 56K system and locates the stack */
0000  0015 /* just underneath itself.                              */
0000  0016
0000  0017
0000  0018 constant true=-1, false=0;
0000  0019
0000  0020 at $c840:
0000  0021    byte fcb(0):
0000  0022    byte fcode:
0000  0023    byte errstat:
0000  0024    byte actstat:
0000  0025    byte drive:
0000  0026    byte fname(8):
0000  0027    byte fext(3):
0000  0028    byte fattribute:
0000  0029    byte byte16:
0000  0030    integer startadd, endadd:
0000  0031    integer fsize:
0000  0032    byte fsecmap:
0000  0033    byte byte24:
0000  0034    byte month, day, year:
0000  0035    integer fcblptr:
0000  0036    integer curpos, currecno:
0000  0037    byte dataindex, randindex:
0000  0038    byte namewkbuf(11):
0000  0039    byte track, sector:
0000  0040    byte startindex:
0000  0041    byte firstdeldir(3):
0000  0042    byte scratch(6):
0000  0043    byte spacecflag, scratch_dummy(4):
0000  0044    byte secbuffer(256);
0000  0045
0000  0046 at $cc0c: byte workdrive;
0000  0047 at $cc0e: byte mm,dd,yy;
0000  0048
0000  0049 origin=$B700; stack=*;
B703  0050
B703  0051 /* GLOBALS SET UP FOR 300 DIRECTORY ENTRIES */
B703  0052
```

10.01.06 SORTED DISK DIRECTORY PROGRAM (continued)

Page 2: SORTED DIRECTORY OF FLEX DISK June 1 1984

```
B703  0053 global
B703  0054    integer nfree, nsec:
B703  0055    integer tablen:
B703  0056    integer tabptr(300):
B703  0057    integer tabsize(300):
B703  0058    byte tabname(2700):   /* 300*9 */
B703  0059    byte tabext(1200):    /* 300*4 */
B703  0060    byte tabmonth(300):
B703  0061    byte tabday(300):
B703  0062    byte tabyear(300);
B70B  0063
B70B  0064 /*  THE FOLLOWING ARE EXTRACTED FROM 'IOSUBS.LIB' */
B70B  0065
B70B  0066 constant cr=$0d,lf=$0a,sp=$20,bs=$08,
B70B  0067
B70B  0068          nul=$00,abt=$03,can=$18,bel=$07,esc=$1b;
B70B  0069
B70B  0070
B70B  0071 procedure putchar(byte char);
B70E  0072    acca=char;
B712  0073    call $cd18; /* FLEX 'PUTCHR'  (HONOURS 'TTYSET' PARAMETERS) */
B715  0074 endproc;
B716  0075
B716  0076
B716  0077 procedure crlf;
B716  0078    putchar(cr);
B71E  0079    putchar(lf);
B726  0080 endproc;
B727  0081
B727  0082
B727  0083 procedure print(byte .string): byte char;
B729  0084    while string
B729  0085       begin
B731  0086          if string='\ then
B73A  0087             begin
B73A  0088                .string=.string+1;
B741  0089                if string >= 'a .and string <= 'z
B751  0090                   then char=string - $20;  /* CONVERT TO UPPER CASE/
B766  0091                   else char=string;
B76E  0092                if char = 'N then crlf;
B778  0093                else begin
B77B  0094                     putchar('\);
B783  0095                     putchar(char);
B78B  0096                     end;
B78B  0097             end;
B78B  0098          else putchar(string);
B798  0099          .string=.string+1;
B79F  0100       end;
B79F  0101 endproc;
B7A4  0102
B7A4  0103
```

10.01.06 SORTED DISK DIRECTORY PROGRAM (continued)

Page 3: SORTED DIRECTORY OF FLEX DISK June 1 1984

```
B7A4  0104 procedure space(integer n);
B7A4  0105    while n<>0
B7A6  0106       begin
B7AD  0107          putchar(sp);
B7B6  0108          n=n-1;
B7BD  0109       end;
B7BD  0110 endproc;
B7C0  0111
B7C0  0112
B7C0  0113 /*  THE FOLLOWING ARE EXTRACTED FROM 'STRSUBS.LIB' */
B7C0  0114
B7C0  0115 procedure strlen(byte .string): integer len;
B7C2  0116    len=0;
B7C7  0117    while string(len) len=len+1;
B7DB  0118 endproc len;
B7E2  0119
B7E2  0120
B7E2  0121 procedure strcopy(byte .string1,.string2): integer index;
B7E4  0122    index=-1;
B7E9  0123    repeat
B7E9  0124       index=index+1;
B7F0  0125       string1(index)=string2(index);
B802  0126    until string2(index)=0;
B80E  0127 endproc;
B811  0128
B811  0129
B811  0130 procedure strcmp(byte .string1,.string2):
B811  0131    byte c1,c2: integer index;
B813  0132    index=-1;
B818  0133    repeat
B818  0134       index=index+1;
B81F  0135       c1=string1(index); c2=string2(index);
B833  0136       if c1=0 .and c2=0 then return 0;
B854  0137       if c1=0 then return -1;
B861  0138       if c2=0 then return 1;
B86E  0139    until c1<>c2;
B874  0140    if c1>c2 then return 1;
B881  0141 endproc -1;
B886  0142
B886  0143 /* HERE'S AN EXAMPLE OF A RECURSIVE PROCEDURE IN PL/9 */
B886  0144
B886  0145 procedure printint(integer n);
B886  0146    if n>=10 then printint(n/10);
B8FD  0147    putchar n\10+'0;
B910  0148 endproc;
B911  0149
```

## 10.01.06 SORTED DISK DIRECTORY PROGRAM (continued)

Page 4: SORTED DIRECTORY OF FLEX DISK June 1 1984

```
B911  0150 procedure printdec(integer n: byte digits,char):
B911  0151    integer nn: byte l;
B913  0152    l=1;
B917  0153    nn=n;
B91B  0154    while nn>=10
B91D  0155    begin
B924  0156       nn=nn/10;
B930  0157       l=l+1;
B932  0158    end;
B932  0159    while l<digits
B936  0160    begin
B93C  0161       putchar char;
B945  0162       l=l+1;
B947  0163    end;
B947  0164    printint n;
B952  0165 endproc;
B955  0166
B955  0167 procedure fmscall(byte code);
B955  0168    fcode=code;
B95A  0169    xreg=.fcb;
B961  0170    call $d406;  /* FLEX 'FMS' */
B964  0171 endproc;
B965  0172
B965  0173 procedure line;
B965  0174    print "----------------------  ";
B98E  0175    print "----------------------  ";
B9B7  0176    print "----------------------\n";
B9E0  0177 endproc;
B9E1  0178
B9E1  0179 procedure sort(integer n):
B9E1  0180    integer i,temp:
B9E1  0181    byte done;
B9E3  0182    repeat
B9E3  0183       done=true;
B9E7  0184       i=0;
B9EC  0185       repeat
B9EC  0186          if strcmp(.tabname(tabptr(i)*9),
BA26  0187                    .tabname(tabptr(i+1)*9))>0 then
BA4D  0188          begin
BA4D  0189             temp=tabptr(i);
BA59  0190             tabptr(i)=tabptr(i+1);
BA72  0191             tabptr(i+1)=temp;
BA81  0192             done=false;
BA83  0193          end;
BA83  0194          i=i+1;
BA8A  0195       until i=n-1;
BA9D  0196    until done;
BAA4  0197 endproc;
BAA7  0198
```

10.01.06 SORTED DISK DIRECTORY PROGRAM (continued)

Page 5: SORTED DIRECTORY OF FLEX DISK June 1 1984

```
BAA7  0199 procedure sorted_directory:
BAA7  0200    integer i: byte j;
BAA9  0201    tablen=0;
BAAE  0202    call $cd48;  /* FLEX 'INDEC' */
BAB1  0203    if accb then drive=xreg;
BABB  0204    else drive=workdrive;
BAC4  0205    fmscall(16);
BACD  0206    fmscall(7);
BAD6  0207
BAD6  0208    nfree=fsize;
BADB  0209    nsec=0;
BAE0  0210
BAE0  0211    space 17;
BAEA  0212    print "Directory (";
BB03  0213    printint mm; putchar '/;
BB17  0214    printint dd; putchar '/;
BB2B  0215    printdec yy,2,'0;
BB3E  0216    print ")\n     Disk: ";
BB5A  0217    i=0;
BB5F  0218    repeat
BB5F  0219       if fname(i) then putchar fname(i);
BB7D  0220       i=i+1;
BB84  0221    until i=8;
BB8B  0222    print " #";
BB9B  0223    printint swap(integer(fattribute))+byte16;
BBB0  0224    print ", Created ";
BBC8  0225    printint fsecmap; putchar '/;
BBDC  0226    printint byte24; putchar '/;
BBF0  0227    printdec month,2,'0;
BC03  0228    call $cd24;  /* FLEX 'PCRLF' */
BC06  0229    line;
BC09  0230
BC09  0231    fmscall(6);
BC12  0232
BC12  0233    while errstat<>8
BC15  0234    begin
BC1B  0235       errstat=0;
BC1E  0236       fmscall(7);
BC27  0237       if fname>0 .and fname<127 then
BC45  0238       begin
BC45  0239          tabptr(tablen)=tablen;
BC51  0240          strcopy(.tabname(tablen*9),"        ");
BC79  0241          i=0;
BC7E  0242          repeat
BC7E  0243             if fname(i) then
BC8C  0244                tabname(tablen*9+i)=fname(i);
BCAB  0245             i=i+1;
BCB2  0246          until i=8;
BCB9  0247          strcopy(.tabext(tablen*4),"   ");
BCDC  0248          i=0;
BCE1  0249          repeat
BCE1  0250             if fext(i) then
BCEF  0251                tabext(tablen*4+i)=fext(i);
BD0E  0252             i=i+1;
BD15  0253          until i=4;
```

10.01.06 SORTED DISK DIRECTORY PROGRAM (continued)

Page 6: SORTED DIRECTORY OF FLEX DISK June 1 1984

```
BD1C  0254          tabsize(tablen)=fsize;
BD2B  0255          nsec=nsec+fsize;
BD32  0256          tabmonth(tablen)=month;
BD3F  0257          tabday(tablen)=day;
BD4C  0258          tabyear(tablen)=year;
BD59  0259          tablen=tablen+1;
BD60  0260       end;
BD60  0261    end;
BD60  0262
BD60  0263    sort tablen;
BD6C  0264
BD6C  0265    i=0; j=0;
BD73  0266    repeat
BD73  0267       print .tabname(tabptr(i)*9);
BD92  0268       putchar '.;
BD9B  0269       print .tabext(tabptr(i)*4);
BDBA  0270       printdec tabsize(tabptr(i)),4,sp;
BDDD  0271       printdec tabmonth(tabptr(i)),3,sp;
BDFF  0272       putchar '/;
BE08  0273       printdec tabday(tabptr(i)),2,sp;
BE2A  0274       putchar '/;
BE33  0275       printdec tabyear(tabptr(i)),2,'0;
BE55  0276       j=j+1;
BE57  0277       if j=3 then
BE5F  0278       begin
BE5F  0279          j=0;
BE61  0280          call $cd24;  /* FLEX 'PCRLF' */
BE64  0281       end;
BE64  0282       else space 2;
BE71  0283       i=i+1;
BE78  0284    until i=tablen;
BE80  0285    if j then call $cd24;  /* FLEX 'PCRLF' */
BE8A  0286    line;
BE8D  0287    print "Files: ";
BEA2  0288    printint tablen;
BEAB  0289    print ", Sectors: ";
BEC4  0290    printint nsec;
BECD  0291    print ", Free: ";
BEE3  0292    printint nfree;
BEEC  0293    call $cd24;  /* FLEX 'PCRLF' */
```

10.01.06 SORTED DISK DIRECTORY PROGRAM (continued)

Page 7: SORTED DIRECTORY OF FLEX DISK June 1 1984

```
PROCEDURES:
putchar              B70E
crlf                 B716
print                B727
space                B7A4
strlen               B7C0   INTEGER
strcopy              B7E2
strcmp               B811   BYTE
printint             B886
printdec             B911
fmscall              B955
line                 B965
sort                 B9E1
sorted_directory     BAA7

DATA:

EXTERNALS:
true                 FFFF
false                0000
fcb                  C840   BYTE
fcode                C840   BYTE
errstat              C841   BYTE
actstat              C842   BYTE
drive                C843   BYTE
fname                C844   BYTE
fext                 C84C   BYTE
fattribute           C84F   BYTE
byte16               C850   BYTE
startadd             C851   INTEGER
endadd               C853   INTEGER
fsize                C855   INTEGER
fsecmap              C857   BYTE
byte24               C858   BYTE
month                C859   BYTE
day                  C85A   BYTE
year                 C85B   BYTE
fcblptr              C85C   INTEGER
curpos               C85E   INTEGER
currecno             C860   INTEGER
dataindex            C862   BYTE
randindex            C863   BYTE
namewkbuf            C864   BYTE
track                C86F   BYTE
sector               C870   BYTE
startindex           C871   BYTE
firstdeldir          C872   BYTE
scratch              C875   BYTE
spacecflag           C87B   BYTE
scratch_dummy        C87C   BYTE
secbuffer            C880   BYTE
workdrive            CC0C   BYTE
mm                   CC0E   BYTE
dd                   CC0F   BYTE
```

10.01.06 SORTED DISK DIRECTORY PROGRAM (continued)

Page 8: SORTED DIRECTORY OF FLEX DISK June 1 1984

```
yy                  CC10   BYTE
cr                  000D
lf                  000A
sp                  0020
bs                  0008
nul                 0000
abt                 0003
can                 0018
bel                 0007
esc                 001B

GLOBALS:
nfree               0000   INTEGER
nsec                0002   INTEGER
tablen              0004   INTEGER
tabptr              0006   INTEGER
tabsize             025E   INTEGER
tabname             04B6   BYTE
tabext              0F42   BYTE
tabmonth            13F2   BYTE
tabday              151E   BYTE
tabyear             164A   BYTE
```

## 10.01.07 BINARY MOVE UTILITY

Page 1: MOVE UTILITY June 1 1984

```
0000  0001 /* MOVE UTILITY */
0000  0002
0000  0003 origin=$C100;  /* RUNS IN FLEX TCA */
C100  0004
C100  0005 global byte erflag, keychar:integer .start, .end, .dest;
C106  0006
C106  0007
C106  0008 include 0.trufalse.def;
C106  0009 include 0.iosubs.lib;
C3EE  0010 include 0.hexio.lib;
C4E3  0011
C4E3  0012
C4E3  0013 procedure cancel;
C4E3  0014    print "\R                                \R";
C512  0015 endproc;
C513  0016
C513  0017
C513  0018 procedure abort_check;
C513  0019    if erflag = true .and keychar = abt
C521  0020       then begin
C52F  0021            crlf;
C532  0022            crlf;
C535  0023            jump $cd03;  /* BACK TO FLEX */
C538  0024            end;
C538  0025 endproc;
C539  0026
C539  0027
C539  0028 procedure get_start:integer start;
C53B  0029    repeat
C53B  0030       cancel;
C53D  0031       print "START ADDRESS: $";
C55B  0032       start = get_hex_address;
C560  0033       abort_check;
C562  0034    until erflag=false;
C568  0035 endproc integer start;
C56D  0036
C56D  0037
C56D  0038 procedure get_end:integer end;
C56F  0039    crlf;
C572  0040    repeat
C572  0041       cancel;
C575  0042       print "END   ADDRESS: $";
C593  0043       end = get_hex_address;
C598  0044       abort_check;
C59B  0045    until erflag=false;
C5A1  0046 endproc integer end;
C5A6  0047
C5A6  0048
C5A6  0049 procedure get_destination:integer destination;
C5A8  0050    crlf;
C5AB  0051    repeat
C5AB  0052       cancel;
C5AE  0053       print "DEST  ADDRESS: $";
C5CC  0054       destination = get_hex_address;
C5D1  0055       abort_check;
C5D4  0056    until erflag=false;
C5DA  0057 endproc integer destination;
```

10.01.07 BINARY MOVE UTILITY (continued)

Page 2: MOVE UTILITY June 1 1984

```
C5DF  0058
C5DF  0059
C5DF  0060 procedure lo_hi_move;
C5DF  0061    repeat
C5DF  0062       dest = start;        /* SHIFT THE DATA */
C5E5  0063       .dest = .dest + 1;   /* BUMP THE POINTERS */
C5EC  0064       .start = .start + 1;
C5F3  0065    until !.start = .end;   /* LOOK FOR END */
C603  0066 endproc;
C604  0067
C604  0068
C604  0069 procedure hi_lo_move;
C604  0070    .dest = !.dest + (.end - .start);  /* ADD SIZE TO DEST */
C61A  0071    repeat
C61A  0072       dest = end;          /* SHIFT THE DATA */
C620  0073       .dest = .dest - 1;   /* BUMP THE POINTERS */
C627  0074       .end = .end - 1;
C62E  0075    until !.end < .start;   /* LOOK FOR END */
C63E  0076 endproc;
C63F  0077
C63F  0078
C63F  0079 procedure move;
C63F  0080
C63F  0081    crlf; crlf;
C645  0082
C645  0083    repeat
C645  0084       .start = get_start;
C64A  0085       .end  = get_end;
C64F  0086       if !.start > .end
C653  0087           then print "\N\NILLEGAL ADDRESSES\B\N\N";
C68A  0088           else break;
C690  0089    forever;
C690  0090
C690  0091    .dest = get_destination;
C697  0092
C697  0093    if !.dest > .start .and .dest <= .end
C6B1  0094           then hi_lo_move;
C6CA  0095           else lo_hi_move;
C6D0  0096
C6D0  0097    crlf;
```

10.01.07 BINARY MOVE UTILITY (continued)

Page 3: MOVE UTILITY June 1 1984

PROCEDURES:

| | | |
|---|---|---|
| monitor | C109 | |
| warms | C10E | |
| getchar | C112 | BYTE |
| getchar_noecho | C118 | BYTE |
| getkey | C11F | BYTE |
| convert_lc | C137 | BYTE |
| get_uc | C15C | BYTE |
| get_uc_noecho | C169 | BYTE |
| putchar | C176 | |
| printint | C17E | |
| remove_char | C223 | |
| input | C23F | INTEGER |
| crlf | C2E3 | |
| print | C2F6 | |
| space | C3D2 | |
| get_hex_nibble | C3EE | BYTE |
| get_hex_byte | C44D | BYTE |
| get_hex_address | C475 | INTEGER |
| put_hex_nibble | C49B | |
| put_hex_byte | C4BB | |
| put_hex_address | C4D0 | |
| cancel | C4E3 | |
| abort_check | C513 | |
| get_start | C539 | INTEGER |
| get_end | C56D | INTEGER |
| get_destination | C5A6 | INTEGER |
| lo_hi_move | C5DF | |
| hi_lo_move | C604 | |
| move | C63F | |

DATA:

EXTERNALS:

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

GLOBALS:

| | | |
|---|---|---|
| erflag | 0000 | BYTE |
| keychar | 0001 | BYTE |
| start | 0002 | INTEGER |
| end | 0004 | INTEGER |
| dest | 0006 | INTEGER |

## 10.01.08 INTEL HEX DUMP ROUTINE

Page 1: HEX DUMP IN INTEL FORMAT FROM A FLEX BINARY FILE June 1 1984

```
0000 0001 /* HEX DUMP IN INTELLEC FORMAT FROM A FLEX BINARY FILE. */
0000 0002 /* OUTPUT IS DIRECTED THROUGH THE STANDARD FLEX PRINTER */
0000 0003 /* DRIVERS TO ENABLE THE USER TO EASILY CUSTOMIZE THE   */
0000 0004 /* OUTPUT HARDWARE CONFIGURATION.                       */
0000 0005
0000 0006 origin=$c200; stack=*;
C203 0007
C203 0008 constant end_of_file=8;
C203 0009
C203 0010 at $c840: byte fcb,error(319);
C203 0011 at $cc09: byte ttyset_pause;
C203 0012
C203 0013 global byte checksum:
C203 0014        integer length,address(0):
C203 0015        byte address_high,address_low:
C203 0016        byte ttyset_pause_save:
C203 0017        byte buffer(255);
C20B 0018
C20B 0019 include 0.flex.lib;
C29C 0020
C29C 0021 constant cr=$0d,lf=$0a,sp=$20,bs=$08,
C29C 0022          nul=$00,abt=$03,can=$18,bel=$07,esc=$1b,
C29C 0023          false=0, true=-1;
C29C 0024
C29C 0025 procedure putchar(byte char);
C29C 0026    acca=char;
C2A0 0027    call $cd18; /* FLEX 'PUTCHR' */
C2A3 0028 endproc;
C2A4 0029
C2A4 0030 procedure init;
C2A4 0031    ttyset_pause_save = ttyset_pause;
C2A9 0032    ttyset_pause = 0;
C2AC 0033    call $ccc0; /* FLEX 'PINIT' */
C2AF 0034 endproc;
C2B0 0035
C2B0 0036 procedure put_char (byte char);
C2B0 0037    acca=char; call $cce4; /* FLEX 'POUT' */
C2B7 0038    if char=lf then return;
C2C0 0039    if char=cr then call $cd24 /* FLEX 'PCRLF' */
C2CB 0040    else putchar(char);
C2D6 0041 endproc;
C2D7 0042
C2D7 0043 procedure put_crlf;
C2D7 0044    put_char(cr);
C2DF 0045    put_char(lf);
C2E7 0046 endproc;
C2E8 0047
C2E8 0048 procedure put_hex (byte digit);
C2E8 0049    digit=(digit and $f)+'0;
C2F0 0050    if digit>'9 then digit=digit+7;
C2FE 0051    put_char(digit);
C306 0052 endproc;
C307 0053
```

10.01.08 INTEL HEX DUMP ROUTINE (continued)

Page 2: HEX DUMP IN INTEL FORMAT FROM A FLEX BINARY FILE June 1 1984

```
C307  0054 procedure put_byte (byte item);
C307  0055    put_hex(shift(item,-4));
C313  0056    put_hex(item);
C31B  0057    checksum=checksum+item;
C321  0058 endproc;
C322  0059
C322  0060 procedure put_address (integer item);
C322  0061    put_byte(swap(item));
C32C  0062    put_byte(item);
C334  0063 endproc;
C335  0064
C335  0065 procedure put_record:
C335  0066    byte count:
C335  0067    integer position,marker;
C337  0068    position=0;
C33C  0069    repeat
C33C  0070       if length-position>=16 then count=16
C349  0071       else count=length-position;
C354  0072       marker=position+count;
C35F  0073       put_crlf;
C362  0074       put_char(':);
C36B  0075       checksum=0;
C36D  0076       put_byte(count);
C375  0077       put_address(address);
C37D  0078       put_byte(0);
C385  0079       repeat
C385  0080          put_byte(buffer(position));
C394  0081          position=position+1;
C39B  0082       until position=marker;
C3A1  0083       put_byte(-(checksum));
C3AE  0084       address=address+count;
C3B9  0085    until position=length;
C3C1  0086 endproc;
C3C4  0087
C3C4  0088 procedure end_record;
C3C4  0089    put_crlf;
C3C7  0090    put_char(':);
C3D0  0091    put_byte(0);
C3D9  0092    put_address(0);
C3E3  0093    put_byte(0);
C3EC  0094    put_byte(0);
C3F5  0095    put_crlf;
C3F8  0096 endproc;
C3F9  0097
C3F9  0098 procedure get_record: byte char: integer i;
C3FB  0099    repeat
C3FB  0100       char=read(.fcb);
C407  0101       if error then return;
C412  0102       if char=$16 then
C41A  0103          begin
C41A  0104             read(.fcb); if error then return;
C42F  0105             read(.fcb); if error then return;
C444  0106          end;
C444  0107    until char=2;
C44A  0108    address_high=read(.fcb);
C456  0109    if error then return;
C461  0110    address_low=read(.fcb);
```

10.01.08 INTEL HEX DUMP ROUTINE (continued)

Page 3: HEX DUMP IN INTEL FORMAT FROM A FLEX BINARY FILE June 1 1984

```
C46D  0111     if error then return;
C478  0112     length=integer(read(.fcb));
C485  0113     if error then return;
C490  0114     i=0;
C495  0115     repeat
C495  0116        buffer(i)=read(.fcb);
C4A9  0117        if error then return;
C4B4  0118        i=i+1;
C4BB  0119     until i=length;
C4C1  0120 endproc;
C4C4  0121
C4C4  0122 procedure hexdump;
C4C4  0123     get_filename(.fcb);
C4CE  0124     if error then return;
C4D7  0125     set_extension(.fcb,0);
C4E5  0126     open_for_read(.fcb);
C4EF  0127     if error then return;
C4F8  0128     set_binary(.fcb);
C502  0129     repeat
C502  0130        get_record;
C505  0131        if error=end_of_file then
C50E  0132           begin
C50E  0133              error=false;
C511  0134              return;
C512  0135           end;
C512  0136        if error then return;
C518  0137        put_record;
C51E  0138     forever;
C51E  0139 endproc;
C521  0140
C521  0141 procedure main;
C521  0142     init;
C524  0143     hexdump;
C526  0144     if error then report_error(.fcb);
C538  0145     close_file(.fcb);
C542  0146     end_record;
C545  0147     ttyset_pause = ttyset_pause_save;
```

10.01.08 INTEL HEX DUMP ROUTINE (continued)

Page 4: HEX DUMP IN INTEL FORMAT FROM A FLEX BINARY FILE June 1 1984

```
PROCEDURES:
flex                C20E
get_filename        C211
set_extension       C21B
report_error        C223
open_for_read       C228
read                C231    BYTE
open_for_write      C23C
write               C245
read_sector         C24C
write_sector        C260
set_binary          C274
close_file          C27C
delete_file         C285
rename_file         C293
putchar             C29C
init                C2A4
put_char            C2B0
put_crlf            C2D7
put_hex             C2E8
put_byte            C307
put_address         C322
put_record          C335
end_record          C3C4
get_record          C3F9
hexdump             C4C4
main                C521

DATA:

EXTERNALS:
end_of_file         0008
fcb                 C840    BYTE
error               C841    BYTE
ttyset_pause        CC09    BYTE
cr                  000D
lf                  000A
sp                  0020
bs                  0008
nul                 0000
abt                 0003
can                 0018
bel                 0007
esc                 001B
false               0000
true                FFFF

GLOBALS:
checksum            0000    BYTE
length              0001    INTEGER
address             0003    INTEGER
address_high        0003    BYTE
address_low         0004    BYTE
ttyset_pause_save   0005    BYTE
buffer              0006    BYTE
```

## 10.01.09 MOTOROLA HEX DUMP ROUTINE

Page 1: HEX DUMP IN MOTOROLA FORMAT FROM A FLEX DISK FILE June 1 1984

```
0000  0001 /* HEX DUMP IN MOTOROLA FORMAT FROM A FLEX DISK FILE.   */
0000  0002 /* OUTPUT IS DIRECTED THROUGH THE STANDARD FLEX PRINTER */
0000  0003 /* DRIVERS TO ENABLE THE USER TO EASILY CUSTOMIZE THE   */
0000  0004 /* OUTPUT HARDWARE CONFIGURATION.                       */
0000  0005
0000  0006 origin=$c200; stack=*;
C203  0007
C203  0008 constant end_of_file=8;
C203  0009
C203  0010 at $c840: byte fcb,error(319);
C203  0011 at $cc09: byte ttyset_pause;
C203  0012
C203  0013 global byte checksum:
C203  0014        integer length,address(0):
C203  0015        byte address_high,address_low:
C203  0016        byte ttyset_pause_save:
C203  0017        byte buffer(255);
C20B  0018
C20B  0019 include 0.flex.lib;
C29C  0020
C29C  0021 constant cr=$0d,lf=$0a,sp=$20,bs=$08,
C29C  0022          nul=$00,abt=$03,can=$18,bel=$07,esc=$1b,
C29C  0023          false=0, true=-1;
C29C  0024
C29C  0025 procedure putchar(byte char);
C29C  0026    acca=char;
C2A0  0027    call $cd18; /* FLEX 'PUTCHR' */
C2A3  0028 endproc;
C2A4  0029
C2A4  0030 procedure init;
C2A4  0031    ttyset_pause_save = ttyset_pause;
C2A9  0032    ttyset_pause = 0;
C2AC  0033    call $ccc0; /* FLEX 'PINIT' */
C2AF  0034 endproc;
C2B0  0035
C2B0  0036 procedure put_char (byte char);
C2B0  0037    acca=char; call $cce4; /* FLEX 'POUT' */
C2B7  0038    if char=lf then return;
C2C0  0039    if char=cr then call $cd24 /* FLEX 'PCRLF' */
C2CB  0040    else putchar(char);
C2D6  0041 endproc;
C2D7  0042
C2D7  0043 procedure put_crlf;
C2D7  0044    put_char(cr);
C2DF  0045    put_char(lf);
C2E7  0046 endproc;
C2E8  0047
C2E8  0048 procedure put_hex (byte digit);
C2E8  0049    digit=(digit and $f)+'0;
C2F0  0050    if digit>'9 then digit=digit+7;
C2FE  0051    put_char(digit);
C306  0052 endproc;
C307  0053
```

10.01.09 MOTOROLA HEX DUMP ROUTINE (continued)

Page 1: HEX DUMP IN MOTOROLA FORMAT FROM A FLEX DISK FILE June 1 1984

```
C307  0054 procedure put_byte (byte item);
C307  0055    put_hex(shift(item,-4));
C313  0056    put_hex(item);
C31B  0057    checksum=checksum+item;
C321  0058 endproc;
C322  0059
C322  0060 procedure put_address (integer item);
C322  0061    put_byte(swap(item));
C32C  0062    put_byte(item);
C334  0063 endproc;
C335  0064
C335  0065 procedure put_record:
C335  0066    byte count:
C335  0067    integer position,marker;
C337  0068    position=0;
C33C  0069    repeat
C33C  0070       if length-position>=16 then count=16
C349  0071       else count=length-position;
C354  0072       marker=position+count;
C35F  0073       put_crlf;
C362  0074       put_char('S);
C36B  0075       put_char('1);
C374  0076       checksum=0;
C376  0077       put_byte(count+3);
C380  0078       put_address(address);
C388  0079       repeat
C388  0080          put_byte(buffer(position));
C397  0081          position=position+1;
C39E  0082       until position=marker;
C3A4  0083       put_byte(not(checksum));
C3AE  0084       address=address+count;
C3B9  0085    until position=length;
C3C1  0086 endproc;
C3C4  0087
C3C4  0088 procedure end_record;
C3C4  0089    put_crlf;
C3C7  0090    put_char('S);
C3D0  0091    put_char('9);
C3D9  0092    put_crlf;
C3DC  0093 endproc;
C3DD  0094
C3DD  0095 procedure get_record: byte char: integer i;
C3DF  0096    repeat
C3DF  0097       char=read(.fcb);
C3EB  0098       if error then return;
C3F6  0099       if char=$16 then
C3FE  0100          begin
C3FE  0101             read(.fcb); if error then return;
C413  0102             read(.fcb); if error then return;
C428  0103          end;
C428  0104    until char=2;
C42E  0105    address_high=read(.fcb);
C43A  0106    if error then return;
C445  0107    address_low=read(.fcb);
C451  0108    if error then return;
C45C  0109    length=integer(read(.fcb));
C469  0110    if error then return;
```

10.01.09 MOTOROLA HEX DUMP ROUTINE (continued)

Page 3: HEX DUMP IN MOTOROLA FORMAT FROM A FLEX DISK FILE June 1 1984

```
C474  0111     i=0;
C479  0112     repeat
C479  0113        buffer(i)=read(.fcb);
C48D  0114        if error then return;
C498  0115        i=i+1;
C49F  0116     until i=length;
C4A5  0117  endproc;
C4A8  0118
C4A8  0119  procedure hexdump;
C4A8  0120     get_filename(.fcb);
C4B2  0121     if error then return;
C4BB  0122     set_extension(.fcb,0);
C4C9  0123     open_for_read(.fcb);
C4D3  0124     if error then return;
C4DC  0125     set_binary(.fcb);
C4E6  0126     repeat
C4E6  0127        get_record;
C4E9  0128        if error=end_of_file then
C4F2  0129           begin
C4F2  0130              error=false;
C4F5  0131              return;
C4F6  0132           end;
C4F6  0133        if error then return;
C4FF  0134        put_record;
C502  0135     forever;
C502  0136  endproc;
C505  0137
C505  0138  procedure main;
C505  0139     init;
C508  0140     hexdump;
C50A  0141     if error then report_error(.fcb);
C51C  0142     close_file(.fcb);
C526  0143     end_record;
C529  0144     ttyset_pause = ttyset_pause_save;
```

10.01.09 MOTOROLA HEX DUMP ROUTINE (continued)

Page 4: HEX DUMP IN MOTOROLA FORMAT FROM A FLEX DISK FILE June 1 1984

```
PROCEDURES:
flex                C20E
get_filename        C211
set_extension       C21B
report_error        C223
open_for_read       C228
read                C231   BYTE
open_for_write      C23C
write               C245
read_sector         C24C
write_sector        C260
set_binary          C274
close_file          C27C
delete_file         C285
rename_file         C293
putchar             C29C
init                C2A4
put_char            C2B0
put_crlf            C2D7
put_hex             C2E8
put_byte            C307
put_address         C322
put_record          C335
end_record          C3C4
get_record          C3DD
hexdump             C4A8
main                C505

DATA:

EXTERNALS:
end_of_file         0008
fcb                 C840   BYTE
error               C841   BYTE
ttyset_pause        CC09   BYTE
cr                  000D
lf                  000A
sp                  0020
bs                  0008
nul                 0000
abt                 0003
can                 0018
bel                 0007
esc                 001B
false               0000
true                FFFF

GLOBALS:
checksum            0000   BYTE
length              0001   INTEGER
address             0003   INTEGER
address_high        0003   BYTE
address_low         0004   BYTE
ttyset_pause_save   0005   BYTE
buffer              0006   BYTE
```

THIS PAGE INTENTIONALLY LEFT BLANK