PASCAL

PROGRAMMA

PRODUCT DESCRIPTION

T I N Y   P A S C A L

USERS MANUAL
VER 1.0

(C) 1979 COPYRIGHT

MARTIN TRACY

AUGUST 1979

PROGRAMMA INTERNATIONAL, INC.
3400 Wilshire Blvd.
Los Angeles, CA 90010

(213) 384-0579 • 384-1116 • 384-1117

P R E F A C E

This reference publication is intended for programmers using the PROGRAMMA TINY PASCAL System. This publication describes how to write Tiny Pascal source statements, system start-up, alteration of system parameters, and error handling.

The original specifications and description of the implementation of the Tiny Pascal System can be obtained from the September, October, and November 1978 issues of BYTE Magazine. The article "A Tiny Pascal Compiler" by Kin-Man Chung and Herbert Yuen may be purchased as a reprint directly from BYTE. Their address is:

BYTE MAGAZINE
70 Main Street
Peterborough, NH 03458

An excellent book to introduce the reader to the PASCAL language is "Programming in PASCAL" by Peter Grogono. The book is available directly from PROGRAMMA International, Inc. or the publisher: Addison-Wesley Publishing Company, Inc.

The reader should be familiar with the hardware manuals and operational procedures of the devices attached to his particular computer configuration.

TABLE OF CONTENTS

What is "Tiny Pascal"?

Tiny Pascal is a modified subset of the Pascal programming language as defined by Kin-Man Chung and Herbert Yuen in BYTE magazine September, October, and November 1978.

Tiny Pascal is an integer only language, similar to Integer BASIC in the Apple II. The comparison between Tiny Pascal and a full Pascal is analagous to the comparison between Integer BASIC and Applesoft.

Tiny Pascal is very well suited for writing games and systems software for the Apple II. Programs created using the Tiny Pascal compiler are compatible with existing Apple Hardware/Software products (i.e., the DOS, printer interfaces, modems, etc.).

Databases created by Tiny Pascal can be manipulated by Applesoft or Integer BASIC. Likewise, data created by Applesoft or Integer BASIC can be read in, manipulated, and written out using Apple's DOS. There are no new DOS commands to learn. You create and maintain disk files from Tiny Pascal in the same manner that you would from BASIC.

What is a "Compiler"?

Both Integer BASIC and Applesoft are interpreters. An interpreter reads one line of text at a time and executes the statement. Interpreters are very effective for interactive program development, unfortunately they are very slow. Compilers, on the other hand, take the textfile created by the user and generate machine code for the specific processor. Compiled programs run very fast (often 1Ø-2Ø times faster) than an interpreted program. The drawbacks to a compiled program include: (1) Program development is no longer interactive, increasing the difficulties associated with debugging your programs. (2) Machine code versions of a program often require more memory space than interpreted versions. (This is especially true if a good interpreter, such as Integer BASIC, is being used.)

Despite these drawbacks often speed is very important so a compiler must be used.

What is "P-code"?

The Tiny Pascal compiler does not generate 65Ø2 machine code. It generates machine code for an imaginary machine called a "P-machine". Since this P-machine does not exist the P-machine has to be emulated by the 65Ø2 microprocessor. In effect the P-machine code gets interpreted by a 65Ø2 program - the P-code interpreter.

Since the P-code is being interpreted it will run much slower than if 65Ø2 machine code were produced. However, the interpretation of P-code is much faster than the interpretation of BASIC so Tiny Pascal programs will run 2-4 times faster than an equivalent BASIC program. P-code has two other advantages. First, P-code is very compact. A program written and compiled in Tiny Pascal will only require 1/2 to 3/4 the space required by an equivalent Integer BASIC program. By comparison, a Tiny Pascal program compiled directly to machine language would probably require 2-3 times the memory (of the Integer BASIC example) just to hold the program.

The second advantage to P-code is the fact that it makes the Tiny Pascal compiler easier to write, thereby freeing up even more memory.

So Tiny Pascal is a compromise between a true compiler and an interpreter. P-code is very compact, like an interpreter, but runs slower than the actual 65Ø2 code.

What do I get when I purchase "Tiny Pascal?"

The Tiny Pascal system includes the P-code interpreter, a Pascal monitor/ command interpreter, the Pascal compiler, an editor, source listings of the editor and monitor, a set of library subroutines which allow you to perform graphics manipulations, read the paddles, switches, etc. (source listing is provided), several Tiny Pascal programming examples including Cannibals (a game) and Detoken (a program which detokenizes Integer BASIC programs).

How is the Tiny Pascal system used?

That's easy! Boot the disk provided. The Apple will clear the screen, print a "coldstart" message and prompt you for a command.

To get a menu describing each of the commands press return. The Pascal monitor will print onto the video screen E(ditor), C(ompiler), U(ser), B(ASIC), ^D(OS), L(OAD), S(AVE), F(ENCES).

The parentheses around all the first characters simply means you only need to press the first character of each command to invoke the desired function. All commands must be terminated by a Return. (This prevents you from automatically executing a function when you accidentlly hit a key.) Note that the Pascal monitor only looks at the first key pressed. EAT <cr>, Elephant <cr>, Egads <cr> (where <cr> is return) all put you into the editor.

What do all these commands mean?

B-B(ASIC): Returns you to BASIC (A/S or Integer, depending). To prevent possible damage to the Tiny Pascal system you should immediately set LOMEM:2Ø48Ø. This will prevent damage to the Pascal system, however any source file or user program may be destroyed. If you wish to prevent possible damage to a textfile you should set LOMEM even higher. (e.g, 32767) To return to the Pascal system type Call 2Ø48.

RESET: Actual system reset. On an Apple II system (without the auto start ROM) RESET will deactivate the DOS and jump to the Apple monitor. If you have an Apple II plus or the auto start ROM, the Reset key is undefined. Sometimes it will boot the disk, other times it will return you to BASIC.

D-D(OS): ^D means control D. This allows you to execute a disk command directly from the Pascal monitor. Simply type control D and then the DOS command followed by <cr>. Note: You do not need to hit return after the control D. Example: command: ^D catalog <cr>

L-L(OAD): Loads a file into memory, beginning at the fence. You can specify a filename in one of two manners. (1) Hit return immediately after the L. The Pascal monitor will prompt you for a filename. (2) Type L (plus any number of non-blank, non- carriage return characters) followed by at least one blank. After the blank specify the filename followed by <cr>.

Examples:

```
     Command: L <cr>
     File: Test

     Command: L Test

     Command: LOAD Test

     etc.
```

E-E(ditor): Involves the Pascal system editor. The editor will be described in more detail later.

S-S(AVE): SAVES the last file edited or the last program compiled onto the disk. Syntax is similar to the L(OAD) command. Warning: You cannot L(OAD) and then immediately S(AVE) a source file. If you do not look at the source file with the editor, the Pascal monitor does not know how long the file is and will try to S(AVE) it with a length of zero. The resulting syntax error returns you to BASIC.

C-C(ompile):Compiles the Tiny Pascal program currently at the FENCE (the fence will be described later, don't worry!). The COMPILE/SYNTAX (S): message indicates that the compiler has two options and that the default option is (s)yntax. The syntax option tells the compiler to go ahead and compile the program but don't generate any P-code. This allows you to make sure that errors do not exist in your program before the actual compilation takes place. To select the default option type "S" or simply hit return. The compiler will begin compiling your program, printing the current line number to the left of the line. When an error is detected the appropriate message will be displayed and you will be returned to the Pascal monitor. You may now enter the editor, correct the problem, then recompile the program using the syntax option once again. When the program compiles successfully you can compile the program using the (C)ompile option. At the command level type C <cr> when the Apple displays "Compile/ SYNTAX(S):" Type "C". This will instruct the compiler to generate P-code for your Tiny Pascal program. Upon pressing "C" the Pascal compiler will ask you where you want the P-code to be stored during compilation. It will suggest the current fence value. Since your textfile is currently sitting at the fence the P-code will wipe out part of your program should you desire to select the default value. Since the program is usually shorter than the source it is possible for the two to co-exist in memory at the same time. When compilation of the program is complete the source file is considered DESTROYED. (So make sure you've saved it to disk beforehand.) When you press "C" for compile the compiler will respond with

```
P-code ADDR (5000):
```

where 5000 is the current (HEX) fence value. If you do not want to wipe out the textfile in memory or you would prefer to have the P-code stored in memory somewhere besides the fence you can specify this location by typing a four digit HEX number after the colon. If the fence is okay with you simply type <cr>. Should you type an illegal hex digit, or more than four hex digits, the Apple will keep at you and make you type the address back in. Note: Backspace is an illegal HEX digit!

Warning: If you specify that the P-code be stored in memory beginning somewhere in the middle of the source file you will destroy the textfile before the compiler gets a chance to compile it. Result: Error 104, undeclared identifier. If you change the P-code ADDRESS make sure that you specify an address beyond the last character in your textfile (the editor provides you with the required information). Once you've informed the compiler where the P-code is to be stored during compilation, you must tell the compiler where the P-code will be stored WHEN THE PROGRAM IS RUN.

When the compiler asks you for

```
P-CODE ORIG (5ØØØ):
```

you can specify this information. The default value will be whatever value you decided on for the P-CODE ADDR previously.

If you override the default with, let's say P-CODE ORIG (5ØØØ):3ØØØ, then you have made a promise to the compiler that you will load the program to that location before trying to run it. You can move the program by using the Apple II system monitor move command or by specifying the address in the DOS BLOAD instruction. Or you can write a Pascal program to move it for you.

Example: let's compile CANNIBAL.S into a useable program.

1) L(OAD) Cannibal.S (i.e, L CANNIBAL.S)
2) Compile (i.e, command:C)
3) Select compile mode (i.e, Compile/ SYNTAX(S):C)
4) Use default values for P-CODE Address and P-code orig.
(i.e, P-CODE ADDR (5ØØØ):<cr>
P-code orig (5ØØØ): <cr>)

The compiler finishes with the status message describing the size and location of the program then returns to the Pascal monitor. You may now save the compiled program onto disk by using the S(AVE) command. Be careful not to name the program after the source file (i.e, Cannibal.S) or you will replace the source file with the code file. By convention source files should have ".S" suffixed to the program name and code files should suffixed with ".C"

Warning: You must compile a program with the (C)ompile option before you can use the S(AVE) command. The S(yntax) option does not produce a program file. Let's name the program "Cannibal.C" and save it onto disk.

```
Command: SAVE Cannibal.C <cr>
```

U-U(ser): is used to run a program.

```
Command:U <cr>
Run ADDR (5ØØØ):
```

The Run ADDR (5ØØØ): informs you that, unless otherwise specified, the current Pascal program at location 5ØØØ will be executed. Once again you can override the default by specifying a four digit Run address. For example, if we specify Run ADDR (5ØØØ):3ØØØ we will end up in the editor (which begins at location 3ØØØ - better make sure a textfile is loaded first however!).

Since we just compiled Caniabal.S at 5ØØØ all we need to do is hit return to begin execution. To return from Cannibal to the Pascal monitor first play Cannibal. If you get impatient (or frustrated!), use the RESET sequence.

F-F(ENCE): sets the lower and upper bounds of the Pascal STACK as well as the fence. The FENCE is described in more detail in the section on more advanced topics.

What does "Tiny Pascal" look like?

Tiny Pascal uses the same program structure as regular Pascal. A Tiny Pascal program begins with a "PROGRAM" statement. The program statement is of the form "PROGRAM progname"; where progname is the program name which is chosen by the programmer. It can be any length but only the first eight characters are significant. Beyond the eighth character all further characters are ignored.

Examples:

```
Program Test;
Program Hires;
Program Thisisalongname;
Program Thisisalongername;
```

Note: As far as the compiler is concerned "Thisisalongname" and "Thisisalongername" are one and the same since the first eight characters match.

Following the program statement comes the constant declarations. Constants are simply handy abbreviations. They do not exist at program run time, and take up no memory space. They are used to increase program readability.

For Example: The constant declaration BS=Ø8; simply says whenever I say "BS" I really mean to use decimal Ø8. To declare some constants type "CONST" after the "Program" statement. After the "CONST" you can declare as many constants as you desire, separated by semicolons.

```
Example:
    CONST BS=Ø8; CR=13;
    TRUE=1; FALSE=Ø;
    LF=1Ø;
```

Do not confuse constants with variable names. You cannot assign a value to a constant anymore than you can assign a value to 8. The constant section is optional, if you don't have any constants you do not put a "Const" in your program.

Following the constant declarations comes the variable declarations. Variable declarations begin with the word "VAR". Following the "VAR" come the individual variable declarations. ALL VARIABLES IN A TINY PASCAL PROGRAM MUST BE DECLARED BEFORE THEY ARE USED. The only data type currently supported by Tiny Pascal is Integer. Arrays of integer are also allowed. To define an integer variable use the following form.

```
VAR Variablename:Integer;
```

As with the "CONST" declarations many variables can be declared by separating the variable declarations by a semicolon.

```
EG;
    VAR I:Integer; J:Integer;
    K:Integer; L:Integer;
```

Alternately, you can declare many variables at the same time as follows.

```
Var I,J,K,L:Integer;
```

Arrays are specified by declaring a variable to be "ARRAY [n] of Integer" where n+1 is the number of elements you wish in the array. Array subscripts begin at Ø (not one!) and go through n. Only single dimension arrays are supported and range checking is not performed at runtime. If your subscript exceeds the maximum size declared you will probably mess up the Pascal stack. No error message will be printed.

Examples:

```
Var I:Array [6] of Integer;
J,K:Array [4] of Integer;
L:Integer;
```

I is an integer array of size 7, J and K are integer arrays of size 5, and L is a simple integer. Your array subscript must be either a numeric constant or a symbolic constant declared in the "Const" declarations. Another variable name is not allowed (as in BASIC). To get the square brackets, "[" and "]" use control T and control Y in the Pascal editor. As with the "Const" declarations, the "Var" declarations are optional.

Following the "Var" declarations come the PROC and FUNC declarations. These will be discussed in greater detail later on.

After the FUNC and PROC declarations comes the Main Program. The main program consists of the reserved word BEGIN followed by Pascal statements which are separated by semicolons. After the last statement in the program comes the reserved word END followed by a period.

What statements are available in "Tiny Pascal?"

The Assignment statement:

The assignment statement ("LET" in BASIC) is of the form:

```
Varname:=<expression>
```

Note the use of ":=" for the assignment statement. No type checking is done so all of the following are valid:

Examples:

```
I:=1Ø;
I:=I+1;
I:=I+'C';
I:=Num-'Ø';
J[1]:=I*2+'I'+I;
```

whenever an Ascii character is encountered, the ASCII code for that character will be used. This is somewhat like the "CHR$" and "ASC" functions in BASIC. The Arithemetic operations are as follows. (Listed from Low precedence to High precedence.)

Low Precedence	+, -, AND, OR
Medium Precedence	*, DIV, MOD, SHL, SHR
High Precedence	Not, - (unary)

A few notes are in order. The assignment I:=J AND K; performs a bitwise AND on all 16 bits of J and K. The result is stored in I. Likewise I:=J OR K; performs a bitwise OR of J and K.

*" is used for multiply, as you would expect. But you must use "DIV" for division. In Pascal "/" is used only for REAL divisions and "DIV" is used only for Integer divisions. Since Tiny Pascal does not support floating point "/" is not used. "MOD", of course, performs the module function. "SHL" and "SHR" provide bit shifting facilities. "SHL" performs a "shift left" and "SHR" performs a "shift right".

Example:

```
I:=J SHL 2;
```

shifts J two bits to the left and places result in I. NOT inverts all the bits, and (-) takes the 2's compliment. Parentheses may be used to provide a change in precedence when required.

The MEM Array.

Often, in a microcomputer environment, it is necessary to access absolute locations in the memory space. In BASIC Peek and Poke provide this facility. In Tiny Pascal a phathom array "MEM" is used. Basically the array "mem" is an array of bytes 64K long. MEM [Ø] corresponds to location zero in memory, MEM [1] corresponds to location one, etc.

To simulate a Peek simply use the MEM array within the expression on the right side of an assignment statement. For example, I:=MEM[33]; is the same as I=Peek(33) in BASIC. To simulate a POKE statement use the MEM on the left side of the assignment statement. Example, MEM[33]:=33; is the same as Poke 33,33 in BASIC.

Hexadecimal Constants

Sometimes it is much more convenient to use a hexadecimal constant than a decimal constant. This is particularly true when accessing absolute addresses in the computer memory.

A 16-bit hexadecimal constant is specified by preceeding a four-digit hexadecimal number with a percent sign (%).

Examples:

```
     I:=%FDED; CR:=%ØØØD;
     I:=%ØØ2Ø;
     J:=%ØØØA;
```

Note that hexadecimal constants must be exactly four characters long, so leading zeros must be typed in.

Handling Characters and Strings

Since Tiny Pascal is integer only you might get the impression that no string handling facilities are provided. Fortunately this is not the case, characters can be handled by Tiny Pascal.

The assignment

```
     I:='A';
```

where I is an integer, places the character 'A' in the low order byte of I and zero's the high order byte. Arrays of integer may be used to hold character strings.

Read and Write

I/O is facilitated by the Pascal Read and Write statements. Tiny Pascal write statements come in four flavors: write character, write decimal value, write string, and write hexadecimal value.

> Write (I): will print the low order byte of I as an ASCII character.
> Write (I#): will print I as a decimal integer.
> Write (I%): will print I as a hexadecimal value.
> Write ('string'): writes the desired string out to the CRT.

Naturally you can specify more than one element in the list by separating the variables/strings with commas. Tiny Pascal does not divide the screen up into "fields" as do the BASIC interpreters. Each element will be printed with no interleaving blanks.

Example: Write ('I=', I#); prints "I=nnnnn" where "nnnnn" is the current value for I.

Note: Write (33); does not display "33" onto the CRT as you might expect. Rather it displays the character "!". Remember, unless you suffix the list element with a "#" or "%" it will be printed as an Ascii character. Also: Write does not automatically eject a return after the last element has been printed. You have to specifically provide the return. This can be done in several ways. For instance, WRITE (I,13); will print the current character in the lower order byte of I and then a carriage return (13 is the decimal value for a carriage return). Possibly a better way to print a return is to define a constant, CR, equal to 13 (CR=13) and now simply use WRITE(I, CR). This will improve the readability of your program considerably.

Using Apple DOS from Tiny Pascal.

You use the Apple DOS in Tiny Pascal in the same manner that you use it in Integer BASIC. To execute a DOS command you simply "write" control-D, followed by a DOS command, followed by a return. The last character printed prior to the control-D must be a return.

Define a constant "CTLD" equal to 4 (the decimal value for control-D). Now you can execute DOS commands as follows:

```
Write (CTLD, 'Catalog', CR): Prints a catalog.
Write (CTLD, 'OPEN Test', CR): Opens a file call test.
Etc.
```

To turn on printers and other I/O devices you can use the DOS command IN# and PR#.

Example:

Write (CTLD, 'PR#2', CR): turns on output device in slot 2.

Write (CTLD, 'IN#',I, CR): turns on the input device whose slot number is contained in the variable "I".

The Read statement is used in a manner similar to the write statement, except you cannot read a string of characters with a single read statement. You must read a single character, pack it into an integer array, and read the next character, etc. This will give the appearance to the user that he is entering a string of characters. Library.S.2, provided, contains several string input/output routines.

The automatic conversion routines "#" and "%" may be used in READ statements. READ(A#) will read a decimal number into the variable A. The user should respond by entering a string of decimal digits followed by a blank, comma, or carriage return. If the input string is empty, that is, if only a blank, comma, or carriage return was entered then A will be set to Ø. If the number is invalid it must be re-entered.

The IF-THEN-ELSE Statement

The IF-THEN-ELSE Statement in Tiny Pascal has the form:

If <condition> Then statement (Else statement).

The ELSE portion is optional. Statement may be either a simple statement (a single Pascal statement) or a compound statement (to be described later).

Examples:

```
     If A=B then write (A#) ELSE write ('A=', A#,  B=', B#);
     If I<=J then if K=J then write (K#)
     ELSE write (J) ELSE write (I);
```

In the second example some ambiguity might arise due to the use of the nested If's and Else's. The simple rule of thumb in this situation is: Each ELSE goes with the last "UN-ELSED" If. In the previous example the "ELSE write (J)" goes with the "If K=J" and the "ELSE write (I)" goes with the "If I<=J".

Relational operators are as follows:

<	: Less Than
<=	: Less Than or Equals
=	: Equals
<>	: Does Not Equal
>=	: Greater Than or Equals
>	: Greater Than

In addition, TRUE is any value other than zero and FALSE is zero (you may want to define two constants "TRUE" and "FALSE" with these values). So now...

```
     I:=1;
     IF I-1 THEN WRITE(I#)
     ELSE WRITE(I+1);
```

will write "I+1" to the CRT< since I-1=Ø (which is false). In the same manner the relational values may be used in assignment statements.

```
     I:=2;
     I:=I=2; sets I to 1;
     I:=I=1Ø; sets I to Ø;
     etc.
```

In addition the operators "AND", OR", and "NOT" may also be used in "If" statements.

Example:

```
     If ((A=B) AND (K=L)) OR (M=N) then write (N#)
```

Up till now very little mention has been made of semicolons other than the fact that they are used to separate statements. Please note that "If <cond> Then <statement 1> Else <statement 2>" itself constitutes ONE statement. You cannot place a semicolon after statement 1 or the Tiny Pascal compiler will think the end of the If statement has been reached. As a result the Else <statement 2> will no longer be connected to the If statement and an error will result.

The While Loop

Pascal supports a looping construct known as the While Loop. It has the form:

```
While <cond> DO <statement>;
```

where statement is a simple statement or a compound statement. <Cond> is any expression returning a TRUE or FALSE value. As long as the condition is true the statement is executed. When the condition becomes false the loop will be exited.

Example:

```
I:=Ø
While (I<=1Ø) DO I:=I+1; (* Delay loop. *)
```

The syntax and use of <cond> is the same for the while loop as it is for the If statement. The while loop tests the condition at the beginning of the loop and then executes the statement if and only if the condition proved to be true. The looping continues until the condition becomes false. If the condition is false when the while is first executed the statement after the DO will not be executed at all.

The Repeat...Until Loop

Unlike the While Loop, statements within a Repeat until loop always get executed at least once. This is because the condition gets tested at the bottom of the loop rather than at the beginning of the loop. There are two other major differences. First, the loop is repeated only if the condition turns out to be false (opposite of the while loop) and second, you are not limited to one simple or compound statement but you can have as many statements as you desire (separated by semicolons). The Repeat...Until Loop has the following form:

```
Repeat
  <statement 1>;
  <statement 2>;
       -
       -
  <statement n>
UNTIL <cond>
```

You may have none, one, or as many statements as you desire between the Repeat and Until as long as they are separated by semicolons.

Example:
```
     Repeat
       Write ('Enter Answer (Y/N)');
       Read (Answer);
     Until (Answer = 'Y') OR (Answer = 'N');
```

THE FOR LOOP

The FOR loop in tiny Pascal is very similar to the FOR/NEXT loop in BASIC. It is of the form:

```
     FOR <varname> := <initialvalue> TO <finalvalue> DO <statement>;
                                       -OR-
     FOR <varname> := <initialvalue> DOWNTO <finval> DO <statement>;
```

As with the WHILE loop only one statement (simple or compound) may follow the DO. There is no stepsize allowed other than 1 or -1. (for a stepsize of one use "TO", for a stepsize of minus one use "DOWNTO").

EXAMPLES:

```
          FOR I:= 1 TO 1Ø DO WRITE('I=',I#);
          FOR I:='A' TO 'Z' DO WRITE(I);
          FOR I:= 1Ø DOWNTO 1 DO WRITE(I#);
```

THE CASE STATEMENT

The CASE statement is a much more powerful version of the ON...GOTO in BASIC. It has the form:

```
          CASE <varname> OF
            <const1>:<statement1>;
            <const2>:<statement2>;
            <const3>:<statement3>;
                      -
                      -
            <constn>:<statementn>
            ELSE <statementn+1>
          END;
```

The ELSE <statementn+1> is optional. Please note that a semicolon is NOT ALLOWED after <statmentn> and <statementn+1>. These statements can be either simple statements or compound statements.

At execution time the Apple II will take the value contained in <varname> and compare it with <const1>. If a match is made <statement1> is executed. Otherwise <varname> gets compared with <const2>, then <const3> etc. until a match is made. If a match is not made amongst all the available constants the ELSE statement (if present) will be executed. If the ELSE statement is not present none of the statements in the CASE statement will be executed, and program execution will continue with the next statment after the CASE statement. EXAMPLES:

```
          I:=Ø;
          CASE I OF
            1:WRITE('I=',I#);
            2:WRITE('J=',I+2#);
            3:WRITE('K=',K%)
            ELSE WRITE('NOT THERE!')
          END;
```

This will write "NOT THERE!" on the terminal.

```
          I:=Ø;
          CASE I OF
            1:WRITE(I#);
            2:WRITE(2*I#)
          END;
```

This will do nothing since a match will not be made and there is no "ELSE" clause.

```
          I:=Ø; CASE I OF
            Ø:WRITE('ZERO');
            1:WRITE('ONE');
            2:WRITE('TWO')
          END;
```

This example prints "ZERO" on the crt.

<const1>, <const2>, ... , <constn> must be constants, they cannot be variables. Should you try and use a variable name you will surely be rewarded with an error message.
If the CASE statement is of the form:

```
     CASE I OF
          <const1>,<const2>,...,<constn>:<statement1>;
          <constn+1>,<constn+2>,...,<constm>:<statement2>;
          ETC.
```

then <statement1> will be executed if a match is made between I and any of <const1> ... <constn>. Likewise <statement2> gets executed if any of <constn+1> ... <constm> match up with I.

COMPOUND STATEMENTS

Up till now we've been limited to one statement is all of our examples except the REPEAT...UNTIL. In many (most!) applications more than one statement is required.

Compound statements may be used anywhere single statements are allowed are are of the form:

```
          BEGIN
            <statement1>;
            <statement2>;
                 -
                 -
                 -
            <statementn>
          END;
```

Examples:

```
FOR I:= 1 TO 1Ø DO BEGIN
  IF (I<=9) THEN WRITE(' ');
  WRITE('I=',I#);
END;

IF I=J THEN BEGIN
  IF K=L THEN WRITE(L#); END ELSE WRITE(I#);
```

Note in particular the use of the BEGIN ... END to force the ELSE to be associated with the first IF statement.

- COMMENTS -

Comments in Pascal are delimited by "(*" and "*)". Comments are allowed any where a space is allowed and the compiler ignors them. Unlike BASIC, comments in Pascal do not exist at run time, and as such do not require any memory in your program. Nor do they degradate program performance. Comments do improve the readability of your program so they should be used generously.

Examples:

```
(* THIS IS A COMMENT *)
FOR (* COMMENTS ARE ALLOWED HERE! *) I:= 1 TO 1Ø DO WRITE(I#);
```

PROCEDURES AND FUNCTIONS

Tiny Pascal supports the use of procedures and integer functions. A procedure (or function) definition is very similar to a program definition. The first line of a procedure definition is

```
PROC <procname> (<optional parameter list>);
```

This defines a procedure by the name <procname>.

Following the PROC statement come the CONST declarations. Constants declared within a procedure cannot be referenced by the external program. These constants are "local" to the current procedure. A procedure, however, is allowed to use constants and variables declared in the main routine. Such constants (and variables) are considered "global". For a complete description of global and local constants, variables, procedures, and functions check out the section on "scope" in any textbook on Pascal. The concept of "scope" is beyond the scope of this paper!

Following the CONST declarations come the variable declarations. once again, any variables defined wil be local and inaccessable by the main procedure.

After the variable declarations come the internal procedure and function declarations. (Yes, you can have a procedure inside a procedure). After the procedure and function declarations (if any) there is a BEGIN, followed by the statements in the procedure, terminated by an "END;".

Procedures are "invoked or "called" simply by using the procedure name as a statement in the program.

EXAMPLE:

```
PROGRAM TEST;
  VAR I:INTEGER;
    PROC CRLF;
    CONST CR=13;
    BEGIN
      WRITE(CR);
    END;
  BEGIN
    WRITE('HELLO THERE'); CRLF;
    WRITE('HOW ARE YOU TODAY?'); CRLF;
END.
```

The program returns from a procedure by encountering an "END" statement. It is not possible to exit from the middle of a procedure or function via a "RETURN" as in BASIC. You always enter a procedure at the top and exit at the bottom.

Parameters are used to pass information to the routine. For instance, suppose we want to simulate the SPC procedure in BASIC. SPC(I) prints I spaces onto the CRT. The procedure to do this could be:

```
PROC SPC(I);
  VAR J:INTEGER; BEGIN
    FOR J:=1 TO I DO WRITE(' ');
END;
```

Now, in our mainline program, if we write
WRITE('HELLO'); SPC(2); WRITE('THERE');
It would print "HELLO THERE" with two blanks between the "HELLO" and THERE". You can have as many parameters in your list as you need, simply separate them with comma's.

```
i.e,
PROC TEST(I,J);
  BEGIN WRITE(I#,J#); END;
```

Is called by TEST(3,2) for example.

Functions are set up in the same manner as procedures. Instead of "PROC" you use "FUNC", and then somewhere in the body of the function you are allowed to make the assignment:

```
     <funcname>:=<expression>;
```

The value will be returned when you exit the subroutine. Functions are invoked by appearing anywhere an expression is legal, some examples:

```
          I:=<funcname>;
          WRITE(<funcname>);
          MEM[<funcname>:=<expression>;
          ETC.
```

In the above examples simply replace <funcname> with the name of the function you wish to use.

EXAMPLE:

```
          PROGRAM TEST;
          VAR I:INTEGER;
          FUNC ADD1(I);
          BEGIN
               ADD1:=I+1;
          END;
          BEGIN
               I:=ADD1(2);
          END.
```

I now equals 3.

- CALL -

Often it is desirable to call machine language programs directly from a Pascal program. This is handled by the builtin procedure "CALL". As an argument, you pass CALL the address of the machine language subroutine, and whatever 6502 machine language program is sitting at that address will be executed. This is very similar to the "CALL" in BASIC.

EXAMPLES:

```
          CALL(-936); - HOMES AND CLEARS SCREEN.
          CALL(-151); - PUTS YOU INTO THE APPLE MONITOR.
```

USING THE PASCAL EDITOR

The editor provided with the Tiny Pascal system is a line oriented editor written by Herbert Yuen. The source for the editor (written in Pascal) is in the file "EDITOR.S".

You may enter the editor by typing "E" while in the Pascal monitor. The Apple II will respond with:

```
          NEW/EDIT(E):
```

By pressing "E" or <cr> you can edit a file existing (created previously or L(OAD)ed in at the Pascal monitor level). By pressing "N" the editor will clear any existing text and place you directly in the "insert" mode.

Once you are in the editor (by using the "E" command or by getting out of the insert mode when editing a N(EW) file) you will be at the editor command level. Valid commands are:

n' refers to a decimal number in the range 1-999.

L:	list entire file.
P:	prints current line.
P^:	prints top line.
P*:	prints last line.
Pn:	prints the next 'n' lines.
R<string>:	replaces current line by <string>.
A<string>:	appends <string> to the end of the current line.
D:	deletes current line.
D^:	deletes the first line.
D*:	deletes the last line.
Dn:	deletes the next 'n' lines.
X:	status, prints size of file etc.
U:	move the line pointer up one line.
Un:	move the line pointer up 'n' lines.
N:	move the line pointer to the next line.
Nn:	move the line pointer past the next 'n' lines
E:	exit the editor.
I:	enter the insert mode (automatic when editing a N(EW) file). All following text is inserted AFTER the current line. You exit the insert mode by typing <cr> as the first character of a new line (to insert a blank line type at least one space prior to the <cr>). If the textfile is empty (when editing a N(EW) file) you must insert at least one line of text before exiting the insert mode.
I^:	Insert text before the first line.
I*:	insert text after the last line.
M:	enter intra-line editing mode. " " denotes a control character. Commands in the intra-line editing mode are:
	^A: copies current character.
	^G: copies entire line.
	^H: backspace one character.
	^S<c>: copies all characters up to <c>.
	^N: re-edit new line.
	<cr>: exit modify mode.

In addition to the Pascal editor you may use the Apple Pie (version 2.Ø) text editing system to create source files for creating tiny Pascal source programs. Since Apple Pie textfiles and Tiny Pascal textfiles are incompatable, Pie textfiles must be converted before attempting to compile a Pascal program created by Pie. To accomplish this run the program "CONVERT/PIE" (a source is provided). Programs created using Apple Pie cannot be modified by the tiny Pascal editor (even after conversion by "CONVERT/PIE".

Since neither Pie nor the tiny Pascal editor allow you to insert control characters into the source code you must use constants with the appropriate value instead of the actual control character. For instance, to display a catalog from a program you could use

```
    WRITE(CTLD,'CATALOG',CR);
```

Where CTLD=4 and CR=13. The "[" and "]" characters are directly available from the keyboard using control-t and control-y.

Tab characters. The compiler and editor both recognize the TAB character (control-I). During listings two blanks are substituted for each TAB character.

-- MORE ADVANCED TOPICS --

- ERROR MESSAGES -

The error messages printed by the compiler are the standard error messages found in PASCAL:USER MANUAL AND REPORT by Jensen & Wirth. In addition error 999 may occassionally appear. Error 999 indicates that the static function/procedure nesting level, or the number of permissable variables in a function/procedure, has been exceeded.

- MAXIMUM NUMBER OF VARIABLES -

The total number of integer variables (including each element of an array) must not exceed 2048 (decimal). If you need an array of 4000 elements you must break it into two arrays of 2000 elements each. one array must be declared at the current level of procedure or function. The second array must be declared within an internal procedure or function. The arrays must have different names (if both are to be accessed). The inner procedure can now "see" both the outer array and its own. The main body of the outer procedure can now be a simple call to the inner procedure.

If a function/procedure is 16 levels deeper than the outermost function it can no longer access variables in the outer function/procedure. This highly unusual situation is flagged with the error 999 message.

- RUN TIME ERRORS -

Runtime errors will return you to the Pascal monitor, which will identify the type of error and the P-code address of its occurance. This can be matched against the compiler listing to determine in which line the error occured.

- SYSTEM FLAGS -

The 6502 P-code interpreter maintains an eight bit flag register at location %000D hex which is used during runtime errors. Four of the eight bits are used to determine whether or not a runtime error will be flagged.

The following bits are defined:

```
        BIT Ø: arithimetic overflow (>32767)
        BIT 1: division by zero (ZERO DIVIDE)
        BIT 6: invalid opcode (*DAMAGE*)
        BIT 7: stack overflow (stack full)
```

The appropriate bit is on (1) if the error is enabled and off (Ø) if disabled. The invalid opcode and stack overflow bits cannot be disabled. The defaults are zero divide enabled and arithimetic overflow disabled. This is because the compiler uses + and - for address arithimetic which sometimes produces arithemetic overflow. Your program may wish to enable arithimetic overflow for a few instructions, the disable it again.

```
        MEM[%ØØØD] := %ØØFF; ... MEM[%ØØØD] := %ØØFE;
```

- PASSING PARAMETERS TO MACHINE LANGUAGE SUBROUTINES VIA CALL -

It is possible to pass certain values to machine language subroutines invoked via the "CALL" statement. Whenever a CALL is made the accumulator is loaded from location %ØØ1A, the X-register is loaded from location %ØØ1B, and the Y-register is loaded from location %ØØ1C. When the machine language subroutine returns the contents of the registers are stored in their respective locations. This allows flexible management of data when calling machine language routines.

One very useful example might be hexadecimal output. Although you can output a hex number directly from Pascal you are forced to output exactly four hex digits with each hex write. Sometimes it would be nice to be able to output only two hex digits at a time. The following program will perform a memory dump outputting only two hex digits at a time.

```
PROGRAM HEXDUMP;
VAR LOCATION, LOWER, UPPER:INTEGER;
BEGIN
    WRITE('INPUT LOWER BOUNDS:'); READ(LOWER%);
    WRITE('INPUT UPPER BOUNDS:'); READ(UPPER%);

    FOR LOCATION:=LOWER TO UPPER DO BEGIN
        MEM[%ØØ1A]:=MEM[LOCATION];
        CALL(%FDDA);
        WRITE(' ');
        IF NOT(LOCATION MOD 8) THEN WRITE(13);
    END;
END;
```

- MINIMAL RUN TIME SYSTEM -

You may wish to overwrite the compiler and editor and use this memory space for your own program. By setting the P-code origin at 18ØØ and then compiling (at 5ØØØ) then moving your program to location 18ØØ you can run your program in a stand-alone environment. You can also use the monitor F(ENCE) command to make more room for variable storage by setting low to 18ØØ and leaving your program at 5ØØØ.

If you would like an autostart capability, overwrite the Pascal monitor with your program. RESET and *8ØØG will then auto-execute your program (at hex %13ØØ). However you must adjust the memory fences by changing the defaults in the interpreter. If you save the P-code interpreter and your program to disk, BRUN <progname> will auto-execute your program. In addition, if the "HELLO" program on your disk BRUN's the program, it will auto execute when the DOS is booted. For more information see the section on the interpreter.

EXTERNAL PROCEDURES AND OVERLAY

Any program may call procedure and functions which are external to it. The program should use the ordinary PROC or FUNC heading when it declares the external procedure or function. However, in place of the BEGIN ... END block of the procedure or function body, a single hexadecimal constant should appear. When the procedure or function is called, control will be transferred to that hex address. The PROGRAM heading of the external procedure or function should match the internal heading of the calling procedure. For example, the Pascal monitor contains the declaration

```
FUNC EDITOR; (* PASCAL EDITOR *)
%3ØØØ;
```

The program line of the editor reads:

```
PROGRAM EDITOR;
```

Neither have any parameters (although both could). After the declaration, the monitor treats EDITOR just as it would any normal internal function.

- VERY HARD TO FIND BUGS -

Programma Pascal does not check array subscripts to see whether or not they are in bounds. If you store a value at position 1Ø of a nine element array you will damage the stack. Since return addresses are stored on the stack, you may find yourself in an embarrasing situation.

Another difficult but to detect is the use of zero instead o "O" (oh) in a variable name, or a hidden control character which does not print.

LIBRARY.S

Phillip Wasson has written some very useful (though untested by Programma) functions and procedures. They are in the file "LIBRARY.S". You may find them to be of interest.

Programma Pascal vs. Supersoft Pascal

Programma Pascal is a derivative of Kin-Man Chung and Herbert Yuen's Tiny Pascal. Supersoft Pascal is also a derivative of Tiny Pascal for the Radio Shack TRS-80 and Northstar computers.

Supersoft Pascal has an additional array called MEMW. Assignments to and from MEMW transfer all 16 bits as opposed to MEM, which only transfer the low order 8 bits. See LIBRARY.S.2 for a procedure and function which simulate MEMW.

Programma Pascal has added the external procedure definition and the PROGRAM header, which work together.

Otherwise the two Pascals are quite compatable at the source level, although their P-codes have been optimized for different environments. It may be necesary to rewrite certain sections of code which have been designed to run on the different computers. However, translating Supersoft Pascal to Programma Pascal will be an order of magnitude easier than translating BASIC.

- DETOKEN -

Detoken is a program written by Randy Hyde which will "detokenize" an Integer BASIC program. It can be used to create BASIC program textfiles which can be edited by APPLE PIE.

- LIBRARY.S.2 -

Contains several string handling routines which mimick many of the string routines in UCSD Pascal. Several other utilities are also given.

- CANNIBALS -

A programming example which demonstrates many of the features of Programma Tiny Pascal.

- THE P-CODES -

A table of P-codes has been provided with this documentation. One special P-code, hex 30, is used as a Pascal breakpoint. When executed hex 30 will exit to the Apple II monitor.

- THE INTERPRETER -

The 6502 P-code interpreter has been carefully designed to be a black box. The only locations which may need adjusting when moving from the 6502 environment to another appear in the JMP vectors and the following six bytes of default values.

```
0800 - JMP COLDSTART
0803 - JMP WARMSTART
0806 - JMP ERROR ENTRY
0809 - JMP READ BYTE
080C - JMP WRITE BYTE
080F - JMP SET MEMORY FENCES

0812 - Default start address of P-code pgm (low order byte).
0813 - Default start address of P-code pgm (hi order byte).
0814 - Default starting PAGE of Pascal stack.
0815 - Default fence PAGE.
0816 - Default HIGH memory page location.
0817 - Default SYSBIT error enable flags (currently hex $FE)
```

The interpreter itself was written using LISA (a 6502 assembler) The interpreter, monitor, compiler, and editor currently reside in locations $800 - $4000. If you wish to use HIRES graphics (and the Apple supplied routines) the current P-code interpreter will prove to be located in the wrong area. On the Pascal disk is a binary file called "PCODE.HIRES", it is assembled beginning at location $4000 in memory so that it will be out of the way of the first HIRES page and the Apple's HIRES routines. To use "PCODE.HIRES" first create and compile your program using the normal Pascal system. Once the syntax errors have been taken care of, save the codefile to disk. Load the binary file "PCODE.HIRES" and then adjust the memory fences so that they are ABOVE your program. Now you can use the "U(SER)" command from the Pascal monitor to run your HIRES program.

If you need the P-code interpreter assembled for a different location please contact Programma and they will be happy to assist you.

MEMORY REQUIREMENTS

Pascal occupies the following memory locations:

ØØØØ - ØØ1F:These are the zero page locations used by the 6502 P-code interpreter.
ØØD8 - ØØDF:Zero page locations reserved fro DOS/Pascal interface.
Ø8ØØ - 11FF:P-code interpreter and the compiler's reserved symbol table.
12ØØ - 12FF:A spare page for storing user builtin functions, patches, and so forth.
13ØØ - 17FF:Pascal monitor. This is a Pascal program. The source code to this program can be found in the file "MONITOR.S", so feel free to modify it and substitute your own improved monitor here. The monitor is actually much smaller than the space provided, so you have plenty of room for growth.
18ØØ - 2FFF:Pascal compiler.
3ØØØ - 3FFF:Pascal editor. You will find the source in "EDIT.S". 4ØØØ - 4FFF:(LOW-FENCE) The stack. All variables created in a Pascal program are stored on this stack. The stack starts at fence-1 and grows towards LOW.
5ØØØ - 95FF:(FENCE-HIGH) User memory. This area is not quite a part of the Pascal memory space. It is used to store text, programs, user subroutines, and so forth. The Pascal monitor, editor, and compiler will not normally access or change any memory above HIGH.

- CHANGING MEMORY LIMITS:LOW, FENCE, AND HIGH -

The F(ENCE) command in the Pascal monitor allows the user to specify the memory limits for a Pascal program. This is similar to setting LOMEM and HIMEM in a BASIC program.

Instead of two memory bounds (as in the LOMEM/HIMEM commands) the Pascal user must set three memory bounds: LOW, FENCE, and HIGH. LOW is the absolute smallest memory location usable by a Pascal program. It is roughly equivalent to LOMEM. HIGH is the absolute highest memory location usable by the Pascal program and is roughly equivalent to HIMEM. FENCE resides somewhere inbetween LOW and HIGH and divides the memory area between LOW and HIGH into two areas. One of these areas (from LOW to FENCE-1) is reserved for variable storage. The other area (from FENCE to HIGH-1) is reserved for program storage.

Should you ever get a stack full error it simply means that you have not reserved enough memory for your variables (which are kept on the stack) and as a result, you need to adjust the FENCE upwards towards HIGH. The default settings are LOW=4ØØØ, FENCE=5ØØØ, and HIGH=96ØØ. This has been optimized for text editing when using the Pascal editor. This will only leave enough room for approxamately 5ØØ-1ØØØ variable locations. Generally, except for smaller programs, you will want to adjust the FENCE to give you more room.

PROPRIETARY NOTICE

Since many users will find Programma Pascal much easier to use (and twice as fast!) as Integer BASIC it should not be too long before programs written in Tiny Pascal begin to appear on the marketplace. Although, in this paper, explicit instructions have been given guiding the user in creating "stand alone/auto execute" Pascal programs, Programma does not release rights on the P-code interpreter. All information presented in this paper is intended for the sole use and enjoyment of the original purchaser. Since we do not want to discourage the propagation of quality software written in Tiny Pascal we can offer you four suggestions when selling your Pascal programs.

1) Sell the source listing (or file) only. The end user must purchase Tiny Pascal in order to use your program.

2) Sell the P-code file only. The end user must purchase either the Tiny Pascal system, or the P-code interpreter alone.

3) Negotiate a license agreement with Programma International to sell the P-code interpreter together with your program. You take care of your own marketing and distribution.

4) Sell your software product through Programma International. Programma will patch your program up so that it will auto-execute. All you have to do is sit back and collect the royalties.

Clarity Tiny Pascal P-code Instruction Set

Least Significant Nybble

Most Significant Nybble	00xx	01xx	10xx	11xx	00xx	01xx	10xx	11xx	
0000	LOL$_{-5}$ LOL$_{-4}$ LOL$_{-3}$ LOL$_{-2}$	LOL$_{-1}$ LOL$_{0}$ LOL$_{1}$ LOL$_{2}$	LOL$_{3}$ LOL$_{4}$ LOL$_{5}$ LOL$_{6}$	LOL$_{7}$ LOL$_{8}$ LOL$_{9}$ LOL$_{10}$	ADD	SUB MUL DIV MOD	AND ---- OR BIT	SHL *n* SHR *n*	0100
0001	STL$_{-5}$ STL$_{-4}$ STL$_{-3}$ STL$_{-2}$	STL$_{-1}$ STL$_{0}$ STL$_{1}$ STL$_{2}$	STL$_{3}$ STL$_{4}$ STL$_{5}$ STL$_{6}$	STL$_{7}$ STL$_{8}$ STL$_{9}$ STL$_{10}$	INC DEC	NEG NOT	EQU NEQ	LT LTE GT GTE	0101
0010	INL$_{-5}$ INL$_{-4}$ INL$_{-3}$ INL$_{-2}$	INL$_{-1}$ INL$_{0}$ INL$_{1}$ INL$_{2}$	INL$_{3}$ INL$_{4}$ INL$_{5}$ INL$_{6}$	INL$_{7}$ INL$_{8}$ INL$_{9}$ INL$_{10}$	(← floating point →) *Version 3.0*				0110
0011	(← indirect →) *Version 2.0*				IN IN$_{10}$ IN$_{16}$	OUT OUT$_{10}$ OUT$_{16}$ OUT *n*	CPY	RTS	0111
1000	CAL$_{0}$ CAL$_{1}$ CAL$_{2}$ CAL$_{3}$	CAL$_{4}$ CAL$_{5}$ CAL$_{6}$ CAL$_{7}$	CAL$_{8}$ CAL$_{9}$ CAL$_{10}$ CAL$_{11}$	CAL$_{12}$ CAL$_{13}$ CAL$_{14}$ CAL$_{15}$	LOD$_{+}$	LOD_		LODX	1100
1001	JMP	JPF	JPT	JSR	STO$_{+}$	STO_		STOX	1101
1010	LIB *one byte LIT*				LDA	LIT	INCA	DECA	1110
1011	JEQ	JNE	JGE	JLT	STA	INT		... EOF ...	1111

One byte instructions

3 byte instructions - flow of control

one byte instructions

3 byte instructions

LOL, STL, INL use local address mode. The least significant nybble specifies the offset from the local contour plus 5. This gives immediate access to the first five parameters and first eight local variables. INL is the local form of INT.
CAL the least significant nybble specifies the static nesting depth of the call.
LDA, STA, JSR the absolute form of LOD, STO and CAL.
JEQ combines the EQU comparison and the JPT (jump if condition true). Similarly, JNE, JGE and JLT are used to optimize CASE and FOR loops.

PROGRAMMA PRODUCT DESCRIPTION

PROGRAMMA

SER Form

PLEASE CIRCLE THE APPROPRIATE PROBLEM AND MEDIA CODES BELOW.

PROBLEM:	MEDIA:
1. MISSING OR INCOMPLETE	A. DOCUMENTATION
2. WRONG PROGRAM RECEIVED	B. MAGNETIC TAPE
3. UNREADABLE DATA	C. CASSETTE TAPE
4. DAMAGED	D. DISK
5. QUESTION	E. CARDS
6. IMPROVEMENT	F. PAPER TAPE
7. INFORMATION	G. DISKETTE
8. PROGRAM PRODUCT PROBLEM	H. PAPER LISTING
9. OTHER ____________	I. OTHER ____________

SOFTWARE PRODUCT NAME:	
VER #:	SER #:
COMPUTER:	ATTACHED EVIDENCE SUPPLIED:
MONITOR/DOS:	1. PROGRAM LISTING 2. HEX MEMORY DUMP
MEMORY:	3. CONSOLE MESSAGES 4. OTHER ____________

DESCRIPTION OF SUSPECTED ERROR/IMPROVEMENT/SUGGESTION/INFORMATION.

SUBMITTED BY:		DATE:
COMPANY:		MAIL STOP:
ADDRESS:		
CITY:	STATE:	ZIP:
COUNTRY:	TELEPHONE:	

PROGRAMMA

NON EXCLUSIVE LIMITED LICENSE AGREEMENT
END USER AGREEMENT

AGREEMENT

This agreement is between the person or organization designated below (LICENSEE) and PROGRAMMA International, Inc., a California corporation, at 3400 Wilshire Boulevard, Los Angeles, California 90010 (PROGRAMMA).

PURPOSE AND CONSTRUCTION

The purpose of this agreement is to convey to LICENSEE a license to use the proprietary computer program listed and described below, together with accompanying copyrighted media material and documentation.

CONSIDERATION AND TERM

This license is granted by PROGRAMMA in return for a fee as described below and LICENSEE's agreement to respect the terms and conditions regarding permitted use of the PRODUCT and of related materials supplied.

The term of this license is 19 years, commencing on the date as signed below. The license may be renewed for an additional term of 19 years at the option of the LICENSEE upon payment of an additional fee of $1.00 to PROGRAMMA at any time during the initial term.

EXCLUSIVE SOURCE

LICENSEE must obtain all product materials through PROGRAMMA or through an AUTHORIZED PROGRAMMA DEALER and no other source. PROGRAMMA product materials include, but are not limited to, manuals, license agreements and media upon which proprietary computer programs are recorded. Except for archival copies, as defined elsewhere in this agreement, LICENSEE shall make no copies, of any kind, of any of the materials furnished by PROGRAMMA, unless specifically authorized to do so in writing signed by an officer of PROGRAMMA International, Inc.

LIQUIDATED DAMAGES

LICENSEE recognizes that PROGRAMMA has expended considerable time and expense to develop PROGRAMMA's products and PROGRAMMA would be damaged by unauthorized copying and reproduction or distribution of PROGRAMMA's product materials. In the event LICENSEE breaches this Agreement by unauthorized copying or reproducing or distributing of PROGRAMMA's product materials, LICENSEE agrees to pay PROGRAMMA as liquidated damages, for each occurance of the unauthorized act of copying or reproducing or distributing of PROGRAMMA's product materials, the sum of $1,000 (One Thousand Dollars) in lawful United States Currency, immediately upon demand by PROGRAMMA.

Furthermore, PROGRAMMA agrees to pay to any person, company, or other lawfully constituted entity that provides information that leads to a successful prosecution and recovery, the sum of $250 (Two Hundred and Fifty). At all times, the decision to evaluate such information as well as the decision to prosecute remains at the sole option of PROGRAMMA.

End User Agreement

Any sum paid will only be paid once, and, in the event of multiple entities, the sum will be distributed at the discretion of PROGRAMMA.

ARCHIVAL COPIES

LICENSEE may make archival copies of those portions of PROGRAMMA product(s) that are provided on machine readable media, provided such copies are for the LICENSEEs personal use and that no more than one (1) such copy is in use at any time unless LICENSEE has paid for multiple copy use as described below. LICENSEE agrees to label each archival copy with a reasonable copy of PROGRAMMA's copyright notice, product name, and LICENSEE serial number as furnished with the vended media. Failure to label each archival copy shall be considered a breach of this Agreement for which liquidated damages are applicable.

MULTIPLE COPY USE

In the event that LICENSEE intends to use PROGRAMMA product or any part thereof on more than one computer, the required fee for each use must be paid. This Agreement licenses the LICENSEE to use the product on a single computer installation.

SOURCE CODE AVAILABILITY AND ACCESS

PROGRAMMA agrees to furnish to the LICENSEE, upon request and for a separate charge listed below, a single copy of the source code used in the preparation of the product. Upon taking possession of the source code, the LICENSEE agrees that the source code shall be subject to the restrictions on the product itself.

PRODUCT INCLUSION AND MODIFICATION

The LICENSEE has the absolute right to modify the product or to include the product as part of another system, limited, however, to the LICENSEE's internal use only. The product as so modified or as included as part of a LICENSEE developed system remains subject to the same restrictions on use, reproduction and disclosure as contained in this Agreement with respect to the product itself. Upon any such modification or inclusion, PROGRAMMA shall be released from any responsibility to maintain the product, except that PROGRAMMA shall continue to disclose to the LICENSEE any errors discovered in the product.

UPDATE POLICY

PROGRAMMA may, from time to time, revise the performance or offer enhanced versions of its product(s) and, in so doing, incur NO obligation to furnish such revisions to any PROGRAMMA customer. PROGRAMMA agrees to furnish to the LICENSEE, upon request and for the cost of media, postage, and handling, a copy of an updated version of the product, provided the product fee remains unchanged. Should the product fee increase in an enhanced version of the product, PROGRAMMA agrees to furnish to the LICENSEE, upon request for the cost of media, postage, and handling and the difference between original product fee and updated product fee, a copy of an enhanced version of the product.

PATENT AND OTHER PROPRIETARY RIGHTS INDEMNITY

PROGRAMMA warrants that the product sold does not infringe upon or violate any patent, copyright, trade secret or any other proprietary right of any third party. In the event of any claim by any third party against LICENSEE, the LICENSEE agrees to promptly notify PROGRAMMA and PROGRAMMA shall defend such claim, in LICENSEE's name, but at PROGRAMMA's expense and shall indemnify LICENSEE against any loss, cost, expense or liability arising out of such claim, whether or not such claim is successful.

ASSIGNMENT OF AGREEMENT

This Agreement is not assignable without written permission from PROGRAMMA. Any attempt to assign any rights, duties or obligations which arise under this Agreement without the permission of PROGRAMMA shall be void.

LIMITED WARRANTY POLICY

PROGRAMMA warrants that all materials furnished by PROGRAMMA constitutes an accurate manufacture of PROGRAMMA products and will replace any such PROGRAMMA furnished material found to be defective, provided such defect is found within ten (10) days of purchase by LICENSEE. However, PROGRAMMA makes NO expressed or implied warranty of any kind with regard to performance or fitness for any particular purpose for any PROGRAMMA product. PROGRAMMA is NOT responsible for any loss or inaccuracy of data of any kind nor for any consequential damages resulting therefrom whether through PROGRAMMA negligence or not. PROGRAMMA will not honor any warranty where PROGRAMMA product has been subjected to physical abuse or used in defective or non-compatible equipment.

PROTECTION AND SECURITY

LICENSEE acknowledges that all material and information which will come into his possession or knowledge in connection with this Agreement or the performance hereof, consists of confidential and proprietary data, whose disclosure to or use by third parties will be damaging. LICENSEE agrees to hold such material and information in strictest confidence, not to make use thereof other than for the performance of this Agreement, to release it only to employees requiring such information, and not to release or disclose it to any other party.

GUARANTEE OF OWNERSHIP

PROGRAMMA warrants that it is the sole owner of the product and has the full power and authority to grant the rights herein granted without the consent of any other person or entity.