WINDRUSH
Micro Systems Ltd.

PL/9

REFERENCE MANUAL

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

PL/9

REFERENCE MANUAL

(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.

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

by Graham Trott

TABLE OF CONTENTS

TABLE OF CONTENTS

(+) in section 7.05.00 ..... (-) in section 7.06.00
(*) in this section and section 7.06.00

TABLE OF CONTENTS

TABLE OF CONTENTS

TABLE OF CONTENTS

TABLE OF CONTENTS

1.00.00 INTRODUCTION

PL/9 has become a very popular language among programmers involved with the day to day writing of control programs for systems or products incorporating the Motorola MC6809.

PL/9 has been designed specifically for the Motorola MC6809 and as a result it takes full advantage of the architecture of this powerful processor. As no tradeoffs have been made to make the 'core' of PL/9 compatible with other processors there is no intention of porting PL/9 to other processors at this time although we are keeping a close eye on the MC68008 (the 8-bit version of the MC68000).

PL/9 is currently only being offered under the FLEX disk operating system as we feel that this single user operating system offers the most solid base for developing control oriented programs. FLEX imposes absolutely no restrictions on what you may or may not do with the MC6809 instruction set. The simplicity of a non-interrupt driven operating system is also highly desirable lest you find yourself spending more time accommodating the requirements of your operating system than you will writing your program. If your program will fit into 48K of memory the FLEX working environment will be hard to beat.

The popularity of PL/9 is, in our opinion, due to six factors:

1. The interactive working environment offered by the co-resident editor/compiler/tracer greatly assists the programmer when debugging, fine tuning, or adding 'bells and whistles' to programs.

   The traditionally lengthy edit-compile-debug, edit-compile-debug, edit-compile-debug cycles ... with time consuming file loading and saving through the disk operating system intervening between each step of the process is now a thing of the past!

2. The syntax of PL/9 is close enough to Pascal and 'C' that if you are familiar with either of these languages you will not have any problems getting used to the syntax of PL/9.

3. The extremely fast compile time. A source file that produces 8K of binary takes less than 60 seconds to compile! This INCLUDES the load time of the compiler, the source, and the saving of the binary object file!

4. The overall performance of the product in code efficiency and execution times is VERY impressive; the SIEVE of Eratosthenes benchmark runs in 9.4 seconds, with a total program size of 192 bytes (678 bytes if all I/O routines are included).

5. NO RUN-TIME OVERHEADS; be they in the form of a license for an interpreter or a license for a mathematics package. The cost of PL/9 includes an un-limited and non-exclusive license to the PL/9 REAL math package.

6. The ease with which programs that start from system power-up can be developed. PL/9 has the built-in capability to intercept ALL of the MC6809 interrupts, including RESET. Gone are the days of generating the system startup and interrupt handling procedures in assembly language. Now the ENTIRE program from system startup to system power down can be handled by one language ... PL/9.

1.00.01 ACKNOWLEDGEMENTS

We sincerely thank Ron Anderson, Author of the 'FLEX USERS NOTES' column in '68 Micro Journal for his authoritative constructive criticism of the earlier versions of this product. If Ron had not taken the time, trouble, and effort to enter into lengthy correspondence with us and offer many suggestions for improvement this product and its documentation would not be what they are today.

We would also like to acknowledge Ron Anderson as the author of algorithms used in the SCIPACK library and for his valuable assistance in improving the internal arithmetic package.

We would also like to acknowledge Matt Scudiere as the author of the coefficients used in the SINE and ARCTAN routines in the SCIPACK library.

Our thanks also to Neil Jarman for his efforts in improving the file handling capabilities and editor features in our assembler product MACE. These features have subsequently been incorporated into PL/9 by the author.

The information given to Windrush by the author was edited by Bill Dickinson to form this manual but he refuses to take the blame for it.

We also thank Motorola for developing THE most powerful 8-bit processor available, the MC6809, and for having the foresight to produce the MC68008. Keep 'em coming you guys!

Our thanks also to Ric Hammond of Smoke Signal Broadcasting and Richard Don of Gimix for having the foresight in seeing the potential of the SS-50C bus and the Motorola MC6809 and producing hardware that was reliable enough to withstand the demands of professional users. Without the reliability offered by the hardware designs and manufacturing standards of these two companies we would have never considered that the SS-50C bus and FLEX was worth supporting.

And last, but by no means least, we would like to thank John Alford of Alford and Associates for developing what we consider to be the most powerful word processing package available; SCREDITOR III. Without this beautiful piece of software this manual would NEVER have been completed.

1.00.02 HOW TO USE THIS MANUAL

This manual has been written as a tutorial on PL/9, aimed at the complete newcomer to structured programming languages in general, and PL/9 in particular. All of the important features of the language are described, with examples given where appropriate.

As the manual is organized as a tutorial the sequence of presentation had to to be arranged to ensure that the topics contained in each section only made reference to preceeding sections wherever possible. This has necessitated various trade-offs in the order of presentation.

To help compensate for the tutorial organization of the various sections in the manual the index to the keyword definitions is arranged in alphabetical order. This has been devised to assist the user when he needs to address a specific topic.

The information in the Language Reference Manual outlining the keywords is not meant to provide masses of detail; it is designed for quick reference. If the information in the Language Reference section does not supply the detail you require consult the equivalent section in the Users Guide. The Users Guide, which is bound in a separate manual, is arranged in roughly the same order and has, in most cases, identical section titles to assist in locating information.

The body of the reference manual can be considered to be in three main sections:

(1) The CONFIGURATION of PL/9 to match your system hardware environment and to suit your own programming requirements.

(2) The TECHNICAL REFERENCE section which covers the following topics:

a. The PL/9 editor.
b. The PL/9 compiler.
c. The PL/9 tracer.
d. The PL/9 language.
e. The PL/9 libraries.

The body of the users guide can be considered to be in two main sections:

(1) The USERS GUIDE which covers six main areas in considerable detail. This section places heavy emphasis on working programs as examples, which, for the most part, are fully explained on a line by line basis:

a. A brief description of data types and sizes.
b. Program structures.
c. A detailed description of PL/9 arithemetic, and data types.
d. Advanced program structures.
e. Bit operations and bit oriented program structures.
f. Hardware oriented program structures.

(2) A selection of WORKING PL/9 programs.

We make no apologies for our frivolity from time to time in this manual. Technical documentation tends to be BORING. We feel that there is already an ample supply of boring manuals and books around. We do not intend to add this one to the collection.

S T O P - L O O K - L I S T E N

You are encouraged to read this manual from cover to cover before you sit down to write any serious programs. If you can't wait to get started refer to the section on configuring PL/9 to your system hardware. Further instructions for those of you without any patience will be found there.

If you encounter ANY difficulties read this entire manual starting from here!

1.00.03 HISTORY OF PL/9

PL/9 was written to meet a need that has been long felt without any real solution having previously been available. It is aimed primarily at the world of control systems, where programmers traditionally have a choice between BASIC and assembly language.

The former is simple to learn but results in slow and usually unreadable programs. Most BASICs also impose run-time interpreter license considerations that will, quite often, price a product out of the market.

Assembly language allows the programmer to get the utmost in performance out of his computer. Programming at this level has the disadvantage of taking a long time to learn, the programs are difficult to write, and can, at times, be almost impossible to debug!

It is against this background that languages such as PL/M (Programming Language for Microprocessors), originally written for the Intel 8080, but subsequently ported to the 8086 (amongst others), appeared a few years ago.

PL/M strikes a happy balance between code efficiency and ease of programming, while providing a degree of control over the hardware of the system that formal teaching languages such as Pascal, and mainframe languages such as 'C' cannot match.

The Motorola 6800 microprocessor was for many years poorly served for software of this type, except on expensive development systems. Then Tom Crosley wrote SPL/M, a subset of PL/M. This gave 6800 programmers the ability to write control programs, without spending more time trying to get around the limitations of the language than actually writing the program. The language was well-received by the 6800 fraternity and is still in widespread use. When the 6809 appeared, however, no comparable language was initially available (although SPL/M has subsequently been ported to the 6809).

1.00.04 EXISTING TOOLS

Pascal and 'C' are now available for use in microprocessors but due to their heritage are often cumbersome in handling the I/O, et. al., in microprocessor applications.

In addition most BASICs and Pascals require run-time interpreters; some of the 'native code' variety even charge you a licence fee to use their math package. Most 'C' language compilers tend to require a very large overhead for the library modules which tend to be difficult to trim down. When these languages are used to generate code for small dedicated control systems it is like using a sledge hammer to crack a nut ... they'll do the job but they can also have undesirable side effects!

Although languages such as SPL/M, Pascal and C are a vast improvement on BASIC or assembly language, they can still be inconvenient to use.

The program development cycle consists of first using a text editor to create the source file, which is then written back to the disk at the end of the edit session. Next the compiler is called, giving the name of the source file to be compiled and any options required, such as whether an object file is wanted. Lastly, the object file is loaded and tested.

Few languages give the user much help in debugging his programs; it is usually a case of putting diagnostic print instructions into the code at crucial points. If an error is found, the whole edit-compile-load cycle must be repeated; this will take several minutes for all but the smallest of programs.

What is needed is some way of speeding up this process; BASIC for all of its deficiencies at least has the advantage that you can run the program without having to go through all that time consuming disc loading and saving, and it is this factor alone that keeps many programmers faithful to BASIC.

1.00.05 A 'BETTER MOUSETRAP'

PL/9 provides a solution to the problem of the edit-compile-load cycle by incorporating an editor into the compiler itself, making it unnecessary to exit the editor and load a separate compiler just to see if there are any syntax errors. This interactive approach is not new; one of the first assemblers for the 6800 was Motorola's CO-RES, which also included an editor and was very popular among small computer users. The inclusion of the editor does require a different approach to designing the compiler; IT IS NOW ESSENTIAL TO CONSERVE MEMORY since the source file is taking up a considerable amount of space. PL/9 is very frugal in its use of memory; each of its 18 symbol tables takes up only as much space as it actually needs (compare this with systems that use hash coding) and the source file is stored in a compacted form that allows the free use of spaces to improve readability.

PL/9 runs to just under 16K bytes leaving you 32K for your source and binary files. In addition a separate trace-debug facility, which loads into the FLEX transient command area, is provided to simplify the program testing cycle. At a command, PL/9 compiles the program into memory but puts extra information into the object code that allows it to retain control over the program while it is being tested. This means that the program can be stopped at any point or run one instruction at a time. Variables can be examined while the program is stopped or running, allowing the programmer the convenience of BASIC. When the program is working satisfactorily, it can be compiled to object code for use in ROM or RAM.

1.00.05 A 'BETTER MOUSETRAP' continued

No understanding of 6809 assembly language programming is needed, although such understanding will help when interfacing to the system hardware.

PL/9 does not require a run-time package for its programs to work, and there are no license fees to pay for the use of compiled programs. All programs compiled by PL/9 can be placed in ROM and will run at any address in memory WITHOUT recompiling them. No memory addresses are reserved by PL/9 for special purposes, so you have complete flexibility.

Library routines are available to perform functions not provided directly by the language. Due to the architecture of PL/9 the users library functions actually appear to be an integral part of the language.

1.00.06 COMPATIBILITY WITH OTHER TEXT EDITORS

Even though PL/9 has its own built in editor there is nothing in its architecture that prevents you from using any editor that produces TSC EDITOR compatible files. Obviously this includes the TSC TEXT EDITOR, but also includes many cursor oriented screen editors such as Alford and Associates SCREDITOR III.

W A R N I N G

IF YOU USE AN EDITOR OTHER THAN THE EDITOR WITHIN PL/9 TO GENERATE YOUR PROGRAMS ENSURE THAT THE MAXIMUM LINE LENGTH DOES NOT EXCEED 127 CHARACTERS. EXCEEDING THIS LINE LENGTH WILL CAUSE THE LINE TO BE TRUNCATED AS IT IS LOADED.

If this truncation takes place the line will have four percent symbols (%%%%) appended to the end of the line as an aid in locating the line. In addition an error message 'LINE TOO LONG' will be posted as the file is loaded.

Many programmers prefer to use the editor they are most familiar with to do the main program entry. Once the vast majority of the program is entered and you enter the 'debugging' or 'fine tuning' stages of your program development work you will find the co-resident editor to be WORTH ITS WEIGHT IN GOLD if you value your time OR wish to keep your blood pressure down!

1.00.07 VARIATIONS BETWEEN VERSIONS

PL/9 VERSION 2.XX January 1982. Initial Release

PL/9 VERSION 3.XX June 1983. Tracer separated from compiler, REALS added to language pointer structures enhanced, upper/lower case supported, 'XREG' pseudo register added, SWI, SWI2, SWI3 and RESET procedures added, improved handling of unsigned bytes and integers, ability to call compiler from FLEX command line added. Improved REAL and INTEGER arithmetic. Manual improved.

PL/9 VERSION 4.XX June 1984. Error messages separated from compiler, STACK manipulation enhanced, subscripted vector code generation improved, compile time options enhanced. More libraries and sample programs provided. Manual expanded.

2.00.00 PRODUCT OVERVIEW

PL/9 is a complete co-resident Editor/Compiler/Trace Debugger for the 6809.

PL/9 features an editor, identical to the one in 'MACE', that is very quick to learn and easy to use. It loads and saves files, finds and changes strings, appends comments, inserts and deletes lines, prints selected lines on the terminal or printer, passes commands to 'FLEX' and calls the co-resident single pass Compiler or Debugger.

PL/9 is a TRUE COMPILER that produces PURE 6809 machine code. PL/9 does not require a run time interpreter, with its associated loss of speed and license costs (as do most BASICs and PASCALs) ...NOR... does PL/9 impose any license fee or restrictions in regard to its MATH module (as does TSC's Native Code Pascal). The code PL/9 produces belongs to you, and you alone...a valuable consideration if you are writing programs to sell or integrate into systems.

PL/9's Trace Debugger allows you to single step or breakpoint a PL/9 program a source line at a time examining variables as you go.

PL/9 is a structured language loosely based upon the control structures found in BASIC, PASCAL, PL/M and 'C' but omitting the exotic data types and type checking. PL/9 has been specifically developed for dedicated control applications in a microcomputer environment. The language is designed to be a step up from assembly language. It retains most of the flexibility and speed of the latter but makes programs, particularly those with structured control arguments, shorter and more readable. PL/9 is largely self documenting owing to its ability to support variable names of up to 127 characters in length.

Functions not supported directly by the PL/9 compiler, such as disk drivers or I/O routines can easily be 'INCLUDEd' in PL/9 programs thus allowing the user to generate new KEY WORDS to suit his own particular requirements; a number of such functions (library modules) are included with the PL/9 compiler.

PL/9 makes extensive use of the STACK for temporary variable storage thus making all PL/9 modules position independent and ROMable. Variables may also be located at fixed positions in memory to facilitate interface with hardware, programs written in other languages such as PASCAL or BASIC, and programs written by several programmers, each with his own allocated variable storage area for parameter passing/scratch.

PL/9 recognizes three distinct data sizes: BYTE (8-bit), INTEGER (16-bit), and REAL (32-bit...8-bit exponent and 24-bit mantissa) floating point accurate to six or seven decimal digits. BYTE and INTEGER values may be signed (twos complement) or unsigned (ones complement). The unsigned BYTE and INTEGER variables have been provided to facilitate the bitwise operations and comparisons that are common in digital I/O. The data types available are:

1. BYTE (signed) in the range of -128 to +127.

2. BYTE (unsigned) in the range of 0 to +255.

3. INTEGER (signed) in the range of -32768 to +32767.

4. INTEGER (unsigned) in the range of 0 to +65,535.

5. REAL (floating point) in the range of +/-1 E-38 to +/-1 E+38.

2.00.00 PRODUCT OVERVIEW (continued)

Data assignment operators are: 'GLOBAL' (are allocated permanent space on the stack and known to all procedures, 'AT' (a fixed absolute address), and 'CONSTANT' (allows you to equate commonly used hex or decimal values to improve readability). An implicit 'LOCAL' assignment operator allows temporary (scratch-pad) variables to be defined for and known by one or more procedures (these variables are not allocated permanent space on the stack).

'ORIGIN' and 'STACK' statements allow the programmer to specify where in memory his object code (on a procedure by procedure basis) and global variables are to be located.

'DPAGE' statements allow the programmer to pre-set the MC6809 direct page register in order to improve the code efficiency of accessing AT variables.

Mathematical expressions: (+), (-), (*), (/), negation (-), modulus (\).

Bitwise operators: (AND), (OR), (EOR), (XOR), (NOT), (SHIFT), (SWAP).

Logical operators: (.AND), (.OR), (.EOR), (.XOR).

Relational operators: (=), (<>), (>=), (<=), (>), (<).

Functions: (SQR), (INT), (INTEGER), (BYTE), (FLOAT).

Address pointers are supported in the forms: '.VARIABLE', '.VARIABLE(COUNT)', '.VARIABLE(COUNT*2)', etc. ('&' may be used in lieu of '.' if desired)

Control statements are: 'IF...THEN...ELSE', 'IF...CASE1...CASE2.(etc)...ELSE', 'BEGIN...END', 'WHILE', 'REPEAT...UNTIL', 'REPEAT...FOREVER', 'CALL', & 'JUMP'.

Control statement terminators supported are: 'RETURN', 'RETURN <condition>', 'BREAK' and 'GOTO'

The 'ASMPROC' and 'GEN' statements (or special files produced by 'MACE') may be used to insert machine code inside of PL/9 procedures to obtain special functions, such as indirect addressing e.g. (JSR [D3E5]) would be coded as: 'GEN $AD,$9F,$D3,$E5;'

Some of the more powerful aspects of PL/9 include direct access to accumulators 'A', 'B', 'D', 'X', 'CCR' (condition code register) and 'STACK'. These pseudo variables may be contained as part of an evaluation or assignment construction and greatly improve and simplify communication with external assembly language procedures.

PL/9 also posseses the ability to intercept the systems RAM (or the MC6809 hardware vectors at $FFF2 - $FFFF) vectors for: SWI, SWI1, SWI2, NMI, FIRQ, and IRQ INTERRUPTS. The MC6809 RESET vector at $FFFE/F can also be intercepted. Thus allowing PL/9 procedures to start the entire system from power-up, service interrupts, etc., WITHOUT RECOURSE TO ASSEMBLY LANGUAGE PROGRAMS.

2.00.00 PRODUCT OVERVIEW (continued)

PL/9 LIBRARY MODULES

Since PL/9 has been designed to produce code for TARGET hardware we have made no assumptions about the I/O environment or memory map of the system the code is to run in. By default this means that we cannot include I/O facilities in the compiler itself. For example if we built a PRINT statement into the compiler we whould also have to build in an OUTCHAR routine. The OUTCHAR routine would HAVE to be configured for a specific hardware environment which would make the program 100% dependant on being run in a particular hardware configuration.

This is clearly undesirable if you are writing programs for a variety of hardware environments. To cater for this programming requirement PL/9 is supplied with a set of I/O libraries configured for the FLEX environment. When these library modules are INCLUDEd in the users programs the routines then APPEAR TO BE AN INTEGRAL PART OF THE LANGUAGE. The beauty of this arrangement is that you can have a library module, say IOSUBS.LIB, configured for the FLEX environement and another library module, say TARGETIO.LIB, with procedures of identical names and functions, configured to match the target hardware environment. Thus the users program may be tested within the development system by simply substituting IOSUBS.LIB whenever the program is compiled. Once the program is debugged you revert to TARGETIO.LIB and compile the program for the target hardware enviroment with the knowledge that 95% of the program is debugged. This 'dual library' approach can also be extended to cater for differences in memory maps between the development system environment and the target hardware environment.

The libraries supplied with PL/9 are as follows:

IOSUBS.LIB This library module has a series of PL/9 procedures to simplify communication via the system console. Functions provided by this module include: (MONITOR), (GETCHAR), (GETCHAR_NOECHO), (GETKEY), (CONVERT_LC), (GET_UC), (GET_UC_NOECHO), (PUTCHAR), (PRINTINT), (INPUT text into a buffer), (CRLF), (PRINT string) and (SPACE).

TERMSUBS.LIB This library module has a series of PL/9 procedures to handle 'intelligent' terminals. As supplied the procedures are configured for a SOROC IQ-120/LEAR SIEGLER ADM3 or ADM5 but they can be quite easily modified for other terminals. (NULLS), (ERASE_EOL), (ERASE_EOP), (CURSOR_COL/ROW), (HOME), (HOME_N_CLEAR), (ATTR_ON) and (ATTR_OFF);

HEXIO.LIB This library module has a series of PL/9 procedures to simplify console I/O with hexadecimal numbers: (GET_HEX_NIBBLE), (GET_HEX_BYTE), (GET_HEX_ADDRESS), (PUT_HEX_NIBBLE), (PUT_HEX_BYTE) and (PUT_HEX_ADDRESS).

BITIO.LIB This module provides two routines for simulating bit oriented I/O via the system console: (BITSIN) and (BITSOUT). It also includes three bit oriented input routines: (BITIN), (BITZ8IN) and (BITZ16IN), and three bit oriented output routines: (BITOUT), (BITZ8OUT), and (BITZ16OUT). These last six routines greatly simplify bit oriented I/O as they completely remove the requirement to perform bitwise AND/OR operations.

2.00.00 PRODUCT OVERVIEW (continued)

HARDIO.LIB This module contains four routines that BASIC programmers should recognize: (PEEK), (DPEEK), (POKE) and (DPOKE). These functions greatly simplify working with absolute memory addresses being referenced by a pointer.

STRSUBS.LIB This library module has a series of PL/9 procedures to simplify string handling. Functions provided by this module include: (STRLEN), (STRCOPY), (STRCAT), (STRCMP) and (STRPOS).

BASTRING.LIB This library contains the familiar string manipulators found in BASIC: (LEFT), (MID) and (RIGHT).

FLEX.LIB This library module has a series of 'ASMPROC' (pseudo assembly language procedures) to simplify the interface with the 'FLEX' disk operating system. Functions provided by this module include: (FLEX), (GET_FILENAME), (SET_EXTENSION), (REPORT_ERROR), (OPEN_FOR_READ), (READ), (OPEN_FOR_WRITE), (WRITE), (SET_BINARY), (CLOSE_FILE), (RENAME_FILE), (DELETE_FILE), (READ_SECTOR) and (WRITE_SECTOR).

SCIPACK.LIB This library module contains various scientific functions to an accuracy of at least 5 decimal digits. Functions provided include: (LN natural log), (LOG base 10 log), (ALOG), (XtoY), (SIN), (COS), (TAN) and (ATN).

REALCON.LIB This library module comprises two floating point conversion routines: 'BINARY' (converts an ASCII string into a REAL number) and 'ASCII' (converts a REAL number into an ASCII string).

REALIO.LIB This library module contains two routines that facilitate console I/O with REAL numbers: (FINPUT) and (FPRINT).

NUMCON.LIB This library module contains four routines that facilitate console I/O with INTEGER numbers: (BINTODEC), (PRDEC), (PRNUM) and (GETNUM).

SORT.LIB This routine contains the 'primitive' required to implement sorting programs. It is similar to the the sort function provided in most 'C' compiler libraries: (SHELLSORT).

2.00.00 PRODUCT OVERVIEW (continued)

FLOATING POINT MATHS MODULE

In large multi-module systems or where several programmers may be involved in the development of a large software package it is often desirable to write, compile, and test program modules (particularly low-level modules) and then insert them in PROM, or at least leave them alone, for the remainder of the software development.

Under normal circumstances PL/9 will generate a MATHS MODULE for every PL/9 procedure file (or groups of procedure files compiled by 'spooling'). Where a large number of PL/9 procedure files are involved 'spooling' can prove to be very time consuming.

A unique capability of PL/9 is the ability to install the MATHS module in a fixed location in memory to be shared by all PL/9 procedures. The relocatable MATHS module produced by PL/9 is just over 1K in size. All entries into the module are made via a jump table to ensure that future additions to the module will be compatible with the current version.

The routines used in the MATHS module supports the 'REAL' numbers in the range of +/-1 E37 to +/- 1 E-38 with 6 or 7 significant (decimal) digits. A summary of the speeds (2 MHz 6809) are as follows:

OPERATION	REALS	INTEGERS
ADD	590 uSec	10 uSec
SUBTRACT	610 uSec	10 uSec
MULTIPLY	450 uSec	80 uSec
DIVIDE	1.31 mSec	450 uSec
SQ ROOT	2.91 mSec	---
SINE	8.41 mSec	---
COSINE	7.91 mSec	---
NATURAL LOG	7.41 mSec	---
X TO Y POWER	23.41 mSec	---

The REALCON.LIB conversion routines allows input, from external sources, in a wide range of formats, e.g. 1234, .052, .531, 3.6E12 -8.4316E-4, etc.

If the answer is >= 1,000,000 or < 0.01 the response will be in scientific notation. If the answer falls between these limits the response will be in decimal with all trailing zeros truncated (100.0000 will be 100).

2.00.01 BENCHMARK PERFORMANCE FIGURES

The figures for products other than PL/9 have been taken from published figures either in 'BYTE' magazine or '68 Micro Journal'. We assume no liability for their accuracy nor do we assume any liability for any omissions. To the best of our knowledge all other products were tested in a 2 MHz system.

Even though PL/9 places well in the benchmark tests these test results should not be used as the sole criteria to compare our product with the rest of the field. For example several of the other products offer number processing capabilities far in excess of PL/9's capabilities.

All tests of PL/9 have been conducted in the following hardware environment:

A Windrush 6809 Phoenix system with the 6809 processor running at 2.0 MHz, a Windrush 5/8, SD/DD, SS/DS disk controller, 5" DS, DD, 80 track, 3 Ms step YE-DATA (Qume) disk drives, I/O stretch (MRDY) enabled, 56K of STATIC RAM with a memory mapped video display emulating a Soroc IQ 120 with an effective baud rate in excess of 38K.

LANGUAGE	SIEVE OF ERATOSTHENES	PRIME NUMBER GENERATOR
ASSEMBLER	5.10	not available
IMS PASCAL (NATIVE)	8.78	not available
OS9 PASCAL (NATIVE)	not available	54
PL/9	9.4	56
INTROL 'C'	11.0	not available
TSC PASCAL (UNIFLEX)	34.0	not available
TSC PASCAL (FLEX)	54.0	59
IMS PASCAL (P-CODE)	105.00	not available
OMEGASOFT PASCAL (FLEX)	not available	72
BASIC-09 (INTERPRETER)	238.00	not available
DUGGERS 'C'	not available	74
OS9 PASCAL (P-CODE)	not available	112
DYNASOFT PASCAL (P-CODE)	309.00	not available
LUCIDATA PASCAL (P-CODE)	735.00	194
TSC XBASIC (UNIFLEX)	810.00	not available
TSC XBASIC (FLEX)	840.00	not available

3.00.00 CONFIGURING PL/9 TO YOUR SYSTEM HARDWARE ENVIRONMENT

This section will provide details of how to install PL/9 in your system.

(1) The first thing you should do upon receipt of this software is to complete and return the registration form that accompanies it. This is very important as it is through registration forms (not sales records) that we keep you informed of any upgrades that are available for the product. We will only support one user per copy sold. If you fail to register your copy with us and then phone for technical support we have no way of knowing whether you are a legitimate user or someone who has come by this product by dishonest means. We will not answer any questions until we have a signed registration form in our files. DO IT NOW!

(2) Remember that this product is licensed for use by a single user on a single computer system. If any of your friends or associates within the same organization wants a copy contact the factory for details of our low cost 'KLONING' service whereby we create klone copies of your disk and manual. Each copy may be registered to a different individual and EACH user will be eligible for the same upgrades, support service, etc. as the original purchaser. THIS IS A LOW COST SERVICE DESIGNED TO KEEP YOU HONEST!

(3) The next thing you should do is to format a fresh disk in the system you will be using PL/9 with. You should then copy ALL of the files from the disk we have supplied you to this new disk. The original disk we supplied should then be put in a safe place as we will require that you return it to us if you want to upgrade your copy of PL/9 at a later date.

(4) The next thing you should do is to copy the following files, which are the files associated with the compiler itself, from the working copy of the PL/9 disk onto your SYSTEM disk, i.e. the disk that normally resides in drive #0. These files MUST be on the drive defined as the SYSTEM drive (as defined by the FLEX 'ASN' command) for the compiler to operate properly

a. PL9 .CMD the compiler
b. PL9_TD .CMD the tracer
c. PL9 .ERR the compiler error message file
d. SETPL9 .CMD the PL/9 configuration program

(5) If you intend to use any of the library files we supply and your SYSTEM disk has adequate spare capacity we suggest that you also copy the following files on to your system disk. These files are not required for the compiler to operate unless one of your PL/9 source files 'INCLUDEs' them. A selection of these files can also be part of your work disk if you desire.

a. TRUFALSE.DEF
b. HEXGLOBL.DEF
c. IOSUBS .LIB
d. TERMSUBS.LIB
e. HEXIO .LIB
f. BITIO .LIB
g. HARDIO .LIB
h. STRSUBS .LIB
i. BASTRING.LIB
j. FLEX .LIB
k. SCIPACK .LIB
l. REALCON .LIB
m. REALIO .LIB
n. NUMCON .LIB
o. SORT .LIB

3.00.00 CONFIGURING PL/9 TO YOUR SYSTEM HARDWARE ENVIRONMENT (continued)

(6) There will also be several files on the disk with the extension '.PL9'. These files are sample programs and need not be copied to your system disk. All files present on the disk and their purpose will be described in a file called 'READ-ME.TXT'. List this file for further information.

(7) The last thing you must do is to configure your copy of PL/9 to match the system you will be using it in. A special program has been provided to greatly simplify the task of configuring PL/9 to your FLEX system and it's terminal. The program is called 'SETPL9.CMD' and it runs like this:

```
+++SETPL9<CR>

*****   PL/9 Configuration Program    *****
        ==========================

      For use with PL/9 version 4.XX

This program allows you to configure PL/9
to your own particular requirements and
those of your computer system.

Some of the questions do not need answers
unless you wish to alter the current
settings. In these cases the existing data
will be displayed. To leave the data <---------- note!
alone just hit <CR>.

        PUT YOUR PL/9 DISK IN DRIVE
        ZERO THEN HIT ANY KEY.....      <---------- hit <CR>


            INPUT WITHOUT ECHO
            ==================

PL/9 requires your keyboard input routine
to return the ASCII value of the key
pressed, in the A-accumulator, WITHOUT the
character being echoed. Early versions of
FLEX9 did not support this feature, so you
should first check whether you have an
"INCHNE" vector (i.e. address of an input-
without-echo routine) at $D3E5. If you need
to exit SETPL9 to examine your system, just
type <CR>.

Does your system have INCHNE at $D3E5? Y    <---------- 'Y' for most systems.
```

If you answer this question 'Y' the prompt just after '***' on the following page will appear.

If you have an early FLEX 9 system produced by SWTP (et al) the INCHNE vector at $D3E5 will not be present. If this is the case answer this question 'N' and the following options will be available.

3.00.00 CONFIGURING PL/9 TO YOUR SYSTEM HARDWARE ENVIRONMENT (continued)

There are three ways to implement input without echo:

1) You can give me the address of a routine in your system that performs input without echo;

2) You can give me the address of a vector that points to your input routine (NOT the address of the routine itself);

3) You can give me the address of your input device (6850 ACIA, 6821 PIA etc.) at which the ASCII key code can be found. It is essential that the act of reading the character clears the "character waiting" flag;

4) You can exit and think about it.

Which do you want to do (1-4)? 2 <---------- if you have an 'SBUG-E' compatible system monitor.

Address of your input vector: $F804 <---------- 'INCHNE' vector

NOTE: If you are not sure of the address you are supplying OPEN THE DISK DRIVE DOORS as an invalid address will cause the system to CRASH!

***

Just to check, type the number "1": 1 <---------- type '1'

I got a "1"!! Did it echo on the screen? N <---------- it should be 'N'

NOTE: If any part of this check fails SETPL9 will return to FLEX.

That's the difficult part over. Now for the easy bits!

KEYBOARD

Next let's set up PL/9 for your keyboard. Each question should be answered with a single keypress or control key combination, e.g. (control H) for backspace:

First press your backspace key..... <---------- CTRL H if in doubt
Now your forward tab key..... <---------- CTRL I if in doubt
Next your line cancel key..... <---------- CTRL X if in doubt
Now your escape key..... <---------- CTRL [if in doubt
And lastly Control C or equivalent... <---------- CTRL C if in doubt

3.00.00 CONFIGURING PL/9 TO YOUR SYSTEM HARDWARE ENVIRONMENT (continued)

PRINTER

Now to set up PL/9 for your printer.

How many listing lines are to be printed on each page (leave some room for top and bottom margins)?
..... (currently 55)? 55 <---------- number then <CR>

Total length (in lines) of each sheet?
..... (currently 66)? 66 <---------- number then <CR>

Does your printer support form feed? Y <---------- Y or N (<CR> = Y)
What HEX character is it?.....$0C <---------- $0C<CR> for most

NOTE: PL/9 obeys the TTYSET 'WD' parameter when it outputs to the system terminal AND the system printer. If you have a 132 column printer and wish to prevent PL/9 from truncating the source lines simply set WD=132 before using PL/9.

STRINGS

In PL/9 as supplied, strings are terminated with nulls, and the library routines are set up to recognise this character. You can change the null to anything else you like (e.g. $04) but many of the library routines will then not work unless modified.

.....Is the null terminator OK? Y <---------- 'Y' for most users.

What HEX character do you want?.....$04 <---------- You get this question if you answer the one above 'N'. $04 is the end of transmission character recognized by most system monitors and FLEX

3.00.00 CONFIGURING PL/9 TO YOUR SYSTEM HARDWARE ENVIRONMENT (continued)

DISK FILES

Now I want to know the default filename extensions and default drive numbers: <--------- hit <CR> to accept existing defaults.

LOAD and SAVE file extension.(currently PL9): <--------- new extension <CR>
and its default drive number..(currently #1): <--------- new drive number

OBJECT (A:O) file extension..(currently BIN): <--------- new extension <CR>
and its default drive number..(currently #1): <--------- new drive number

INCLUDE file extension.....(currently LIB): <--------- new extension <CR>
and its default drive number..(currently #0): <--------- new drive number

LISTING (A:L) file extension.(currently OUT): <--------- new extension <CR>
and its default drive number..(currently #1): <--------- new drive number

READ and WRITE use a file called: "1.SCRATCH.SCR".
.....is this O.K.? <--------- Y or N (<CR> = Y)

If you answer the above prompt 'N' you will be asked the following:

R/W scratch file name?...(currently SCRATCH): <--------- new name <CR>
R/W scratch file extension?..(currently SCR): <--------- new extension <CR>
and its default drive number..(currently #1): <--------- new drive number

SYSTEM INTERRUPT VECTORS

Lastly, I need to know where your interrupt vectors are. These are the RAM vectors used by your system monitor.

RESET (ROM=$FFFE) vector (fixed at: $FFFE)

NMI (ROM=$FFFC) vector (currently: $E740): <--------- 4 HEX digits <CR>
SWI (ROM=$FFFA) vector (currently: $E746): <--------- 4 HEX digits <CR>
IRQ (ROM=$FFF8) vector (currently: $E744): <--------- 4 HEX digits <CR>
FIRQ (ROM=$FFF6) vector (currently: $E742): <--------- 4 HEX digits <CR>
SWI2 (ROM=$FFF4) vector (currently: $E748): <--------- 4 HEX digits <CR>
SWI3 (ROM=$FFF2) vector (currently: $E74A): <--------- 4 HEX digits <CR>

Your copy of PL/9 is now configured!

+++

3.00.00 CONFIGURING PL/9 TO YOUR SYSTEM HARDWARE ENVIRONMENT continued

The hex addresses supplied as the default RAM vectors are compatible with all versions of the Windrush MC6809 system monitor 'GT-BUG9'.

If your system uses a GIMIX GMXBUG, Version 2, or an SSB MON-69D, or an SWTP SBUG-E the following interrupt vectors should be correct. As we have no control over what alterations these manufacturers make to their products this information is supplied for GUIDANCE only. We do not assume any responsibility for its accuracy. Consult the documentation supplied with your system to be on the safe side.

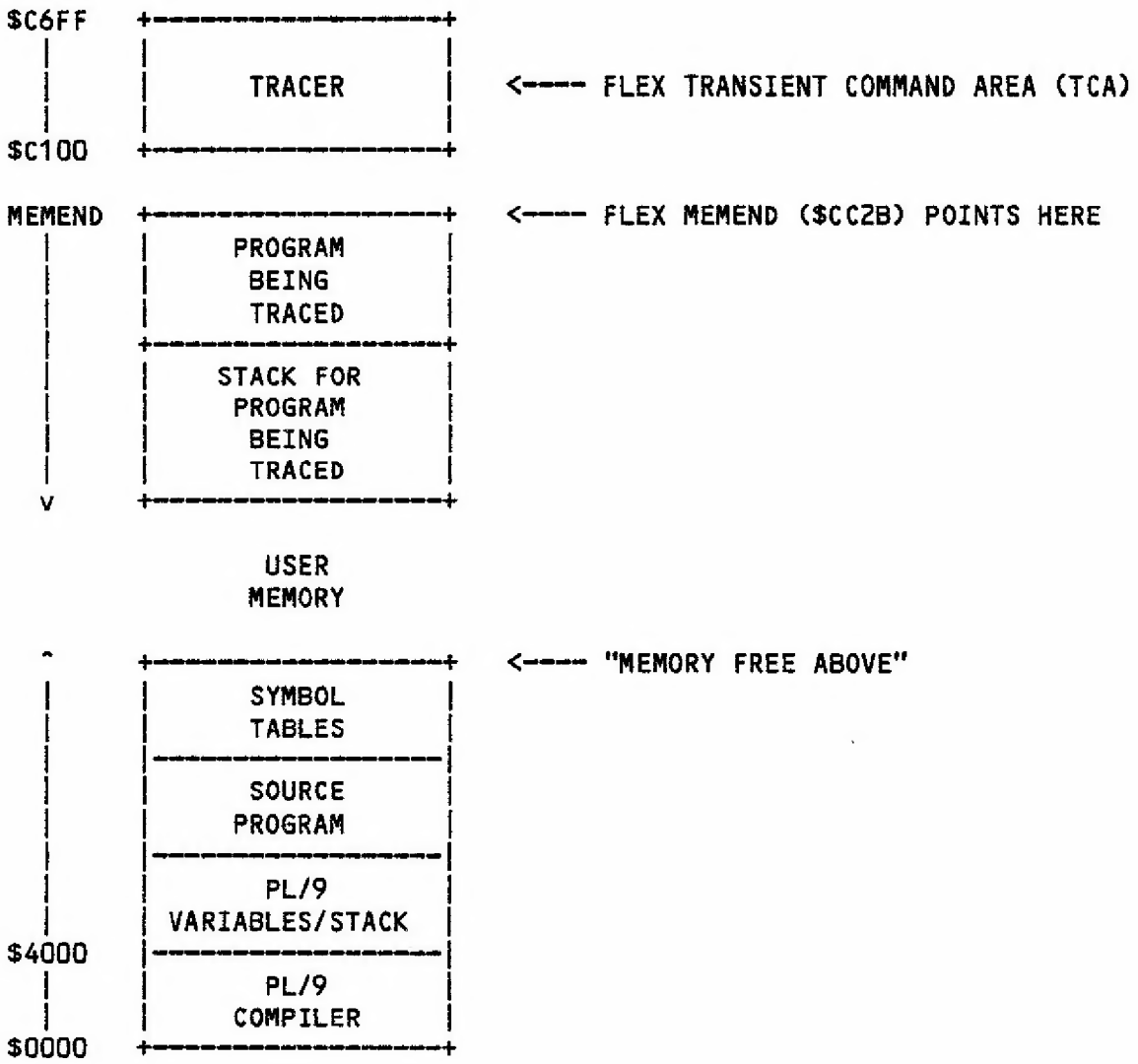
VECTOR	GIMIX	SSB	SWTP
NMI	$E408	$F3C0	?????
SWI	$DFCA	$F3CA	$DFCA
IRQ	$DFC8	$F3C8	$DFC8
FIRQ	$DFC6	$F3C6	$DFC6
SWI2	$DFC4	$F3C4	$DFC4
SWI3	$DFC2	$F3C2	$DFC2

N O T E

If you compile a file using the 'R' option (see Compiler Reference) the MC6809 hardware interrupt vectors at $FFF2 through $FFFF will be substituted for the RAM vectors specified.

3.01.00 MEMORY MAP

PL/9 uses memory as follows:



PLEASE READ THIS

From this memory map it should not be difficult to ascertain that a program being run under the tracer cannot have a massive amount of variable storage ALLOCATED on the stack NOR can the program USE a massive amount of stack. For example the statement 'GLOBAL BYTE BUFFER(32000);' would cause the program to relocate the stack right into the middle of of your source program and clobber it the moment the program was run.

It is up to the programmer to ensure that there is adequate space on the stack before invoking the tracer. You should remember also that although the tracer runs in the FLEX TCA it also makes use of some of the facilities in the compiler which resides between $0000 and $3FFF. If your program writes into this area (or relocates the stack into this area) the result will undoubtedly be a system crash.

3.02.00 PL/9 AND THE FLEX ENVIRONMENT

PL/9 makes certain assumptions about the FLEX environment it is running in, and assumes that you have configured it for your system hardware. Generally PL/9 and its associated library modules should be placed on your SYSTEM disk (drive 0), duplicate (or modified) copies can be maintained on your software development work disk if desired as this will reduce the number of alternate disk accesses during compilation and the attendant 'head banging'.

The TTYSET 'PS' (pause) should be turned off (+++TTYSET,PS=NO<CR>). Leaving the TTYSET pause function enabled can cause annoying stops when listing lines in the PL/9 editor and strange stoppages during program execution while in the tracer.

The TTYSET 'WD' (width) should be set to the width of your printer or terminal which ever is the greater. If, for example, you leave the WD value set to 80 columns PL/9 will truncate all terminal (A:T) printer (A:P) and listing (A:L) output at 80 columns. If you have a 132 column printer we suggest that you set WD=132 and leave it there. The TTYSET WD parameter is not normally used by FLEX unless some program exceeds the width of the system console at which time FLEX will generate a CR-LF.

3.02.01 PL/9 AND FLEX PRINTER DRIVERS

A set of TSC standard printer drivers should be loaded into memory at $CCC0 - $CCFF (e.g. +++GET,PRINT.SYS<CR>). If you don't know what we mean by 'standard' see your FLEX User's Manual for details of 'standard' serial and parallel printer drivers. If you have ANY problems with your printer and PL/9 it is almost certainly going to be due to non-standard printer drivers.

The printer drivers will be accessed via the standard entry points (INIT=$CCC0 and POUT=$CCE4), if your printer drivers reside elsewhere these locations should point to your routines via an extended jump (7E XXXX). The printer driver routines must preserve the registers specified by TSC.

A problem arises for users of SouthWest Technical Products' FLEX, which has a different way of handling printer drivers to that used by most other versions of FLEX. SWTP FLEX requires the programmer to use the "P" command to cause output to be sent to the printer. This method will not work satisfactorily with PL/9. The recommended action is either to generate a printer driver that PL/9 can use (see above) or to direct output to a disc file for later printing.

3.02.02 PL/9 AND FLEX ITSELF

It is important to note that PL/9 expects, no demands, that your system have FLEX implemented properly. This means that the INCHNE routine (pointed to by the address held in $D3E5) must strip the parity bit (the most significant bit) of the incoming character as specified by TSC and that all routines within FLEX preserve the registers as specified by TSC.

We have tested this program on FLEX from SSB for their DCB-4A controller, FLEX from GIMIX for their DD-5 disk controller and DMA disk controller. It has also been tested on all Windrush FLEX systems. Therefore we are satisfied that the program will work 'as advertised' if your system has had FLEX implemented in accordance with TSC's standards.

3.03.00 JUST TO PROVE THAT IT WORKS

This section is a quick exercise just to prove to you that the product works. Before starting this exercise you must have the following files on the disk in drive #0.

a. PL9 .CMD
b. PL9_TD .CMD
c. PL9 .ERR
d. IOSUBS .LIB

First we will call PL9 off the system disk so that we can use it in the interactive mode:

```
+++PL9 <CR>
```

PL9 will greet you with its startup banner:

```
PL/9 Compiler, Version X.XX
Serial #XXXX
    #
```

The hash symbol '#' signifies that the EDITOR is in command mode and waiting for a command. First we must instruct the editor to enter the 'INSERT LINE' mode of the editor so we can type in our test program. We do this with the 'I' command thus:

```
    #I<CR>        ..... type 'I' followed by <CR>
```

PL9 will then respond with a '+' to tell you that you are in the 'INSERT' mode. We will now enter the text of the program. You can use 'BACK SPACE' to correct any errors on the line before you type <CR>. If you hit <CR> before you spot the error you will have to fix it later using the 'O' (OVERLAY) command or delete the line and re-enter it. To use these latter facilities of the EDITOR you will have to read section four.

```
    +ORIGIN=$C100; /* PUT IN FLEX TCA */<CR>
    +<CR>
    +INCLUDE 0.IOSUBS.LIB;<CR>
    +<CR>
    +PROCEDURE CONFIDENCE;<CR>
    +   PRINT "\n\nHELLO WORLD\n";<CR>
    +<ESCAPE>
```

Hitting <ESCAPE> when the cursor is adjacent to the '+' symbol means you wish to exit the INSERT LINE mode and return to editor command mode. PL9 signifies this mode by the appearance of the hash symbol '#' once again. Now just to make sure that we have typed the program in correctly lets list it. We do this with the command '1P22'. This command means 'go to line number 1 and print 22 lines on the system console':

```
    #1P22<CR>
0001 ORIGIN=$C100; /* PUT IN FLEX TCA */
0002
0003 INCLUDE 0.IOSUBS.LIB;
0004
0005 PROCEDURE CONFIDENCE;
0006    PRINT "\n\nHELLO WORLD\n";
0007 /EOF
    #
```

3.03.00 JUST TO PROVE THAT IT WORKS (continued)

NOTE: The '\n' sequence used three times in the above string means NEW LINE. The 'PRINT' routine (in IOSUBS.LIB) will interpret this sequence and generate a carriage-return followed by a line feed.

Now we should save a copy of the source file on disk for later use. We do this by typing:

```
#S=0.HELLO.PL9<CR>
```

This command will cause a disk access to drive #0 as the compiler saves the source file to a disk file called 'HELLO.PL9'.

Next we should compile the file so that we can use it. We do this by typing:

```
#A:0=0.HELLO.CMD<CR>

Memory Free Above $XXXX
Last PC Value was $XXXX
```

This command will cause a disk access to drive #0 as the compiler first reads and compiles the library file 'IOSUBS.LIB' and then compiles the source file resident in memory to a disk file called 'HELLO.CMD'. PL/9 will automatically generate the return to FLEX (JMP $CD03) at the end of the program as we have left off the last 'ENDPROC'. PL/9 will also automatically generate the transfer address of the program. The mechanisms that control these actions of the compiler will be discussed in subsequent sections of this manual.

If any errors are detected by the compiler they will be reported and the compiler will stop at the line containing the error. If you hit <CR> the compiler will abort the compilation and return to command mode POINTING to the line where the error occurred. If you hit <SPACE> the compiler will continue until either it completes the compilation or encounters another error.

If the program compiled without errors we are now ready to return to FLEX. We do this by typing:

```
    #X<CR>
IS TEXT SECURE? Y <------- type 'Y'

+++
```

Now we can invoke the program we just compiled by typing:

```
+++HELLO<CR>
```

The program will load and then print:

```
HELLO WORLD

+++
```

Well done! You have just written and compiled your first program with PL/9.

Now you might want to try a couple of the examples in the next section which demonstrate some of the other features of PL/9.

3.03.01 OTHER MODES OF OPERATION

PL/9 has two distinct modes of operation. The first is calling it from the FLEX command line and giving it the name of the input file, the output file(s), and the compile option(s). The second is the interactive mode where PL/9 is called as a FLEX command, i.e. (+++PL9<CR>) and used in much the same manner as BASIC. This latter mode was the mode we used in the previous section.

Whatever mode of operation you use PL/9 will load into memory and start up. A banner will appear, announcing the name of the compiler, the version number and the serial number of your copy. Any questions to your supplier concerning the product should quote both of these numbers.

When using PL/9 in the inter-active mode a hash (#) symbol will appear on the next line, indented by four spaces. This is the prompt that signifies the editors readiness to accept commands. The editor is where you will always be if you are not actually testing a program using the tracer. Commands are available to allow you to create or alter a source file, to load from or save to floppy disc, to pass commands to FLEX and to tell the compiler part of the program to do its job. The reference sections contain a full description of each of the editor commands, compiler commands, and tracer commands. It is to these sections that you should go to gain familiarity with this part of the system.

PL/9 can alternatively be invoked from FLEX and instructed to load a file, compile it and return to FLEX, much in the same way as other compilers. Suppose that you have a program called HELLO.PL9 on your working drive, that you wish to compile to generate a command file HELLO.CMD on the system drive, and that you also wish to have a listing, complete with machine code, sent to your printer. The command line that would achieve all this is as follows:

```
+++PL9,0.HELLO.PL9,0=0.HELLO.CMD,P,C<CR>
```

Note that in order for this particular example to work, a suitable printer driver must have been installed in FLEX before the command is given.

You can also instruct PL/9 to do the same job but instead of directing the listing output to a printer the listing may be directed to a disk file for later use.

```
+++PL9,0.HELLO.PL9,0=0.HELLO.CMD,L=0.HELLO.OUT,C<CR>
```

PL/9 will generate a listing file on drive zero called HELLO.OUT, which can then be spooled to the printer or "P,LIST"ed.

You can also instruct PL/9 to do the same job but instead of directing the listing output to a printer the listing may be directed to the system console.

```
+++PL9,0.HELLO.PL9,0=0.HELLO.CMD,C,T<CR>
```

More information on this subject is contained in section five.

THIS PAGE INTENTIONALLY LEFT BLANK

4.00.00 THE PL/9 EDITOR

One of the strongest features of PL/9 is its built-in editor, which cuts out much of the loading and saving that makes using other compilers a time-consuming business. The editor is broadly similar to, and compatible with, the TSC text editing system, although the commands are not identical. Anyone familiar with the latter should have little difficulty in using the PL/9 editor.

The user is encouraged to make free use of spaces to improve the readability of a program, since these are stored in a compacted form and do not take up any extra room in memory or on disc.

The editor prints line numbers while listing the source file; these numbers are not part of the file, however, and are not saved on disc. It also allows editing of a line as it is being entered, by means of the following control characters:

TAB.....generates three spaces.

BACKSPACE...moves the cursor to the left destructively one place.

CANCEL.....erases the entire line.

ESCAPE.....in the left column terminates the insert session.

RETURN.....generates a new line.

The default key values supplied may not suit your terminal. The SETPL9 program, described in the previous section, provides a convenient method of altering the keycodes PL/9 recognizes to those available on your terminal.

PL/9 will accept ANY text file that is stored on disk in the TSC TEXT EDITOR FORMAT. Many cursor oriented text editors/word processors, e.g. SCREDITOR III, save text in this format.

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                                                                       *
*   IF YOU USE AN EDITOR OTHER THAN THE PL/9 RESIDENT EDITOR  YOU       *
*   MUST ENSURE THAT THE LINE LENGTH NEVER EXCEEDS 127 CHARACTERS       *
*                                                                       *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

The PL/9 editor will not allow you to enter a source line longer than 127 characters, nor will it allow you to use the 'Z' command to append two lines that will create a line longer than 127 characters, nor will it allow you to use the 'C' command to change a string that will create a line longer than 127 characters.

If you have created a source file in another editor that contains lines with more than 127 characters you will be greeted with the message 'LINE TOO LONG' whenever you use the 'L' (LOAD) or 'R' (READ) commands of the editor. The offending line will be truncated to 127 characters with the last four characters in the line set to "%%%%". The four percent symbols present in the offending line(s) will enable you to use the 'F' (FIND) command to locate the lines.

4.00.00 THE PL/9 EDITOR (continued)

If you use an editor other than the TSC TEXT EDITOR or SCREDITOR III to enter your programs and have problems with PL/9's editor don't blame us! The fault lies with the file format produced by your editor. The STYLOGRAPH disk file format is typical example of a non-standard format that will be absolutely useless to PL/9 unless you are very careful when you are using it, i.e. ensure that each and every line terminates in a carriage return!

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                                                                           *
*  All PL/9 PROGRAMS may be entered in UPPER-CASE or lower-case letters.    *
*                                                                           *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

There are two programs presented in the 'USERS GUIDE' (section ten) that will convert a text file from lower-case to UPPER-CASE and vice versa but will leave the letters inside of strings and comments alone.

4.00.01 DETAILED DESCRIPTION OF EDITOR COMMANDS

Each of the editor commands is described fully in the following paragraphs. This section groups the various editor commands by function. A command summary can also be found in a separate document.

Generally speaking the PL/9 editor does not support multiple command entry. Edit commands must be entered singly i.e. the command followed by a carriage return. There are a few exceptions to this rule which will be described as they are encountered.

4.00.02 EDITOR COMMAND SYMBOLS

The following symbols will be used as part of the definition of the editing, compiler and tracer commands:

<CR> represents a carriage return.

<> symbols are used to enclose a variable.

<NUMBER> represents a decimal number such as 36 or 192, and which defaults to one if it is omitted.

<TARGET> represents the decimal number of lines specified by the command, and defaults to one if none is given.

<#TARGET> represents the decimal line number specified by the command.

[] symbols indicate that the enclosed data is optional and may be omitted, in which case a default value is usually supplied by PL/9.

The TRACE (T) command and COMPILE (A) command are called from within the editor. Each of these commands is described in their own sections.

MODE CONTROL

The following commands are only active when the (#) prompt is present.

I

INSERT lines into the file. The editor will change its prompt from (#) to (+) to remind you that you are now in the insert mode. Every line you type from now on will be added to the file immediately ABOVE the current line. This may seem strange at first if you are used to the TSC editor, but it has the advantage that having added a line to the file the current line is still the same one as before. It also enables you to insert a line above line number 1, something which is very awkward with the TSC editor. The current line and the rest of the file will be moved to make room for the new line, so the file will always be in sequence, i.e. it is automatically re-numbered each time lines are added.

When you have finished adding lines, ensure that the cursor is at the left margin and press the <ESCAPE> key. You will then be returned to the editor command level, with the prompt restored to a (#). Your current line will now have a new (larger) number because of the extra lines inserted above it.

X

EXIT to FLEX. You will be prompted 'IS TEXT SECURE?' which must be answered 'Y' to enter FLEX. Any other response will return you to the editor.

M

MONITOR. Enter the ROM System Monitor. A warning message will be posted along with instructions on how to re-enter PL/9 through the warm start address.

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                                                         *
*  NEVER RE-ENTER PL/9 AT THE COLD START ADDRESS $0000.   *
*                                                         *
*  Doing so will cause the existing file to be erased!    *
*                                                         *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

See the section on file start and file end markers at the end of this section of the manual for techniques to use in the event that you accidentally loose a file.

/<COMMAND>

Execute a FLEX command. Warning: Only use commands that reside in the Utility Command Area, or you may risk bombing PL/9, with possibly disastrous results.

The only commands we recommend are: CAT, DIR, LIST, DELETE, RENAME, ZAP, TTYSET, and ASN. Using COPY, for example, is a definite no-no!

LINE POSITIONING COMMANDS

The following commands are active only when the (#) prompt is present.

<NUMBER>[COMMAND]

Make line <NUMBER> the current line, then execute the command (optional) that follows the number.

1 or ^

Go to the first line in the file.

B or !

Go to the bottom (/EOF) of the file.

+<NUMBER>

Move down <NUMBER> lines from the current position.

-<NUMBER>

Move up <NUMBER> lines from the current position. This command may be followed by the print command, i.e. -23P23<CR> would back up 23 lines and print 23 lines.

<CR>

Display the current line.

<ESCAPE>

Hitting the escape key will cause the next line in the file to be displayed and to become the current line. This provides a convenient method of 'stepping' through your file a line at a time. If you are already at the bottom of the file then you will get /EOF printed every time.

FILE ORIENTED COMMANDS

N

NEW file. This command erases the file currently in memory to make room for a new file. PL/9 will prompt with "ARE YOU SURE?" to prevent you from inadvertently erasing your file.

The file is not actually deleted from memory when 'N' is used. The file is 'erased' by setting the end of file marker to beginning of the text buffer. See the section on file start and file end markers at the end of this section of the manual for techniques to use in the event that you accidentally loose a file.

P<TARGET>

PRINT a number of lines of the file on the terminal, starting at the current line. The last line printed becomes the current line. Examples:

#P50<CR>	Print 50 lines, starting at the current line.
#142P8<CR>	Print 8 lines, starting with line 142.
#P<CR>	Print some more lines.

In the last example, the number of lines printed will be the same as that specified by the last P command. When you start up PL/9, this number will be preset to one less than that given by the TTYSET DP count (see your FLEX manual). This facility allows you to scan your file N lines at a time, by pressing P then <CR> repeatedly.

D<TARGET> or D<#TARGET>

DELETE line(s). The first form will delete the requested NUMBER of lines from the file starting with the current line. Lines above the current line are unaffected and it does not matter if you specify a target that is beyond the bottom of the file.

The second form deletes all of the lines starting with the current line TO AND INCLUDING, the line number followed by the #

If D is typed by itself then only the current line is deleted. After deletion, the editor displays the new current line. Examples:

#D15	Delete 15 lines, starting at the current line.
#81D3	Delete 3 lines, starting at line 81. The line that was previously line 84 will become the current line, which will still be numbered 81.
#43D#72	Delete from line 43 to 72 inclusive. The line that was previously line 73 will become the current line.

CAUTION: Be very careful when deleting multiple lines as the file will automatically be renumbered after EACH line is deleted. When you wish to delete several lines simply start at the highest line number and work toward the lowest.

LINE EDITING

The following commands are active only when the (#) prompt is present.

O<CHAR>

OVERLAY the current line. This command is useful when a change has to be made near the start of a line. PL/9 displays the line in question, then prompts immediately underneath with a > symbol. You can then type in a new line containing only the characters you wish to change, in their correct positions under the displayed line. If <CHAR> is omitted, then spaces typed in the new line indicate characters that should be left alone; if <CHAR> is supplied then it becomes the character that you type to leave the corresponding one in the original as it was. The following example first shows the current line which is then overlayed twice:

```
0123    IF COUNT > 5 THEN CHAR = 'X      /* TEST CASE */
   #O
0123    IF COUNT > 5 THEN CHAR = 'X      /* TEST CASE */
   >            =                  ;
0123    IF COUNT = 5 THEN CHAR = 'X;     /* TEST CASE */

   #O-
0123    IF COUNT = 5 THEN CHAR = 'X;     /* TEST CASE */
   >-----------------------------------SPECIAL CASE */
0123    IF COUNT = 5 THEN CHAR = 'X;     /* SPECIAL CASE */
   #
```

E

EDIT the current line. This command causes PL/9 to display the line and to leave the cursor at the end, as if you had just typed it in but had not yet pressed <CR>. The line can then be altered by backspacing or by adding more text.

=<TEXT>

Delete the current line and put in its place the remainder of the command line. Example:

```
   #52=   REPEAT COUNT=COUNT-1 UNTIL COUNT=0;
   #52
0052    REPEAT COUNT=COUNT-1 UNTIL COUNT=0;
   #
```

L I N E E D I T I N G

\

SPLIT up the current line. Every semicolon in the line will be taken as a line delimiter. Text from that point on will be made into a new line, indented the same as the current line. An example should make this clear; suppose that this is the current line:

```
0095    COUNT=0; FLAG=FALSE; BUFFER_POINTER=2;
   #\
0095    COUNT=0;

   #P3
0095    COUNT=0;
0096    FLAG=FALSE;
0097    BUFFER_POINTER=2;
   #
```

Z

CONCATENATE two lines, i.e. remove the carriage return from the end of the current line. If there are leading spaces at the start of the following line then these will be reduced to a single space. This command performs the inverse function of the \ command above, restoring the line to its original state. The two commands together help the programmer to produce a neat and readable source file.

N O T E

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                                                                     *
*   IF  YOU  ATTEMPT TO USE THIS COMMAND TO CONCATENATE TWO LINES     *
*   THAT WILL CREATE A LINE THAT IS GREATER THAN  127  CHARACTERS     *
*   LONG PL/9 WILL REFUSE TO DO IT AND COMPLAINS WITH THE MESSAGE     *
*   'LINE TOO LONG'.                                                  *
*                                                                     *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

G L O B A L E D I T I N G

The following commands are active only when the (#) prompt is present.

F<NUMBER>/<STRING1>

FIND the next <NUMBER> occurrences of <STRING1>, starting with the line following the current line. If <NUMBER> is omitted it defaults to one. Any delimiter can be used in place of the / symbol. Examples:

#F20/IF	Find the next 20 occurrences of IF.
#F,/What?/	Find the next occurrence of /What?/.
#^F!/HELLO/	Find every occurrence of HELLO from the top of the file (^) but excluding the top line, to the bottom of the file (!).

C<NUMBER>/<STRING1>/<STRING2>

CHANGE the next <NUMBER> occurrences of <STRING1> into <STRING2>, starting with the current line. Only the first occurrence of <STRING1> will be changed on any one line. Again any delimiter is allowed. Examples:

#C/THIS/THAT	Change THIS into THAT in the current line.
#93C;WILE;WHILE	Change WILE in line 93 into WHILE.
#^C!/THESE/THOSE	Change every occurrence of THESE to THOSE.

NOTE 1: If more than one occurrence of <STRING1> in is present on a given line only the first occurrence will be altered by CHANGE, the second occurrence will be ignored.

NOTE 2: (F)ind and (C)hange operate by setting up two buffers, one for <STRING1> and the other for <STRING2>, every time either command is called. If an incomplete command line is typed, only the specified data will be updated. For example, F23 instructs PL/9 to find the next 23 occurrences of the previously defined <STRING1>, while C8;VAR (note the missing second delimiter) will change the next 8 occurrences of VAR into whatever <STRING2> had been previously set to.

This facility simplifies making global changes where the same string occurs more than once on a line.

NOTE 3: <STRING1> may contain one or more "?" as "don't care" characters. For example, ^F!/VAR? finds all occurrences of VAR1, VAR2, VAR3 etc.

NOTE 4: The reason that 'F' does not operate on the current line is so that you can use 'F' and 'C' in succession to selectively change strings.

D I S K F I L E H A N D L I N G

The following commands are active only when the (#) prompt is present.

Q or ?

QUERY the default filenames. PL/9 maintains a table of filenames which are used in conjunction with the LOAD (L), SAVE (S), READ (R), WRITE (W), COMPILE to OBJECT (A:O) and COMPILE to LISTING (A:L) commands. If QUERY is used after invoking PL/9 from the FLEX command line before loading any source file the defaults defined by the configuration program SETPL9 will be presented thus:

```
Present defaults are:
=====================

L/S = 1.          .PL9
R/W = 1.SCRATCH   .SCR
A:O = 1.          .BIN
A:L = 1.          .OUT
```

Now suppose that you wish to load a source file named 'TEST.PL9' that resides on drive #1. Since drive #1 is the default drive and '.PL9' is the default extension all you would have to do is issue the command:

```
L=TEST<CR>
```

If you then QUERY the default filenames you will get:

```
Present defaults are:
=====================

L/S = 1.TEST      .PL9
R/W = 1.SCRATCH   .SCR
A:O = 1.TEST      .BIN
A:L = 1.TEST      .OUT
```

Note that the filename for the L/S, A:O and the A:L commands now echo the filename just specified. This GLOBAL filename updating will occur whenever you specify a filename when using the L, S, and A:O command but will not occur when using the A:L command. The the default drive number and default extension are INDIVIDUALLY updated whenever the L, S, or A:O commands are issued.

In summary whenever you specify a filename you can give any combination of drive number, filename and extension, and PL/9 will take whatever you omit from the current default for the command you are using. When you specify a new value for any of the defaults for L, S, and A:O the existing default will be replaced with the new value. Using SAVE (S) as an example:

```
#S<CR>              Save to 1.TEST.PL9
#S=0<CR>            Save to 0.TEST.PL9
#S=.TXT.1           Save to 1.TEST.TXT (note the order)
#S=0.JUNK<CR>       Save to 0.JUNK.TXT
#S=.PL9<CR>         Save to 0.JUNK.PL9
#S<CR>              Save to 0.JUNK.PL9
```

D I S K F I L E H A N D L I N G

Q or ? (continued)

If we now used the Q command the default values would be as follows:

```
Present defaults are:
=====================

L/S = 0.JUNK     .PL9
R/W = 1.SCRATCH  .SCR
A:0 = 1.JUNK     .BIN
A:L = 1.JUNK     .OUT
```

Note that the default filename has been GLOBALLY upated even though we only used the SAVE (S) command. Note also that the default drive number and file extension have been INDIVIDUALLY updated. The same thing will happen whenever you use the LOAD (L) command or the COMPILE to OBJECT (A:0) command.

The A:L command can be used to direct listing output to an alternative drive, and alternative filename or an alternative file extension. For example:

```
A:L=0.TEMP.TXT<CR>
```

This command, does not alter any of the default values however.

The default values of the READ (R) and WRITE (W) command are 100% alterable and will be updated whenever any new information is given after either of these commands. For example:

```
#W23<>              Write to 1.SCRATCH.SCR
#W23=0<CR>          Write to 0.SCRATCH.SCR  (default drive now 0)
#W23=TEMP<CR>       Write to 0.TEMP.SCR     (default filename now TEMP)
#W23=.TMP<CR>       Write to 0.TEMP.TMP     (default extension now .TMP)

#R=1.TEST.TXT       Read file 1.TEST.TXT (the default drive, file name,
                    and file extension will be updated accordingly.
```

NOTE 1: Whenever using L, S, R, W, A:0, or A:L the filename can be expressed fully in either of the two FLEX standard forms. For example:

```
S=1.MYFILE.TXT

   or

S=MYFILE.TXT.1
```

NOTE 2: QUERY does not show the default drive number of file extension for the INCLUDE directive as determined by the configuration program SETPL9.

D I S K F I L E H A N D L I N G

The following commands are active only when the (#) prompt is present.

L[=<FILENAME>]

LOAD a disc file. The default drive and file extension specified when configuring PL/9 with the SETPL9 command need not be supplied. The filename supplied will become the default name to be used by further Load, Save, Assemble to Object or Assemble to Listing file commands.

S[=<FILENAME>]

SAVE the file on disc in TSC editor format. The editor will over-write any existing file of the same name. Filenames have the default extension ".PL9" (but this can be changed using SETPL9). If the filename is omitted then the name of the file that was loaded will be used again, allowing files to be loaded, modified and re-saved without the name having to be typed more than once.

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                                                                     *
*  PL/9 does not make backup copies of files; if you require a        *
*  backup you must create it explicitly (e.g. S=FILE.BAK).            *
*                                                                     *
*  Alternatively you can invoke the FLEX RENAME command from          *
*  within PL9 and rename the file just loaded into the editor         *
*  (e.g. /RENAME,1.TEST.PL9,1.TEST.BAK).                              *
*                                                                     *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

W<TARGET>[=<FILENAME>] or W<#TARGET>...

WRITE part of a file to disc. As for (S) except that PL/9 writes only the specified number of lines, starting at the current line in the first form, and writes from the current line to the specified line in the second form. The default filename in this case is 1.SCRATCH.SCR, but this may be changed using SETPL9.

R[=<FILENAME>]

READ in a file, inserting it into the buffer immediately above the current line. The default filename is 1.SCRATCH.SCR, as for (W). These two commands enable block moves to be made safely, by writing part of the file to disc and then re-loading it at the new position. This technique for block copy-move operations may be a bit inconvenient at times but it does away with the overhead of reserving a large chunk of memory for a seldom used text buffer.

RECOVERING A FILE IN MEMORY

If your System Monitor has a memory dump facility that also displays the contents of memory in ASCII on the VDU screen you stand a 50-50 chance of recovering a file that has been lost though an accidental use of the 'N' command or re-entering PL/9 through the cold start entry point at $0000.

This same technique can also save a file in memory when a system crash occurs, but this time the odds are about 1 in 10 that you will be successful.

The first case concerns an accidental use of the 'N' command or a cold start of PL/9. In both of these cases you can be confident that the original file is still present in memory and intact. What you have to do is enter your system monitor and dump the memory contents out to your VDU starting at the memory location CONTAINED in $4000/1. This is the beginning of file marker. As you work your way through the file you should recognize the text of your source file. Keep searching until you find the last line of the file. The memory location that you are interested in is the location of the first byte past the carriage return ($0D) in the last line. Once you locate this position in the file make a note of the memory location. Use your system monitor memory examine and change facility to alter the contents of $4002/3 to the memory address just noted. Now warm start PL/9 by a JUMP to $0003. Your file should be back to normal.

The second case concerns recovering a file when a system crash has occurred. In these circumstances the following course of action should be followed to the letter.

(1) Hit hardware RESET.

(2) Examine the contents of memory location $4002/3 and make a note of the address pointed to.

(3) Re-boot FLEX using a disk that does not have a STARTUP file on it. This is very important unless you are absolutely 100% positive that your STARTUP file does not cause the memory below say $B800 to be altered.

(4) Use the 'GET' command to load PL/9, i.e. +++GET,PL9.CMD<CR>

(5) Enter your system monitor. Use the system monitor dump memory command to display the contents of memory starting about 500 bytes or so before the address noted in step (2). Work your way up to the end of the file as described in the earlier recovery instructions and verify that the address noted does in fact point to the end of the text file. If it doesn't then go back to the beginning of the file and start working your way up it until the text becomes junk. Make a note of the address of the byte following the address of the last sensible line in the file. Insert this address into to memory location $4002.

(6) Open both disk drive doors, unless you like to live dangerously!

(7) Warm start PL/9 by jumping to $0003.

(8) With a bit of luck you will have recovered your file or at least a reasonable part of it. Save it out to disk with a full file specification, i.e. #S=1.CRASH.SAV<CR>

5.00.00 THE PL/9 COMPILER

The compiler resides in memory with the editor and may be considered to be an integral part of the editor when used in the interactive mode. The following commands are available whenever the (#) prompt is present:

T

Invoke the trace/debugger. PL/9 will compile the program without generating a listing, putting the code into memory, then will jump to the trace/debugger (described in its own section). No options are allowed with this command. The trace/debugger is covered in greater detail in the following section.

A

Compile the edit file without any listing, printout or object file. Generally used to perform a quick syntax/typographical error check.

A[:<options>]

Compile the file resident in memory. (A is used mainly for ergonomic compatibility with the MACE assembler.) Options are as follows:

A:C

Display the code generated for each source statement. The code generated for "INCLUDE" files (q.v.) will not be displayed. This option only makes sense when used with the 'T', 'P' or 'L' options.

A:M

Write object code directly into memory. PL/9 will not allow itself, its edit file or any of its tables to be over- written, and complains with the message "CAN'T WRITE TO $MMMM", where MMMM is the address of the attempted write. See the diagram of memory usage, in section three, for information on what areas of memory are used by PL/9 and its tracer.

A:N

Compile with a cross reference listing only. Defaults to terminal but may be directed to the printer (A:N,P) or to an output file (A:N,O).

A:P

Compile with a printer listing. The page number and the date will be printed at the top of each page. If there is a comment (i.e. /* ... */) on the first line of the program then this will be printed at the top of each listing page as a title, otherwise the PL/9 startup banner will be printed. The FLEX TTYSET 'WD' parameter is obeyed. Any lines longer than the defined 'WD' value will be truncated on the listing.

A:T

Compile with a listing on the terminal; no titles or page numbers will be printed. The FLEX TTYSET 'WD' parameter is obeyed. Any lines longer than the defined 'WD' value will be truncated on the listing.

A:L[=<FILENAME>]

Write the compile listing to disc into the named file. Use the Q command to see what default drive and extension will be used; if you don't like them use SETPL9 to change them. As for the (A:P) command titles and page numbers will be printed at the top of each page. This option is provided primarily to produce a text file for print 'spooling' or merging into a word processor text file. The file produced can also be 'P,LIST'ed.

A:O[=<FILENAME>]

Write object code to disc, overwriting any existing file of that name. Use the Q command to see what default drive and extension will be used; if you don't like them then use SETPL9 to change them.

A:<N1>-<N2>

If one of the T, P or L options is in force, the compiler can be requested to generate output for only the specified range of source line numbers. No symbol table will be output in this case. If the -<N2> is omitted then only one line will be generated.

A:$XXXX

When using either the 'M' or 'O' options it is frequently useful to be able to offset the program (for example when the object code is to be put into an EPROM and there is no RAM on the development system at the required address). The offset $XXXX is added to the program counter value (as printed on the object listing).

A:R

This option directs the compiler to use the MC8809 hardware vectors at $FFF2 through $FFFF in lieu of the RAM vectors you defined when using SETPL9. This option only has an effect if you use the following procedure names in your program: RESET, NMI, FIRQ, IRQ, SWI, SWI2 or SWI3. This facility enables you to test your program within the development system using its RAM vectors and then generate a file for use in the target hardware environment by invoking this option. This option only makes sense when used with 'A:O'.

Compile options may be strung together, as in the following examples:

A:P,100-200

Compile to the printer, generating a listing only for lines 100-200.

A:C,T,281

Compile to the terminal, displaying the generated object code for line 281 only.

A:M,$4000

Compile to memory, loading the program at a location offset by $4000 from any origin specified.

A:O,L

Generate a binary file and a listing file, both files having the names given by the Q command.

A:O=1.MYFILE.BIN,L=0.MYFILE.OUT

As above but override the default drive numbers, file names and file extensions.

A:O,R

Generate a binary file, using the default drive number, file name and file extension, but substitute the MC6809 hardware vectors ($FFF2 - $FFFF) for the RAM vectors defined by SETPL9.

NOTE: Any one, or all of the listing options (T, P or L) may be in force at any given time. i.e. it is possible to specify 'A:T,P,L,C' and generate a listing on the terminal, a listing on the printer and a disk file all with the code generated by PL/9 displayed if this is what you require.

CALLING THE COMPILER FROM FLEX

The PL/9 compiler may also be called from FLEX with multiple options specified, as the following examples illustrate:

+++PL9,1.FILENAME.EXT<CR>

Compile the file specified reporting any errors.

+++PL9,FILENAME,T<CR>

Compile the file specified using the default drive number and file extension that PL/9 was configured for using the SETPL9 command to the system console obeying TTYSET 'WD'.

+++PL9,FILENAME,C,P<CR>

Compile the file specified to the system printer obeying TTYSET 'WD' and displaying the generated object code

+++PL9,FILENAME,O,R<CR>

Compile the file specified to an output file (binary) substituting the MC6809 hardware vectors for the RAM vectors defined by SETPL9. The output file will have the same name as 'FILENAME' but will use the default drive and file extension specified by SETPL9.

+++PL9,FILENAME,O=0.OBJECT.BIN,R<CR>

As above but override the default drive and extension.

+++PL9,FILENAME,C,L,O<CR>

Compile the file but direct the output listing, with generated object code displayed, to a disk file using the default drive number and file extension defined by SETPL9. Also produce an object file on disk in the same manner.

+++PL9,FILENAME,C,L,O,M<CR>

Compile the file as above but also compile the file into memory.

NOTE: Any one, or all of the listing options (T, P or L) may be in force at any given time. i.e. it is possible to specify 'A:T,P,L,C' and generate a listing on the terminal, a listing on the printer and a disk file all with the code generated by PL/9 displayed if this is what you require.

ERROR HANDLING

When PL/9 detects an error it stops compiling and prints a message from the list below, followed by the source line in which the error was found, then by a caret (up arrow) under the position it had reached in that source line. This is usually within one or two character positions of the point at which the error was detected, although some errors may not be detected until the following line (for example when a semicolon is omitted). Once the error has been reported the compiler waits for the user to type a response. A carriage return will cause the editor to be re-entered at the line in which the error was detected, while any other input will cause compilation to resume after the first semicolon following the error. (This means that further errors may be reported as a result of a keyword such as BEGIN being skipped, leaving a mismatched END later on in the program.)

Most of the error messages are self-explanatory, but each is outlined below:

")" EXPECTED

The compiler was expecting a right parenthesis at this point.

"," EXPECTED

The compiler was expecting a comma at this point.

";" EXPECTED

The compiler was expecting a statement delimiter at this point.

"=" EXPECTED

The compiler was expecting an assignment operator at this point.

"BYTE", "INTEGER" OR "REAL" EXPECTED

The syntax so far has led the compiler to expect a BYTE, INTEGER or REAL identifier; for example if a colon is typed instead of a semicolon at the end of a GLOBAL statement.

ILLEGAL DECLARATION

You will get this message if you attempt to use the GLOBAL or DPAGE directives after the compiler has generated any code.

NO ACTIVE PROCEDURE

A statement has been encountered within a procedure that may only reside outside a procedure.

NOT ALLOWED OUTSIDE A PROCEDURE

A statement has been encountered outside a procedure that may only reside within a procedure.

NUMBER OR CONSTANT EXPECTED

The symbol encountered should have been a number or a constant. For example, it is not permissible to assign a variable to a constant.

ONLY ONE GLOBAL DECLARATION!

There may be only one GLOBAL statement in a program.

READ-ONLY VARIABLE

An attempt has been made to assign a value to a data variable, i.e. one declared by the BYTE, INTEGER or REAL data declaration.

SYMBOL ALREADY EXISTS

The variable or procedure name indicated has already been declared, earlier on in the program.

"THEN" EXPECTED

An IF statement has been encountered that either has a missing THEN or some other error causing the compiler to miss the THEN.

THIS PROCEDURE DOESN'T RETURN A VALUE

An attempt has been made to use a procedure as a function subroutine without a return value having been specified.

TOO MANY BREAKS

Only ten BREAKs may be pending at any one time.

UNDEFINED SYMBOL

The variable or procedure name indicated has not been declared.

WARNING: SYMBOL ALREADY USED FOR ANOTHER VARIABLE TYPE

Usually caused by trying to give a CONSTANT the same name you have given a procedure.

WRONG PARAMETER TYPE

In a procedure call, the parameters supplied do not match those declared with the PROCEDURE statement. This message is also used to report errors when using pointers incorrectly.

WRONG VARIABLE TYPE

You will get this error message if you put a procedure name on the left side of an assignment expression, if JUMP or CALL attempt to use anything other than an integer or if you attempt to use a variable that has not been declared as a pointer as a pointer.

OTHER ERRORS

It is worth noting that there are some errors that may not be detected until considerably after they occur. Errors such as a missing END or a missing semicolon are unlikely to be detected immediately. A missing " at the end of a string can cause the whole of the rest of the program to be treated as a very long string! In cases such as these there is no substitute for human ingenuity - you shouldn't expect PL/9 to do ALL your thinking for you! If a visual inspection fails to locate the error, try removing instructions successively back up the program until the problem is located.

It is also worth noting that the compiler will not object to you giving a CONSTANT one of the names reserved for keywords. For example it will readily accept

```
CONSTANT RETURN=$0D;
```

knowing full well that 'RETURN' is a reserved word. Genearlly speaking the expressions that will use a constant will use it in such a way that the compiler will not confuse it with a keyword. Thefore this abuse of keyword names is not considered to be an error to the compiler. If you decide to use keywords as constants, IN SPITE OF OUR WARNINGS NOT TO, you should be sure of what you are doing first!

THIS PAGE INTENTIONALLY LEFT BLANK

6.00.00 THE PL/9 TRACE/DEBUG FACILITY

The PL/9 symbolic trace/debugger allows the user to single-step, breakpoint or run his program and to examine variables as the program proceeds. The principle of operation is that in trace mode PL/9 inserts calls to the tracer and information relating to the program into the object code at every line of source enabling the debugger to retain control over the running program. The penalty paid for this facility is that the program runs somewhat slower, to an extent dependant partly upon the program itself but mainly upon the number of trace/debug facilities used. The trace instructions are not inserted into the code generated for INCLUDEd procedures; these will run at full speed since it is assumed that they will have been previously debugged.

The tracer for PL/9 is a separate file, called PL9_TD.CMD and PL/9 expects to find it on the system drive (usually drive 0). If it is not present then you will not be able to trace programs. It loads into the FLEX command area between $C100 and $C6FF, so please ensure that the program you are tracing does not make any reference to this area. It is quite permissible for the program to have an ORIGIN within this area as ORIGIN statements are ignored by the tracer. Refer to the PL/9 memory map in section 3.01.00 for further details.

W A R N I N G !

The ENDPROC on the last (main) procedure should not be present when using the tracer. The tracer 'senses' the JUMP to FLEX that is put at the end of the main procedure as meaning that the program has ended. If the ENDPROC is used this jump will be replaced by an RTS instruction which will cause the tracer to crash when it reaches the end of the program. Alternatively you may leave the ENDPROC in and place the statement 'JUMP $0003;' just before the ENDPROC while you are tracing a program. This will cause the tracer to re-enter PL/9 via its warm start address when the program ends.

C A U T I O N !

Any procedure that 'polls' the system console keyboard via a routine similar to 'GETKEY' (see the IOSUBS library descriptions) will not work properly when using the tracer. This is due to the fact that the tracer is also 'polling' the keyboard looking for a CONTROL C.

N O T E

The way the tracer works will often cause it NOT to print the last line in a program and/or not print the variables associated with the last line in the procedure when single stepping the program. This 'foible' of the tracer can be avoided by simply placing the statement 'ACCB=ACCB;' below the last source line in the program. This statement may be left in your program if desired as the compiler will not generate any code for it.

USING THE TRACER

To invoke the tracer, load the file as usual and check (using the A command) that there are no errors, then type T. PL/9 will compile the file, putting the object code at the top of available memory, ignoring any ORIGIN statements. It will then indicate how much space is available between the top of the symbol table and the start of the program (this is the area in which global and local variables are stored). The tracer will then be loaded from the system drive, then a startup banner will be printed. The first executable line (the last procedure declaration in the program) will be displayed and the tracer will stop and and wait for instructions. Note that there are no options associated with the T command.

```
    #T

XXXXX BYTES AVAILABLE FOR STACK

0182 PROCEDURE MAIN;              /*  THIS IS THE MAIN PROGRAM  */
    &
```

The debug command mode is indicated by the editor prompt changing from # to &.

MODE CONTROL COMMANDS

<ESCAPE>

Typing <ESCAPE> at the start of the line will return control to the editor.

E

EDITOR. This command also causes a return to the editor.

X

EXIT to the disk operating system.

M

MONITOR. Exit to the ROM system monitor.

SOURCE FILE RELATED COMMANDS

<CR>

Display the current line i.e. the source line about to be executed. The entire line is printed, complete with any comments.

<NUMBER>P<TARGET>

Print part of the source file. This command is the same as the editor P command and has no effect on the execution of the program; it is included as an aid to the programmer deciding where to put breakpoints etc.

TRACER CONTROL COMMANDS

G

GO. Run the program, continuing until:

(1) a breakpoint is encountered.
(2) a control C is typed.
(3) the program ends.

S

SINGLE-STEP the program. Execution will stop at each source line, the line will be displayed and the tracer will wait for a key to be pressed before continuing. A space will cause the line to be executed, while anything else will cause a return to tracer command level.

R<NUMBER>

RUN <NUMBER> lines of the program. Execution can be terminated as for G above, or will stop after the specified number of lines has been executed.

T<NUMBER>

TRACE <NUMBER> lines of the program. Each source line will be printed before it is executed; otherwise as for R (which does not display the source lines). If <NUMBER> is zero, the result will be as if G had been typed, except that lines with breakpoints will not cause a break but will instead cause a printout of the source line, with execution then continuing automatically.

W<NUMBER>

WAIT at each line for a time dependant on the value of <NUMBER>. This allows a program to be slowed down when in TRACE so that its effects can more clearly be seen.

N<N1>-<N2>[,<N3>-<N4>....]

NO TRACE. The tracer will not stop at any line in any of the ranges specified. This command is very useful in that it can be made to skip delay loops, initialization sequences and procedures that have already been debugged.

Q

QUIT. Restart the program without re-compiling it.

BREAKPOINTS

B

Clear all breakpoints.

B<N1>,<N2>...

Set breakpoints at the specified line(s). Existing breakpoints are kept active; the only way to remove them is to clear all breakpoints using B by itself.

VARIABLES

?<VARIABLE LIST>

Print the values of specified program variables. Simple variables and vector elements (with numeric indices, not other variables) can be specified, and may be separated by either a semicolon (print on the same line) or a comma (start a new line). Examples:

Print all three values on one line:

```
&?CHAR;COUNT;POS
```

Print the first three values on one line and the fourth on the next line:

```
&?BUFFER;BUFFER(1);BUFFER(2),COUNT
```

Print each value on a separate line:

```
&?CHAR1,CHAR2,CHAR3
```

If a variable is specified that either does not exist or is not known to the current procedure then no value is printed.

D<VARIABLE LIST>

Print variable values whenever a source line is displayed. The values are specified as for (?) above and are printed before the source line. To prevent variable printing, use the command D with no list. The variable list is "remembered" from one compilation to the next.

7.00.00 PL/9 LANGUAGE REFERENCE MANUAL

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

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

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

7.00.01 COMMENTS

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

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

7.00.02 SYMBOLS

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

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

7.01.00 PL/9 KEYWORD DESCRIPTIONS

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

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

\+ Covered in the section on BIT OPERATORS.

\- Covered in the section on FUNCTIONS.

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

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

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

7.01.01 CONSTANT

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

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

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

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

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

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

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

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

7.01.02 AT

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

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

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

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

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

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

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

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

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

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

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

This allows you to generate constructions such as:

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

to shift data around in the absolute memory map.

7.01.03 ORIGIN

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

```
ORIGIN = $C100;
```

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

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

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

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

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

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

ORIGIN=$2000;
```

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

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

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

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

7.01.04 STACK

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

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



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

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

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

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

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

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

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

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

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

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

7.01.05 GLOBAL

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

Typical GLOBAL declarations are as follows:

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

GLOBAL BYTE BUFFER(80), FLAG, COUNT;

GLOBAL REAL VALUE1, VALUE2: BYTE FLAG;

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

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

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

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

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

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

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

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

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

7.01.06 DPAGE

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

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

```
DPAGE=$E0;
```

Alternatively you can assign DPAGE via a constant.

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

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

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

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

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

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

7.01.07 BYTE, INTEGER and REAL

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

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

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

INTEGER PROCEDURES .PROC1, .PROC2, .PROC3;

REAL PI 3.14159265;

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

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

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

7.01.08 INCLUDE

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

```
INCLUDE FNAME;
```

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

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

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

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

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

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

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

7.01.09 A PL/9 PROGRAM HEADER

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

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

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

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

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

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

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

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

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

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

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

7.01.10 PROCEDURE

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

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

```
PROCEDURE ONE;
```

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

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

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

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

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

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

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

7.01.10 PROCEDURE (continued)

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

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

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

Pointers are ALWAYS 16-bit quantities. The variables that pointers point to may be BYTE, INTEGER or REAL (or vectors of these variable types) and must always be declared as such. viz: (REAL .POINTER1:INTEGER .POINTER2:BYTE .POINTER3) declares that three pointers are being passed to the procedure. One is pointing to a REAL, one is pointing to an INTEGER, and one is pointing to a BYTE. They can be single variables of the size indicated or they may be elements of a vector.

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

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

PLEASE NOTE

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

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

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

7.01.10 PROCEDURE (continued)

PROCEDURES AS VARIABLES AND FUNCTIONS

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

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

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

```
VAR1 = PROCVAR1;

VAR1(PROCVAR1) = VAR2;

VAR1 = VAR2*PROCVAR1;
```

get the idea yet?

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

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

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

```
PROCVAR1 = 1;
```

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

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

now we can use the construction:

```
PROCVAR2 = 'A;
```

7.01.10 PROCEDURE (continued)

PROCEDURES AS VARIABLES AND FUNCTIONS (continued)

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

```
VAR1 = PROCVAR2;
```

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

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

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

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

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

```
VAR1 = VAR2;

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

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

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

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

7.01.11 ENDPROC, RETURN, and ENDPROC END

These keywords provide the mechanism for terminating PROCEDURES.

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

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

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

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

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

ENDPROC

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

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

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

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

7.01.11 ENDPROC, RETURN, and ENDPROC END (continued)

ENDPROC (continued)

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

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

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

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

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

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

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

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

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

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

7.01.11 ENDPROC, RETURN, and ENDPROC END (continued)

ENDPROC (continued)

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

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

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

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

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

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

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

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

This latter construction illustrates that you can use the construction:

```
ENDPROC <EXPRESSION>;
```

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

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

ENDPROC CON;        (where CON is a constant)

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

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

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

7.01.11 ENDPROC, RETURN, and ENDPROC END (continued)

RETURN

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

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

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

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

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

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

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

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

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

7.01.11 ENDPROC, RETURN, and ENDPROC END (continued)

RETURN (continued)

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

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

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

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

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

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

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

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

7.01.11 ENDPROC, RETURN, and ENDPROC END (continued)

RETURN (continued)

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

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

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

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

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

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

7.01.11 ENDPROC, RETURN, and ENDPROC END (continued)

ENDPROC END

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

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

If it is desired, however, that the program behave in its entirety as a subroutine (so that it can be called from BASIC, for example) then it is important that the stack pointer and 6809 registers are returned with the values the PL/9 program was entered with.

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

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

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

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

N O T E

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

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

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

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


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

                  [ ELSE <STATEMENT> ]
```

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

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

IF VALUE = 66.3 THEN ...

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

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

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

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

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

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

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

7.01.13 BEGIN...END

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

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

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

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

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

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

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

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

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

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

LOGICAL OPERATORS MAY NOT BE BRACKETED!

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

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

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

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

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

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

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

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

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

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

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

7.01.15 WHILE

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

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

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

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

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

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

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

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

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

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

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

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

7.01.16 REPEAT...UNTIL REPEAT...FOREVER

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

REPEAT <STATEMENT> UNTIL <CONDITION>

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

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

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

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

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

7.01.17 BREAK

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

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

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

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

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

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

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

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

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

7.01.18 GOTO

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

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

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

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

7.01.19 CALL

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

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

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

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

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

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

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

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

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

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

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

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

```
GLOBAL INTEGER TABLE(3);

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

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

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

7.01.20 JUMP

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

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

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

7.01.21 GEN

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

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

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

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

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

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

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

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

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

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

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

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

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


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

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

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

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

7.01.22 ASMPROC

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

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

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

```
ASMPROC ONE;                              /*  No parameters passed      */

ASMPROC TWO(BYTE);                        /*  One BYTE parameter passed */

ASMPROC THREE(INTEGER,INTEGER,BYTE);      /*  Three parameters passed;
                                              two INTEGER and one BYTE  */

ASMPROC FOUR(REAL);                       /*  One REAL parameter passed */
```

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

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

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

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

7.01.22 ASMPROC (continued)

R E M E M B E R

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

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

```
ASMPROC FOUR: BYTE;                      /*  Returns a BYTE value  */

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

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

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

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

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

7.01.22 ASMPROC (continued)

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

```
GLOBAL INTEGER RNDVAL(2);

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

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

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

```
IVAL=RANDOM;

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

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

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

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

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

7.01.22 ASMPROC (continued)

HOW DATA IS PASSED TO AND FROM PROCEDURES

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

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

```
                                    GLOBAL INTEGER I1, I2, I3;
PSHS  CC,D,DP,X,Y,U
LEAS  -6,S (allocate global storage)
TFR   S,Y  (point 'Y' at base of global storage)

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

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

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

7.01.22 ASMPROC (continued)

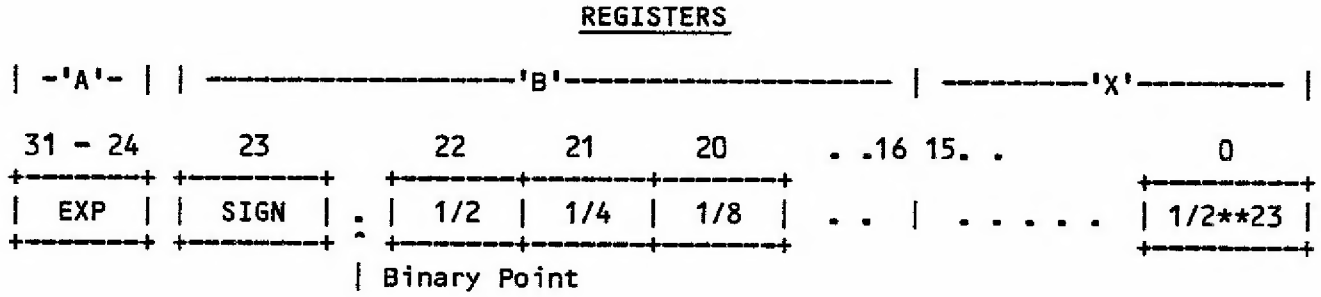
THE ARCHITECTURE OF REAL (FLOATING-POINT) NUMBERS

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

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

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

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



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

7.01.22 ASMPROC (continued)

EXAMPLES OF REAL NUMBER BINARY FORMATS

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

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

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

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

1.0 is 01 40 00 00

Some more numbers:

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

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

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

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

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

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

7.01.23 ACCA, ACCB, ACCD, XREG and STACK

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

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

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


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


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

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

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

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

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

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

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

7.01.24 CCR

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

For example, to set the carry flag, use

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

To clear the IRQ interrupt flag, use:

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

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

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

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

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

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

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

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

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

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

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

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

7.01.25 MATHS

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

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

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

```
MATHS = $E800;
```

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

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

NOTE

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

7.02.00 INTERFACING WITH ASSEMBLY LANGUAGE (a plug for MACE)

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

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

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

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

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

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

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

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

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

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

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

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

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

7.03.00 PL/9 VARIABLES

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

7.03.01 ARITHMETIC QUANTITIES

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

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

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

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

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

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

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

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

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

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

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

7.03.02 UNSIGNED BYTES AND INTEGERS

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

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

BYTE in the range of 0 to 255.

INTEGER in the range of 0 to 65,535.

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

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

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

7.03.02 UNSIGNED BYTES AND INTEGERS (continued)

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

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

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

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

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

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

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

HEX NUMBERS

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

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

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

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

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

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

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

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

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

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

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

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

THE EXCLAMATION MARK (!)

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

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

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

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

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

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

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

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

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

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

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

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

Other examples are:

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

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

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

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

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

INTEGER

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

For example:

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

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

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

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

```
INTVAR = READ_PORT;
```

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

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

This function is also discussed in section 7.06.00.

EXAMPLES OF UNSIGNED OPERATIONS

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

EXAMPLES OF UNSIGNED OPERATIONS

```
0059
0060
0061    /* COMPARE BYTES WITH INTEGERS (THEY ALL REQUIRE THE !) */
0062
0063    IF !$80 = I5 THEN PRINT "\N13";
0064
0065    IF !A = I5 THEN PRINT "\N14";
0066
0067    IF !B1 = I5 THEN PRINT "\N15";
0068
0069    IF !$80 = I4 THEN PRINT "\N16";
0070
0071
0072    /* UNSIGNED ADDITION WITH BYTES AND INTEGERS (THEY ALL REQUIRE THE !) */
0073
0074    IF !I5+$10 = B1+$10 THEN PRINT "\N17";
0075
0076    IF !B1+$10 = I5+$10 THEN PRINT "\N18";
0077
0078    IF !I5+B2 = I6+B1 THEN PRINT "\N19";
0079
0080    IF !(I5+B2) - (I6+B1) = (B1+B2) - (I5+I6) THEN PRINT "\N20";
0081
0082
0083    /* COMPARE A BYTE WITH THE SUM OF TWO BYTES WHERE THE RESULT OVERFLOWS
0084       A BYTE VARIABLE.  IN THIS CASE PL/9 WILL CONVERT THE OFFENDING SUM
0085       TO AN INTEGER BY SIGN EXTENSION UNLESS YOU USE THE (!).          */
0086
0087     B1 = 100;
0088     B2 = 75;
0089     B3 = 120;
0090
0091     IF !B3 < B1 + B2 THEN PRINT "\N21";
0092
0093
```

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

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

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

7.03.03 DATA TYPES

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

GLOBAL VARIABLES

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

LOCAL VARIABLES

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

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

ABSOLUTE VARIABLES

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

CONSTANTS

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

READ-ONLY DATA

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

7.03.04 POINTERS

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

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


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

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

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

THERE'S MORE

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

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

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

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

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

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

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

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

7.03.04 POINTERS (continued)

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

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

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

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

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

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

we can develop several constructions within a procedure:

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

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

```
BP1 = BP2;
```

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

S U M M A R Y

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

7.03.04 POINTERS (continued)

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

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

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

The latter is a much more efficient construction than:

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

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

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

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

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

OR

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

OR

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

7.03.04 POINTERS (continued)

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

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

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

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

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

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

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

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

7.03.04 POINTERS (continued)

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

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

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

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

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

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

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

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

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

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

Do the same thing in reverse.

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

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

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

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

7.03.04 POINTERS (continued)

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

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

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

```
AT $E060:INTEGER DA_CONV;

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

   DATA1=$1234;
   DATA2=$4567;

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

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

```
DATA2 = POINTER1;
```

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

```
POINTER1 = DATA2;
```

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

```
POINTER1 = POINTER2;
```

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

7.03.04 POINTERS (continued)

POINTERS TO STRINGS

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

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

                    OR

        PRINT "Choice (Y/N)?";

                              OR

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

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

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

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

Whenever the message needs to be printed you simply enter:

```
PRINT(IVAR);
```

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

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

Then, whenever you need the message you simply enter:

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

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

```
IVAR = .PROMPT;

PRINT(IVAR);
```

7.04.00. ARITHMETIC AND OPERATOR PRECEDENCE RULES

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

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

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

N O T E

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

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

would produce code that would:

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

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

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

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

7.05.00 BIT OPERATORS AND, OR, EOR (XOR)

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

A summary of the functions are as follows:

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

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

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

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

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

is completely different in meaning to

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

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

7.06.00 PL/9 FUNCTIONS

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

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

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

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

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

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

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

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

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

7.06.00 PL/9 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.

7.07.00 ANATOMY OF PL/9 PROGRAMS

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

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

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

7.07.01 VARIABLE ALLOCATION

7.07.02 GLOBAL VARIABLES

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

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

The code generated is

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

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

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

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

7.07.03 LOCAL VARIABLES

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

7.07.04 ABSOLUTE VARIABLES

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

7.07.05 DATA

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

7.07.06 ASSIGNMENTS

7.07.07 SIMPLE ASSIGNMENTS

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

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

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

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

7.07.08 VECTORS

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

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

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

3. Add D to X.

4. Load B or D via X.

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

4. Push the value in X onto the stack.

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

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

7.07.09 PROCEDURE CALLS

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

7.07.10 PROCEDURES

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

7.07.11 BUILT-IN ARITHMETIC FUNCTIONS

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

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

7.07.12 REGISTER PRESERVATION

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

7.07.13 POINTERS

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

And you can subscript them:

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

7.07.13 POINTERS (continued)

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

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

OR

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

Now lets examine the basic constructions possible within each group.

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

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

7.07.13 POINTERS (continued)

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

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

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

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

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

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

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

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

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

7.07.13 POINTERS (continued)

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

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

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

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

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

POINTERS TO PROCEDURES

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

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

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

```
PROCEDURE PROC_ONE;
.
.
ENDPROC;


ORIGIN = $XXXX;

INTEGER _ONE .PROC_ONE;
```

WARNING: THESE VECTOR TABLES ARE NOT POSITION INDEPENDANT!

7.07.14 MEMORY USE BY PL/9 DURING COMPILATION

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

7.07.15 DISK BINARY FILES

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

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

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

8.00.00 PL/9 LIBRARIES TECHNICAL REFERENCE MANUAL

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

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

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

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

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

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

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

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

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

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

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

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

8.00.00 PL/9 LIBRARIES TECHNICAL REFERENCE MANUAL (continued)

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

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

N O T I C E

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

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

N O T I C E

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

8.00.01 TRUE - FALSE - MEM DEFINITIONS

N O T I C E

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

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

TRUE - FALSE - MEM DEFINITIONS V:4.00

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


EXTERNALS:

true                FFFF
false               0000
mem                 0000   BYTE
```

8.01.00 IOSUBS.LIB

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

8.01.01 MONITOR

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

8.01.02 WARMS

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

8.01.03 GETCHAR

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

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

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

8.01.04 GETCHAR_NOECHO

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

GETCHAR_NOECHO may be treated in a manner identical to GETCHAR.

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

8.01.05 GETKEY

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

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

8.01.06 CONVERT_LC

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

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

8.01.07 GET_UC

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

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

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

8.01.08 GET_UC_NOECHO

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

8.01.09 PUTCHAR

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

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

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

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

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

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

8.01.10 PRINTINT

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

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

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

8.01.11 REMOVE_CHAR

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

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

8.01.12 INPUT

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

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

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

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

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

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

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

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

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

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

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

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

8.01.12 INPUT (continued)

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

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

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

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

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

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

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

8.01.13 CRLF

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

```
CRLF;
```

8.01.14 PRINT

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

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

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

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

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

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

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

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

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

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

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

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

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

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

8.01.14 PRINT (continued)

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

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

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

8.01.15 SPACE

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

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

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

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

```
0000  0001 /*  SYSTEM CONSOLE INPUT/OUTPUT ROUTINES  V:4.00  */
0000  0002
0000  0003
0000  0004 constant nul = $00,
0000  0005          abt = $03,
0000  0006          bel = $07,
0000  0007           bs = $08,
0000  0008           lf = $0a,
0000  0009           cr = $0d,
0000  0010          can = $18,
0000  0011          esc = $1b,
0000  0012           sp = $20;
0000  0013
0000  0014
0000  0015 procedure monitor;
0003  0016   gen $6e,$9f,$d3,$f3;  /* JMP [$D3F3]  ('MONIT') */
0007  0017 endproc;
0008  0018
0008  0019
0008  0020 procedure warms;
0008  0021    jump $cd03; /* FLEX WARM START ENTRY POINT */
0008  0022 endproc;
000C  0023
000C  0024
000C  0025 procedure getchar;
000C  0026    call $cd15; /* FLEX 'GETCHR' */
000F  0027 endproc acca;
0012  0028
0012  0029
0012  0030 procedure getchar_noecho;
0012  0031    gen $ad,$9f,$d3,$e5;  /* JSR [INCHNE] */
0016  0032 endproc acca;
0019  0033
0019  0034
0019  0035 procedure getkey;
0019  0036    call $cd4e; /* FLEX 'STAT' */
001C  0037    if ccr and $04  /* IMPLICIT <> 0 */
001E  0038       then acca = nul;
0029  0039       else getchar_noecho;
002E  0040 endproc acca;
0031  0041
0031  0042
0031  0043 procedure convert_lc(byte char);
0031  0044    if char >= 'a .and char <= 'z
003F  0045       then char = char - $20;
0053  0046 endproc char;
0056  0047
0056  0048
0056  0049 procedure get_uc:byte inchar;
0058  0050 endproc convert_lc(getchar);
0063  0051
0063  0052
0063  0053 procedure get_uc_noecho:byte inchar;
0065  0054 endproc convert_lc(getchar_noecho);
0070  0055
0070  0056
```

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

```
0070  0057 procedure putchar(byte char);
0070  0058    acca = char;
0074  0059    call $cd18; /* FLEX 'PUTCHR'  (HONOURS 'TTYSET' PARAMETERS) */
0077  0060 endproc;
0078  0061
0078  0062
0078  0063 procedure printint(integer n);
0078  0064    if n < 0
007A  0065       then begin
0081  0066               putchar '-;
0089  0067               n = -n;
0091  0068            end;
0091  0069    if n >= 10 then printint n/10;
0109  0070    putchar n\10 + '0;
011C  0071 endproc;
011D  0072
011D  0073 procedure remove_char;
011D  0074    putchar(bs);
0126  0075    putchar(sp);
012F  0076    putchar(bs);
0138  0077 endproc;
0139  0078
0139  0079
0139  0080 procedure input(byte .buffer,length):byte char:integer pos;
013B  0081    pos = 0;
0140  0082    repeat
0140  0083       char = getchar_noecho;
0145  0084       if char
0145  0085          case bs
014A  0086               then begin
014E  0087                    if pos > 0
0150  0088                       then begin
0157  0089                            remove_char;
0159  0090                            pos = pos - 1;
0160  0091                            end;
0160  0092                       else putchar(bel);
016C  0093                    end;
016C  0094          case can
0171  0095               then begin
0175  0096                       while pos > 0
0177  0097                          begin
017E  0098                             remove_char;
0180  0099                             pos = pos - 1;
0187  0100                          end;
0187  0101                    end;
0189  0102       if char >= sp
018B  0103          then if pos < length
0193  0104                  then begin
01A2  0105                          putchar(char);
01AB  0106                          buffer(pos) = char;
01B5  0107                          pos = pos + 1;
01BC  0108                       end;
01BC  0109                  else putchar(bel);
01C8  0110    until char = cr;
01D0  0111    buffer(pos) = 0;
01D8  0112 endproc .buffer;
```

```
01DD  0113
01DD  0114
01DD  0115 procedure crlf;
01DD  0116    putchar(cr);
01E6  0117    putchar(lf);
01EF  0118 endproc;
01F0  0119
01F0  0120
01F0  0121 procedure print(byte .string): byte char;
01F2  0122    while string /* IMPLICIT <> 0 (NULL) */
01F2  0123       begin
01FA  0124          if string = '\
01FD  0125             then begin
0203  0126                  .string = .string + 1;
020A  0127                  if string >= 'a .and string <= 'z
021A  0128                     then char = string - $20;  /* FORCE UPPER CASE */
022F  0129                     else char = string;
0237  0130                  if char
0237  0131                     case 'N then crlf;
0242  0132                     case 'B then putchar(bel);
0254  0133                     case 'L then putchar(lf);
0266  0134                     case 'R then putchar(cr);
0278  0135                     case 'E then putchar(esc);
028A  0136                     case '0 then putchar(nul);
029C  0137                     else begin
029F  0138                             putchar('\);
02A8  0139                             putchar(string);
02B2  0140                          end;
02B2  0141                  end;
02B2  0142             else putchar(string);
02BF  0143          .string = .string + 1;
02C6  0144       end;
02C6  0145 endproc;
02CC  0146
02CC  0147
02CC  0148 procedure space(integer n);
02CC  0149    while n > 0
02CE  0150       begin
02D5  0151          putchar(sp);
02DE  0152          n = n - 1;
02E5  0153       end;
02E5  0154 endproc;
```

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

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

DATA:

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

GLOBALS:
```

8.02.00 INTELLIGENT TERMINAL LIBRARY

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

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

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

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

CONTROL CODES USED BY THIS LIBRARY

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

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

8.02.01 NULLS

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

8.02.00 INTELLIGENT TERMINAL LIBRARY (continued)

8.02.02 ERASE EOL

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

8.02.03 ERASE EOP

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

8.02.03 CURSOR

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

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

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

8.02.04 HOME

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

8.02.05 HOME N CLR

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

8.02.06 ATTR ON

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

8.02.07 ATTR OFF

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

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

```
0000  0001 /* INTELLIGENT TERMINAL HANDLER LIBRARY  V:4.00  */
0000  0002
0000  0003
0000  0004 include 0.iosubs.lib;
02E8  0005
02E8  0006
02E8  0007 /* THIS LIBRARY IS CONFIGURED FOR A SOROC IQ120/LEAR SIEGLER ADM-5 */
02E8  0008
02E8  0009
02E8  0010 procedure nulls:byte count;
02EA  0011    count = 1;
02EE  0012    while count /* IMPLICIT <> 0 */
02EE  0013       begin
02F5  0014          putchar(nul);
02FE  0015          count = count - 1;
0300  0016       end;
0300  0017 endproc;
0305  0018
0305  0019
0305  0020 procedure erase_eol;
0305  0021    putchar(esc);
030E  0022    putchar('T);
0317  0023    nulls;
0319  0024 endproc;
031A  0025
031A  0026
031A  0027 procedure erase_eop;
031A  0028    putchar(esc);
0323  0029    putchar('Y);
032C  0030    nulls;
032E  0031 endproc;
032F  0032
032F  0033
032F  0034 procedure cursor(byte column,row);
032F  0035    putchar(esc);
0338  0036    putchar('=);
0341  0037    putchar(sp + row);      /* OFFSET OF $20 */
034C  0038    putchar(sp + column);   /* OFFSET OF $20 */
0357  0039    nulls;
0359  0040 endproc;
035A  0041
035A  0042
035A  0043 procedure home;
035A  0044    cursor(0,0);
0366  0045 endproc;
0367  0046
0367  0047
0367  0048 procedure home_n_clr;
0367  0049    home;
0369  0050    erase_eop;
036B  0051 endproc;
036C  0052
036C  0053
```

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

VISA 30/40 CODE SEQUENCES

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

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


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


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


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


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

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

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

PROCEDURES:

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

DATA:

EXTERNALS:

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

GLOBALS:

8.03.00 HEXIO LIBRARY

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

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

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

W A R N I N G

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

HEX-IO LIBRARY GLOBAL DEFINITIONS

The HEXGLOBL.DEF file looks like this:

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


GLOBALS:

erflag                0000   BYTE
keychar               0001   BYTE
```

8.03.01 GET_HEX_NIBBLE

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

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

8.03.02 GET_HEX_BYTE

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

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

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

8.03.03 GET_HEX_ADDRESS

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

8.03.04 PUT_HEX_NIBBLE

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

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

would print 'F' on the system console screen.

8.03.05 PUT_HEX_BYTE

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

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

8.03.06 PUT_HEX_ADDRESS

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

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

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

```
0000  0001 /*  HEX-IO LIBRARY  V:4.00  */
0000  0002
0000  0003
0000  0004 include 0.hexglobl.def;
0006  0005 include 0.trufalse.def;
0006  0006 include 0.iosubs.lib;
02EE  0007
02EE  0008 <---------------------------------------- NOTE:  Sudden   jump   in
02EE  0009 procedure get_hex_nibble:byte inchar;           address  caused  by
02F0  0010    inchar = get_uc;                             included libraries
02F5  0011    keychar = inchar;
02F9  0012    erflag = true;
02FD  0013    if inchar >=  '0 .and inchar <= '9
030B  0014       then begin
0319  0015            inchar = inchar - '0;
031F  0016            erflag = false;
0321  0017            end;
0321  0018       else if inchar >=  'A .and inchar <= 'F
0332  0019               then begin
0340  0020                    inchar = inchar - '7;
0346  0021                    erflag = false;
0348  0022                    end;
0348  0023 endproc inchar;
034D  0024
034D  0025
034D  0026 procedure get_hex_byte:byte inchar;
034F  0027       inchar = shift(get_hex_nibble,4);
0357  0028       if erflag = true
0359  0029          then return;
0362  0030       inchar = inchar or get_hex_nibble;
0370  0031 endproc inchar;
0375  0032
0375  0033
0375  0034 procedure get_hex_address:integer inchar;
0377  0035       inchar = swap(integer(get_hex_byte));
037E  0036       if erflag = true
0380  0037          then return;
0389  0038       inchar = inchar or integer(get_hex_byte);
0396  0039 endproc inchar;
039B  0040
039B  0041
039B  0042 procedure put_hex_nibble(byte outchar);
039B  0043    outchar = (outchar and $0f) + '0;  /* CONVERT TO ASCII */
03A3  0044    if outchar > '9
03A5  0045       then outchar = outchar + 7;       /* A-F OFFSET */
03B1  0046    putchar(outchar);
03BA  0047 endproc;
03BB  0048
03BB  0049
03BB  0050 procedure put_hex_byte(byte outchar);
03BB  0051    put_hex_nibble(shift(outchar,-4)); /* FIRST DIGIT */
03C7  0052    put_hex_nibble(outchar);           /* LAST DIGIT */
03CF  0053 endproc;
03D0  0054
03D0  0055
```

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

```
03D0  0056 procedure put_hex_address(integer outchar);
03D0  0057    put_hex_byte(swap(outchar)); /* FIRST TWO DIGITS */
03DA  0058    put_hex_byte(byte(outchar)); /* LAST TWO DIGITS */
03E2  0059 endproc;
```

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

PROCEDURES:

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

DATA:

EXTERNALS:

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

GLOBALS:

erflag	0000	BYTE
keychar	0001	BYTE

8.04.00 BITIO LIBRARY

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

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

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

The bit positions are numbered as follows:

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

8.04.01 BITSIN

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

```
INTVAR=BITSIN;
```

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

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

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

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

8.04.02 BITSOUT

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

```
BITSOUT(INTVAR);
```

8.04.03 BITIN

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

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

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

8.04.04 BITOUT

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

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

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

8.04.05 BITZ8IN

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

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

```
AT $E004: BYTE ACIA_STATUS;

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

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

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

                OR

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

8.04.06 BITZ16IN

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

8.04.07 BITZ80UT

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

```
BITZ80UT(.PORT,6,1);   or   BITZ80UT .PORT,6 = 1;
```

8.04.08 BITZ160UT

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

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

```
0000 0001 /*  BIT-IO LIBRARY  V:4.00  */
0000 0002
0000 0003
0000 0004 include 0.iosubs.lib;
02E8 0005
02E8 0006 constant zero = $30,
02E8 0007          one  = $31;
02E8 0008
02E8 0009 integer mask $0001,$0002,$0004,$0008,$0010,$0020,$0040,$0080,
02F4 0010             $0100,$0200,$0400,$0800,$1000,$2000,$4000,$8000;
0308 0011
0308 0012
0308 0013 procedure bitsin:byte count,inchar:integer bitchar;
030A 0014    count = 16;
030E 0015    bitchar = 0;
0313 0016    repeat
0313 0017       repeat
0313 0018         inchar = getchar_noecho;
0318 0019       until inchar = zero  .or  inchar = one
0326 0020       putchar(inchar); /* ECHO 0/1 */
033B 0021       if inchar = one
033D 0022          then bitchar = bitchar or mask(count - 1);
035B 0023       putchar($20);
0364 0024       count = count - 1;
0366 0025   until count = 0;
036C 0026 endproc bitchar;
0371 0027
0371 0028
0371 0029 procedure bitsout(integer bitchar):byte count;
0373 0030   count = 16;
0377 0031   repeat
0377 0032      if bitchar and mask(count - 1)  /* IMPLICIT <> 0 */
038A 0033         then putchar(one);
039E 0034         else putchar(zero);
03AA 0035      putchar($20);
03B3 0036      count = count - 1;
03B5 0037    until count = 0;
03BB 0038 endproc;
03BE 0039
03BE 0040
03BE 0041 procedure bitin(integer data:byte position):byte status;
03C0 0042    if data and mask(position) /* IMPLICIT <> 0 */
03D1 0043       then status = 1;
03E0 0044       else status = 0;
03E5 0045 endproc status;
03EA 0046
03EA 0047
03EA 0048 procedure bitout(integer data:byte position, status);
03EA 0049    if status    /* IMPLICIT <> 0 */
03EA 0050       then data = data or mask(position);
0408 0051       else data = data and not(mask(position));
0424 0052 endproc integer data;
0427 0053
0427 0054
```

```
0427  0055 procedure bitz8in(byte .data:byte position):byte status;
0429  0056    if data and mask(position) /* IMPLICIT <> 0 */
043C  0057       then status = 1;
044B  0058       else status = 0;
0450  0059 endproc status;
0455  0060
0455  0061
0455  0062 procedure bitz16in(integer .data:byte position):byte status;
0457  0063    if data and mask(position)  /* IMPLICIT <> 0 */
0469  0064       then status = 1;
0478  0065       else status = 0;
047D  0066 endproc;
0480  0067
0480  0068
0480  0069 procedure bitz8out(byte .data:byte position, status);
0480  0070    if status  /* IMPLICIT <> 0 */
0480  0071       then data = data or mask(position);
04A1  0072       else data = data and not(mask(position));
04C0  0073 endproc;
04C1  0074
04C1  0075
04C1  0076 procedure bitz16out(integer .data:byte position, status);
04C1  0077    if status  /* IMPLICIT <> 0 */
04C1  0078       then data = data or mask(position);
04E1  0079       else data = data and not(mask(position));
04FF  0080 endproc;
```

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

PROCEDURES:

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

DATA:

mask	02E8	INTEGER

EXTERNALS:

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

GLOBALS:

8.05.00 HARDIO LIBRARY

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

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

8.05.01 PEEK

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

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

The first and last forms are the most common use of PEEK as the one in the middle can be constructed as 'BVAR=PORT;' In the last example ADDRESS would be an INTEGER variable maintained/updated by a procedure and therefore there is no way of telling what its value is. In this instance we are passing the VALUE contained in the variable ADDRESS NOT the memory location of ADDRESS.

8.05.02 DPEEK

This routine is passed a pointer to an INTEGER variable and returns the contents of the memory location, i.e. it returns a 16-bit variable. The syntax is identical to PEEK.

8.05.03 POKE

This routine is passed a pointer to a BYTE variable and a BYTE value to be placed in the memory location pointed to:

```
POKE($1000,$FF);    or    POKE(ADDRESS,BVAR);
```

In the second example ADDRESS and BVAR would be an INTEGER variable and a BYTE variable, respectively, which would be updated by a procedure.

8.05.04 DPOKE

This routine is passed a pointer to an INTEGER variable and an INTEGER value to be placed in the memory location pointed to:

```
DPOKE($1000,$1234);    or    DPOKE(ADDRESS,IVAR);
```

Page 1: HARD-IO LIBRARY V:4.00 June 1 1984

```
0000 0001 /* HARD-IO LIBRARY V:4.00 */
0000 0002
0000 0003
0000 0004 procedure peek(byte .pointer);
0003 0005 endproc pointer;
0007 0006
0007 0007
0007 0008 procedure dpeek(integer .pointer);
0007 0009 endproc pointer;
000B 0010
000B 0011
000B 0012 procedure poke(byte .pointer, char);
000B 0013    pointer = char;
0010 0014 endproc;
0011 0015
0011 0016
0011 0017 procedure dpoke(integer .pointer, char);
0011 0018    pointer = char;
0016 0019 endproc;
```

Page 2: HARD-IO LIBRARY V:4.00 June 1 1984

PROCEDURES:

peek	0003	BYTE
dpeek	0007	INTEGER
poke	000B	
dpoke	0011	

DATA:

EXTERNALS:

GLOBALS:

8.06.00 STRSUBS (STRING SUBROUTINES) LIBRARY

This package provides you with a set of minimal functions to perform string handling and is based on the string functions usually supplied with C compilers. If you are a BASIC programmer have a look at the library module called 'BASTRING.LIB', as it contains functions to perform operations similar to LEFT$, RIGHT$ and MID$.

Strings in PL/9 can be of any length (they are NOT limited to 255 characters) and are by convention terminated in a null. Strings generated by PL/9 always have the null; you don't have to specify one yourself. You can tell PL/9 to use another character in lieu of the null when you run the SETPL9 program if this is what your application requries. If you do this you MUST change all of the library routines that expect to find a null at the end of a string, e.g. 'PRINT' in IOSUBS.

In the following statement:

```
PRINT "HELLO";
```

the string generated is six characters long; five for the word and one for the null at the end. At the risk of being tedious, I should remind you that when you declare a buffer that is to be used to hold a string, always dimension it to one larger than the longest string you will want to put in it.

8.06.01 STRLEN(.STRING)

Measures the supplied string and returns its length as an INTEGER. The argument is shown here as a pointer, but as long as it is 16 bits and points to the string in question it may take any of the following forms:

```
IVAR=STRLEN(.STRING);
POINTER=.STRING;  IVAR=STRLEN(POINTER);
IVAR=STRLEN("How long is this string?");
```

The function does not count the null when measuring it and is smart enough to cope with a null string, returning zero.

8.06.02 STRCOPY(ARG1, ARG2)

Copies the string represented by the SECOND argument into the location pointed to by the FIRST. Note the order; if you don't like it then feel free to re-write the function. If the buffer allocated for ARG1 is smaller than the actual length of ARG2 then the program will happily walk all over your valuable data or eat its own return address, so BE CAREFUL!

8.06.03 STRCAT(ARG1, ARG2)

Puts the second string onto the end of the first. It calls STRLEN and STRCOPY and is a good example of how to write cryptic PL/9 programs! Unfortunately, to make the function more readable also makes it larger and slower, a sad fact of life. Again, no checks are made that the resulting string will fit into the space available. (What do you expect from a FREE library pack?)

8.06.04 STRCMP(ARG1, ARG2)

Compares two strings, for length and for content. The result of the comparison is either -1, 0 or 1 according to these rules:

If ARG1 = ARG2 then return 0.
If ARG1 alphabetically succeeds ARG2 then return 1.
If ARG1 alphabetically preceeds ARG2 then return -1.

NOTE: 'FULL' will preceed 'FULLY'.

8.06.05 STRPOS(ARG1, ARG2)

Searches ARG1 to see if it can find ARG2 contained within. If it can, it returns the position (zero upwards) in ARG1 at which it found the start of ARG2. If ARG1 does not contain ARG2 then the value -1 is returned.

Page 1: STRING FUNCTIONS LIBRARY V:4.00 June 1 1984

```
0000  0001 /*  STRING FUNCTIONS LIBRARY  V:4.00  */
0000  0002
0000  0003
0000  0004 procedure strlen(byte .string): integer len;
0005  0005    len = 0;
000A  0006    while string(len) /* IMPLICIT <> 0 (NULL) */
0012  0007       len = len + 1;
001E  0008 endproc len;
0025  0009
0025  0010
0025  0011 procedure strcopy(byte .string1, .string2): byte .return_value;
0027  0012    .return_value = .string1;
002B  0013    repeat
002B  0014       string1 = string2;
0031  0015       .string1 = .string1 + 1;
0038  0016       .string2 = .string2 + 1;
003F  0017    until string1(-1) = 0;
004C  0018 endproc .return_value;
0051  0019
0051  0020
0051  0021 procedure strcat(byte .string1, .string2);
0051  0022    strcopy(.string1(strlen(.string1)),.string2);
0067  0023 endproc .string1;
006A  0024
006A  0025
006A  0026 procedure strcmp(byte .string1, .string2):
006A  0027    byte c1, c2: integer index;
006C  0028    index = -1;
0071  0029    repeat
0071  0030       index = index + 1;
0078  0031       c1 = string1(index);
0082  0032       c2 = string2(index);
008C  0033       if c1 = 0 .and c2 = 0 then return 0;
00AD  0034       if c1 = 0 then return -1;
00BA  0035       if c2 = 0 then return 1;
00C7  0036    until c1 <> c2;
00CD  0037    if c1 > c2 then return 1;
00DA  0038 endproc -1;
00DF  0039
00DF  0040
00DF  0041 procedure strpos(byte .string1, .string2):
00DF  0042    byte c1, c2: integer i, j, k;
00E1  0043    i = 0;
00E6  0044    repeat
00E6  0045       j = i;
00EA  0046       k = 0;
00EF  0047       repeat
00EF  0048          c1 = string1(j);
00F9  0049          c2 = string2(k);
0103  0050          if c2 = 0 then return i;
0110  0051          if c1 = 0 then return extend(-1);
011E  0052          j = j + 1;
0125  0053          k = k + 1;
012C  0054       until c1 <> c2;
0132  0055       i = i + 1;
0139  0056    forever;
0139  0057 endproc;
```

Page 3: STRING FUNCTIONS LIBRARY V:4.00 June 1 1984

PROCEDURES:

strlen	0003	INTEGER
strcopy	0025	INTEGER
strcat	0051	INTEGER
strcmp	006A	BYTE
strpos	00DF	INTEGER

DATA:

EXTERNALS:

GLOBALS:

8.07.00 BASTRING (BASIC STRING) LIBRARY

These routines are near equivalents of the BASIC string handling functions LEFT$, RIGHT$, etc. Note that most of these routines alter the string they are given as a parameter. If the string is part of the program (as opposed to being an alterable variable) is is not advisable to use one of these functions directly (it would not work at all if the program were in ROM!) Instead you should first use STRCOPY to copy the string to a working buffer (see the example program on the following page). See the REALCON.LIB file for number conversions.

8.07.01 LEFT

This procedure is used in the basic form: X = LEFT(.STRING,N). "LEFT" returns a pointer to the string, truncated to the length specified by 'N' which is defined as an integer.

8.07.02 RIGHT

This procedure is used in the basic form: X = RIGHT(.STRING,N). "RIGHT" returns a pointer to the last N characters of the specified string.

8.07.03 MID

This procedure is used in the basic form: X = MID(.STRING,N,M). "MID" returns a pointer to M characters of the string, starting at character N. To do the equivalent of BASIC's MID$(A$,I) use "MID(.STRING,N,32767)".

8.07.04 SAMPLE PROGRAM

The following program uses all of the functions in this library and should assist the user in generating his own constructions using them.

```
INCLUDE 0.IOSUBS.LIB;
INCLUDE 0.STRSUBS.LIB;
INCLUDE 0.BASTRING.LIB;

PROCEDURE EXAMPLE: BYTE .X, .Y, .Z, BUF(80);
   STRCOPY(.BUF,"ABCDEFGHIJKLMNOPQRSTUVWXYZ");
   CRLF;
   PRINT .BUF;
   SPACE 10;
   PRINT "ORIGINAL STRING\N";
   .X=LEFT(.BUF,18);
   PRINT .X;
   SPACE 18;
   PRINT ".X=LEFT(.BUF,18);\N";
   .Y=RIGHT(.X,10);
   PRINT .Y;
   SPACE 26;
   PRINT ".Y=RIGHT(.X,10);\N";
   .Z=MID(.Y,3,5);
   PRINT .Z;
   SPACE 31;
   PRINT ".Z=MID(.Y,3,5);\N";
   STRCOPY(.BUF,"ABCDEFGHIJKLMNOPQRSTUVWXYZ");
   CRLF;
   PRINT .BUF;
   SPACE 10;
   PRINT "Original String\N";
   PRINT MID(RIGHT(LEFT(.BUF,18),10),3,5);
   SPACE 31;
   PRINT "MID(RIGHT(LEFT(.BUF,18),10),3,5);\N";
ENDPROC;
```

Page 1: "BASIC" STRING FUNCTIONS LIBRARY V:4.00 June 1 1984

```
0000  0001 /*  "BASIC" STRING FUNCTIONS LIBRARY  V:4.00  */
0000  0002
0000  0003
0000  0004 include 0.strsubs.lib;
013E  0005
013E  0006
013E  0007 procedure left(byte .string: integer n);
013E  0008    if n < 0
0140  0009       then n = 0;
014C  0010    if strlen(.string) > n
0155  0011       then string(n) = 0;
0163  0012 endproc .string;
0166  0013
0166  0014
0166  0015 procedure right(byte .string: integer n):integer l;
0168  0016    if n < 0
016A  0017       then n = 0;
0176  0018    l = strlen(.string);
0181  0019    if n > l
0183  0020       then n = l;
018D  0021 endproc .string(l - n);
019A  0022
019A  0023
019A  0024 procedure mid(byte .string: integer n, m):integer l;
019C  0025    l = strlen(.string);
01A7  0026    n = n - 1;
01AE  0027    if n < 0
01B0  0028       then n = 0;
01BC  0029    if n > l
01BE  0030       then n = l;
01C8  0031    if m < 0
01CA  0032       then m = 0;
01D6  0033    if n + m > l
01DA  0034       then m = l - n;
01E6  0035    string(n + m) = 0;
01F0  0036 endproc .string(n);
```

Page 2: "BASIC" STRING FUNCTIONS LIBRARY V:4.00 June 1 1984

PROCEDURES:

strlen	0003	INTEGER
strcopy	0025	INTEGER
strcat	0051	INTEGER
strcmp	006A	BYTE
strpos	00DF	INTEGER
left	013E	INTEGER
right	0166	INTEGER
mid	019A	INTEGER

DATA:

EXTERNALS:

GLOBALS:

8.08.00 FLEX LIBRARY

This library file contains procedures that help you to write programs that interface with the FLEX operating system. They are all ASMPROCs, generated by the MACE assembler, and perform the functions indicated. This library, like all of the other library modules, has been processed by 'LCASE' to convert the file to lower case. We mention this because MACE will normally generate upper-case 'GEN' statements and hex numbers. You may alter any of these routines to your own requirements or add more routines that you find useful. The MACE assembler (no apologies for the plug) has facilities for generating ASMPROCs which may save you some work.

8.08.01 FLEX

This is a routine that when called (just use the single word FLEX;) returns you to the operating system via the warm start entry point at $CD03. You could just as easily use JUMP $CD03; (but NOT JUMP FLEX; unless you remove the ASMPROC and put in CONSTANT FLEX=$CD03;). This routine is identical in function to the routine 'WARMS' in the IOSUBS library.

8.08.02 GET_FILENAME(.FCB)

This procedure requires the FLEX Line Buffer Pointer at $CC14 to contain the starting address of a string of characters that are presumed to be a valid filename. FCB is a 320 byte File Control Block that you must allocate (or you may use the System FCB at $C840-C97F). The ASMPROC calls the FLEX routine that checks the filename for validity and copies it into the name area of the FCB. If there is anything wrong with the filename, the FCB Error byte (the second byte into the FCB) will contain 21 (which will cause ILLEGAL FILE SPECIFICATION to be printed if REPORT_ERROR is called), otherwise the byte will be clear.

8.08.03 SET_EXTENSION(.FCB, CODE)

This procedure allows you to add one of the standard filename extensions (see your FLEX manual) if none was specified during a call to GET_FILENAME.

8.08.04 REPORT_ERROR(.FCB)

This procedure causes FLEX to print an error message dependant upon the value of the second byte of the FCB. If ERRORS.SYS is present on the system drive then an error message will be selected from that file, otherwise a numeric error code will be printed. The best way to use this function is put 'IF FCB(1) THEN REPORT_ERROR;' after every call to one of the other routines in this package. This will cause the error byte to be tested, and if non-zero an error message is printed. In all of the example programs that use FLEX.LIB you will find that at the start of the program there is a declaration:

```
AT $C840: BYTE FCB, ERROR(319); possibly followed by
AT $CC14: INTEGER LINE_POINTER;
```

8.08.04 REPORT_ERROR(.FCB) (continued)

The first of these declares the system FCB by a small trick that defines its length as 320 bytes total, 319 of them as error bytes! The effect, however, is that FCB is 320 bytes long and the second element (FCB(1)) is the error byte. You can then say IF ERROR THEN REPORT_ERROR after any call to a FLEX interface routine.

8.08.05 OPEN_FOR_READ(.FCB)

This procedure asks FLEX to open the file already specified in the FCB (by a call to GET_FILENAME) for reading.

8.08.06 OPEN_FOR_WRITE(.FCB)

This procedure asks FLEX to open a file for writing. The file must not already exist on the disk or an error will result.

8.08.07 SET_BINARY(.FCB)

This procedure tells FLEX that the file just opened is binary, not text.

8.08.08 READ(.FCB)

This is a function that reads the next byte from the (already opened) file specified by FCB. If an attempt is made to READ past the end of the file then the error byte will be set to 8. READ returns a BYTE value, being the character read from the file, so you can use CHAR=READ(.FCB).

8.08.09 WRITE(.FCB, CHAR)

This procedure writes the BYTE value CHAR to the (already open) file specified by FCB.

8.08.10 CLOSE_FILE(.FCB)

This procedure closes the file specified by FCB, whether it has been open for read or write.

8.08.11 READ_SECTOR(.FCB, DRIVE, TRACK, SECTOR)

This is is a low-level routine that reads the sector at DRIVE, TRACK and SECTOR into the data area of the FCB. This routine should be used with caution and only if you understand what you are doing.

8.08.12 WRITE_SECTOR(.FCB, DRIVE, TRACK, SECTOR)

This is the corresponding write routine.

8.08.13 DELETE_FILE(.FCB)

This procedure deletes the specified file, preserving the filename in FCB (which FLEX would otherwise alter). This routine has its main use in opening an already-existing file for write.

8.08.14 RENAME_FILE(.FCB)

This procedure renames the file specified by FCB, using as the new name the contents of the FCB Scratch Bytes (see your FLEX manual). It is assumed that you will already have set up the FCB correctly.

Page 1: FLEX INTERFACE LIBRARY V:4.00 June 1 1984

```
0000 0001 /*  FLEX INTERFACE LIBRARY  V:4.00  */
0000 0002
0000 0003
0000 0004 asmproc flex;
0003 0005    gen $7e,$cd,$03;      /*          JMP   $CD03     */
0006 0006
0006 0007
0006 0008 asmproc get_filename(integer);
0006 0009    gen $ae,$62;          /*          LDX   2,S       */
0008 0010    gen $bd,$cd,$2d;      /*          JSR   $CD2D     */
000B 0011    gen $25,$02;          /*          BCS   *+4       */
000D 0012    gen $6f,$01;          /*          CLR   1,X       */
000F 0013    gen $39;              /*          RTS             */
0010 0014
0010 0015
0010 0016 asmproc set_extension(integer, byte);
0010 0017    gen $ae,$63;          /*          LDX   3,S       */
0012 0018    gen $a6,$62;          /*          LDA   2,S       */
0014 0019    gen $bd,$cd,$33;      /*          JSR   $CD33     */
0017 0020    gen $39;              /*          RTS             */
0018 0021
0018 0022
0018 0023 asmproc report_error(integer);
0018 0024    gen $ae,$62;          /*          LDX   2,S       */
001A 0025    gen $7e,$cd,$3f;      /*          JMP   $CD3F     */
001D 0026
001D 0027
001D 0028 asmproc open_for_read(integer);
001D 0029    gen $ae,$62;          /*          LDX   2,S       */
001F 0030    gen $86,$01;          /*          LDA   #1        */
0021 0031    gen $a7,$84;          /*          STA   ,X        */
0023 0032    gen $7e,$d4,$06;      /*          JMP   FMS       */
0026 0033
0026 0034
0026 0035 asmproc read(integer): byte;
0026 0036    gen $34,$10;          /*          PSHS  X         */
0028 0037    gen $ae,$64;          /*          LDX   4,S       */
002A 0038    gen $bd,$d4,$06;      /*          JSR   FMS       */
002D 0039    gen $1f,$89;          /*          TFR   A,B       */
002F 0040    gen $35,$90;          /*          PULS  X,PC      */
0031 0041
0031 0042
0031 0043 asmproc open_for_write(integer);
0031 0044    gen $ae,$62;          /*          LDX   2,S       */
0033 0045    gen $86,$02;          /*          LDA   #2        */
0035 0046    gen $a7,$84;          /*          STA   ,X        */
0037 0047    gen $7e,$d4,$06;      /*          JMP   FMS       */
003A 0048
003A 0049
003A 0050 asmproc write(integer, byte);
003A 0051    gen $ae,$63;          /*          LDX   3,S       */
003C 0052    gen $a6,$62;          /*          LDA   2,S       */
003E 0053    gen $7e,$d4,$06;      /*          JMP   FMS       */
0041 0054
0041 0055
```

```
0041  0056 asmproc read_sector(integer, byte, byte, byte);
0041  0057     gen $ae,$65;          /*          LDX   5,S       */
0043  0058     gen $86,$09;          /*          LDA   #9        */
0045  0059     gen $a7,$84;          /*          STA   ,X        */
0047  0060     gen $a6,$64;          /*          LDA   4,S       */
0049  0061     gen $a7,$03;          /*          STA   3,X       */
004B  0062     gen $a6,$63;          /*          LDA   3,S       */
004D  0063     gen $e6,$62;          /*          LDB   2,S       */
004F  0064     gen $ed,$88,$1e;      /*          STD   30,X      */
0052  0065     gen $7e,$d4,$06;      /*          JMP   FMS       */
0055  0066
0055  0067
0055  0068 asmproc write_sector(integer, byte, byte, byte);
0055  0069     gen $ae,$65;          /*          LDX   5,S       */
0057  0070     gen $86,$0a;          /*          LDA   #10       */
0059  0071     gen $a7,$84;          /*          STA   ,X        */
005B  0072     gen $a6,$64;          /*          LDA   4,S       */
005D  0073     gen $a7,$03;          /*          STA   3,X       */
005F  0074     gen $a6,$63;          /*          LDA   3,S       */
0061  0075     gen $e6,$62;          /*          LDB   2,S       */
0063  0076     gen $ed,$88,$1e;      /*          STD   30,X      */
0066  0077     gen $7e,$d4,$06;      /*          JMP   FMS       */
0069  0078
0069  0079
0069  0080 asmproc set_binary(integer);
0069  0081     gen $ae,$62;          /*          LDX   2,S       */
006B  0082     gen $86,$ff;          /*          LDA   #$FF      */
006D  0083     gen $a7,$88,$3b;      /*          STA   59,X      */
0070  0084     gen $39;              /*          RTS             */
0071  0085
0071  0086
0071  0087 asmproc close_file(integer);
0071  0088     gen $ae,$62;          /*          LDX   2,S       */
0073  0089     gen $86,$04;          /*          LDA   #4        */
0075  0090     gen $a7,$84;          /*          STA   ,X        */
0077  0091     gen $7e,$d4,$06;      /*          JMP   FMS       */
007A  0092
007A  0093
007A  0094 asmproc delete_file(integer);
007A  0095     gen $ae,$62;          /*          LDX   2,S       */
007C  0096     gen $86,$0c;          /*          LDA   #12       */
007E  0097     gen $a7,$84;          /*          STA   ,X        */
0080  0098     gen $e6,$04;          /*          LDB   4,X       */
0082  0099     gen $bd,$d4,$06;      /*          JSR   FMS       */
0085  0100     gen $e7,$04;          /*          STB   4,X       */
0087  0101     gen $39;              /*          RTS             */
0088  0102
0088  0103
0088  0104 asmproc rename_file(integer);
0088  0105     gen $ae,$62;          /*          LDX   2,S       */
008A  0106     gen $86,$0d;          /*          LDA   #13       */
008C  0107     gen $a7,$84;          /*          STA   ,X        */
008E  0108     gen $7e,$d4,$06;      /*          JMP   FMS       */
```

Page 3: FLEX INTERFACE LIBRARY V:4.00 June 1 1984

```
PROCEDURES:
flex                0003
get_filename        0006
set_extension       0010
report_error        0018
open_for_read       001D
read                0026    BYTE
open_for_write      0031
write               003A
read_sector         0041
write_sector        0055
set_binary          0069
close_file          0071
delete_file         007A
rename_file         0088

DATA:

EXTERNALS:

GLOBALS:
```

8.09.00 SCIPACK LIBRARY

A C K N O W L E D G E M E N T S

The algorithms used in this library were originally published in '68 Micro Journal and were developed by Ronald Anderson. Subsequent improvements by Ron Anderson have used the SINE and ARCTAN coefficents developed by Matt Scudiere. These were also published in '68 Micro Journal

The routines in this pack provide you with the common scientific functions encountered in engineering programs. Their accuracy is usually better than five significant (decimal) digits and they are reasonably fast, bearing in mind that you are working with an 8-bit micro, not a mainframe! Each of the functions returns a REAL value; the argument supplied may be of any type and is converted, as necessary, to REAL.

The technique used is that of polynomial approximation, whereby a polynomial (A+B*X+C*X*X+....) is computed that fits the function concerned over a limited range. Scaling is done to get the argument into the necessary range. This technique is used by nearly all BASIC interpreters and is more code-efficient than any other method, as well as being faster than most. The polynomial has no more than ten terms, and is arranged in such a way as to require only N multiplications and N additions.

The procedure _POLY does most of the work; it is passed the scaled operand and the address of the appropriate coefficient table, with a BYTE value to tell it how many terms to calculate.

8.09.01 LN(ARG)

Computes the natural (Naperian) logarithm of its argument. The argument must be positive and non-zero, otherwise LN returns zero. It is advisable for you to do your own error checking before calling any of these functions.

8.09.02 LOG(ARG)

Calculates the base 10 logarithm of its argument, by calling LN then multiplying by LOG(E).

8.09.03 EXP(ARG)

Exponentiates its argument, which must be between -88 and +87 or overflow will occur. In such a case the function returns the largest or smallest number it can.

8.09.04 ALOG(ARG)

Evaluates the base 10 antilog of the argument, by multiplying by LN(10) then calling EXP.

8.09.05 XTOY(ARG1, ARG2)

Is the only function that requires two arguments. Its purpose is to raise ARG1 to the power of ARG2, which it does by multiplying the natural log of ARG1 by ARG2 and then exponentiating. ARG2 can have any value, but ARG1 must be positive and non-zero. Note that it is possible to find the square root of a number using XTOY(ARG,0.5), but PL/9's built-in SQR function is much faster and somewhat more accurate.

8.09.06 SIN(ARG)

Returns the sine of the argument, which is assumed to be in radians.

8.09.07 COS(ARG)

Returns the cosine of the argument, also assumed to be in radians.

8.09.08 TAN(ARG)

Computes SIN(ARG)/COS(ARG) but does not check for COS(ARG) being zero.

8.09.09 ATN(ARG)

Computes the arctangent of the argument, giving the result in radians.

Page 1: SCIENTIFIC FUNCTIONS PACKAGE V:4.00 June 1 1984

```
0000 0001 /*  SCIENTIFIC FUNCTIONS PACKAGE  V:4.00  */
0000 0002
0000 0003
0000 0004 /*
0000 0005                       ACKNOWLEDGEMENTS
0000 0006                       ===============
0000 0007
0000 0008    The routines in this package were largely written by
0000 0009    Ron Anderson, using data supplied by  Matt  Scudiere
0000 0010    and published in April 1983 Micro Journal.
0000 0011
0000 0012 */
0000 0013
0000 0014
0000 0015 real _pio2 1.5707963;
0007 0016 real _e 2.7182818;
000B 0017 real _log2 0.69314718;
000F 0018
000F 0019 procedure _poly(real op, .table: byte count): real temp;
0011 0020    temp = table(count);
043B 0021    repeat
043B 0022       count = count - 1;
043D 0023       temp = temp * op + table(count);
0460 0024    until count = 0;
0466 0025 endproc real temp;
0470 0026
0470 0027
0470 0028 real log_coeff
0470 0029    0,
0474 0030    0.9999964,
0478 0031   -0.4998741,
047C 0032    0.3317990,
0480 0033   -0.2407338,
0484 0034    0.1676541,
0488 0035   -0.09532939,
048C 0036    0.03608849,
0490 0037   -0.006453544;
0494 0038
0494 0039 procedure ln(real op): byte n;
0496 0040    if op <= 0
04A3 0041       then return real 0;
04B7 0042    gen $e6,$63;   /* LDB 3,S GET EXPONENT OF OP */
04B9 0043    n = accb - 1;  /* ADJUST EXP TO BE 1 */
04BD 0044    accb = 1;
04BF 0045    gen $e7,$63;   /* STB 3,S PUT EXP BACK IN OP */
04C1 0046 endproc real _poly(op - 1, .log_coeff,8) + n * _log2;
04F9 0047
04F9 0048
04F9 0049 procedure log(real op);
04F9 0050 endproc real ln(op) * 0.4342944;
0512 0051
0512 0052
```

Page 2: SCIENTIFIC FUNCTIONS PACKAGE V:4.00 June 1 1984

```
0512  0053 real exp_coeff
0512  0054    0,
0516  0055    0.9999999,
051A  0056    0.4999999,
051E  0057    0.1666700,
0522  0058    0.04165734,
0526  0059    0.00830140,
052A  0060    0.00151500,
052E  0061    0.000116;
0532  0062
0532  0063 procedure exp(real op): real k;
0534  0064    if op > 87
0541  0065       then return real 1E38;
0555  0066    if op < -88
0565  0067       then return real 1E-38;
0579  0068    k = 1;
0586  0069    while op > _log2
058B  0070       begin
0599  0071          op = op - _log2;
05AD  0072          gen $6c,$e4;  /*  INC ,S    K=K*2  */
05AF  0073       end;
05AF  0074    while op < 0
05BE  0075       begin
05C5  0076          op = op + _log2;
05D9  0077          gen $6a,$e4;  /*  DEC ,S    K=K/2  */
05DB  0078       end;
05DB  0079 endproc real (_poly(op, .exp_coeff,7) + 1) * k;
060B  0080
060B  0081
060B  0082 procedure alog(real op);
060B  0083 endproc real exp(op * 2.302585);
0625  0084
0625  0085
0625  0086 procedure xtoy(real op1,op2);
0625  0087    if op2 = 0
0632  0088       then return real 1;
0644  0089    if op1 < 0
0651  0090       then return real 0;
0663  0091 endproc real exp(ln(op1) * op2);
0681  0092
0681  0093
```

```
0681  0094 real sin_coeff
0681  0095    1.0,
0685  0096   -0.1666666,
0689  0097    8.333332E-3,
068D  0098   -1.9852E-4,
0691  0099    2.8255E-6,
0695  0100   -3.70E-8;
0699  0101
0699  0102 procedure sin(real op): byte negative, quadrant;
069B  0103    if op = 0
06A8  0104       then return real 0;
06BC  0105    quadrant = fix(int(op / _pio2));
06D3  0106    op = op - quadrant * _pio2;
06F0  0107    negative = quadrant and 2;  /* NON-ZERO FOR QUADS 2,3 */
06F6  0108    if quadrant and 1           /* IMPLICIT <> 0 */
06F8  0109       then op = _pio2 - op;    /* QUADS 1,3 */
0713  0110    op = op * _poly(op * op, .sin_coeff,5);
073E  0111    if negative                 /* IMPLICIT <> 0 */
073E  0112       then op = -op;
0752  0113 endproc real op;
075C  0114
075C  0115
075C  0116 real cos_coeff
075C  0117    1.0,
0760  0118   -0.5,
0764  0119    0.041666642,
0768  0120   -1.3888397E-3,
076C  0121    2.47609E-5,
0770  0122   -2.605E-7;
0774  0123
0774  0124 procedure cos(real op): byte negative, quadrant;
0776  0125    if op = _pio2
077B  0126       then return real 0;
0796  0127    quadrant = fix(int(op / _pio2));
07AD  0128    op = op - quadrant * _pio2;
07CA  0129    negative = 0;
07CC  0130    if quadrant = 1 .or quadrant = 2
07DA  0131       then negative = 1;
07EC  0132    if quadrant and 1           /* IMPLICIT <> 0 */
07EE  0133       then op = _pio2 - op;
0809  0134    op = _poly(op * op, .cos_coeff,5);
082C  0135    if negative                 /* IMPLICIT <> 0 */
082C  0136       then op = -op;
0840  0137 endproc real op;
084A  0138
084A  0139
084A  0140 procedure tan(real op);
084A  0141 endproc real sin(op) / cos(op);
0868  0142
0868  0143
```

```
0868  0144 real atn_coeff
0868  0145    1.0,
086C  0146   -0.3333315,
0870  0147    0.1999355,
0874  0148   -0.1420890,
0878  0149    0.1065626,
087C  0150   -0.07528964,
0880  0151    0.04290961,
0884  0152   -0.01616574,
0888  0153    0.002866226;
088C  0154
088C  0155 procedure atn(real op): byte sign, reciprocal;
088E  0156    if op < 0
089B  0157       then begin
08A2  0158               op = -op;
08AF  0159               sign = 1;;
08B3  0160            end;
08B3  0161       else sign = 0;
08B8  0162    if op > 1
08C5  0163       then begin
08CC  0164               op = 1/op;
08E1  0165               reciprocal = 1;;
08E5  0166            end;
08E5  0167       else reciprocal = 0;
08EA  0168    op = op * _poly(op * op, .atn_coeff,8);
0915  0169    if reciprocal                 /* IMPLICIT <> 0 */
0915  0170       then op = _pio2 - op;
0930  0171    if sign                       /* IMPLICIT <> 0 */
0930  0172       then op = -op;
0944  0173 endproc real op;
```

Page 5: SCIENTIFIC FUNCTIONS PACKAGE V:4.00 June 1 1984

PROCEDURES:

_poly	000F	REAL
ln	0494	REAL
log	04F9	REAL
exp	0532	REAL
alog	060B	REAL
xtoy	0625	REAL
sin	0699	REAL
cos	0774	REAL
tan	084A	REAL
atn	088C	REAL

DATA:

_pio2	0003	REAL
_e	0007	REAL
_log2	000B	REAL
log_coeff	0470	REAL
exp_coeff	0512	REAL
sin_coeff	0681	REAL
cos_coeff	075C	REAL
atn_coeff	0868	REAL

EXTERNALS:

GLOBALS:

8.10.00 REALCON LIBRARY

This library file contains two routines that are essential when inputing or outputing REAL numbers via the system console. They convert between ASCII strings and the binary format used by PL/9 to hold REAL numbers.

8.10.01 BINARY(.STRINGPTR, .POSITIONPTR)

Is called with two arguments, both BYTE. The first is the address of (i.e. pointer to) the ASCII string that is to be converted into binary. The second argument, also a pointer, is the address of a BYTE location that contains the position in the string at which to start converting. Conversion finished whenever the routine comes across a character that it can't make sense of, for example a space or a null. When this happens, the second argument is left set to the position in the string at which this character was found, allowing the calling program to work its way along a number of items in a buffer.

BINARY returns a REAL value and can therefore be used in any REAL expression in the same way as any other REAL value. Examples of its use:

```
INPUT(.BUFFER,80);              /* Get a line of input */
POSITION = 0;                   /* Set up the starting position */
X=BINARY(.BUFFER,.POSITION);    /* Convert the number */
```

A point to note is that since INPUT returns as its value the first argument it was given, the above can be simplified:

```
POSITION=0;
X=BINARY(INPUT(.BUFFER,80),.POSITION);
```

This combines the calls to the two routines.

If you frequently want to get a line of input and read a single number from it, use the following procedure:

```
PROCEDURE INNUM:BYTE POS, BUF(20);
   POS=0;
ENDPROC REAL BINARY(INPUT(.BUF,20),.POS);
```

See the routine 'FINPUT' in the 'REALIO' library for further information.

8.10.02 ASCBIN(.STRINGPTR)

This routine uses BINARY to convert an ASCII string pointed to by STRINGPTR to a REAL number. This procedure simplifies converting ASCII strings as the starting position pointer is automatically provided.

8.10.03 ASCII(VAR, .BUFFER)

Is the complementary routine that converts a REAL number into an ASCII string in the BYTE buffer whose address is supplied, terminating it with a null. The chief use for this routine is to print out a REAL number, which is achieved by the following procedure:

```
PROCEDURE OUTNUM(REAL X): BYTE BUF(20);
   PRINT(ASCII(X,.BUF));
ENDPROC;
```

See the routine 'FPRINT' in the 'REALIO' library for further information.

As with INPUT, ASCII returns as its "value" the second parameter passed to it; this is in the example then passed directly to PRINT. Note that PL/9 is happy to accept any of the following:

```
   PRINT(ASCII(X,.BUF));                PRINT ASCII(X,.BUF);
   PRINT = ASCII(X,.BUF);
```

```
Page  1:  FLOATING-POINT CONVERSION ROUTINES  V:4.00              June 1 1984

 0000  0001 /*  FLOATING-POINT CONVERSION ROUTINES  V:4.00  */
 0000  0002
 0000  0003
 0000  0004 procedure binary(byte .buffer: byte .posptr):
 0003  0005    real acc: byte flag, msign, esign, exponent, expt, char;
 0005  0006    acc = 0;
 0429  0007    flag = 0;   <---------------- NOTE: SUDDEN  JUMP  IN PROGRAM SIZE
 042B  0008    msign = 0;                          IS CAUSED BY PL/9  GENERATING
 042D  0009    esign = 0;                          THE CODE FOR THE 'REAL' MATHS
 042F  0010    expt = 0;                           RUN TIME LIBRARY.
 0431  0011    exponent = 0;
 0433  0012    if buffer(posptr) = '-
 043D  0013       then begin
 0443  0014               msign = -1;
 0447  0015               posptr = posptr + 1;
 044A  0016            end;
 044A  0017 loop:
 044A  0018    char = buffer(posptr);
 0456  0019    posptr = posptr + 1;
 0459  0020    while char >=  '0 .and char <= '9
 0467  0021       begin
 0475  0022          acc = acc * 10 + (char - '0);
 049C  0023          if flag                    /* IMPLICIT <> 0 */
 049C  0024             then exponent = exponent - 1;
 04A5  0025          char = buffer(posptr);
 04B1  0026          posptr = posptr + 1;
 04B4  0027       end;
 04B4  0028
 04B4  0029    if char = '. .and flag = 0
 04C4  0030       then begin
 04D2  0031               flag = -1;
 04D6  0032               goto loop;
 04D9  0033            end;
 04D9  0034
 04D9  0035    if char = 'E .or char = 'e
 04E7  0036       then begin
 04F5  0037               if buffer(posptr) = '-
 04FF  0038                  then begin
 0505  0039                          esign = -1;
 0509  0040                          posptr = posptr + 1;
 050C  0041                       end;
 050C  0042               char = buffer(posptr);
 0518  0043               posptr = posptr + 1;
 051B  0044               while char >= '0 .and char <= '9
 0529  0045                  begin
 0537  0046                     expt = expt * 10 + char - '0;
 056E  0047                     char = buffer(posptr);
 057A  0048                     posptr = posptr + 1;
 057D  0049                  end;
 057D  0050               if esign              /* IMPLICIT <> 0 */
 057F  0051                  then expt = -expt;
 058E  0052               exponent = exponent + expt;
 0594  0053            end;
 0594  0054
 0594  0055    posptr = posptr - 1;
```

```
0597  0056     while exponent > 0
0599  0057        begin
059F  0058           acc = acc * 10;
05B4  0059           exponent = exponent - 1;
05B6  0060        end;
05B6  0061     while exponent < 0
05BA  0062        begin
05C0  0063           acc = acc/10;
05D5  0064           exponent = exponent + 1;
05D7  0065        end;
05D7  0066     if msign                          /* IMPLICIT <> 0 */
05D9  0067        then acc = -acc;
05ED  0068 endproc real acc;
05F7  0069
05F7  0070
05F7  0071 procedure ascbin(byte .buffer):byte posptr;
05F9  0072    posptr = 0;
05FB  0073 endproc real binary(.buffer, .posptr);
060F  0074
060F  0075
060F  0076 procedure ascii(real op: byte .buffer):
060F  0077    integer pointer:
060F  0078    byte ptr, exponent, count, carry, first, last, digit;
0611  0079    pointer = .op;
0617  0080    exponent = -1;
061B  0081    ptr = 0;
061D  0082    if op = 0
062A  0083       then begin
0631  0084               buffer = '0;
0636  0085               buffer(1) = 0;
063F  0086               return .buffer;
0644  0087            end;
0644  0088    if op < 0
0651  0089       then begin
0658  0090               op = -op;
0666  0091               buffer(ptr) = '-;
0671  0092               ptr = ptr + 1;
0673  0093            end;
0673  0094    first = ptr;
0677  0095    while op >=  1
0684  0096       begin
068B  0097          op = op/10;
06A1  0098          exponent = exponent + 1;
06A3  0099       end;
06A3  0100    while op < 0.1
06B2  0101       begin
06B9  0102          op = op * 10;
06CF  0103          exponent = exponent - 1;
06D1  0104       end;
06D1  0105
```

Page 3: FLOATING-POINT CONVERSION ROUTINES V:4.00 June 1 1984

```
06D1  0106 /* DE-NORMALIZE THE REAL NUMBER */
06D1  0107
06D1  0108    gen $ae,$e4;          /* DENORM   LDX   ,S        */
06D5  0109    gen $68,$03;          /*          ASL   3,X       */
06D7  0110    gen $69,$02;          /*          ROL   2,X       */
06D9  0111    gen $69,$01;          /*          ROL   1,X       */
06DB  0112    gen $6d,$84;          /*          TST   ,X        */
06DD  0113    gen $27,$0a;          /*          BEQ   *+12      */
06DF  0114    gen $64,$01;          /* :1       LSR   1,X       */
06E1  0115    gen $66,$02;          /*          ROR   2,X       */
06E3  0116    gen $66,$03;          /*          ROR   3,X       */
06E5  0117    gen $6c,$84;          /*          INC   ,X        */
06E7  0118    gen $26,$f6;          /*          BNE   :1        */
06E9  0119
06E9  0120 /* GENERATE SEVEN DECIMAL DIGITS BY REPEATED MULTIPLICATION BY TEN */
06E9  0121
06E9  0122    count = 7;
06ED  0123    repeat
06ED  0124       gen $ae,$e4;          /* DIGIT    LDX   ,S        */
06EF  0125       gen $ec,$84;          /*          LDD   ,X        */
06F1  0126       gen $34,$06;          /*          PSHS  D         */
06F3  0127       gen $ec,$02;          /*          LDD   2,X       */
06F5  0128       gen $8d,$16;          /*          BSR   DOUBLE    */
06F7  0129       gen $8d,$14;          /*          BSR   DOUBLE    */
06F9  0130       gen $e3,$02;          /*          ADDD  2,X       */
06FB  0131       gen $ed,$02;          /*          STD   2,X       */
06FD  0132       gen $35,$06;          /*          PULS  D         */
06FF  0133       gen $e9,$01;          /*          ADCB  1,X       */
0701  0134       gen $a9,$84;          /*          ADCA  ,X        */
0703  0135       gen $ed,$84;          /*          STD   ,X        */
0705  0136       gen $8d,$06;          /*          BSR   DOUBLE    */
0707  0137       gen $e6,$84;          /*          LDB   ,X        */
0709  0138       gen $6f,$84;          /*          CLR   ,X        */
070B  0139       gen $20,$09;          /*          BRA   *+11      */
070D  0140
070D  0141       gen $68,$03;          /* DOUBLE   ASL   3,X       */
070F  0142       gen $69,$02;          /*          ROL   2,X       */
0711  0143       gen $69,$01;          /*          ROL   1,X       */
0713  0144       gen $69,$84;          /*          ROL   ,X        */
0715  0145       gen $39;              /*          RTS             */
0716  0146
0716  0147       digit = accb;
0718  0148       buffer(ptr) = digit + '0;
0725  0149       ptr = ptr + 1;
0727  0150       count = count - 1;
0729  0151    until count = 0;
072F  0152
072F  0153    buffer(ptr) = 0;
0738  0154    ptr = ptr - 1;
073A  0155    last = ptr;
073E  0156    carry = 0;
0740  0157    if buffer(ptr) >= '5
0749  0158       then carry = 1;
```

```
0753  0159    repeat
0753  0160       ptr = ptr - 1;
0755  0161       buffer(ptr) = buffer(ptr) + carry;
0762  0162       if buffer(ptr) > '9
076B  0163          then buffer(ptr) = '0;
077C  0164          else carry = 0;
0781  0165    until carry = 0 .or ptr = first;
079B  0166
079B  0167    if carry                          /* IMPLICIT <> 0 */
079B  0168       then begin
07A2  0169               count = last + 1;
07A8  0170               repeat
07A8  0171                  buffer(count+1) = buffer(count);
07BE  0172                  count = count - 1;
07C0  0173               until count < ptr;
07C6  0174               buffer(ptr) = '1;
07D1  0175               exponent = exponent + 1;
07D3  0176            end;
07D3  0177    buffer(last) = 0;
07DC  0178
07DC  0179    if exponent > 5 .or exponent < -2
07EA  0180       then begin
07F8  0181               count = last + 1;
07FE  0182               repeat
07FE  0183                  buffer(count+1) = buffer(count);
0814  0184                  count = count - 1;
0816  0185               until count = first;
081C  0186               buffer(count+1) = '.;
0829  0187               while buffer(last) = '0
0832  0188                  last = last - 1;
083A  0189               if buffer(last) = '.
0845  0190                  then last = last - 1;
084D  0191               buffer(last+1) = 'E;
085A  0192               last = last + 2;
0860  0193               if exponent < 0
0862  0194                  then begin
0868  0195                          exponent = -exponent;
0870  0196                          buffer(last) = '-;
087B  0197                          last = last + 1;
087D  0198                       end;
087D  0199               if exponent > 9
087F  0200                  then begin
0885  0201                          buffer(last) = exponent/10 + '0;
08FA  0202                          last = last + 1;
08FC  0203                       end;
08FC  0204               buffer(last) = exponent - exponent/10 * 10 + '0;
0925  0205               buffer(last+1) = 0;
0930  0206               return .buffer;
0935  0207            end;
0935  0208
```

```
0935  0209       else if exponent >= 0
093A  0210               then begin
0940  0211                       ptr = first + 1;
0946  0212                       while exponent > 0
0948  0213                          begin
094E  0214                             ptr = ptr + 1;
0950  0215                             exponent = exponent - 1;
0952  0216                          end;
0952  0217                       count = last;
0958  0218                       while count >= ptr
095A  0219                          begin
0960  0220                             buffer(count+1) = buffer(count);
0976  0221                             count = count - 1;
0978  0222                          end;
0978  0223                       buffer(ptr) = '.;
0985  0224                    end;
0985  0225
0985  0226               else begin
0988  0227                       if exponent = -2
098A  0228                          then begin
0990  0229                                  count = last + 1;
0996  0230                                  repeat
0996  0231                                     buffer(count+1) = buffer(count);
09AC  0232                                     count = count - 1;
09AE  0233                                  until count < first;
09B4  0234                                  last = last + 1;
09B6  0235                                  buffer(first) = '0;
09C1  0236                               end;
09C1  0237                       count = last + 1;
09C7  0238                       repeat
09C7  0239                          buffer(count+2) = buffer(count);
09DD  0240                          count = count - 1;
09DF  0241                       until count < first;
09E5  0242                       last = last + 1;
09E7  0243                       buffer(first) = '0;
09F2  0244                       buffer(first+1) = '.;
09FF  0245                    end;
09FF  0246
09FF  0247    while buffer(last) = '0
0A08  0248       begin
0A0E  0249          buffer(last) = 0;
0A17  0250          last = last - 1;
0A19  0251       end;
0A19  0252    if buffer(last) = '.
0A24  0253       then buffer(last) = 0;
0A33  0254 endproc .buffer;
```

Page 6: FLOATING-POINT CONVERSION ROUTINES V:4.00 June 1 1984

PROCEDURES:

binary	0003	REAL
ascbin	05F7	REAL
ascii	060F	INTEGER

DATA:

EXTERNALS:

GLOBALS:

8.11.00 REAL I/O LIBRARY

This library provides two procedures to simplify console input and output of REAL numbers. These procedures are the two procedures outlined in the description of the REALCON library. The reason we placed these two procedures in a separate library was to maintain compatibility with older versions of the compiler libraries.

8.11.01 FINPUT

This procedure will expect to get a line of input from the terminal representing a real number. e.g. 23456932, 0.123456, -456.986, 2.47609E-5, -1.388457E-3, etc. The ASCII string input will be returned by this procedure as a REAL number which may be evaluated or assigned as required by your application.

8.11.02 FPRINT

This procedure does the inverse of the above. It is passed a REAL number and prints it out on the system console. If the REAL number is greater than or equal to 1,000,000 or less than 0.01 the response will be scientific notation. If the number falls between these limits the response will be in decimal with all trailing zeros truncated (100.0000) will be 100.

```
Page  1:  FLOATING-POINT INPUT/OUTPUT ROUTINES  V:4.00          June 1 1984

 0000  0001 /*  FLOATING-POINT INPUT/OUTPUT ROUTINES  V:4.00  */
 0000  0002
 0000  0003
 0000  0004 include 0.iosubs.lib;
 02E8  0005 include 0.realcon.lib;
 0CC0  0006
 0CC0  0007
 0CC0  0008 procedure finput: byte ptr, buffer(20);
 0CC3  0009    ptr = 0;
 0CC5  0010 endproc real binary(input(.buffer, 20), .ptr);
 0CE5  0011
 0CE5  0012
 0CE5  0013 procedure fprint(real op): byte buffer(20);
 0CE8  0014    print(ascii(op, .buffer));
 0CFE  0015 endproc;
```

Page 2: FLOATING-POINT INPUT/OUTPUT ROUTINES V:4.00 June 1 1984

PROCEDURES:

monitor	0003	
warms	0008	
getchar	000C	BYTE
getchar_noecho	0012	BYTE
getkey	0019	BYTE
convert_lc	0031	BYTE
get_uc	0056	BYTE
get_uc_noecho	0063	BYTE
putchar	0070	
printint	0078	
remove_char	011D	
input	0139	INTEGER
crlf	01DD	
print	01F0	
space	02CC	
binary	02E8	REAL
ascbin	08DC	REAL
ascii	08F4	INTEGER
finput	0CC0	REAL
fprint	0CE5	

DATA:

EXTERNALS:

nul	0000
abt	0003
bel	0007
bs	0008
lf	000A
cr	000D
can	0018
esc	001B
sp	0020

GLOBALS:

8.12.00 NUMCON LIBRARY

This library provides procedures to simplify console input and output of INTEGERs.

8.12.01 BINTODEC

BINTODEC is an assembler routine that converts a supplied integer into an ASCII decimal number between 0 and 65535 and puts it into the buffer whose address is supplied, terminating it with a NULL.

8.12.02 PRDEC

PRDEC prints an integer as a decimal number between 0 and 65535.

8.12.03 PRNUM

PRNUM prints an integer as a signed decimal number between -32768 and 32767.

8.12.04 GETNUM

GETNUM works its way along the text in the buffer whose address is supplied and returns the number contained therein. The number may have a leading minus sign; a $ signifies hexadecimal. Examples: 25 -9003 $1234 -$A7 0 50817.

8.12.05 TRY THIS

If you want to get some insight into how these routines work try entering the following and running it under the PL/9 tracer:

```
0001 include 0.trufalse.def;
0002 include 0.iosubs.lib;
0003 include 0.numcon.lib;
0004
0005 procedure test:byte buffer(20):integer i;
0006     i = getnum(input(.buffer,20));
0007     crlf; crlf;
0008     prnum(i);
0009     space(5);
0010     prdec(i);
```

Page 1: NUMERICAL CONVERSION AND I/O ROUTINES V:4.00 June 1 1984

```
0000  0001 /*  NUMERICAL CONVERSION AND I/O ROUTINES  V:4.00  */
0000  0002
0000  0003
0000  0004 include 0.trufalse.def;
0000  0005 include 0.iosubs.lib;
02E8  0006
02E8  0007
02E8  0008 asmproc bintodec(integer,integer);
02E8  0009    gen $cc,$00,$04;     /*          LDD   #$0004     */
02EB  0010    gen $34,$26;         /*          PSHS  D,Y        */
02ED  0011    gen $ae,$66;         /*          LDX   6,S        */
02EF  0012    gen $ec,$68;         /*          LDD   8,S        */
02F1  0013    gen $31,$8c,$2b;     /*          LEAY  TABLE,PCR  */
02F4  0014    gen $6f,$84;         /* L1       CLR   ,X         */
02F6  0015    gen $63,$84;         /*          COM   ,X         */
02F8  0016    gen $6c,$84;         /* L2       INC   ,X         */
02FA  0017    gen $a3,$a4;         /*          SUBD  ,Y         */
02FC  0018    gen $24,$fa;         /*          BCC   L2         */
02FE  0019    gen $e3,$a1;         /*          ADDD  ,Y++       */
0300  0020    gen $34,$04;         /*          PSHS  B          */
0302  0021    gen $e6,$84;         /*          LDB   ,X         */
0304  0022    gen $26,$04;         /*          BNE   L3         */
0306  0023    gen $6d,$61;         /*          TST   1,S        */
0308  0024    gen $27,$04;         /*          BEQ   L4         */
030A  0025    gen $8d,$0e;         /* L3       BSR   L5         */
030C  0026    gen $6c,$61;         /*          INC   1,S        */
030E  0027    gen $35,$04;         /* L4       PULS  B          */
0310  0028    gen $6a,$61;         /*          DEC   1,S        */
0312  0029    gen $26,$e0;         /*          BNE   L1         */
0314  0030    gen $8d,$04;         /*          BSR   L5         */
0316  0031    gen $6f,$84;         /*          CLR   ,X         */
0318  0032    gen $35,$a6;         /*          PULS  D,Y,PC     */
031A  0033    gen $cb,$30;         /* L5       ADDB  #$30       */
031C  0034    gen $e7,$80;         /*          STB   ,X+        */
031E  0035    gen $39;             /*          RTS              */
031F  0036    gen $27,$10,$03,$e8; /* TABLE    FCB   $27,$10,$03,$E8 */
0323  0037    gen $00,$64,$00,$0a; /*          FCB   $00,$64,$00,$0A */
0327  0038
0327  0039
0327  0040 procedure prdec(integer n): byte buffer(6);
0329  0041    bintodec(n, .buffer);
0335  0042    print(.buffer);
033E  0043 endproc;
0341  0044
0341  0045
0341  0046 procedure prnum(integer n): byte buffer(6);
0343  0047    if n < 0
0345  0048       then begin
034C  0049               putchar('-);
0355  0050               n = -n;
035D  0051            end;
035D  0052    bintodec(n, .buffer);
0369  0053    print(.buffer);
0372  0054 endproc;
0375  0055
```

```
0375  0056
0375  0057 procedure getnum(byte .buffer):
0375  0058    integer ptr, n:
0375  0059    byte sign, base, char, done;
0377  0060    ptr = 0;
037C  0061    sign = false;
037E  0062    n = 0;
0383  0063    base = 10;
0387  0064    done = false;
0389  0065    while buffer(ptr) = sp
0391  0066       ptr = ptr + 1;
039E  0067    if buffer(ptr) = '-
03A8  0068       then begin
03AE  0069               sign = true;
03B2  0070               ptr = ptr + 1;
03B9  0071            end;
03B9  0072    if buffer(ptr) = '$
03C1  0073       then begin
03C7  0074               base = 16;
03CB  0075               ptr = ptr + 1;
03D2  0076            end;
03D2  0077    repeat
03D2  0078       char = buffer(ptr) - '0;
03DE  0079       if char > 9 .and char < 17
03EC  0080          then done = true;
03FE  0081       if char > 16
0400  0082          then if base = 10
0408  0083                  then done = true;
0412  0084                  else char = char - 7;
041B  0085       if char < 0 .or char > $f
0429  0086          then done = true;
043B  0087       if not(done)              /* IMPLICIT <> 0 */
043E  0088          then begin
0443  0089                  n = n * base + char;
0476  0090                  ptr = ptr + 1;
047D  0091               end;
047D  0092    until done;
0484  0093    if sign                      /* IMPLICIT <> 0 */
0484  0094       then n = -n;
0493  0095 endproc n;
```

Page 3: NUMERICAL CONVERSION AND I/O ROUTINES V:4.00 June 1 1984

PROCEDURES:

monitor	0003	
warms	0008	
getchar	000C	BYTE
getchar_noecho	0012	BYTE
getkey	0019	BYTE
convert_lc	0031	BYTE
get_uc	0056	BYTE
get_uc_noecho	0063	BYTE
putchar	0070	
printint	0078	
remove_char	011D	
input	0139	INTEGER
crlf	01DD	
print	01F0	
space	02CC	
bintodec	02E8	
prdec	0327	
prnum	0341	
getnum	0375	INTEGER

DATA:

EXTERNALS:

true	FFFF	
false	0000	
mem	0000	BYTE
nul	0000	
abt	0003	
bel	0007	
bs	0008	
lf	000A	
cr	000D	
can	0018	
esc	001B	
sp	0020	

GLOBALS:

8.13.00 SORT LIBRARY

A C K N O W L E D G E M E N T

These routines are based on a series of sorting algorithms presented in the May 1983 issue of BYTE magazine, from page 482 onwards.

This library is provided as a primitive to enable you to develop more complex sorting routines. As supplied the library sorts a VECTOR of REAL numbers into ascending order. It can be easily modified to sort a VECTOR of INTEGERs or a VECTOR of BYTEs.

The standard form is:

```
SHELLSORT(.VECTOR, NUMBER_OF_ELEMENTS_TO_BE_SORTED);
```

Page 1: SHELL SORT LIBRARY ROUTINE V:4.00 June 1 1984

```
0000  0001 /*  SHELL SORT LIBRARY ROUTINE  V:4.00  */
0000  0002
0000  0003
0000  0004 procedure shellsort(real .data: integer size):
0003  0005    integer i, j, d:
0003  0006    real temp;
0005  0007    i = size;
0009  0008    d = 16383;
000E  0009    while i < 16384
0010  0010       begin
0017  0011          d = shift(d,-1);
001D  0012          i = shift(i,1);
0023  0013       end;
0023  0014    repeat
0025  0015       i = 0;
002A  0016       repeat
002A  0017          j = i;
002E  0018          repeat
002E  0019             if data(j) <= data(j + d)
0462  0020                then break;
046B  0021             temp = data(j);
047D  0022             data(j) = data(j + d);
04A0  0023             data(j + d) = temp;
04BA  0024             j = j - d;
04C0  0025          until j < 0;
04C9  0026          i = i + 1;
04D0  0027       until i = size - d;
04E2  0028       d = shift(d,-1);
04E8  0029    until d = 0;
04F1  0030 endproc;
```

Page 2: SHELL SORT LIBRARY ROUTINE V:4.00 June 1 1984

PROCEDURES:
shellsort 0003

DATA:

EXTERNALS:

GLOBALS:

THIS PAGE INTENTIONALLY LEFT BLANK