NASA Technical Memorandum 85752

Pascal/48 Reference Manual

January 1984

Central Scientific Computing Complex Document M-3

NASA
National Aeronautics and Space Administration

Langley Research Center
Hampton, Virginia 23665

NASA Technical Memorandum 85752

Central Scientific
Computing Complex
Document M-3

P A S C A L / 4 8

R E F E R E N C E   M A N U A L

John C. Knight
Analysis and Computation Division
May 1981

Revised
Roy W. Hamm
Computer Sciences Corporation
January 1984

PASCAL/48 REFERENCE MANUAL

1. INTRODUCTION AND OVERVIEW 1

2. LEXICAL DEFINITIONS 2

3. CONSTANTS AND VARIABLES 4

4. PROGRAM STRUCTURE 7
   - 4.1 Label Declaration Part 9
   - 4.2 Constant Definition Part 10
   - 4.3 Variable Declaration Part 11
   - 4.4 Value Part 14
   - 4.5 Procedure Declaration Part 16
   - 4.6 Compound Statement 19

5. OPERATORS 20

6. EXPRESSIONS 23

7. PREDECLARED VARIABLES 26

8. STATEMENTS 28
   - 8.1 Assignment Statement 29
   - 8.2 IF Statement 30
   - 8.3 WHILE Statement 32
   - 8.4 REPEAT Statement 34
   - 8.5 FOR Statement 35
   - 8.6 CASE Statement 37
   - 8.7 GOTO Statement 39
   - 8.8 Procedure Call Statement 40
   - 8.9 Predeclared Procedures 41

9. INTERRUPTS 43

10. USE OF THE COMPILER 45

11. APPENDIX A - SAMPLE PROGRAMS 48

12. APPENDIX B - SAMPLE LISTING 60

13. APPENDIX C - SYNTAX DIAGRAMS 70

14. APPENDIX D - ERROR MESSAGES 77

15. INDEX 85

1 INTRODUCTION AND OVERVIEW

Pascal/48 is a programming language for the Intel MCS-48 series of microcomputers. In particular, it can be used with the Intel 8748. It is designed to allow the programmer to control most of the instructions being generated and the allocation of storage. The language has sufficient expressive power that it can be used instead of assembly language in most applications while allowing the user to exercise the necessary degree of control over hardware resources.

A thorough knowledge of the MCS-48 hardware (Microcontroller Handbook, Intel Corporation, May 1983) is assumed in this manual. Some knowledge of Pascal would be useful.

Although it is called Pascal/48, the language differs in many ways from Pascal. The program structure and statements of the two languages are similar. The major differences are in the expression mechanism and the data types.

The syntax diagrams in this manual precisely describe the syntax of the Pascal/48 language. Upper case quantities and special symbols indicate items which are actually written in a Pascal/48 program and lower case quantities are names of other diagrams. They must be located and used to generate valid instances of the named diagram. Below is the diagram for 'program' (see also section 4) and it indicates that a Pascal/48 program consists of the word PROGRAM followed by an ident, followed by a semicolon, followed by a block, followed by a period. Ident and block are names of other diagrams because they are in lower case. Starting with the diagram named 'program', any path through the diagram defines a syntactically correct program.

```
 <<<  program  >>>

---> PROGRAM ---> ident ---> ; ---> block ---> . ----->
```

For reference purposes, all of the syntax diagrams are repeated in Appendix C.

2 LEXICAL DEFINITIONS

Pascal/48 is a free format language. Column numbers have no meaning except that the compiler only reads columns 1 to 72 inclusive of the input records (usually card images). The number of columns read can be changed with a compiler option (see section 10). Blanks may not be used inside identifiers, reserved words, constants or multi-character operators but otherwise can be used freely to improve program readability. Blank lines are specifically allowed. A blank character is assumed between input records and so a single word cannot be continued across two input records.

Identifiers are defined by the following syntax diagram:

```
 <<<  ident  >>>


---> letter ---+-------------+------>
               ^             |
               |             |
               +<-- letter <--+
               |             |
               +<-- digit  <--+
               |             |
               +---   _    <--+
```

letter means a letter of the alphabet,
digit means 0,1,2,3,4,5,6,7,8, or 9.

Example:

COUNTER   MY_PROGRAM   DIGIT3   are valid identifiers.

Identifiers can be of any length but only the first ten characters are used by the compiler. Thus identifiers must be unique in the first ten characters.

Certain words called reserved words are used to build statements and other constructs and may not be used as variable names.

Example:

The identifiers IF, THEN, and ELSE are reserved.

The list of reserved words is:

AND	ARRAY	BEGIN	CASE
CONST	DOWNTO	DO	ELSE
END	FOR	GOTO	IF
LABEL	NOT	OF	OR
PROCEDURE	PROGRAM	REPEAT	THEN
TO	UNTIL	VALUE	VAR
WHILE			

Comments are any sequence of characters beginning with the special symbol (* and ending with the special symbol *). Comments may appear anywhere that a blank may appear.

Example:

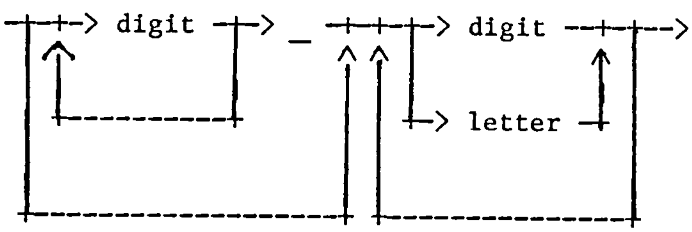
```
(* THIS IS A COMMENT *)
```

Comments may extend over several lines. It is important to correctly terminate a comment with the *) characters. If they are not present or extend past the last column the compiler reads (usually 72), the rest of the program will be taken as comment.

3 CONSTANTS AND VARIABLES

Variables may be scalars or arrays. Scalar variables can be of type integer, character or Boolean. Integer variables occupy one word (eight-bit byte) and can take on values in the range 0 to 255 inclusive. The syntax of integer constants is:

<<< int_const >>>



Example:

8748	Decimal representation.
16_222C	Hexadecimal representation of 8748.
8_21054	Octal representation of 8748.

If the first integer and underscore are present they designate the base of the constant. The second integer is then a constant in that base. The base is specified in decimal. Letters are used in the conventional way for constants with bases larger than 10. The base may be any integer between 2 and 36 inclusive but a warning is issued for any base other than 2, 8 or 16. If there is only one integer in the constant (i.e., no base is specified) then it is assumed to be a decimal constant.

Boolean variables occupy one word and can take on the values TRUE and FALSE. TRUE and FALSE are the Boolean constants. The internal representation of TRUE is any eight bits with the least significant bit one. The internal representation of FALSE is any eight bits with the least significant bit zero. A special provision to specify a Boolean as a bit reference is provided (see section 4.3) to permit packing several Boolean variables in one word. In this case all eight bits are considered significant so the compiler will generate extra code to preserve the bit pattern of the word.

Character variables occupy one word and take on values defined by the 128 character ASCII subset. Character constants which are printable on standard CYBER computers are represented as a single character contained within quotes, except for the quote character which is represented as two quotes within quotes. A single character is used for upper case letters, digits and special characters. Lower case letters are represented by the string LC_ followed by the corresponding upper case letter.

Example:

'A'	Upper case A.
'%'	Percent sign.
LC_A	Lower case A (i.e., a).
''''	The quote character.

Other non-printable character constants are represented by the corresponding ASCII name. These names are:

ASCII Hex Code	Name	ASCII Hex Code	Name
00	NUL	10	DLE
01	SOH	11	DC1
02	STX	12	DC2
03	ETX	13	DC3
04	EOT	14	DC4
05	ENQ	15	NAK
06	ACK	16	SYN
07	BEL	17	ETB
08	BS	18	CAN
09	HT	19	EM
0A	LF	1A	SUB
0B	VT	1B	ESC
0C	FF	1C	FS
0D	CR	1D	GS
0E	SO	1E	RS
0F	SI	1F	US
7B	L_BRACE	7E	TILDE
7C	BAR	7F	DEL
7D	R_BRACE		

Arrays are limited to one dimension. They can be of any size within the limits of the machine. Array elements can be any scalar type except for a bit specified Boolean. Arrays must be indexed when referenced and the resulting element used anywhere that a scalar of the element type can be used. The notation for indexing is the array name followed by either a scalar variable or an integer constant surrounded by brackets. There are no array constants. However, strings of characters can be used in the VALUE part of a program (see section 4.4) to initialize an array of characters.

Example:

Suppose A is an array.
A[I] - the Ith element of A.
A[5] - the 5th element of A.

For the rules governing the use of arrays in expressions see section 6. The declaration of arrays is covered in section 4.3.

4 PROGRAM STRUCTURE

The syntax of a Pascal/48 program is:

<<< program >>>

```
---> PROGRAM ---> ident ---> ; ---> block ---> . ----->
```

<<< block >>>

```
-+-> lab_part -+-+-> con_part -+-+-> var_part -+---+
 |             ^ |             ^ |             ^   |
 |             | |             | |             |   |
 +-------------+ +-------------+ +-------------+   |
                                                   |
          +----------------------------------------+
          |
          |
          +-+-> val_part -+-+-> proc_part -+---> cmpnd_stmt ----->
            |             ^ |              ^
            |             | |              |
            +-------------+ +--------------+
```

A program describes a set of instructions which can be placed in an MCS-48 computer and executed. As with an MCS-48 assembly language program there is no operating system or other software to support a Pascal/48 program. It is the user's responsibility to ensure that a program is totally self-contained.

The following is a very simple Pascal/48 program:

Example:

```
(* ECHO BUS TO PORT2 *)

PROGRAM DEMO;

   BEGIN
      WHILE TRUE DO
         PORT2 := BUS;
   END.
```

This program reads BUS and outputs the value to PORT2. It continues to do this forever. The compiler will generate an unconditional branch to the program instructions at location 0. When the computer is reset, this will branch past the memory locations reserved in the hardware for interrupt processing. For this example, the program instructions will be placed in memory beginning at address 9.

At the end of a Pascal/48 program, an unconditional branch to itself instruction is generated. This helps to prevent an incorrect program from executing data or uninitialized ROM.

Each of the subsections of the remainder of section 4 will conclude with expanded versions of this program. The expanded versions are merely examples of the syntax and may not be meaningful.

4.1 LABEL DECLARATION PART

The syntax of the label declaration part is:

```
<<<  lab_part  >>>

---> LABEL -+-> int_const -+-> ; ----->
            ^              |
            |              |
            +----- , <-----+
```

Example:

```
LABEL 1,100,1000;
```

The purpose of the label declaration part is to declare integer constants which will be used as statement labels. The labels can then be used in GOTO statements. All the labels used in a program must be declared. The use of labels and GOTO statements is discouraged because other control statements (WHILE, IF etc) are provided and their use is preferred.

No code is generated for a label declaration part and labels bear no relation whatsoever to machine addresses.

Example:

```
(* USE GOTO AND LABEL *)

PROGRAM DEMO;

   LABEL 100;

   BEGIN
      100: PORT2 := BUS;
      GOTO 100;
   END.
```

This program generates the same machine instructions as the previous version of the program which used a WHILE statement. Efficient programs do not need to use GOTO statements.

4.2 CONSTANT DEFINITION PART

The syntax of the constant definition part is:

```
  <<<  con_part  >>>

---> CONST -+-> ident ---> = ---> constant ---> ; -+--->
            ^                                       |
            |                                       |
            +---------------------------------------+
```

Example:

```
CONST  MASK       = 2_01111111;
       COUNT_REG  = 5;
       LOOP_COUNT = 100;
       ESC_CHAR   = '_';
```

The meaning of a constant definition is the association of an identifier with a constant value. Subsequent references to the identifier are equivalent to references to the constant. This is a similar capability to an assembly language EQUATE. No code is generated for a constant definition part.

Example:

```
(* OUTPUT THE 7 LEAST SIGNIF.  BITS OF BUS TO PORT2 *)

PROGRAM DEMO;

   CONST MASK = 2_01111111;

   BEGIN
      WHILE TRUE DO
         PORT2 := BUS AND MASK;
   END.
```

4.3 VARIABLE DECLARATION PART

The syntax of the variable declaration part is:

<<< var_part >>>

```
---> VAR -+-+-> ident ---> addr_spec -+-> : ---> type_spec ; -+--->
         ^ ^                          |                       |
         | +----------- , <-----------+                       |
         +----------------------------------------------------+
```

<<< addr_spec >>>

```
         +---------------+
         |               v
---> [ --+-> int_const -+-+-> , --+-> RAM  --+-> ] ----->
                        |         |          ^
                        |         +-> ROM  ->+
                        |         |          |
                        |         +-> XRAM ->+
                        |                    |
                        +--------------------+
```

<<< type_spec >>>

```
-+-> array_spec ---> OF -+-> scl_type ----->
 |                       ^
 +-----------------------+
```

<<< array_spec >>>

```
---> ARRAY ---> [ ---> int_const ---> .. ---> int_const ---> ] ----->
```

<<< scl_type >>>

```
-----+--> INTEGER ---------------------------------+--->
     |                                             ^
     |                                             |
     +--> BOOLEAN -+------------------------+-->+
     |             |                        ^   |
     |             |                        |   |
     |             +-> [ -> int_const -> ] -+   |
     |                                          |
     +--> CHAR ---------------------------------+
```

The variable declaration part of a program declares variables which a program can use. The address specification is required for each variable and indicates the address to be used for the variable and the type of memory it will occupy; random access - RAM, read only - ROM, external random access - XRAM. The memory type is optional and RAM is assumed if no type is specified. RAM addresses are in the range 0 to 63 inclusive, but may be expanded by the data RAM option (see section 10). XRAM address must be in the range 0 to 255 inclusive. ROM addresses must be in the range 0 to 255 inclusive. Variables which are declared in the main part of a program are called global variables. Global ROM variables are allocated space in page 3 and their address is a displacement into page 3. An address must be specified for global variables. They are accessable throughout the entire program. Variables may also be declared inside procedures (see section 4.5). They are called local variables. Local ROM variables are allocated space in the page in which the procedure resides. Their address specification is required but the actual integer constant specifying the address may be omitted. If present, it is a displacement into that page. If absent, the ROM variable will be allocated space automatically, ahead of the instructions for the procedure. It is the programmer's responsibility to relocate instructions and place variables in ROM so that the required access is possible. Relocation can be achieved with the origin option (see section 10). Booleans may be defined as a bit within a word. The value of the integer constant following the scalar type BOOLEAN specifies the bit of the word and is limited to the range 0 to 7. The array specification is not allowed with the bit specification of Boolean variables.

Example:

```
VAR X[4],Y[5]     : INTEGER;
    Z[6]          : BOOLEAN;
    S[7]          : BOOLEAN[0];
    T[7]          : BOOLEAN[1];
    A[0,ROM]      : ARRAY[1..10] OF INTEGER;
    CH[20,XRAM]   : CHAR;
```

In the above example, X and Y are one word integer variables which will occupy words 4 and 5 of RAM. Z is a Boolean variable which will occupy word 6 of RAM. S and T are Boolean variables which will occupy bits 0 and 1 of word 7 of RAM. A is an array with ten elements each of which is an integer. Assuming it is global, it will occupy ten words of ROM beginning at word 300(hex), i.e., the first word of the third page. CH is a character variable which will occupy word 20 of external RAM.

Since the hardware defines RAM addresses 0 to 7 inclusive as registers, the address specification allows the programmer control of the registers. Registers 0 and 1 should not be used by the programmer. They are reserved for use by the compiler and if used explicitly may produce unpredictable results. If register bank one (see section 8.9) is selected, RAM addresses 24 and 25(dec) will become registers 0 and 1 respectively. Note that RAM addresses 8 through 23(dec) are used by the hardware for a stack of procedure call return addresses and other data. The programmer should not use these addresses unless explicit modification of the stack is required. Warning messages are issued for variables defined at RAM addresses 0, 1 and 8 thru 23 but not for register bank one RAM addresses 24 and 25.

No code is generated for a variable declaration part.

Example:

```
(* ADD 7 LEAST SIGNIF. BITS OF BUS TO REG 4 *)
(* STORE RESULT IN REG 4 AND OUTPUT TO PORT 2 *)

PROGRAM DEMO;

   CONST MASK = 2_01111111;

   VAR X[4] : INTEGER;

   BEGIN
      X := 0;
      WHILE TRUE DO
         BEGIN
         X := BUS AND MASK + X;
         PORT2 := X;
         END;
   END.
```

4.4 VALUE PART

The syntax of the value part is:

```
 <<<  val_part  >>>

---> VALUE -+-> ident ---> = ---> val_con --- ; -+--->
            ^                                    |
            |                                    |
            +------------------------------------+


 <<<  val_con  >>>

-+-> ( -+-+-> int_const ---> OF -+-> constant -+-> ) -+--->
 |      ^ |                      ^             |      ^
 |      | |                      |             |      |
 |      | +----------------------+             |      |
 |      |                                      |      |
 |      +--------------- , <-------------------+      |
 |                                                    |
 +-------------------> constant ----------------------+
```

The value part associates a constant with a previously declared variable which is in ROM. The value is placed into ROM with the program thereby defining the variable. This is the only way to assign a value to a variable in ROM. No executable code is generated for a value part.

The identifier preceeding the equals sign is the variable to be initialized. The value is either a single quantity (for a scalar) or a list contained in parentheses (for an array). A replication factor can be used to generate many copies of a value in a list. It precedes the value and is separated from it by the reserved word OF. It indicates the number of copies of the value that are required.

A special case is provided for the initialization of arrays of characters where all the characters required are printable. Instead of writing each character within quotes and separating each pair in the list with a comma, the characters may be written as a string within a single pair of quotes. Note that non-printable characters may not appear in the string and must be listed individually if required.

Example:

```
VAR MESSAGE[0,ROM] : ARRAY[1..11] OF CHAR;

VALUE MESSAGE = ('ENTER DATA',CR);
```

Example:

```
VAR     X[0,ROM] : INTEGER;
        Y[1,ROM] : ARRAY[1..10] OF INTEGER;
VALUE   X = 5;
        Y = (3 OF 1, 4 OF 2, 5, 8, 11);
   (* X IS THE FIRST WORD OF PAGE 3. IT IS PRESET TO 5 *)
   (* Y IS NEXT TEN WORDS. IT IS SET TO                *)
   (*       1, 1, 1, 2, 2, 2, 2, 5, 8, 11              *)
```

Example:

```
(* SIMULATION OF A 7447 BCD TO 7 SEGMENT DECODER  *)
(* BCD INPUT IS ON LEAST SIGNIF.  NIBBLE OF PORT 1*)
(* GENERATES 7 SEGMENT DRIVER SIGNALS ON PORT 2   *)

PROGRAM BCD_TO_7SEG;

   CONST MASK = 16_OF;

   VAR TABLE [0,ROM] : ARRAY [1..16] OF INTEGER;
       DIGIT[2]      : INTEGER;

   VALUE TABLE = (2_00000001, 2_01001111, 2_01101101,
                  2_00000110, 2_01001100, 2_00100100,
                  2_01100000, 2_00001111, 2_00000000,
                  2_01110010, 6 OF 2_11111111);

   BEGIN
      WHILE TRUE DO
         BEGIN
         DIGIT := PORT1 AND MASK;
         PORT2 := TABLE[DIGIT];
         END;
   END.
```

4.5 PROCEDURE DECLARATION PART

The syntax of the procedure declaration part is:

```
 <<<  proc_part  >>>

-+-> proc_hdr ---> proc_body ---> ; -+--->
 ^                                   |
 |                                   |
 +-----------------------------------+
```

```
 <<<  proc_hdr  >>>

---> PROCEDURE ---> ident ---> ; ----->
```

```
 <<<  proc_body  >>>

-+-> EXTERNAL ---> [ ---> int_const ---> ] -+--->
 |                                          ^
 |                                          |
 +--------------> FORWARD ---------------->+
 |                                          |
 |                                          |
 +--------------> block  -------------------+
```

Procedures are merely subroutines. They may be called from the main program or other procedures.

The machine code for a procedure will be generated following the previous procedure or at location 9 if it is the first procedure. The origin option (see section 10) can be used to change the location counter at any point in a program but an important special use is to relocate procedures. This facility must be used to relocate procedures appropriately in the various pages of ROM. Many MCS-48 instructions are only able to generate "within page" addresses. The boundary (default) option will ensure that no problems will occur with page boundaries, however, the machine code generated will not be optimal. If the boundary problems are nested redundant code will probably be generated.

It is the programmer's responsibility to adjust the location counter to ensure optimal code at the page boundaries. If the boundary option is not selected and the compiler encounters difficulty with a page boundary, a diagnostic will be issued and it will be necessary for the programmer to explicitly relocate the code as necessary.

The body of a procedure may take one of three forms. If it is the word FORWARD then it indicates that the procedure will be defined lower down in the program but it informs the compiler of its existence. This allows forward references to procedures. If a FORWARD declaration is not present, a procedure must be defined before it can be called.

If the body of a procedure is the word EXTERNAL followed by an integer inside brackets, it indicates that the procedure will be produced separately from this program (an assembly language subroutine for example) but will occupy this location at execution time. The integer following the word EXTERNAL indicates the size of the procedure in bytes and this amount of space is left empty by the compiler.

The third form a procedure body may take is a block and so all of the components of a program may appear inside a procedure (see section 4). Any identifier declared inside a procedure is only known (i.e., can only be used) inside that procedure or others nested inside it. Pascal/48 uses the same scope rules for identifiers as Pascal (Pascal User Manual and Report, Second Edition: Jensen, Kathleen; Wirth, I. Niklas; 1974; Springer-Verlag Berlin Heidelburg New York).

Example:

```
PROCEDURE CONVERT;

   CONST ASCII_ZERO = 16_30;

   VAR DIGIT[10] : INTEGER;

   BEGIN
      DIGIT := DATA_ASCII - ASCII_ZERO;
      IF DIGIT > 10 THEN
         DIGIT := DIGIT - 7;
      DATA_HEX := DIGIT;
   END;
```

In this example, the constant ASCII_ZERO and the variable DIGIT can only be used inside procedure CONVERT. The procedure CONVERT takes a character in a variable called DATA_ASCII which is assumed to be the character representation of a hex digit and converts it to the corresponding binary integer, e.g., character 1 (ASCII hex code 31) is converted to binary 1. The result is placed in the variable DATA_HEX. The variable DIGIT is not required but is used as an example. Variables declared in procedures must still have an address specified for them and storage will be allocated statically.

The Pascal/48 compiler will generate machine code for each procedure in order and place it into ROM beginning at address 9 if the origin option is not used. A program may contain any number of procedures within the memory limits of the machine. It is not necessary (nor is it possible) to write an RET instruction at the end of a procedure. The compiler will automatically generate the RET instruction when the END statement of a procedure body is encountered.

Example:

```
PROGRAM ECHO;

   PROCEDURE COPY;

      CONST MASK = 2_10111110;

      VAR X[2] : INTEGER;

      BEGIN
         X := PORT1 AND MASK;
         PORT2 := X;
      END (* COPY *);

   BEGIN
      WHILE TRUE DO
         COPY;
   END.
```

4.6 COMPOUND STATEMENT

The syntax of a compound statement is:

```
 <<<  cmpnd_stmt  >>>

---> BEGIN -+-> statement -+-> END ----->
            ^              |
            |              |
            +---- ; <------+
```

This is just a sequence of statements (see section 8) separated by semicolons. In the case of a program, this compound statement constitutes the body of the program. The sequence of statements will be compiled into machine code and placed into ROM following the machine code for the last procedure. A branch instruction to the beginning of the code for this compound statement is placed into ROM location 0.

For a procedure, the compound statement constitutes the body of the procedure and it is the sequence of statements which will be executed when the procedure is called. The BEGIN-END pairs contained in examples in other sections of this manual are examples of compound statements.

5 OPERATORS

Operators are used to build expressions and they give access to many of the MCS-48 instructions. Operators are either monadic or dyadic. Monadic operators have one operand and dyadic operators have two operands.

Each monadic operator may be used in two different ways in expressions (see section 6) but the operator actually generates either zero or one instruction. A monadic operator can only be used with an operand of the correct type and the appropriate instruction is applied with the operand in the accumulator. The compiler generates the instructions necessary to evaluate the operand (if it is an expression) or load the operand into the accumulator (if it is a variable) prior to generating the instruction for the monadic operator. The monadic operators and their meanings are given by the following table:

OPERATOR	OPERAND TYPE	RESULT TYPE	MEANING
DEC_ADJ	Integer	Integer	Decimal adjust.
NOT	Int/Bool	Int/Bool	Invert each bit.
CHR	Integer	Character	Character representation of integer.
ORD	Any scl.	Integer	Integer representation of any scalar type (in particular character).
ADDR_PAGE	Program label	Integer	Page number in which a program label occurs.
ADDR_WORD	Program label	Integer	Word offset in which a program label occurs.

DEC_ADJ performs the normal decimal adjustment algorithm necessary to generate a BCD result after binary arithmetic with BCD quantities. NOT merely performs a bit by bit inversion. CHR and ORD are for switching types between type INTEGER and CHARACTER. They allow programmers to indicate explicitly that arithmetic on scalar, non-integer quantities is required. In particular, ORD(FALSE) is 0 and ORD(TRUE) is 1. ADDR_PAGE and ADDR_WORD take a programmer defined label as their operand and return the page number and within-page word offset respectively of that label. The labels must be between 0 and 255. The results of these operators are the actual machine address corresponding to the label. These operators do not generate any code. They give access to machine addresses so that in the few necessary cases, machine addresses can be used in a Pascal/48 program. A possible use would be the explicit modification of the hardware stack to provide a non-standard return from an interrupt (see section 9).

Example:
```
        Y := DEC_ADJ(X+1);
        Z := ORD CH + 1;
        C := CHR Z;
```

In the above example, one is added to X, the result is decimal adjusted, and assigned to Y. The numeric representation of the character CH is made available, one is added to it, and the result assigned to Z. The character corresponding to the integer in Z is assigned to the character variable C.

The rules governing the use of monadic operators in expressions are given in section 6.

Dyadic operators are either one or two special characters, or a meaningful sequence of letters. In either case they are written between their operands (infix) in the usual way. Each operator generates either one or a small number of MCS-48 machine instructions. The dyadic operators and their meanings are given by the following table:

OPERATOR	MEANING
+	Eight-bit addition.
-	Eight-bit subtraction (see below).
++	Eight-bit addition with carry (see below).
--	Eight-bit subtraction with borrow (see below).
<	Test for less than.
<=	Test for less than or equals.
=	Test for equality.
>=	Test for greater than or equals.
>	Test for greater than.
<>	Test for inequality.
AND	Logical and.
OR	Logical or.
XOR	Logical exclusive or.
ROTL	Rotate left without carry (carry not affected).
ROTR	Rotate right without carry (carry not affected).
ROTLC	Rotate left through carry.
ROTRC	Rotate right through carry.
SHL	Shift left, insert zeros (carry affected).
SHR	Shift right, insert zeros (carry affected).
BIT	Select bit.

For the operator -, subtraction means negate the right operand and add. Negate means complement each bit and add one (two's compliment). The operators add with carry (++) and subtract with borrow (--) should only be used in a double precision context with the eight-bit addition and subtraction. To insure the setting of the carry bit, the expression should be simple.

Example:

```
A[2] := A[2] + B[2];
A[1] := A[1] ++ B[2];
(* DOUBLE PRECISION ADDITION *)

A[2] := A[2] - B[2];
A[1] := A[1] -- B[1];
(* DOUBLE PRECISION SUBTRACTION *)
```

For the rotate and shift operators, the left operand is the operand to be acted on and the right operand gives the rotate or shift count. These operators are implemented with some special case analysis. For example, a rotate without carry by four is implemented as a SWAP NIBBLES instruction. The rules governing the use of dyadic operators in expressions are given in section 6.

The operands for the dyadic operators must be of type integer, except the logical operators which take either Boolean or integer operands. The result type is integer for all of the dyadic operators except the BIT and relational operators which produce Boolean results and the logical operators which produce Boolean results if their operands were Boolean.

The dyadic operator BIT is used for bit selection. Its left operand must be an integer and its right operand must be an integer constant in the range 0 to 7. It is particularly useful in testing individual bits of the I/O ports.

Example:

```
IF PORT1 BIT 7 THEN . . .

WHILE PORT1 BIT 2 DO . . .
```

6 EXPRESSIONS

The syntax of an expression is:

```
<<<  expression  >>>

-+-> monadic_op ---> ( ---> expression ---> ) -+-+->
 |                                              ^ |
 |                                              | |
 ++-+-> monadic_op --+-----+--> constant ----->+ |
  ^ |                ^     |                    | |
  | |                |     |                    | |
  | +----------------+     +--> variable ------+ |
  |                                               |
  |                                               |
  +--------------- dyadic_op <--------------------+
```

```
<<<  variable  >>>

               +-----------------------------+
               |                             |
               |                             v
---> ident  --+-> [ -+-> int_const -+-> ] -+---->
                     |               ^
                     |               |
                     +->   ident    -+
```

Expressions define sequences of MCS-48 instructions for operating on data. The rules which govern expressions are designed to allow flexibility in these expressions but allow the programmer to retain control of machine resources.

Recall that any path through the syntax diagram labelled <<< expression >>> yields a valid expression. A variable is either an identifier (i.e., a scalar variable) or an array element. An array element can have either a constant or a variable index.

There is no operator hierarchy and dyadic operators in an expression are executed strictly in order from left to right. Note that parentheses cannot be used to force a priority on dyadic operator evaluation. For a monadic operator whose operand is in parentheses (and can therefore be an expression), the operand is evaluated first and then the monadic operator is applied. For a monadic operator whose operand is not in parentheses (and must therefore be a constant or variable) the monadic operator is applied directly to the operand. The result may then become an operand of a dyadic operator.

Example:

```
DEC_ADJ(X+1)
```

Valid expression. Add X and 1 and decimal adjust the result.

```
DEC_ADJ X+1
```

Valid expression. Decimal adjust X and add 1 to the result.

```
DEC_ADJ(X+1) AND Y
```

Valid expression. Add X and 1, decimal adjust the result, and mask Y with the ensuing result.

```
DEC_ADJ(ORD CH1 + ORD CH2)
```

Valid expression. Decimal adjust the sum of the ordinals of the character variables CH1 and CH2.

```
DATA[I] ++ DATA[I+1]
```

Invalid expression. Array index must be a constant or a variable. This expression would have to be broken into two parts - first I+1 would have to stored in a variable; second DATA would have to indexed by that variable and then added to DATA[I].

```
A + B[1] - C ROTL 2 ++ TABLE[3]
```

Valid expression. Add A and the first element of B, subtract C, rotate left two, add the third element of array TABLE with carry.

```
PORT1 + PORT2 + TABLE[4] XOR 16_82
```

Valid expression. Read PORT1 and PORT2 and add, add fourth element of array TABLE, exclusive or the result with the hexadecimal constant 82.

7 PREDECLARED VARIABLES

Access to many parts of the MCS-48 computer is provided by a set of predeclared variables. These are variables which can be used in programs without being declared by the programmer. They represent data items or similar which are not part of the machine's memory. Use of a predeclared variable in an expression causes the data item to be referenced and assignment to the variable causes the data item to be set.

Example:

CARRY is a predeclared variable.

```
CARRY := TRUE;      (* THIS TURNS THE CARRY BIT ON *)

IF CARRY THEN X := X + 1
      ELSE X := 0;
(* THIS INCREMENTS X IF THE CARRY BIT IS ON *)
(* OTHERWISE IT SETS X TO ZERO              *)
```

Example:

TIMER is a predeclared variable.

```
TIMER := 100;
(* THIS SETS THE MACHINE'S TIMER TO 100 *)
```

The table below shows all of the predeclared variables, their type and meaning.

VARIABLE	TYPE	MEANING
ACCUMULATOR	Typeless	The accumulator.
BUS	INTEGER	The bus.
CARRY	BOOLEAN	The accumulator carry bit.
COUNTER	INTEGER	The built-in counter.
FLAG0	BOOLEAN	Flag 0.
FLAG1	"	Flag 1.
INT	"	Interrupt pin.
PORT1	INTEGER	I/O port 1.
PORT2	"	I/O port 2.
PORT4	"	I/O port 4.
PORT5	"	I/O port 5.
PORT6	"	I/O port 6.
PORT7	"	I/O port 7.
PSW	"	Program status word.
TIMER	"	The built-in timer.
TIMER_FLAG	BOOLEAN	Timer flag.
T0	"	Pin T0.
T1	"	Pin T1.

The predeclared variable ACCUMULATOR is typeless and can be assigned to any scalar variable and can have any scalar value assigned to it. It is used when the accumulator must be stored or initialized (in handling interrupts for example - see section 9). Great care must be exercised in dealing with the accumulator and carry since the compiler will generate instructions which will affect these variables for almost every statement.

The predeclared variables PORT1 through PORT7 and BUS are particularly important because they give access to most of the machine's input/output features. Assignment to them causes data to be output and reference to them causes data to be read.

Example:

```
DATA := PORT1;
PORT2 := 2_10101100;
```

Single bits can be set on a port or the BUS by referencing and assigning the appropriate predeclared variable in the same statement.

Example:

```
PORT1 := PORT1 AND 2_00000001;

(* THIS WILL GENERATE    ANL P1,#1 *)
```

8 STATEMENTS

The syntax of a statement is:

```
<<< statement >>>

-+-> int_const ----> : -+-+----> assgn_stmt ---------+----->
 |                      ^ |                          ^
 |                      | |                          |
 +----------------------+ +----> if_stmt    -------->+
                          |                          |
                          +----> while_stmt -------->+
                          |                          |
                          +----> rept_stmt  -------->+
                          |                          |
                          +----> for_stmt   -------->+
                          |                          |
                          +----> case_stmt  -------->+
                          |                          |
                          +----> call_stmt  -------->+
                          |                          |
                          +----> goto_stmt  -------->+
                          |                          |
                          +----> cmpnd_stmt -------->+
                          |                          |
                          +--------------------------+
```

Note that a statement may be labelled only once and that a statement can be empty. Empty statements are used in many examples in this manual. Also, since a compound statement is defined to be a statement, anywhere that a statement can appear a compound statement can appear. Each of the statements is explained in the following sections.

8.1 ASSIGNMENT STATEMENT

The syntax of the assignment statement is:

```
<<< assgn_stmt >>>

---> variable ---> := ---> expression ----->
```

The expression is evaluated and the variable is given that value. If the variable is an array element, the array index is evaluated before the expression. The type of the expression must match the type of the variable. Note that Boolean expressions and assignment are specifically allowed. Assignments to bit specified Boolean variables will generate code to preserve the bit pattern of the word and modify only the bit referenced. For the definitions of variables and expressions, and the rules governing the construction of expressions see section 6.

Example:

```
        TIMER := 100;          (* SET TIMER TO 100 *)
        X     := A + B SHL 1;  (* ADD A AND B, SHIFT LEFT 1 *)
        V[1]  := V[1] + C;     (* ADD C TO V[1] *)
        A[I]  := B[I];         (* COPY B[I] TO A[I] *)
```

8.2 IF STATEMENT

The syntax of the IF statement is:

```
 <<<  if_stmt  >>>

---> IF ---> expression ---> THEN ---> statement --+
                                                   |
                                                   |
                                                   |
        +----------------------------+-------------+
        |                            |
        |                            |
        |                            v
        +---> ELSE ---> statement ---+---->
```

The expression must be of type Boolean. It is evaluated and if it is true, the statement following the THEN is executed. If it is false, the statement following the ELSE is executed if an ELSE is present, otherwise nothing is executed.

Example:

```
IF A[I] > MAX THEN MAX := A[I];

(* MAX IS REPLACED BY A[I] IF A[I] EXCEEDS MAX *)
```

Example:

```
IF RESET THEN PORT1 := 0 ELSE PORT1 := DATA;

(* RESET IS A BOOLEAN - ITS VALUE IS TESTED *)
```

Example:

```
IF COUNT > LIMIT THEN
   BEGIN
   COUNT := 1;
   PORT1 := NEWDATA;
   END
ELSE
   BEGIN
   COUNT := COUNT + 1;
   PORT1 := OLDDATA;
   END
(* NOTE THE USE OF COMPOUND STATEMENTS.  *)
```

Example:

```
IF A < B THEN
   IF B < C THEN
      PORT1 := C
   ELSE PORT1 := B
ELSE
   IF A < C THEN
      PORT1 := C
   ELSE PORT1 := A
(* THIS EXAMPLE OUTPUTS THE LARGEST *)
(* OF THREE INTEGERS A, B, AND C    *)
```

Care should be taken with nested if-then-else constructs in avoiding the problem of the "dangling else." An else clause is associated with the nearest if. Consequently, in the following example, the variable C would be undefined if A is false even though the indenting might lead the reader to infer that C should be assigned 1 when A is false.

Example:

```
IF A THEN
   IF B THEN
      C := 0
ELSE
   C := 1
(* THIS EXAMPLE DEMONSTRATES THE POSSIBILITY *)
(* OF A DANGLING ELSE CLAUSE                 *)
```

To prevent the problem of the "dangling else" a compound statement (BEGIN-END pair) is necessary.

Example:

```
IF A THEN
   BEGIN
   IF B THEN
      C := 0
   END
ELSE
   C := 1
(* THIS EXAMPLE DEMOSTRATES THE METHOD  *)
(* OF PREVENTING A DANGLING ELSE CLAUSE *)
```

8.3 WHILE STATEMENT

The syntax of the WHILE statement is:

```
  <<<  while_stmt  >>>

 ---> WHILE ---> expression ---> DO ---> statement ----->
```

The expression is evaluated. It must be of type Boolean. The statement is repeatedly executed until the expression becomes false. If its value is false at the beginning, the statement is not executed at all.

Example:

```
(* INTEGER DIVISION BY REPEATED SUBTRACTION *)

DIV := 0;

WHILE NUMERATOR > DENOMINATOR DO
   BEGIN
   DIV := DIV + 1;
   NUMERATOR := NUMERATOR - DENOMINATOR;
   END;

REMAINDER := NUMERATOR;
```

A useful special case of the WHILE statement involves using the Boolean constant TRUE as the expression. Since TRUE is always true (by definition) this generates an infinite loop. This construct is used in the example program in section 4.

Another useful special case is the use of one of the I/O ports (or the BUS) as part of the expression, provided the port is being used for input. Evaluation of the expression causes the port to be read and, since its value changes independently of the microcomputer, the WHILE statement can be used to wait for changes in the port. The statement part of the WHILE statement can be empty and so the effect is continuous looping until a certain input occurs.

Example:

```
WHILE PORT1 > 0 DO;

(* LOOP UNTIL PORT1 CHANGES TO ALL ZEROS *)
```

Example:

```
WHILE PORT2 BIT 7 DO;

(* LOOP UNTIL PORT2 BIT 7 GOES LOW *)
```

Example:

```
CONST STATUS = 7;
          .
          .
WHILE NOT(PORT1 BIT STATUS) DO;

(* LOOP UNTIL PORT1 BIT 7 GOES HIGH *)
```

8.4 REPEAT STATEMENT

The syntax of the REPEAT statement is:

```
  <<<  rept_stmt  >>>

---> REPEAT -+-> statement -+-> UNTIL ---> expression ----->
             ^              |
             |              |
             +------ ; <----+
```

The expression is evaluated. It must be of type Boolean. The sequence of statements between the symbols REPEAT and UNTIL is executed repeatedly (and at least once) until the expression becomes true.

Example:

```
I := 0;
REPEAT
   I := I + 1;
   R := R - S;
UNTIL R < S;

(* INTEGER DIVIDE BY REPEATED SUBTRACTION *)
(*         ASSUMES R > S INITIALLY        *)
```

A useful special case of the REPEAT statement has an empty statement between the REPEAT and UNTIL (i.e., no text). In this case, the condition is repeatedly tested until it is true. As with the WHILE statement, if the expression involves the I/O ports or the BUS, this allows a loop to be written which will continue to execute until an outside event occurs.

Example:

```
REPEAT
UNTIL PORT1 = 0;
```

In this example, PORT1 will be read and compared with zero and this will be repeated until PORT1 becomes zero.

8.5 FOR STATEMENT

The syntax of the FOR statement is:

```
<<<  for_stmt  >>>

---> FOR ---> for_list ---> DO ---> statement ----->

<<<  for_list  >>>

---> ident ---> := ---> con_var -+->   TO    -+--> con_var ----->
                                 |            ^
                                 |            |
                                 +-> DOWNTO --+
```

The FOR statement indicates that a statement called the controlled statement is to be executed zero or more times. The identifier is called the control variable. The first con_var in the definition of for_list is called the initial value and the second is called the final value. These values must be of type integer.

The initial value expression and the final value expression are evaluated. The control variable is given the initial value. If the initial value is greater than (less than) the final value in the TO (DOWNTO) case, the controlled statement is not executed. Otherwise, the control variable is given a progression of values up to (down to) the final value and the controlled statement is executed for each one.

Example:

```
FOR I := 1 TO 100 DO
   PORT1 := I;

(* OUTPUT THE INTEGERS 1 THROUGH 100 IN SEQUENCE ON PORT1 *)
```

Example:

```
SUM := 0;
FOR I := 1 TO 10 DO
   SUM := A[I] + SUM;

(* ADD UP THE ELEMENTS OF AN ARRAY *)
```

Example:

```
FOR I := 10 DOWNTO 1 DO
   BEGIN
   A[I] := 0;
   B[I] := 0;
   C[I] := 0;
   END;
(* INITIALIZE ARRAYS *)
```

Example:

```
FOR I := 200 DOWNTO 1 DO
   FOR J := 200 DOWNTO 1 DO;
```

The last example generates two nested Decrement and Jump on Non-Zero instructions (DJNZ) if I and J are registers. Nothing will be executed by these loops but they constitute a convenient form of 'busy wait', i.e., a delay (see also the predeclared procedure DELAY in section 8.9).

The FOR statement is implemented with a DJNZ instruction if the DOWNTO case is used, the final value is the constant 1, and the control variable is a register. This is the most efficient form of counting loop on the MCS-48 computers. Other forms of the FOR statement are implemented with explicit incrementation for the arithmetic and explicit comparison for the loop control.

The value of the control variable is undefined after the FOR statement completes execution, i.e., the programmer cannot assume it will have any particular value.

8.6 CASE STATEMENT

The syntax of the CASE statement is:

```
<<<  case_stmt  >>>

                                  +------------------+
                                  |                  |
                                  |                  v
---> CASE ---> expression ---> OF -+--> case_list_elt -+--> END ----->
                                  ^                  |
                                  |                  |
                                  +------- ; <-------+


<<<  case_list_elt  >>>

-+-> constant -+--> : ---> statement ----->
 ^             |
 |             |
 +---- , <-----+
```

The expression (called the selector) is evaluated. The statement labelled with a constant (called a case label) equal to the selector value is executed. Upon completion of the selected statement, control is transferred to the end of the case statement, provided no GOTO statement was executed. If the selector value is not within the limits of the case labels, the results are undefined. It is the programmer's responsibility to ensure that this does not occur. The case statement is implemented with a table of the addresses of the statements in the case list elements. This table is indexed with the expression. The size of the table is determined by the range of constant values used as case labels but not their absolute values. Thus the smaller the range of values, the smaller the size of the table generated. Case labels must have ordinal values in the range 0 to 255. A case statement must not cross a page boundary. When the compiler corrects for a page boundary problem the whole case statement is shifted to the beginning of the next page.

It is important to realize that the case labels are not statement labels. They cannot be used as the targets of GOTO statements and their scope is limited to the case statement in which they appear.

Example:

```
CASE I OF
   1: PORT1 := X;    (* EXECUTE THIS IF I = 1 *)
   2: PORT2 := Y;    (* EXECUTE THIS IF I = 2 *)
   3,4: BEGIN
          .
          .       (* EXECUTE THIS IF I = 3 OR 4 *)
          .
        END;
   5: PORT1 := 2;    (* EXECUTE IF I = 5 *)
END
```

Example:

```
CH := CHR(PORT1);
(* READ A CHARACTER FROM PORT1 *)

CASE CH OF
   'D': PORT2 := DATA[I];
   'C': DATA[I] := PORT2;
   'M': IF DATA[J] > 128 THEN
          BEGIN
          DATA[I] := PORT2;
          DATA[K] := BUS;
          END;
END
(* NOTE THE TABLE RANGE IS 11.          *)
(* 'E'..'L' TRANSFER CONTROL TO THE END *)
(* OF THE CASE.  'A', 'B', 'N'..'Z'     *)
(* PRODUCE UNDEFINED RESULTS.           *)
```

8.7 GOTO STATEMENT

The syntax of the GOTO statement is:

```
  <<<  goto_stmt  >>>

 ---> GOTO ---> int_const ----->
```

Execution of a GOTO statement causes to control to be transferred to the statement labelled with the integer constant. This constant has to have been declared in a LABEL part.

It is possible to branch to any label defined on any statement. This means that it is possible to jump into or out of a procedure. If this is done, the Pascal/48 compiler will generate the necessary unconditional branch only. The stack will remain unchanged. It is the programmer's responsibility to modify the stack and PSW as necessary in these special cases. The use of labels and GOTO statements is discouraged because other control statements (WHILE, IF etc) are provided and their use is preferred.

Example:

```
GOTO 1;
(* BRANCHES TO STATEMENT LABELLED WITH 1 *)
```

8.8 PROCEDURE CALL STATEMENT

The syntax of a PROCEDURE CALL statement is:

```
 <<<  call_stmt  >>>

---> ident ----->
```

A procedure call statement causes the named procedure to be executed and then control returns to the statement following the call. The call is effected with an MCS-48 CALL instruction which places the return address on the hardware stack. As in assembly language programming, the programmer must take care not to overflow the stack since it wraps around and will overwrite previous stack frames. Recall that the stack has a capacity of eight frames.

8.9 PREDECLARED PROCEDURES

Certain procedures are predeclared (i.e., do not have to be declared by the programmer in order to be used) and they provide access to some of the MCS-48 instruction set. They may be called anywhere that a procedure call can occur and the effect is to generate a either one or a small number of MCS-48 instructions. Each takes a variable number of parameters (i.e., one or more as the user chooses) but the parameters are of a special type in some cases.

The procedures are:

ENABLE

Use:	Enable interrupts, timer or clock output.
Parameters:	COUNT_INT, TIMER_INT, EXTERN_INT, CLOCK_OUT.
Example:	ENABLE(COUNT_INT, EXTERN_INT); (* ENABLE COUNTER AND EXTERNAL INTERRUPTS *)

DISABLE

Use:	Disable interrupts.
Parameters:	COUNT_INT, TIMER_INT, EXTERN_INT.
Example:	DISABLE(COUNT_INT, EXTERN_INT); (* DISABLE COUNTER AND EXTERNAL INTERRUPTS *)

DELAY

Use:	Generate a delay loop. Since this depends on crystal frequency, see the M option in section 10.
Parameters:	Integer expression(s) (delay(s) in millisecs).
Example:	(*$M3.6 SET XTAL FREQUENCY *) . . DELAY(100); (* GENERATE 100 MILLISEC DELAY *)

START

Use:	Start the timer or counter.
Parameters:	TIMER, COUNTER.
Example:	START(TIMER); (* TURN ON TIMER *)

STOP

Use:	Stop the timer or counter.
Parameters:	TIMER, COUNTER.
Example:	STOP(TIMER); (* TURN OFF TIMER *)

INCREMENT

Use:	Increment the value of variable(s).
Parameters:	Any integer variable(s).
Example:	INCREMENT(I,PORT1); (* INCREMENT I AND PORT1 *)

DECREMENT

Use:	Decrement the value of variable(s).
Parameters:	Any integer variable(s).
Example:	DECREMENT(I,PORT1); (* DECREMENT I AND PORT1 *)

SELECT

Use:	Select register banks.
Parameters:	REG_BANK0, REG_BANK1.
Example:	SELECT(REG_BANK1); (* SELECT REGISTER BANK 1 *)

COMPLEMENT

Use:	Complement the value of variable(s).
Parameters:	Any Boolean variable(s).
Example:	COMPLEMENT(F0,CARRY,B); (* COMPLEMENT F0, CARRY AND B *)

INLINE

Use:	Inline insertion of code into the object file.
Parameters:	Any integer constant(s).
Example:	INLINE(16_27,16_97); (* INSERT INSTRUCTION TO CLEAR THE *) (* ACCUMULATOR AND CARRY BIT *)

Note that the DELAY procedure generates loops to cause the machine to perform a 'busy wait'. The maximum value for any one operand is 255 so the maximum delay is 255 milliseconds. The delay is determined by the computers operating frequency and to implement DELAY correctly the compiler needs to know the crystal frequency (see the M option in section 10). Since the DELAY capability depends on the execution time of the machine instructions generated, the compiler will not correct any page boundary problems (see section 4.5) and will issue an error message. The programmer must then relocate the code as necessary. The INCREMENT, DECREMENT and COMPLEMENT procedures are implemented using the corresponding instructions of the machine where possible. This means that INCREMENT(I) is considerably more efficient than the assignment statement I := I + 1 and <u>does not affect the carry bit</u>. Similarly the efficiency of COMPLEMENT(B) for a bit specified Boolean is considerable compared to B := NOT B. The INLINE procedure allows the programmer to enter machine instructions directly into the object file. This facility is provided as a precaution in the event that the required instructions cannot be generated from other features of Pascal/48. The INLINE procedure should seldom be needed.

9 INTERRUPTS

Interrupt handlers are written as procedures with special names. The names are TIMER_INT or COUNT_INT for timer and counter interrupts EXTERN_INT for external interrupts. If procedures by these names are present, they should not be called anywhere in the program. If they are, the compiler will issue a warning but will generate the required instructions for the call. The compiler will generate machine code for for these procedures and place branch instructions to them at address 3 (external) and address 7 (timer or counter). When an interrupt occurs, the effect is an automatic call to the appropriate procedure. When the procedure completes, control returns to the instruction following the one which preceded the interrupt. This return is done by an RETR instruction which the compiler generates. If no interrupt handler is provided, the compiler will generate an unconditional branch to location 3 at location 3 and to location 7 at location 7. Thus, if an interrupt occurs and no handler was provided, the machine will enter an infinite loop at the appropriate address.

Interrupts are enabled and disabled by calling Pascal/48 predeclared procedures (see section 8.9). The compiler will <u>not</u> generate any instructions in the interrupt handling procedure to switch register banks or save and restore the accumulator. These actions are not always required and are left to the programmer. Register banks can be switched with the SELECT predeclared procedure and the accumulator can be saved and restored using the predeclared variable ACCUMULATOR. The interrupt handlers will reset the register bank when the control is returned to the appropriate code.

Example:

```
PROCEDURE TIMER_INT;

   VAR SAVE_ACC[60] : INTEGER;

   BEGIN
      SELECT(REG_BANK1);
      SAVE_ACC := ACCUMULATOR;
         .
         .
         .
      (* REGISTER BANK IS RESET AUTOMATICALLY *)
      ACCUMULATOR := SAVE_ACC;
   END (* TIMER_INT *);
```

It is not always appropriate to return to the point of interruption following an interrupt. For example, the external interrupt may be used to perform some sort of partial reset of the computer. In such cases, it may be preferable to return to a statement in the program which is labelled. One way to achieve this is to modify the hardware stack within the interrupt handler and the allow the RETR instruction at the end of the interrupt handling procedure to transfer control to the labelled statement. The following example shows how this can be done.

Example:

```
PROGRAM EXAMPLE;

   LABEL 100;

   PROCEDURE EXTERN_INT;

      VAR I[3], J[4] : INTEGER;
          STACK[8]   : ARRAY[0..15] OF INTEGER;

      BEGIN
         I := PSW - 1 AND 16_07 SHL 1;
         (* DECREMENT THE PROGRAM STATUS WORD. *)
         (* MASK THE STACK POINTER BITS WHICH  *)
         (* POINT TO THE STACK PAIRS.  SHIFT   *)
         (* FOR THE INDEX IN THE STACK ARRAY.  *)

         STACK[I] := ADDR_WORD 100;
         (* THE FIRST STACK ENTRY CONTAINS THE *)
         (* LOCATION WITHIN THE PAGE.          *)

         INCREMENT(I);
         J := STACK[I] AND 16_F8 + ADDR_PAGE 100;
         STACK[I] := J;
         (* THE SECOND STACK ENTRY CONTAINS    *)
         (* THE PAGE NUMBER IN THE LOWER THREE *)
         (* BITS.                              *)

      END (* EXTERN_INT *);

   BEGIN
     .
     .
    100:
     .
     .
   END.
```

10 USE OF THE PASCAL/48 COMPILER

The Pascal/48 cross compiler is an indirect access file on the Langley CYBER/NOS complex and is called PAS48 on the user number MICRO. It uses a single input file which contains the program to be compiled. It produces three output files which contain the listing, the object file, and an assembly language program equivalent to the machine instructions which the compiler generated. This last file could be assembled to produce the same object file as the compiler produced. The necessary NOS control cards are:

```
GET,PAS48/UN=MICRO.
PAS48,input,output,object,assembl.
```

The lower case names are the default files which the Pascal/48 compiler will use if no file is specified by the user. All files are rewound before a compilation.

The listing that the compiler generates shows the source program annotated with the corresponding line numbers and the approximate value of the location counter as it was at the beginning of each printed line. The value is approximate because of the non-uniform mapping between the source text and the generated instructions. If the compiler corrects for a page boundary problem the location counter will be further affected, since the listing is generated before the machine code is modified. The location counter values are useful in determining the points in programs where page boundaries are crossed.

Following the source program listing is the cross reference of identifiers and a map of the variables of RAM by their location. The RAM map provides an indication of the overloading of variables. Following this is the pseudo-assembly listing of the instructions that the compiler generated. Each instruction has a line number in its comment field and once again, these line numbers are only approximate.

Several options are available to control the Pascal/48 compiler's output. Options may be included in source programs inside comments or they may appear on the execute line. If they are used inside comments, a dollar sign must follow the opening (* and the options follow the dollar sign. Many options may appear in which case they are separated by commas.

Example:

```
(*$A+,I+ TURN ON ASSEMBLY LIST AND ROM IMAGE *)
```

Options may follow the last file name on the execute line and are separated from it by a slash.

Example:

```
PAS48,MYPROG,LISTING,MYOBJ/A+,X-,I+.
```

The options and their meaning are given in the following table:

A+	Turn assembly listing on (default).
A-	Turn assembly listing off.
B+	Code is modified for conditional jumps cross page boundaries (default).
B-	Code is not modified and errors emitted.
B=	Restore previous setting.
C+	Read 80 columns of the input records.
C-	Read 72 columns (default).
Cn	Read 'n' columns (10 < n < 120).
C=	Restore previous setting.
Dn	Number of internal data RAM-64 byte blocks (1 <= n <= 4). Default is 1.
H+	Turn on complete page headings (default).
H-	Turn page headings off.
H	Turn on partial headings (page ejects).
H=	Restore previous setting.
I+	Generate ROM image listing.
I-	No ROM image listing (default).
L+	Turn listing on (default).
L-	Turn listing off, error messages listed.
L*	Page eject.
L=	Restore previous setting.
L'cs'	cs is a character string used as a title.
Mn	Crystal frequency of 'n' hertz (if an integer) or megahertz (if n contains a decimal point). Default is 5.9904 MHz.
O+	Origin at next page boundary.
On	Origin at address 'n'.
P+	Print 52 lines per page.
P-	Print 40 lines per page (default).
P=	Restore previous setting.
Pn	'n' lines per page (11 < n < 1000).
R+	ROM size is eight 256-byte pages.
R-	ROM size is four pages (default).
T+	Equivalent to A-,H-,L+,X- (suitable for terminals).
T-	Restore previous listing option settings.
T=	Restore previous T option setting.
W+	Print warning messages (default).

W-	Suppress warning messages.
W=	Restore previous setting.
X+	Turn on cross reference listing (default).
X-	Turn off cross reference listing.

The assembly listing is a pseudo-assembly listing of the instructions that the compiler generated. The ROM image listing shows the contents of the ROM as it would look if the object program were loaded into an MCS-48 computer. The page heading and line count options are designed to allow the terminal user to disable the headings and page control because they are of most use when the listing is sent to a line printer. Since options can be specified on the execute line, the listing format can be changed temporarily when a compilation is being done at a terminal and the results displayed on the terminal. The cross reference is a cross reference of the identifiers used in the program. Extended program memory addressing has not been implemented in the Pascal/48 language, so the ROM size is limited to eight 256-byte pages. The crystal frequency option should only be included in the source program inside a comment and not on the execute line. The CYBER Control Language limits fields to seven characters so defining of the frequency in hertz is impossible. The decimal point in the megahertz notation terminates the the execution line and therefore the option list is also terminated.

11 APPENDIX A - SAMPLE PROGRAMS

Four programs are included here as examples of the use of Pascal/48. The first takes ASCII characters from a keyboard and translates them into Morse code. The remaining examples are different methods of turning the MCS-48 microcomputer into a simple stopwatch. These programs are examples and are intended to be used for reference purposes. None of the programs is written to be particularly functional or efficient.

The Morse code example reads an ASCII character on PORT1 and outputs Morse code on pin seven of PORT2. The BUS is not used. The seven least significant bits of PORT1 contain the character and the most significant bit is on while a key is pushed. The keyboard is assumed to generate both upper and lower case letters and the program takes account of this. The Morse code corresponding to the letters is contained in an array in ROM and the end of the code for each letter is indicated by a one followed by all zeros. A one in the code indicates a DAH and a zero a DIT. For example, the letter A is coded as the bit string 01100000 which gives a DIT then a DAH and then the end code.

The first stopwatch program assumes that a control switch is connected to bit seven of the BUS. Pushing the switch changes this bit from zero to one. The output is presented on PORT1 and PORT2 as BCD digits. The upper half of PORT1 is a minute count, the lower half and the upper half of PORT2 is a second count, and the lower half of PORT2 is a tenths of a second count. Upon initialization the display is all zeros. Pushing the switch once causes the watch to start counting in minutes, seconds and tenths of a second. Pushing the switch a second time causes the watch to stop counting and pushing it a third time resets the watch. The second stopwatch program uses the decimal adjust capability for a BCD counter, and outputs the minute count on PORT1 and the second count on PORT2. The third program uses an INTERSIL ICM 7218 LED driver for output displays.

-- Example Program 1 --

```
(*  THIS PROGRAM CONVERTS ASCII KEYBOARD INPUT TO MORSE CODE OUTPUT   *)

PROGRAM MORSE_CODE;

   CONST
      FIFTY_MILLISECONDS = 50;

   VAR
      CH[4] : INTEGER;


   PROCEDURE GET_KEY;

      VAR
         CODETABLE[,ROM] : ARRAY [0 .. 63] OF INTEGER;

      VALUE
         CODETABLE =
           (2_00000000, 7 OF 2_10000000,
            2_10110110, 2_10110110, 2_10000000, 2_10000000,
            2_11001110, 2_10000000, 2_01010110, 2_10010100,
            2_11111100, 2_01111100, 2_00111100, 2_00011100,
            2_00001100, 2_00000100, 2_10000100, 2_11000100,
            2_11100100, 2_11110100, 2_11100010, 2_10101010,
            2_10000000, 2_10000000, 2_10000000, 2_00110010,
            2_10000000, 2_01100000, 2_10001000, 2_10101000,
            2_10010000, 2_01000000, 2_00101000, 2_11010000,
            2_00001000, 2_00100000, 2_01111000, 2_10110000,
            2_01001000, 2_11100000, 2_10100000, 2_11110000,
            2_01101000, 2_11011000, 2_01010000, 2_00010000,
            2_11000000, 2_00110000, 2_00011000, 2_01110000,
            2_10011000, 2_10111000, 2_11001000, 2_10000000,
            2_10000000, 2_10000000, 2_10000000, 2_00000000);

      BEGIN  (* GET_KEY *)
         REPEAT
         UNTIL PORT1 BIT 7;   (* PIN 7 HIGH MEANS KEY PUSHED *)

         CH := PORT1 AND 2_01111111;

         REPEAT
         UNTIL NOT PORT1 BIT 7;   (* WAIT FOR PIN 7 LOW *)

         (* CONVERT LOWER CASE LETTERS TO UPPER CASE *)
         IF CH > 16_60 THEN CH := CH - 16_40
         ELSE CH := CH - 16_20;

         CH := CODETABLE[CH]   (* LOOK UP MORSE CODE *)
      END;  (* GET_KEY *)
```

-- Example Program 1 --

```
PROCEDURE SEND;

   VAR
      I[2] : INTEGER;


   PROCEDURE DIT;

      BEGIN
         PORT2 := PORT2 AND 2_11111110;
         DELAY(FIFTY_MILLISECONDS);
         PORT2 := PORT2 OR 2_00000001;
         DELAY(FIFTY_MILLISECONDS)
      END;  (* DIT *)


   PROCEDURE DAH;

      BEGIN
         PORT2 := PORT2 AND 2_11111110;
         DELAY(150);        (* 150 MILLISEC DELAY *)
         PORT2 :=PORT2 OR 2_00000001;
         DELAY(FIFTY_MILLISECONDS)
      END;  (* DAH *)


   BEGIN  (* SEND *)
      IF CH = 0 THEN   (* MEANS ILLEGAL CHARACTER *)
         FOR I := 8 DOWNTO 1 DO
            DIT
      ELSE WHILE CH <> 2_10000000 DO
         BEGIN
         IF NOT CH BIT 7 THEN
            DIT
         ELSE DAH;
         CH := CH SHL 1
         END  (* WHILE *)
   END;  (* SEND *)


BEGIN  (* THE MAIN PROGRAM: MORSE_CODE *)
   PORT2 := PORT2 AND 2_11111110;
   PORT1 := PORT1 OR 2_00000001;
   WHILE TRUE DO
      BEGIN
      GET_KEY;
      SEND
      END  (* WHILE *)
END.  (* THE MAIN PROGRAM *)
```

-- Example Program 2 --

```
PROGRAM STOP_WATCH;

   (* MAKE MCS-48 INTO A SIMPLE STOP WATCH      *)
   (* INPUT  - BIT 7 OF THE BUS                 *)
   (* OUTPUT - UPPER HALF PORT 1 MINUTES        *)
   (*          LOWER HALF PORT 1 TENS OF SECS   *)
   (*          UPPER HALF PORT 2 UNITS OF SECS  *)
   (*          LOWER HALF PORT 2 TENTHS OF SECS *)

   CONST
      TEN_MILLISEC = 128;

   VAR
      DIGITS[60]       : ARRAY[1..4] OF INTEGER;
      SWITCH_COUNT[2], INT_COUNT[3], I[4]
                       : INTEGER;



   PROCEDURE DISPLAY;

      (* DISPLAY TO 9:59.9 ON PORT 1 AND PORT 2 *)

      BEGIN
         PORT1 := DIGITS[1] SHL 4 + DIGITS[2];
         PORT2 := DIGITS[3] SHL 4 + DIGITS[4];
      END (* DISPLAY *);
```

-- Example Program 2 --

```
PROCEDURE TIMER_INT;

VAR
   SAVE_ACC[7] : INTEGER;

   BEGIN
      SAVE_ACC := ACCUMULATOR;
      TIMER := TEN_MILLISEC;
      START(TIMER);

      IF INT_COUNT = 9 THEN
         BEGIN
         INT_COUNT := 0;

         IF DIGITS[4] = 9 THEN
            BEGIN
            DIGITS[4] := 0;
            IF DIGITS[3] = 9 THEN
               BEGIN
               DIGITS[3] := 0;
               IF DIGITS[2] = 5 THEN
                  BEGIN
                  DIGITS[2] := 0;
                  IF DIGITS[1] = 9 THEN
                     DIGITS[1] := 0
                  ELSE
                     INCREMENT(DIGITS[1])
                  END
               ELSE
                  INCREMENT(DIGITS[2])
               END
            ELSE
            INCREMENT(DIGITS[3])
            END
         ELSE
            INCREMENT(DIGITS[4])
         END
      ELSE
         INCREMENT(INT_COUNT);

      DISPLAY;
      ACCUMULATOR := SAVE_ACC;
   END (* TIMER_INT *);
```

-- Example Program 2 --

```
BEGIN
   SWITCH_COUNT := 0;

   WHILE TRUE DO
      BEGIN
      CASE SWITCH_COUNT OF
         0 : BEGIN
             FOR I := 4 DOWNTO 1 DO
                DIGITS[I] := 0;
             DISPLAY;
             END (* 0 *);
         1 : BEGIN
             INT_COUNT := 0;
             TIMER := TEN_MILLISEC;
             START(TIMER);
             ENABLE(TIMER_INT);
             END (* 1 *);
         2 : BEGIN
             STOP(TIMER);
             DISABLE(TIMER_INT);
             END (* 2 *);
      END (* CASE *);

      WHILE BUS BIT 7 DO;

      WHILE NOT(BUS BIT 7) DO;

      INCREMENT(SWITCH_COUNT);
      IF SWITCH_COUNT = 3 THEN SWITCH_COUNT := 0;
      END (* WHILE *)
END (* STOP_WATCH *).
```

-- Example Program 3 --

```
PROGRAM STOP_WATCH;

   (* MAKE MCS-48 INTO A SIMPLE STOP WATCH *)
   (* INPUT  - BIT 7 OF THE BUS            *)
   (* OUTPUT - PORT 1 CONTAINS MINUTES     *)
   (*          PORT 2 CONTAINS SECONDS     *)

   CONST
      TEN_MILLISEC = 128;

   VAR
      SWITCH_COUNT[2] : INTEGER;
      INT_COUNT[4]    : INTEGER;
      SEC[5]          : INTEGER;
      MIN[6]          : INTEGER;


   PROCEDURE DISPLAY;

      (* DISPLAY BCD TO 99:59 ON PORTS 1 AND 2 *)

      BEGIN
         PORT1 := MIN;
         PORT2 := SEC;
      END (* DISPLAY *);
```

-- Example Program 3 --

```
PROCEDURE TIMER_INT;

   (* WHEN THE TIMER FLAG IS SET, THEN THE INT_COUNT *)
   (* REGISTER IS INCREMENTED ONE.  IF INT_COUNT IS  *)
   (* FILLED THIS PROCEDURE INCREMENTS THE SECONDS   *)
   (* AND DISPLAYS THEM.                             *)

VAR
   SAVE_ACC[7] : INTEGER;

   BEGIN
      SAVE_ACC := ACCUMULATOR;
      TIMER := TEN_MILLISEC;
      START(TIMER);

      IF INT_COUNT = 99 THEN
         BEGIN
         INT_COUNT := 0;
         IF SEC = 16_59 THEN
            BEGIN
            SEC := 0;
            MIN := DEC_ADJ(MIN + 1)
            END
         ELSE
            SEC := DEC_ADJ(SEC + 1)
         END
      ELSE
         INCREMENT(INT_COUNT);

      DISPLAY;
      ACCUMULATOR := SAVE_ACC;
   END (* TIMER_INT *);
```

-- Example Program 3 --

```
 BEGIN
    SWITCH_COUNT := 0;

    WHILE TRUE DO
       BEGIN
       CASE SWITCH_COUNT OF
          0 : BEGIN
              SEC := 0;
              MIN := 0;
              DISPLAY;
              END (* 0 *);
          1 : BEGIN
              INT_COUNT := 0;
              TIMER := TEN_MILLISEC;
              START(TIMER);
              ENABLE(TIMER_INT);
              END (* 1 *);
          2 : BEGIN
              STOP(TIMER);
              DISABLE(TIMER_INT);
              END (* 2 *);
       END (* CASE *);

       WHILE BUS BIT 7 DO;

       WHILE NOT(BUS BIT 7) DO;

       INCREMENT(SWITCH_COUNT);
       IF SWITCH_COUNT = 3 THEN SWITCH_COUNT := 0;
       END (* WHILE *)
END (* STOP_WATCH *).
```

-- Example Program 4 --

```
PROGRAM STOP_WATCH;

   (* THIS PROGRAM IS THE SAME AS THE PREVIOUS EXAMPLE *)
   (* EXCEPT IT USES AN INTERSIL ICM 7218 LED DRIVER   *)
   (* TO OUTPUT TO 7-SEGMENT DSPLAYS, AND IT ALSO      *)
   (* DISPLAYS .01 SECOND.  THE OUTPUT IN THIS EXAMPLE *)
   (* USES THE BUS AND BIT 0 OF PORT 1 TO CONTROL THE  *)
   (* 7218.  INPUT IS ON BIT 7 OF PORT 2.              *)

   CONST
      TEN_MILLISEC = 128;

   VAR
      SWITCH_COUNT[2] : INTEGER;
      INT_COUNT[3]    : INTEGER;
      SEC[4]          : INTEGER;
      MIN[5]          : INTEGER;


   PROCEDURE DISPLAY;

      (* DISPLAY WITH THE INTERSIL ICM 7218 LED DRIVER *)
      (* DISPLAY TO 99:59.99                           *)

   VAR
      DSPL[46] : ARRAY[1..8] OF INTEGER;
      I[6]     : INTEGER;

      BEGIN
         DSPL[1] := MIN ROTR 4 AND 16_F;        (* TENS OF MIN  *)
         DSPL[2] := MIN AND 16_F;               (* UNITS OF MIN *)
         DSPL[3] := 15;                         (* NOT USED     *)
         DSPL[4] := SEC ROTR 4 AND 16_F;        (* TENS OF SEC  *)
         DSPL[5] := SEC AND 16_F;               (* UNITS OF SEC *)
         DSPL[6] := 15;                         (* NOT USED     *)
         DSPL[7] := INT_COUNT ROTR 4 AND 16_F;  (* 0.1 SEC      *)
         DSPL[8] := INT_COUNT AND 16_F;         (* 0.01 SEC     *)
         PORT1   := 16_01; (* MODE LINE OF 7218 TO PORT1 BIT 0 *)
         BUS     := 16_90; (* MODE WORD FOR B CODE             *)
         PORT1   := 16_00; (* MODE LINE LOW TO START DATA      *)
         FOR I := 8 DOWNTO 1 DO (* LOAD ALL 8 DISPLAY BITS     *)
            BUS  := DSPL[I] OR 16_80

      END (* DISPLAY *);
```

-- Example Program 4 --

```
PROCEDURE TIMER_INT;

   (* WHEN THE TIMER FLAG IS SET, THEN THE INT_COUNT *)
   (* REGISTER IS INCREMENTED ONE.  IF INT_COUNT IS  *)
   (* FILLED THIS PROCEDURE INCREMENTS THE SECONDS   *)
   (* AND DISPLAYS THEM.                             *)

VAR
   SAVE_ACC[7] : INTEGER;

   BEGIN
      SAVE_ACC := ACCUMULATOR;
      TIMER := TEN_MILLISEC;
      START(TIMER);

      (* DECIMAL ADJUST WILL SET THE CARRY *)
      (* BIT IF OVERFLOW IS DETECTED.      *)
      INT_COUNT := DEC_ADJ(INT_COUNT + 1);
      IF CARRY THEN
         IF SEC = 16_59 THEN
            BEGIN
            SEC := 0;
            MIN := DEC_ADJ(MIN + 1)
            END
         ELSE
            SEC := DEC_ADJ(SEC + 1);

      DISPLAY;
      ACCUMULATOR := SAVE_ACC;
   END (* TIMER_INT *);
```

-- Example Program 4 --

```
BEGIN
   SWITCH_COUNT := 0;

   WHILE TRUE DO
      BEGIN
      CASE SWITCH_COUNT OF
         0 : BEGIN
             INT_COUNT := 0;
             SEC := 0;
             MIN := 0;
             DISPLAY;
             END (* 0 *);
         1 : BEGIN
             TIMER := TEN_MILLISEC;
             START(TIMER);
             ENABLE(TIMER_INT);
             END (* 1 *);
         2 : BEGIN
             STOP(TIMER);
             DISABLE(TIMER_INT);
             END (* 2 *);
      END (* CASE *);

      WHILE PORT2 BIT 7 DO;

      WHILE NOT(PORT2 BIT 7) DO;

      INCREMENT(SWITCH_COUNT);
      IF SWITCH_COUNT = 3 THEN SWITCH_COUNT := 0;
      END (* WHILE *)
END (* STOP_WATCH *).
```

12 APPENDIX B - SAMPLE LISTING

The fourth program of Appendix A is included here as an example of the output listing and object file generated by the Pascal/48 cross compiler. The Langley CYBER/NOS control cards used to generate the files from the input source STOPWAT are:

```
GET,STOPWAT.
GET,PAS48/UN=MICRO.
PAS48,STOPWAT,LISTING,OBJECT/I+,P48.
```

The file OBJECT is the object code of the stopwatch program. The appropriate record format, which is generated for the object file, includes the record mark, byte count, address, and type, the program memory, and the record checksum. All records of program memory are 16 bytes long starting at an even 16-byte address. The object code of the stopwatch program is:

```
:100000000470000403000004 4DFD47530FB82EA0F8
```

The file LISTING is the output listing of the compiler. The compiler options, I+ and P48, are selected to include the image of the object code and 48 lines per page for documentation purposes.

```
NASA, LANGLEY RESEARCH CENTER   PASCAL MCS-48  VERSION 83/12/30  PAGE  1
                                 84/01/03.     14.44.36.         CSC/NASA

009    PROGRAM STOP_WATCH;
009
009       (* THIS PROGRAM IS THE SAME AS THE PREVIOUS EXAMPLE *)
009       (* EXCEPT IT USES AN INTERSIL ICM 7218 LED DRIVER   *)
009       (* TO OUTPUT TO 7-SEGMENT DSPLAYS, AND IT ALSO      *)
009       (* DISPLAYS .01 SECOND.  THE OUTPUT IN THIS EXAMPLE *)
009       (* USES THE BUS AND BIT 0 OF PORT 1 TO CONTROL THE  *)
009       (* 7218.  INPUT IS ON BIT 7 OF PORT 2.              *)
009
009       CONST
009          TEN_MILLISEC = 128;
009
009       VAR
009          SWITCH_COUNT[2] : INTEGER;
009          INT_COUNT[3]    : INTEGER;
009          SEC[4]          : INTEGER;
009          MIN[5]          : INTEGER;
009
009
009
009       PROCEDURE DISPLAY;
009
009          (* DISPLAY WITH THE INTERSIL ICM 7218 LED DRIVER *)
009          (* DISPLAY TO 99:59.99                           *)
009
009       VAR
009          DSPL[46] : ARRAY[1..8] OF INTEGER;
009          I[6]     : INTEGER;
009
009          BEGIN
009             DSPL[1] := MIN ROTR 4 AND 16_F;       (* TENS OF MIN *)
010             DSPL[2] := MIN AND 16_F;              (* UNITS OF MIN*)
016             DSPL[3] := 15;                        (* NOT USED    *)
01A             DSPL[4] := SEC ROTR 4 AND 16_F;       (* TENS OF SEC *)
021             DSPL[5] := SEC AND 16_F;              (* UNITS OF SEC*)
027             DSPL[6] := 15;                        (* NOT USED    *)
02B             DSPL[7] := INT_COUNT ROTR 4 AND 16_F; (* 0.1 SEC     *)
032             DSPL[8] := INT_COUNT AND 16_F;        (* 0.01 SEC    *)
038             PORT1   := 16_01; (* MODE LINE OF 7218 TO PORT1 BIT 0*)
03B             BUS     := 16_90; (* MODE WORD FOR B CODE            *)
03E             PORT1   := 16_00; (* MODE LINE LOW TO START DATA     *)
040             FOR I := 8 DOWNTO 1 DO (* LOAD ALL 8 DISPLAY BITS    *)
042                BUS  := DSPL[I] OR 16_80
042
042          END (* DISPLAY *);
04D
04D
04D
```

```
NASA, LANGLEY RESEARCH CENTER   PASCAL MCS-48  VERSION 83/12/30  PAGE  2
                                 84/01/03.      14.44.37.         CSC/NASA

04D     PROCEDURE TIMER_INT;
04D
04D        (* WHEN THE TIMER FLAG IS SET, THEN THE INT_COUNT *)
04D        (* REGISTER IS INCREMENTED ONE.  IF INT_COUNT IS  *)
04D        (* FILLED THIS PROCEDURE INCREMENTS THE SECONDS   *)
04D        (* AND DISPLAYS THEM.                             *)
04D
04D     VAR
04D        SAVE_ACC[7] : INTEGER;
04D
04D        BEGIN
04D           SAVE_ACC := ACCUMULATOR;
04E           TIMER := TEN_MILLISEC;
051           START(TIMER);
052
052           (* DECIMAL ADJUST WILL SET THE CARRY *)
052           (* BIT IF OVERFLOW IS DETECTED.      *)
052           INT_COUNT := DEC_ADJ(INT_COUNT + 1);
057           IF CARRY THEN
059              IF SEC = 16_59 THEN
05E                 BEGIN
05E                 SEC := 0;
060                 MIN := DEC_ADJ(MIN + 1)
060                 END
067              ELSE
067                 SEC := DEC_ADJ(SEC + 1);
06C
06C           DISPLAY;
06E           ACCUMULATOR := SAVE_ACC;
06F        END (* TIMER_INT *);
070
070
070
070     BEGIN
072        SWITCH_COUNT := 0;
074
074        WHILE TRUE DO
074           BEGIN
074           CASE SWITCH_COUNT OF
078              0 : BEGIN
078                  INT_COUNT := 0;
078                  SEC := 0;
07A                  MIN := 0;
07C                  DISPLAY;
080                  END (* 0 *);
082              1 : BEGIN
082                  TIMER := TEN_MILLISEC;
085                  START(TIMER);
```

```
NASA, LANGLEY RESEARCH CENTER    PASCAL MCS-48   VERSION 83/12/30  PAGE  3
                                   84/01/03.      14.44.40.          CSC/NASA

086                    ENABLE(TIMER_INT);
087                    END (* 1 *);
089                2 : BEGIN
089                    STOP(TIMER);
08A                    DISABLE(TIMER_INT);
08B                    END (* 2 *);
08D            END (* CASE *);
090
090            WHILE PORT2 BIT 7 DO;
096
096            WHILE NOT(PORT2 BIT 7) DO;
09B
09B            INCREMENT(SWITCH_COUNT);
09C            IF SWITCH_COUNT = 3 THEN SWITCH_COUNT := 0;
0A3            END (* WHILE *)
0A5      END (* STOP_WATCH *).
```

```
NASA, LANGLEY RESEARCH CENTER    PASCAL MCS-48   VERSION 83/12/30  PAGE  4
SYMBOLIC CROSS REFERENCE MAP      84/01/03.      14.44.41.         CSC/NASA


MODULE         DEC  REFERENCES
STOP_WATCH       1

VARIABLE       TYPE-LENGTH DEC  REFERENCES
ACCUMULATO     T(PRE,   1)       60   77
BUS            I(PRE,   1)       40   43
CARRY          B(PRE,   1)       67
PORT1          I(PRE,   1)       39   41
PORT2          I(PRE,   1)      105  107
TIMER          I(PRE,   1)       61   62   95   96  100
TRUE           B(LIT,   1)       85

VARIABLE       TYPE-LENGTH DEC  REFERENCES
INT_COUNT      I(REG,   1)  15   37   38   66   66   89
MIN            I(REG,   1)  17   31   32   71   71   91
SEC            I(REG,   1)  16   34   35   68   70   74   74   90
SWITCH_COU     I(REG,   1)  14   83   87  109  110  110
TEN_MILLIS     I(LIT,   1)  11   61   95


MODULE         DEC  REFERENCES
DISPLAY         21    76    92

VARIABLE       TYPE-LENGTH DEC  REFERENCES
DSPL           I(RAM,   8)  27   31   32   33   34   35   36   37   38   43
I              I(REG,   1)  28   42   43


MODULE         DEC  REFERENCES
TIMER_INT       49

VARIABLE       TYPE-LENGTH DEC  REFERENCES
SAVE_ACC       I(REG,   1)  57   60   77


PROCEDURE LIST - INDENTATION INDICATES LEVEL OF NEST

  STOP_WATCH
     DISPLAY
     TIMER_INT
```

```
NASA, LANGLEY RESEARCH CENTER   PASCAL MCS-48  VERSION 83/12/30  PAGE  5
RAM DATA MAP                      84/01/03.     14.44.41.        CSC/NASA

RAM MODULE      VARIABLE    TYPE        LEN   DEC REFERENCES
02  STOP_WATCH  SWITCH_COU  INTEGER       1    14    83    87   109   110   110

RAM MODULE      VARIABLE    TYPE        LEN   DEC REFERENCES
03  STOP_WATCH  INT_COUNT   INTEGER       1    15    37    38    66    66    89

RAM MODULE      VARIABLE    TYPE        LEN   DEC REFERENCES
04  STOP_WATCH  SEC         INTEGER       1    16    34    35    68    70    74
                                                     74    90

RAM MODULE      VARIABLE    TYPE        LEN   DEC REFERENCES
05  STOP_WATCH  MIN         INTEGER       1    17    31    32    71    71    91

RAM MODULE      VARIABLE    TYPE        LEN   DEC REFERENCES
06  DISPLAY     I           INTEGER       1    28    42    43

RAM MODULE      VARIABLE    TYPE        LEN   DEC REFERENCES
07  TIMER_INT   SAVE_ACC    INTEGER       1    57    60    77

RAM MODULE      VARIABLE    TYPE        LEN   DEC REFERENCES
2E  DISPLAY     DSPL        INTEGER       8    27    31    32    33    34    35
                                                     36    37    38    43
```

NASA, LANGLEY RESEARCH CENTER PASCAL MCS-48 VERSION 83/12/30 PAGE 6
OBJECT CODE TABLE 84/01/03. 14.44.42. CSC/NASA

	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
000	04	70	00	04	03	00	00	04	4D	FD	47	53	0F	B8	2E	A0
010	23	0F	5D	B8	2F	A0	B8	30	B0	0F	FC	47	53	0F	B8	31
020	A0	23	0F	5C	B8	32	A0	B8	33	B0	0F	FB	47	53	0F	B8
030	34	A0	23	0F	5B	B8	35	A0	23	01	39	23	90	02	27	39
040	BE	08	FE	03	2D	A8	F0	43	80	02	EE	42	83	AF	23	80
050	62	55	23	01	6B	57	AB	E6	6C	23	59	DC	96	67	27	AC
060	23	01	6D	57	AD	04	6C	23	01	6C	57	AC	14	09	FF	93
070	27	D7	27	AA	FA	03	8D	B3	27	AB	27	AC	27	AD	14	09
080	04	90	23	80	62	55	25	04	90	65	35	04	90	78	82	89
090	0A	37	F2	96	04	90	0A	F2	9B	04	96	1A	23	03	DA	96
0A0	A3	27	AA	04	74	04	A5	00	00	00	00	00	00	00	00	00

```
LOC   OBJ CODE   SOURCE STATEMENT

000   04 70      L000:   JMP     L070
                         ORG     003H
003   04 03      L003:   JMP     L003
                         ORG     007H
007   04 4D      L007:   JMP     L04D
                 ;  PROCEDURE DISPLAY
009   FD         L009:   MOV     A,R5              ; LINE  21
00A   47                 SWAP    A                 ; LINE  31
00B   53 0F              ANL     A,#0FH            ; LINE  31
00D   B8 2E              MOV     R0,#2EH           ; LINE  31
00F   A0                 MOV     @R0,A             ; LINE  31
010   23 0F              MOV     A,#0FH            ; LINE  32
012   5D                 ANL     A,R5              ; LINE  32
013   B8 2F              MOV     R0,#2FH           ; LINE  32
015   A0                 MOV     @R0,A             ; LINE  32
016   B8 30              MOV     R0,#30H           ; LINE  33
018   B0 0F              MOV     @R0,#0FH          ; LINE  33
01A   FC                 MOV     A,R4              ; LINE  34
01B   47                 SWAP    A                 ; LINE  34
01C   53 0F              ANL     A,#0FH            ; LINE  34
01E   B8 31              MOV     R0,#31H           ; LINE  34
020   A0                 MOV     @R0,A             ; LINE  34
021   23 0F              MOV     A,#0FH            ; LINE  35
023   5C                 ANL     A,R4              ; LINE  35
024   B8 32              MOV     R0,#32H           ; LINE  35
026   A0                 MOV     @R0,A             ; LINE  35
027   B8 33              MOV     R0,#33H           ; LINE  36
029   B0 0F              MOV     @R0,#0FH          ; LINE  36
02B   FB                 MOV     A,R3              ; LINE  37
02C   47                 SWAP    A                 ; LINE  37
02D   53 0F              ANL     A,#0FH            ; LINE  37
02F   B8 34              MOV     R0,#34H           ; LINE  37
031   A0                 MOV     @R0,A             ; LINE  37
032   23 0F              MOV     A,#0FH            ; LINE  38
034   5B                 ANL     A,R3              ; LINE  38
035   B8 35              MOV     R0,#35H           ; LINE  38
037   A0                 MOV     @R0,A             ; LINE  38
038   23 01              MOV     A,#01H            ; LINE  39
03A   39                 OUTL    P1,A              ; LINE  39
03B   23 90              MOV     A,#90H            ; LINE  40
03D   02                 OUTL    BUS,A             ; LINE  40
03E   27                 CLR     A                 ; LINE  41
03F   39                 OUTL    P1,A              ; LINE  41
040   BE 08              MOV     R6,#08H           ; LINE  42
042   FE         L042:   MOV     A,R6              ; LINE  43
```

```
 LOC   OBJ CODE   SOURCE STATEMENT

 043   03 2D               ADD     A,#2DH          ; LINE  45
 045   A8                  MOV     R0,A            ; LINE  45
 046   F0                  MOV     A,@R0           ; LINE  45
 047   43 80               ORL     A,#80H          ; LINE  45
 049   02                  OUTL    BUS,A           ; LINE  45
 04A   EE 42               DJNZ    R6,L042         ; LINE  45
 04C   83                  RET                     ; LINE  45
                  ;  PROCEDURE TIMER_INT
 04D   AF         L04D:    MOV     R7,A            ; LINE  49
 04E   23 80               MOV     A,#80H          ; LINE  61
 050   62                  MOV     T,A             ; LINE  61
 051   55                  STRT    T               ; LINE  62
 052   23 01               MOV     A,#01H          ; LINE  63
 054   6B                  ADD     A,R3            ; LINE  66
 055   57                  DA      A               ; LINE  66
 056   AB                  MOV     R3,A            ; LINE  66
 057   E6 6C               JNC     L06C            ; LINE  67
 059   23 59               MOV     A,#59H          ; LINE  68
 05B   DC                  XRL     A,R4            ; LINE  68
 05C   96 67               JNZ     L067            ; LINE  68
 05E   27                  CLR     A               ; LINE  69
 05F   AC                  MOV     R4,A            ; LINE  70
 060   23 01               MOV     A,#01H          ; LINE  71
 062   6D                  ADD     A,R5            ; LINE  72
 063   57                  DA      A               ; LINE  72
 064   AD                  MOV     R5,A            ; LINE  72
 065   04 6C               JMP     L06C            ; LINE  72
 067   23 01      L067:    MOV     A,#01H          ; LINE  73
 069   6C                  ADD     A,R4            ; LINE  74
 06A   57                  DA      A               ; LINE  74
 06B   AC                  MOV     R4,A            ; LINE  74
 06C   14 09      L06C:    CALL    L009            ; LINE  75
 06E   FF                  MOV     A,R7            ; LINE  77
 06F   93                  RETR                    ; LINE  78
                  ;  PROGRAM STOP_WATCH
 070   27         L070:    CLR     A               ; LINE  79
 071   D7                  MOV     PSW,A           ; LINE  82
 072   27                  CLR     A               ; LINE  83
 073   AA                  MOV     R2,A            ; LINE  83
 074   FA         L074:    MOV     A,R2            ; LINE  84
 075   03 8D               ADD     A,#8DH          ; LINE  87
 077   B3                  JMPP    @A              ; LINE  87
 078   27         L078:    CLR     A               ; LINE  88
 079   AB                  MOV     R3,A            ; LINE  90
 07A   27                  CLR     A               ; LINE  91
 07B   AC                  MOV     R4,A            ; LINE  91
```

```
 LOC   OBJ  CODE   SOURCE  STATEMENT

 07C    27                  CLR        A                          ; LINE  92
 07D    AD                  MOV        R5,A                       ; LINE  92
 07E    14  09              CALL       L009                       ; LINE  92
 080    04  90              JMP        L090                       ; LINE  93
 082    23  80     L082:    MOV        A,#80H                     ; LINE  94
 084    62                  MOV        T,A                        ; LINE  95
 085    55                  STRT       T                          ; LINE  96
 086    25                  EN         TCNTI                      ; LINE  97
 087    04  90              JMP        L090                       ; LINE  98
 089    65         L089:    STOP       TCNT                       ; LINE  99
 08A    35                  DIS        TCNTI                      ; LINE 101
 08B    04  90              JMP        L090                       ; LINE 102
 08D    78         L08D:    DB          78H                       ; LINE 103
 08E    82                  DB          82H                       ; LINE 103
 08F    89                  DB          89H                       ; LINE 103
 090    0A         L090:    IN         A,P2                       ; LINE 104
 091    37                  CPL        A                          ; LINE 105
 092    F2  96              JB7        L096                       ; LINE 105
 094    04  90              JMP        L090                       ; LINE 105
 096    0A         L096:    IN         A,P2                       ; LINE 106
 097    F2  9B              JB7        L09B                       ; LINE 107
 099    04  96              JMP        L096                       ; LINE 107
 09B    1A         L09B:    INC        R2                         ; LINE 108
 09C    23  03              MOV        A,#03H                     ; LINE 110
 09E    DA                  XRL        A,R2                       ; LINE 110
 09F    96  A3              JNZ        L0A3                       ; LINE 110
 0A1    27                  CLR        A                          ; LINE 110
 0A2    AA                  MOV        R2,A                       ; LINE 110
 0A3    04  74     L0A3:    JMP        L074                       ; LINE 111
 0A5    04  A5     L0A5:    JMP        L0A5                       ; LINE 112
```

13 APPENDIX C - SYNTAX DIAGRAMS

<<< program >>>

```
---> PROGRAM ---> ident ---> ; ---> block ---> . ----->
```

<<< ident >>>

```
---> letter ---+---------------+------>
               ^               |
               |               |
               +<-- letter <---+
               +<-- digit  <---+
               +---    _   <---+
```

<<< int_const >>>

```
-+-+-> digit -+-> _ -+-+-+-> digit --+-+-->
 | ^          |      ^ ^ |           ^ |
 | |          |      | | |           | |
 | +----------+      | | +-> letter -+ |
 |                   | |               |
 +-------------------+ +---------------+
```

<<< block >>>

```
-+-> lab_part -+-+-> con_part -+-+-> var_part -+---+
 |             ^ |             ^ |             ^   |
 +-------------+ +-------------+ +-------------+   |
                                                   |
     +---------------------------------------------+
     |
     +-+-> val_part -+-+-> proc_part -+---> cmpnd_stmt ----->
       |             ^ |              ^
       +-------------+ +--------------+
```

```
<<<  lab_part  >>>

---> LABEL -+-> int_const -+-> ; ----->
            ^              |
            |              |
            +----- , <-----+
```

```
<<<  con_part  >>>

---> CONST -+-> ident ---> = ---> constant ---> ; -+--->
            ^                                      |
            |                                      |
            +--------------------------------------+
```

```
<<<  var_part  >>>

---> VAR -+-+-> ident ---> addr_spec -+-> : ---> type_spec ; -+--->
          ^ ^                         |                       |
          | |                         |                       |
          | +----------- , <----------+                       |
          |                                                   |
          +---------------------------------------------------+
```

```
<<<  addr_spec  >>>

          +----------------+
          |                |
          |                v
---> [ -+-> int_const -+-+-> , -+-> RAM  --+-> ] ----->
                       |        |          ^
                       |        |          |
                       |        +-> ROM  ->+
                       |        |          |
                       |        |          |
                       |        +-> XRAM ->+
                       |                   |
                       +-------------------+
```

```
<<<  type_spec  >>>

-+-> array_spec ---> OF -+-> scl_type ----->
 |                       ^
 |                       |
 +-----------------------+


<<<  array_spec  >>>

---> ARRAY ---> [ ---> int_const ---> .. ---> int_const ---> ] ----->


<<<  scl_type  >>>

-----+--> INTEGER ----------------------------------+--->
     |                                              ^
     |                                              |
     +--> BOOLEAN -+----------------------------+-->+
     |             |                            ^   |
     |             |                            |   |
     |             +-> [ -> int_const -> ] -----+   |
     |                                              |
     +--> CHAR -------------------------------------+


<<<  val_part  >>>

---> VALUE -+-> ident ---> = ---> val_con --- ; -+--->
            ^                                    |
            |                                    |
            +------------------------------------+


<<<  val_con  >>>

-+-> ( -+-+-> int_const ---> OF -+-> constant -+-> ) -+--->
 |      ^ |                      ^             |      ^
 |      | |                      |             |      |
 |      | +----------------------+             |      |
 |      |                                      |      |
 |      +-------------- , <--------------------+      |
 |                                                    |
 +------------------> constant -----------------------+
```

<<< proc_part >>>

```
-+-> proc_hdr ---> proc_body ---> ; -+--->
 ^                                   |
 |                                   |
 +-----------------------------------+
```

<<< proc_hdr >>>

```
---> PROCEDURE ---> ident ---> ; ----->
```

<<< proc_body >>>

```
-+-> EXTERNAL ---> [ ---> int_const ---> ] -+--->
 |                                          ^
 |                                          |
 +--------------> FORWARD ----------------->+
 |                                          |
 |                                          |
 +--------------> block  -------------------+
```

<<< cmpnd_stmt >>>

```
---> BEGIN -+-> statement -+-> END ----->
            ^              |
            |              |
            +---- ; <------+
```

<<< expression >>>

```
-+-> monadic_op ---> ( ---> expression ---> ) -+-+->
 |                                             ^ |
 |                                             | |
 ++-+-> monadic_op -+-----+--> constant ----->+ |
  ^ |               ^     |                   | |
  | |               |     |                   | |
  | +---------------+     +--> variable ------+ |
  |                                             |
  |                                             |
  +--------------- dyadic_op <------------------+
```

```
<<<  variable  >>>

            +---------------------------+
            |                           |
            |                           v
---> ident -+-> [ -+-> int_const -+-> ] -+--->
                   |              ^
                   |              |
                   +->   ident   -+
```

```
<<<  statement  >>>

-+-> int_const ----> : -+-+----> assgn_stmt ---------+----->
 |                      ^ |                          ^
 |                      | |                          |
 +----------------------+ +----> if_stmt     -------->+
                          |                          |
                          +----> while_stmt  -------->+
                          |                          |
                          +----> rept_stmt   -------->+
                          |                          |
                          +----> for_stmt    -------->+
                          |                          |
                          +----> case_stmt   -------->+
                          |                          |
                          +----> call_stmt   -------->+
                          |                          |
                          +----> goto_stmt   -------->+
                          |                          |
                          +----> cmpnd_stmt  -------->+
                          |                          |
                          +--------------------------+
```

```
<<<  assgn_stmt  >>>

---> variable ---> := ---> expression ----->
```

```
<<<  if_stmt  >>>

---> IF ---> expression ---> THEN ---> statement --+
                                                   |
                                                   |
                                                   |
          +----------------------------+-----------+
          |                            |
          |                            |
          |                            v
          +---> ELSE ---> statement ---+--->
```

```
<<<  while_stmt  >>>

---> WHILE ---> expression ---> DO ---> statement ----->
```

```
<<<  rept_stmt  >>>

---> REPEAT -+-> statement -+-> UNTIL ---> expression ----->
             ^              |
             |              |
             +------ ; <----+
```

```
<<<  for_stmt  >>>

---> FOR ---> for_list ---> DO ---> statement ----->
```

```
<<<  for_list  >>>

---> ident ---> := ---> con_var -+->   TO   -+-> con_var ----->
                                 |           ^
                                 |           |
                                 +-> DOWNTO -+
```

<<< case_stmt >>>

```
                                          +-----------------+
                                          |                 |
                                          |                 v
---> CASE ---> expression ---> OF -+--> case_list_elt -+--> END ----->
                                   ^                    |
                                   |                    |
                                   +------- ; <---------+
```

<<< case_list_elt >>>

```
-+-> constant -+--> : ---> statement ----->
 ^             |
 |             |
 +----- , <----+
```

<<< goto_stmt >>>

```
---> GOTO ---> int_const ----->
```

<<< call_stmt >>>

```
---> ident ----->
```

14 APPENDIX D - ERROR MESSAGES

Four classes of error messages are provided to aid the Pascal/48 programmer in generating correct programs. The lower case notation in the messages are compiler derived from the Pascal/48 program source.

ch	A single character.
id	The significant characters of an identifier.
sym	An internal Pascal/48 symbol.
num	A decimal number.
hex	A hexadecimal number.

The internal symbols are the reserved words, special symbols, and terminals of the language grammar. The terminals include the special notation of <CONST> for a constant and <IDENT> for an identifier.

WARNING AT LINE num

The rules of the language are not violated, however, the code generated may not be what the user intended. The assembly listing should be checked for correctness.

BASE num IS NOT 2, 8 OR 16

The base specified is non-standard and may detract from the meaning of the program.

CASE STATEMENT CODE IS SHIFTED hex LOCATIONS TO NEXT PAGE

A page boundary was crossed during a CASE statement. The complete CASE statement is moved to the next page boundary.

GOTO num IN id POSSIBLE PSW ERROR

A GOTO out of a procedure has occurred but the compiler will not modify the program status word or the program stack.

id INTERRUPT HANDLER REFERENCED

Interrupt routines should be referenced through the hardware interrupt mechanism and not with software references.

INTERUPT REDEFINED BY id

The hardware locations for the timer interrupt and counter interrupt interface are the same. Either the timer or counter interrupt can be defined in one program.

LINES FOLLOWING END OF PROGRAM IGNORED

Non-comments follow the program terminating period.

id LOCATED WITHIN COMPILER TEMPORARIES

Registers 0 and 1 are used by the compiler and should not be used by the programmer.

id LOCATED WITHIN PROGRAM STACK
RAM locations 8 through 24 contain the program stack and redefinition of the stack will affect the procedure communication.

MAXIMUM STRING LENGTH num EXCEEDED
A character string has exceeded the compiler limits and will be truncated.

ORIGIN OF hex LESS THAN LOCATION hex
The current location counter is greater than the specified origin option value.

id REDEFINED BY PROGRAM
A predeclared variable has been redefined by the program for the current scope.

ROM VARIABLE id NOT INITIALIZED
A ROM variable has not been initialized with the VALUE statement.

TOO FEW VALUES SPECIFIED FOR id
An array of a ROM variable is not completely initialized with the VALUE statement.

UNREFERENCED LABEL : num
A label has been defined but has not been referenced within the scoping limits.

UNREFERENCED PROCEDURE id
A procedure has been defined but has not been referenced within the scoping limits.

SEMANTIC ERROR AT LINE num

The rules of the language are violated and the code generation is terminated. The program semantics and syntax will be continually checked.

ADDRESS SPECIFICATION REQUIRED FOR id

Declarations of variables must include an address specification.

ARRAY REFERENCE EXPECTED FOR id

The variable has been declared as an array and all references to the variable must be indexed.

CONSTANT num EXCEEDS RANGE FOR id

The range of a hardware memory device has been exceeded by the constant.

CONTROL VARIABLE id IS REDEFINED

The control variable of a FOR loop has been redefined within the loop.

DIGIT ch GREATER THAN BASE num

A digit in a constant is greater than the base specified.

EMPTY INPUT FILE

The source file specified is not local to the control point.

num ERRORS PER LINE EXCEEDED

The internal stack for errors per line is exceeded. The errors are continued for the error summary but not noted in the source listing.

FORWARD PROCEDURE id NOT DEFINED

A procedure has been declared as a forward reference but has not been defined.

GLOBAL ROM REQUIRES ADDRESS

The optional address specification for ROM variables is required for global variables.

IDENTIFIER id DECLARED TWICE

The identifier has been declared more than once within the current scope level.

IDENTIFIER id NOT DECLARED

The identifier referenced has not been declared.

ILLEGAL ADDRESS REFERENCE num

An arrayed ROM variable initialization crosses the page boundary in the VALUE statement.

ILLEGAL BASE num SPECIFIED
A base value of 0 or greater than 35 is specified.

ILLEGAL CHARACTER ch
An invalid Pascal/48 character is encountered and is ignored.

ILLEGAL MEMORY REFERENCE id
The identifier is not a valid memory reference. Variable declarations must be RAM, ROM or XRAM. The optional address specification for variables and the value initialization of variables must be ROM.

ILLEGAL STRING REFERENCE
A character string is only valid in the VALUE statement to initialize arrays of characters.

INDEX num OUT OF BOUNDS FOR id
The index exceeds the declared array limits of the variable.

INVALID ARRAY REFERENCE FOR id
The variable has been referenced as an array but has not been declared as an array.

INVALID BIT BOOLEAN REFERENCE
A bit specified Boolean has been reference in an illegal context.

INVALID BIT REFERENCE num
A bit reference can only be in the range 0-7.

INVALID EXPRESSION REFERENCE
An expression is encountered in a predeclared procedure reference.

INVALID LITERAL EXPRESSION
A forward label reference with an ADDR_WORD or ADDR_PAGE monadic operation is contained in a literal expression. The monadic operations are treated as literals so the compiler attempts to evaluate the literal expression and therefore would destroy the internal code for patching the forward reference.

INVALID LOOP CONTROL VARIABLE id
A loop control variable can only be a simple integer.

INVALID SCOPE REFERENCE FOR id
The Pascal scoping rules have been violated with the identifier reference.

INVALID TRANSLATE OPERATOR FOR OPERATION id
An invalid internal operator has been generated. This error should not occur and if it does it should be reported.

INVALID TYPE SPECIFICATION id
Only BOOLEAN, INTEGER and CHAR types are legal.

id IS NOT A CONSTANT
The identifier referenced has not been declared as a constant.

id IS NOT A PREDECLARED PROCEDURE
Arguments are only valid for the predeclared procedures.

LOW BOUND EXCEEDS HIGH BOUND
The array specification is in error.

MULTIDECLARED LABEL : num
A label number is declared more than once.

MULTIDEFINED CASE LABEL : num
A case label has been assigned to more than one statement.

MULTIDEFINED LABEL : num
A label has been assigned to a statement more than once.

NULL STRING INVALID
Strings must contain one or more characters.

PAGE BOUNDARY ERROR ENCOUNTERED AT hex REFERENCED AT hex
A conditional jump cross a page boundary has been generated by the compiler. The programmer can either select the boundary compiler option or adjust the code with the origin compiler option.

PARSE TERMINATED
A compiler abort has occurred. The listing is continued but no supplement information is provided beyond this point.

READ ONLY VARIABLE id
Assignments cannot be made to ROM variables or specialized predeclared variables.

REDEFINED ROM AT LOCATION hex
ROM is redefined by overloaded variables or code is generated at a ROM variable data location.

REFERENCE MUST BE id
The identifier specifies the structure allowed in the context of the source.

RUNAWAY NESTED COMMENT
The beginning comment symbol (* has not been matched by the terminating symbol *).

STATEMENT TOO COMPLICATED
The internal stack limits have been exceeded. This implies the nesting levels are excessive or a statement is too complex for the compiler parsing scheme.

TOO MANY VALUES SPECIFIED FOR id
The initialization list in a VALUE statement exceeds the range of the array.

id TYPE IS EXPECTED
The identifier specifies the only data type allowed in the source structure.

id TYPE IS UNEXPECTED
Mixed mode has occurred and the type specified is the first conflicting type.

UNABLE TO PATCH LABEL num ADDRESS
The compiler was unable to patch the label address for a forward reference by the ADDR_WORD or ADDR_PAGE monadic operators.

UNDECLARED LABEL : num
A label referenced has not been declared.

UNDEFINED LABEL : num
A label referenced has not been assigned to a statement.

UNEXPECTED IDENTIFIER id
Either an identifier is not expected in the context or a key word is expected.

SYNTAX ERROR AT LINE num

The grammar rules of the language are violated. An attempt is made to fix the syntax and the parsing is continued. Syntax errors occur in pairs. The first error points to the symbol where the error is detected. The recovery scheme scans ahead in the source and an attempt is made to fix the error with respect to the context. The second error specifies the fix to obtain the correct grammar syntax with respect to the symbol specified in the first error. The fix of the syntax error may not be what the programmer intended so the code generation and semantic checking are terminated.

<EOF> ENCOUNTERED DURING RECOVERY

While attempting a syntax error recovery an end-of-file was encountered. This error commonly occurs when the BEGIN-END pairs of compound statements do not match.

EXPECTED SYMBOL : sym

The symbol specified is necessary in the source context and to obtain exceptable syntax the symbol is inserted.

RECOVER FAILURE LIMIT EXCEEDED

More than three recovery failures has occurred in the program. It is assummed that any errors occurring beyond this point will be meaningless. The parsing is terminated but the listing is continued.

RECOVERY ATTEMPTED AT SYMBOL : sym

The symbol specified violates the syntax rules of the grammar. An attempt will be made to correct the syntax by deleting or replacing the symbol, or by inserting a symbol before the symbol specified.

RECOVERY FAILED, CONTINUE SCAN AT : sym

Usually more than one syntax error is encountered in a statement and the recovery method cannot correct the error with a simple deletion, insertion or replacement. The compiler searches forward until parsing can be resumed.

SUBSTITUTED SYMBOL : sym

The symbol specified replaces another symbol in the source to obtain exceptable syntax.

UNEXPECTED SYMBOL : sym

The symbol specified is not valid in the source context and to obtain exceptable syntax the symbol is deleted.

COMPILER ERROR AT LINE num

Compiler errors should not occur and indicate that the compiler is in error. If encountered such errors should be reported. Compiler errors are caused by semantic errors which generate erroneous internal variable values. The compiler should define exceptable internal data when an error condition is encountered. Abnormal termination of the compiler other than the compiler halt with a dayfile message should also be reported.

NUMBER num id num

Compiler information for problem evaluation.

INDEX

Accumulator, 20, 26, 27, 43.

ADDR_PAGE, 20, 44.

ADDR_WORD, 20, 44.

Array, 5, 6, 11, 12, 13, 14, 15.

ASCII, 4, 5.

Assembly, 45, 46, 47.

Assignment, 26, 27, 29, 42.

BCD, 20.

BEGIN-END, 3, 19, 31.

BIT, 21, 22.

Boolean, 4, 5, 12, 13, 30.

Boundary, 16, 17, 37, 42, 45, 46.

BUS, 26, 27, 32, 34.

CARRY, 26, 27.

CASE OF, 3, 37, 38.

Character, 2, 4, 5, 12, 13, 14.

CHR, 20, 21.

CLOCK_OUT, 41.

Comments, 3, 45.

COMPLEMENT, 42.

Constant, 4, 10, 14.

COUNTER, 26, 41.

COUNT_INT, 41, 43.

DECREMENT, 42.

DEC_ADJ, 20, 21.

DELAY, 41, 42.

DISABLE, 41.

ENABLE, 41.

Expression, 23, 24, 25.

EXTERNAL, 16, 17.

EXTERN_INT, 41, 43, 44.

Flags, 26.

FOR TO(DOWNTO) DO, 3, 35, 36.

FORWARD, 16, 17.

Frequency, 41, 42, 46, 47.

GOTO, 3, 9, 39.

Identifier, 2.

IF THEN ELSE, 2, 3, 30, 31.

INCREMENT, 42.

INLINE, 42.

Input, 45, 46.

Integer, 4, 5, 12, 13.

Interrupts, 41, 43, 44.

Label, 9, 20, 37, 39.

Listing, 45, 46, 47.

NOT, 4, 20.

Object, 42, 45, 47.

Operand, 20, 21, 22, 23, 24, 42.

Operator, 20, 21, 22, 23, 24.

Option, 45, 46, 47.

ORD, 20, 21.

Origin, 12, 16, 18, 46.

Output, 45.

Page, 12, 13, 16, 20, 37, 42.

PAS48, 45.

Pin, 26.

PORT1, 26, 27, 32, 34.

PORT2, 26, 27, 32, 34.

Ports, 26.

Predeclared Procedure, 36, 41, 42, 43.

Predeclared Variable, 26, 27.

Procedure, 16, 17, 18, 40, 41, 42, 43, 44.

Program, 7, 8.

PSW, 26, 39, 44.

RAM, 11, 12, 13, 45.

Registers, 13, 36, 42.

REG_BANK0, 42.

REG_BANK1, 42, 43.

REPEAT UNTIL, 3, 34.

Reserved Words, 2, 3.

ROM, 8, 11, 12, 13, 14, 16, 17, 46, 47.

Rotation, 21, 22.

Scope, 17, 37.

SELECT, 42, 43.

Shift, 21, 22.

Source, 45, 47.

Stack, 13, 20, 39, 40, 43.

START, 41.

Statement, 19, 28, 29, 30, 32, 34, 35, 36, 37, 39, 40.

STOP, 41.

TIMER, 26, 41.

TIMER_FLAG, 26.

TIMER_INT, 41, 43.

Type, 4, 5, 12.

TO, 26.

T1, 26.

VALUE, 3, 5, 14, 15.

Variable, 2, 4, 5, 11, 12, 13, 14, 26, 27.

WHILE DO, 3, 32, 33.

XRAM, 11, 12.

1. Report No. NASA TM 85752	2. Government Accession No.	3. Recipient's Catalog No.
4. Title and Subtitle Pascal/48 Reference Manual		5. Report Date January 1984
		6. Performing Organization Code 505-31-83-02
7. Author(s) John C. Knight and Roy W. Hamm		8. Performing Organization Report No. Central Scientific Computing Complex M-3
		10. Work Unit No.
9. Performing Organization Name and Address NASA Langley Research Center Hampton, VA 23665		11. Contract or Grant No.
		13. Type of Report and Period Covered Technical Memorandum
12. Sponsoring Agency Name and Address National Aeronautics and Space Administration Washington, DC 20546		14. Sponsoring Agency Code
15. Supplementary Notes		

16. Abstract

Pascal/48 is a programing language for the Intel MCS-48 series of microcomputers. In particular, it can be used with the Intel 8748. It is designed to allow the programer to control most of the instructions being generated and the allocation of storage. The language can be used instead of assembly language in most applications while allowing the user the necessary degree of control over hardware resources. Although it is called Pascal/48, the language differs in many ways from Pascal. The program structure and statements of the two languages are similar, but the expression mechanism and data types are different.

The Pascal/48 cross-compiler is written in Pascal and runs on the CDC CYBER NOS system. It generates object code in Intel hexadecimal format that can be used to program the MCS-48 series of microcomputers.

This reference manual defines the language, describes the predeclared procedures, lists error messages, illustrates use, and includes language syntax diagrams.

17. Key Words (Suggested by Author(s)) microprocessor, microcomputer, compiler, Pascal, and HOL		18. Distribution Statement Unclassified--Unlimited Subject Category 61	
19. Security Classif. (of this report) Unclassified	20. Security Classif. (of this page) Unclassified	21. No. of Pages 90	22. Price* A05

* For sale by the National Technical Information Service, Springfield, Virginia 22161