V ASSEMBLY LANGUAGE PROGRAMMING

This section describes how to:

- Use the Kyan Pascal assembler ("as")
- Include assembly code routines in a Pascal program
- Access parameters in Procedures
- Access values returned by Functions

This section does not explain how to write assembly language programs. It is intended primarily for programmers who already know how to write assembly code and wish to use it in their Pascal programs.

USE OF THE KYAN PASCAL ASSEMBLER

Kyan Pascal features a special purpose macro assembler called "as". This assembler is optimized for maximum speed of assembly by limiting features. Symbol table listings, cross reference listings, nested macros, identifiers of greater than 6 characters, and linkable object code modules are not supported by "as". If you need these features, we recommend that you use Kyan's Macro Assembler.

The Kyan Pascal compiler normally pipes its assembly language output to the assembler, which in turn produces an executable object file. If the -s option is used on the compiler, the output of the compiler will not be assembled. Instead, the compiler will generate an assembly language text file to disk named "P.OUT". This assembly language file can then be modified as required for special applications. When modifications are complete, the assembler is then used as described below to generate an executable file. (NOTE: To save the P.OUT file, you must rename it using a ".S" suffix or the assembler will overwrite it).

For advanced programmers, Kyan Software makes available an optimizer and source code version of the Kyan Pascal Runtime Library. When used with the source library, the assembler will conditional assemble only the library routines required in the compiled application program. This feature saves memory and permits large applications to run in one program segment. It also allows the programmer to modify standard Pascal procedures and functions to meet special programming requirements.

RUNNING THE ASSEMBLER

The assembler "as" can be run in two ways.

Option 1: Type "as" at the prompt(%) and type <RETURN>. Then, enter the pathname of the file to be assembled along with any of the options listed on the Help Screen.

Option 2: Type "as" at the prompt(%), the pathname, and the desired options on the same line. Then press <RETURN>.

The assembler has two options which are listed below:

-l	Produces a listing.
- o pathname	Renames the output file.

The assembler listing and error messages can be redirected in two ways as shown below:

> pathname	Redirects output to a file
> 1 - 7	Redirects output to a slot

An example of the above assembler options and output redirection would be: *% as xyz -l -o mno >abc*

With this command, the assembler would: (1) assemble the file named "xyz"; (2) produce an executable file named "mno"; and, (3) produce a listing of the program with the filename "abc".

You can stop the assembler at any time during assembly by pressing the <ESC> key. You will be returned to the system prompt (%).

NAMING AN EXECUTABLE FILE

The assembler has several features which control the name generated for an executable file.

Option 1:	Anytime the "-o" option is used, the executable file is given the name assigned in the option statement.
Option 2:	Any source filename that ends in ".s" will result in an executable file with the same filename with the ".s" stripped off.
Option 3:	If neither of the above options apply, then the executable file will be named "a.out".

GENERATING SYSTEM PROGRAMS AND RELOCATING PROGRAMS FOR HIRES

To make the Pascal compiler generate a system file, include the following instructions before the first line of the Pascal program.

```
#a          (* locate # symbol in column 1 *)
_SystemFile
#
```

Under default conditions the compiler generates a binary file which starts at location $800 in memory. If you want to use HiRes graphics, you must change starting locations. The following steps should be taken.

```
#a
_UsesHires
#
```

Please refer to the memory maps in Appendix B for more information.

ASSEMBLY LANGUAGE ROUTINES

Kyan Pascal accepts assembly language routines as part of the Pascal program. This enables the programmer to include routines that are not restricted by the structure of standard Pascal. Five rules govern the use of assembly language in a Pascal program.

1. Assembly language routines must appear within the body of a Program, Procedure, or Function. That is, they must appear between a BEGIN/END block.

2. To indicate the difference between the Pascal program and the assembly code, assembler routines must begin with a **#a** label and end with a **#** label. The **#** sign must be placed in the first column, and the **a** must appear in the second column. Lines contained between these labels are left untouched by the Pascal compiler and are integrated into the final assembly language output file exactly as they are written.

3. All labels used in assembly language must begin in column 1.

4. Labels used as part of the assembler routine must not begin with an underscore character (_). The compiler uses labels with the format "_xxxxx". Consequently, if you use labels beginning with an underscore, the program may fail.

5. If the X register is used by an assembly language routine, it must first be saved and then later restored. The X register is used by the compiler as the system stack pointer. If it is used and not restored, the compiler will lose track of all variable references (i.e., your program will crash).

In summary, to use assembly language statements:

1. Place all code between BEGIN/END statements
2. Include all code between **#a** and **#** labels
3. Begin all labels in column 1
4. Do not use labels that begin with an underscore (_).
5. Save and restore the X register

ASSEMBLER DIRECTIVES

Kyan Pascal supports 28 assembler directives. Directives are also known as pseudo-code since they appear in the assembly language listing but are not part of the language of the microprocessor. Instead they are terms understood by the assembler itself. The directives are:

Kyan Pascal Assembler Directives

Symbol	Description
ORG	Origin -- indicates that the assembled code should start at the specified location in memory (e.g., ORG $4000; start at $4000)
EQU	Equate a label with a value which will be assigned to the label whenever it appears in the program. In effect, EQU defines a constant (e.g., A EQU $FF; define A to be $FF)
DB DW	Define Byte and Define Word are used to build tables and strings that reside in other parts of the assembly program. When the program executes, it sets the index register to the values identified by these directives. These values indicate where the table or string resides in memory. (e.g., DW $FF00; put $00 in next byte and $FF in following byte).
>	Least Significant Byte (LSB) is used with a label or specific value to indicate the least significant byte of a 2-byte hexadecimal number (e.g., >$FF01 = $0001 or, if WLABEL = $11EE, then >WLABEL = 00EE).
<	Most Significant Byte (MSB) is used with a label or specific value to indicate the most significant byte of a 2-byte hexadecimal number (e.g., <FF01 = $00FF, or, if WLABEL = $11EE, then <WLABEL = $0011).

* The asterisk is used to determine the current value of the program counter during assembly.

& The ampersand is used with a digit (e.g., &1, &2) to represent a macro parameter.

DS Define Storage saves space for the number of bytes in the expression field (e.g., ds 5;reserves 5 bytes).

STR String counts the number of characters in the expression fields and puts that number in the first byte followed by the ASCII values of each character in the string (e.g., str 'abc'; first byte is 3 followed by ascii values of a, b, and c).

IFDEF If Defined assembles the code following the directive if the identifier in the expression field is defined (e.g., ifdef abc; assemble what follows if abc is defined).

IFNDEF If Not Defined assembles the code following the directive if the identifier in the expression is not defined (e.g., ifndef abc; assemble what follows if abc is not defined).

IFEQ If Equal assembles the code following the directive if the expression is equal to zero (e.g., ifeq x-1;assemble what follows if x was previously defined to be 1).

IFNE If Not Equal assembles the code following the directive if the expression is not equal to zero (e.g., ifne x; assemble what follows if x was previously defined and not equal to zero).

ELSE Else optionally follows one of the IF- directive and reverses the conditional assembly (e.g., ifeq y .... else .... endif; assemble what follows "else" if y is defined as not equal to 0, otherwise don't assemble what follows).

ENDIF End If directive ends the conditional assembly associated with IF- or IF- ELSE directives (e.g., ifdef abc .... endif; end the conditional assembly associated with the IFDEF).

INCLUDE Include file in expression field (e.g., include xyz ; include the file xyz).

LST ON Listing On turns on the listing at that point.

LST OFF Listing Off turns off the listing at that point.

DSECT Data Section defines an area of memory for data only. For example, define a data area in high memory:

```
dsect
ram equ $b000
abc ds 2000
dfg ds 1000
dend
```

DEND Data End ends the definition of the area in memory reserved for data only.

MACRO Macro definition follows. For example, define the macro chrout:

```
macro chrout
ora $80
jsr cout
endm
```

ENDM Macro definition ends.

MEX ON Macro EXpansion ON for listing.

MEX OFF Macro EXpansion OFF for listing.

SYS SYStem is used to make the executable file a system file (e.g., sys ; make the following assembly language program a system file).

ASC ASCii is used to put the ASCII values to the string in the expression field in the bytes following the ASC directive (e.g., asc 'ok' ; put ASCII value of '0' and 'k' in next 2 bytes).

DFLAG Define FLAG is used to define or redefine the variable in the expression field. The value of the definition has no meaning. The DFLAG directive is used with the IFDEF and IFNDEF directives to assemble code required by one or more already assembled macros or code segments.

Do not use parentheses in assembler directives. Expressions are evaluated from left to right, and no one operator takes precedence over another.

ASSEMBLY CODE AND PROCEDURES

Data is normally transmitted to a Procedure in the form of parameters. The formal parameter list that is part of the Procedure declaration defines the data the Procedure expects to receive from the main program or the calling routine. The main program transmits the actual data in the Actual Parameter List that accompanies the call to the Procedure.

For example, the Procedure declaration *AddXY(X,Y: Real);* expects to receive two real numbers from the calling routine. The main program might call this procedure with the statement *AddXY(3.5,4);*.

If the Procedure is an assembly code routine, it must read the parameters from the stack which contains the list of parameters passed to the Procedure.

Every time a Procedure is called by another routine, Pascal creates a call frame on the stack to hold the parameters being passed to the Procedure. The zero page location Stack Pointer (_SP) and the local variable pointer (_Local) are used to reference the stack.

The Assembler can identify parameters only by their position in the Stack; but, since the Assembler cannot locate those values by itself, the programmer must identify each parameter by telling the Assembler its location in the Stack. This means that the position of each parameter must be calculated manually before it is used in an assembly language routine.

To calculate the position of each parameter in the Stack you need to understand the structure of the stack and the number of bytes used to store different types of parameters.

THE STACK

When parameters are passed to a Procedure, they are placed in a stack. Each value entered on the Stack pushes a preceding value down the Stack. It's like putting pennies in a coin-change holder with a counter. The first penny you put in is pushed down by the next one, and the counter records how many pennies you have saved.

If you have saved 5 pennies, the first one is at the bottom of the coin-holder and the counter indicates that you have saved 5 cents. Note that penny number one is at the bottom of your stack .

The following diagram illustrates this structure.

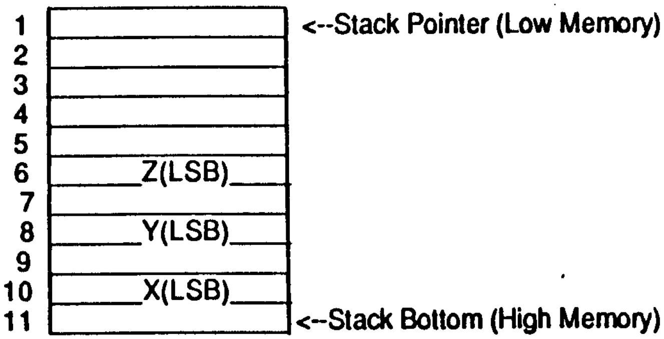
```
 __________   Stack Top
|_penny 5__|
|_penny 4__|
|_penny 3__|
|_penny 2__|
|_penny 1__|
              Stack Bottom
```

The Stack passed to the Procedure is identical in structure to the coin holder. The first parameter passed to the Procedure is located at the bottom of the stack. The others are pushed on top of it as they are passed to the Procedure.

Examine the following Procedure declaration and the corresponding diagram of the Stack that would hold the parameters passed to it from the calling routine. Remember that the Stack is set aside in memory and that _SP is used to keep track of the current value of the stack pointer.

```
PROCEDURE StackSample(X,Y,Z: Integer);
```

The parameter list passed to this Procedure would be stored in the Stack as



The five empty locations at the top of the Stack are used by the computer to store information about the Stack itself. When using the Stack, however, these positions must always be included in calculating the position of the parameters stored in the Stack.

For now, just make sure you understand that the first parameter, X, is stored in positions #10 and #11, that Y is in positions #8 and #9, and Z in positions #6 and #7. The Least Significant Byte (LSB) is the lower number and the Most Significant Byte (MSB) is the higher number.

To locate a specific item in the parameter list, calculate how many bytes of memory separate the Stack pointer from the value you want to locate. Since Pascal programs and subroutines always declare the parameters before the body of the program or subroutine, it is simple to calculate positions of parameters on the stack. The only problem with calculating the position of the parameter is that each type of parameter takes up a different amount of space.

The following list indicates how many bytes of memory are required to store a type of data.

MEMORY STORAGE ALLOCATION

DATA TYPE	BYTES ALLOTTED
I. Real	8
Integer	2
Char	1
Boolean	1
Pointer	2
II. ARRAY[1..n] OF Integer	2*n
ARRAY[1..n] OF Char	n
ARRAY[1..n] OF Boolean	n
III. Value Parameter (Real)	8
Value Parameter (Integer)	2
Value Parameter (Char, Boolean)	1
Value Parameter (ARRAY[1..n] OF Integer)	2*n
Value Parameter (ARRAY[1..n] OF Char or Boolean)	n
IV. Variable Parameter (Address of parameter) (All types)	2

Make sure you understand the amount of memory required by each data type before you try calculating positions of parameters in the Stack. There are 4 groups of data types. The first consists of the predefined data types. The second group consists of ARRAYs of the predefined types. The third contains the data types when they are passed as Values to a Procedure or Function. The last indicates the size allotted for any Variable passed to a Procedure or Function.

If a Procedure or Function uses a Real number, that number takes up 8 bytes of memory. If a Procedure or Function uses an ARRAY [1..10] OF Integer, the array uses 20 bytes of memory (2*n). If the Procedure or Function is passed an Integer Value in the Parameter list, the Value requires 2 bytes of memory. If a Procedure or Function is passed any type of Variable, each Variable uses 2 bytes of memory.

Storage of Real Numbers

Kyan Pascal stores Real numbers as Binary Coded Decimals (BCD) and allots them 8 bytes of storage. The 8 bits contained in each byte are divided into 2 parts which are called NIBBLES. The part with bits 0 through 3 is called the Low Order Nibble. The part with bits 4 through 7 is called the High Order Nibble. For example:

```
                    (1 Byte = 8 Bits)
|<------------------------------------------------------------->|
7       6       5       4       3       2       1       0
|<------High Order Nibble--------|--------- Low Order Nibble ------>|
```

When Real numbers are stored, they are organized as follows:

Byte	Low Order Nibble	High Order Nibble
0	1st significant digit	Bit 4: sign of exponent 0 = + / 1 = - Bit 5: sign of number 0 = + / 1 = - Bits 6 & 7: always zero
1	3rd significant digit	2nd significant digit
2	5th " "	4th " "
3	7th " "	6th " "
4	9th " "	8th " "
5	11th " "	10th " "
6	13th " "	12th " "
7	2nd digit of exponent	1st digit of exponent

The decimal point for the Real number is automatically placed between 0 and 1.

A Sample Stack Calculation

A sample stack calculation should clarify the concepts explained above. The following Procedure receives an Integer from the calling routine. Imagine that the Procedure is simply going to double that value.

```
PROCEDURE Double(Number: Integer);
```

The calling routine might contain a statement like *Double(10);* If the Procedure contains assembly code, you must read the value passed to Number by calculating Number's position in the Stack. After pushing the Number on the Stack, 5 more bytes are added which contain information for using the call frame. Therefore, to calculate the location of Number, add 5 to the position of the Stack Pointer.

If the Procedure Double used 2 Values, its declaration would be

```
PROCEDURE Double(Number1, Number2: Integer);
```

The Stack containing the Parameter Values Number1 and Number2 would look like

Top of Stack

1.	
2.	
3	
4	
5	
6	Number2(lsb)
7	Number2(msb
8	Number1(lsb)
9	Number1(msb)

Bottom of Stack

Since you know that Number1 and Number2 are Integers and that Integers takes 2 bytes of memory, you should be able to calculate that the least significant byte (lsb) of Number1 is 8 bytes brom the top of the Stack. The lsb of Number2 is 6 bytes from the top of the Stack.

RULE: Each parameter is entered in the Stack as it is encountered. Successive parameters are put on top of previous parameters.

The Stack Pointer and the Label "LOCAL"

You may have wondered about the 5 bytes pushed on the top of the Stack. The first two are the subroutine return linkage. They contain the address that points to the memory location with the next executable instruction. The next 2 bytes contain the address of the previous stack pointer. The last byte is the lexical level of the current procedure or function.

The Kyan Assembler uses 2 predefined labels, **_SP** and **_Local**, which always contain the address of the current and previous Variables Stacks. A third predefined label, **_T**, contains the address of temporary zero page memory that the assembly routine may use as workspace. There can be up to 15 temporary bytes beginning at location _T and continuing to _T+14.

The absolute locations of these labels are

```
_SP          EQU   4
_LOCAL       EQU   2
_T           EQU  16
```

These labels can be used by the assembly routine to access values placed on the Stack.

ASSEMBLY LANGUAGE PROCEDURES AND VALUE PARAMETERS

To access Value Parameters passed to an assembly code Procedure, determine the offset from the Stack Pointer to the value parameter being passed. First, load the accumulator with the least significant byte of the Value and store it in workspace _T. Then repeat the process to get and store the most significant byte of the Value. Repeat the process for each Value you want to access. You must access Values by loading their least and then most significant bytes because the 6502 processor can only handle one 8-bit piece of data at a time.

The following Procedure is passed three integer parameters by a calling routine. It also uses two integer variables that are local to the Procedure. The assembly routine accesses the third Value Parameter, C, and loads it into _T and _T+1.

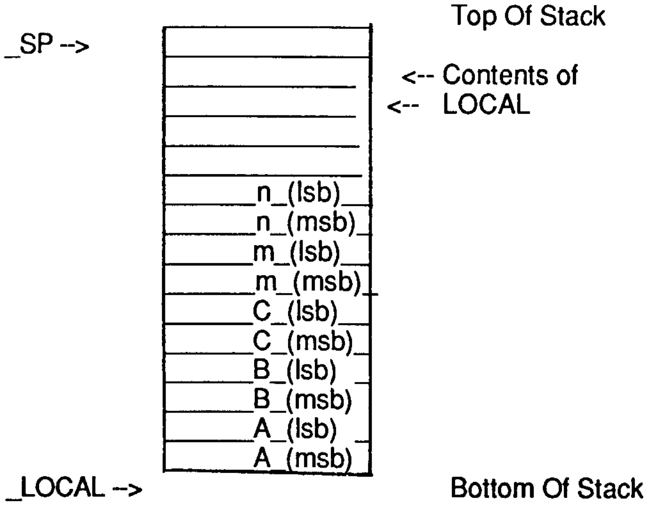
```
******************************************************
PROCEDURE  Access(A, B, C: Integer);
VAR
 m,n : Integer;
BEGIN
#a
    LDY #9    (* the offset from _SP to "C" *)
    LDA  (_SP),Y
    STA  _T
    INY
    LDA  (_SP),Y
    STA  _T+1
#
END;
******************************************************
```

COMMENTS

1. Three Integer values, A, B, and C are passed to the Procedure by the calling routine.

2. The Procedure contains two local integer values, **m** and **n**.

3. The Parameters Stack has the following structure.



4. Since all of the Values and Local variables are Integers, each element uses 2 bytes of memory. (Refer to the List in the previous section for memory size used by the different data types.)

5. The top 5 stack positions contain the Stack Pointer and the address of the Bottom of the Stack.

6. The offset from _SP to C is the total of the 5 bytes at the top of the Stack, plus 2 bytes each for the integer variable, m and n; i.e., 9 bytes.

7. The statement LDY#9 loads the offset value to C in the Y register.

8. The statement LDA (_SP),Y uses the offset from the _SP to load the least significant byte of the Integer Value being passed, i.e. C, into the accumulator.

9. That value is then stored in the temporary workspace, _T, for use within the Procedure.

10. Finally, the Y register is incremented and then used to provide the offset to the most significant byte of the Value C.

The general rule for calculating the position of a Value Parameter in the Stack is

```
Offset (from _SP)  =  5 bytes  + bytes of memory used by values
                                  above the Value desired
```

The following example shows another example of passing parameters by value and accounting for local variable definitions.

```
Procedure (x: integer);
Var b: boolean;
BEGIN
  ....
  ....
#A
  ....
#
END;
```

```
_SP +5     Add 5 bytes for overhead
    +1     Add 1 byte for local boolean, b
    +6     LSB of X
    +7     MSB of X
```

The number of bytes used for the local variables must be added to calculate the offset from _SP for a passed parameter.

LOCAL

You can also use the Label _LOCAL to calculate the position of a Value Parameter in the Stack. Remember that the predefined, absolute value of _LOCAL is 2. _LOCAL is the address of the bottom of the stack.

The formula for calculating the position of a Value Parameter using the Label _LOCAL is

```
Offset(from _LOCAL) = Bytes used by the desired Value
                      +
                      Bytes below the Value on the Stack
```

The offset must be subtracted from the value of _LOCAL to reference the parameter.

Passing Value Parameters: Summary

So far, we have discussed only Values passed in the Parameter list. Assembly code, as we have noted, cannot identify the Pascal concept of SCOPE that is usually involved with Parameters.

Consequently, when a Pascal routine calls a subroutine, it passes the values in a Stack. Since all Pascal programs and subroutines list the declaration before the body statements, the position of the passed values is easily calculated.

A word of caution: never try to calculate the stack location of values based upon their relative positions in the program. That stack only contains the values passed to the subroutine. It cannot be used to access other values or variables used by the main program.

PROCEDURES AND VARIABLE PARAMETERS

An Assembly language Procedure reads Variable parameters from the stack in much the same way that it reads Values. The only difference is that when Variables are passed to a subroutine, the Stack contains only the address of the Variables being passed. In effect, the position in the stack that contains the Variable actually contains a pointer to that variable. Since all pointers require 2 bytes of memory, all Variable parameters in the Stack require 2 bytes of memory.

Once the Variable's address is determined, use the temporary workspace to store the actual variable. If data manipulations within the Procedure change the value of the Variable, the new value can be stored back into memory by referencing the address stored in the temporary workspace.

The following sample procedure expects to receive the identifiers, or names, of 3 integer Variables. It then reads the address of Variable C, which is the third Variable in the Parameter List. Once the address is identified, the actual value of the Variable is read. The calling routine might execute a command such as

```
        XYZ(Length, Width, Height);
```

The 3 Variables will be read into the Parameter Stack as the items identified as A, B, and C.

```
*************************************************************
PROCEDURE XYZ(Var A, B, C : Integer);

BEGIN
#a
  LDY #5          (* Load offset to C *)
  LDA (_SP),Y     (* Get LSB of ADDRESS of C *)
  STA _T          (* Save ADDRESS in _T *)
  INY
  LDA (_SP),Y     (* Get MSB of ADDRESS of C *)
  STA _T+1                 (* Save ADDRESS in _T+1 *)
  LDY #0
  LDA (_T),Y      (* Get LSB of C *)
  STA ...
  ...
#
END:
*************************************************************
```

COMMENTS

1. Since C is the last Variable passed to the Stack, it is 5 bytes from the top of the stack.

2. LDY #5 sets the offset in the Y register.

3. The accumulator is then loaded with the least significant byte of the address that refers to the Variable C.

4. The LSB of C's address is loaded into the temporary workspace, _T.

5. The Y register is incremented, and the most significant byte of C's address is read.

6. The MSB of C's address is then read and stored in _T+1.

7. Once the address of C has been determined, the value addressed by _T is loaded into the accumulator.

The offset to C is calculated in the usual manner:

+ 5 bytes	offset from the Stack Pointer
+0	local Variables
5 bytes	total offset to LSB of C's address

The offset to the MSB of C is 6.

ASSEMBLER ROUTINES AND FUNCTIONS

Like a Procedure, a Function also receives data in the parameter list. It transmits the value calculated by the Function, however, in a slightly different manner. The manner is determined by the structure of the Function subroutine.

Remember that a Function receives data in the Parameter List, and then, after performing calculations that may include local variables, stores the resultant value in the location identified by the Function's name. The Parameter Stack for a Function, therefore, contains the Function's identifier.

As a rule, the Function's identifier is placed on the Parameter Stack after the passed parameters and before the local variables.

To calculate the position of the Function's identifier, use the same memory allotment guidelines that you used for Procedures.

The following sample Function receives an Integer Value from the calling routine and returns an Integer which is identified by the Function's identifier. It also uses a local variable which is also an Integer.

```
**************************************************
FUNCTION Test(A: Integer): Integer;

VAR
 B : Integer;

BEGIN
Test := 0;  (* Assignment required for ISO compatibility *)
...
END;
**************************************************
```

The Stack for this Function is diagrammed below. It is more detailed than previous Stack illustrations because it illustrates the byte structure of each Parameter item. Remember that the Function's identifier is passed to the Stack after the parameters and before the local variables.

```
                ____________   <--  Top of Stack
 _SP ->        |____________|  <--  Lexical Level
               |____________|  <--  LSB of old _SP
               |____________|  <--  MSB of old _SP
               |____________|  <--  LSB of RETURN
               |____________|  <--  MSB of RETURN
               |_____B______|  <--  LSB of B
               |_____B______|  <--  MSB of B
               |____Test____|  <--  LSB of Function Identifier
               |____Test____|  <--  MSB of Function Identifier
               |_____A______|  <--  LSB of A
               |_____A______|  <--  MSB of A
 _LOCAL ->                     <--  Bottom Of Stack
```

A Sample Function In Assembly Language

The following Function, XYZ, is passed three Integer Values and returns a Boolean value with the Function's name. Note that a Boolean requires only 1 byte of memory, so locations on the Stack are calculated accordingly. It also declares a local Variable, X.

The code in the sample Function reads the Value of B. After performing a series of undetermined calculations, it theoretically determines a Boolean value. That value is stored in the accumulator. The accumulator is then stored in the Stack location associated with the Function's identifier.

NOTE: Conformance to the ISO standard dictates that the compiler return an error whenever it encounters a FUNCTION written entirely in assembly language. This occurs because ISO requires any function identifier to be explicitly assigned a return value. For this reason, a dummy assignment using the function identifier should be included as the first line of a function.

```
************************************************************
FUNCTION  XYZ(A, B, C: Integer): Boolean;

VAR
 X: Integer;

BEGIN
   XYZ := TRUE;  (* Required for ISO compatibility *)
#a
  LDY #10        (* The offset to B *)
  LDA (_SP),Y  (* Put LSB of B in Accumulator *)
  STA  _T         (* Store LSB of B in workspace *)
  INY
  LDA (_SP),Y  (* Put MSB of B in Accumulator *)
  STA  _T+1     (* Put MSB of B in workspace *)
   .
   .
   .
  LDA ...          (* Put Boolean value in Accumulator*)
  LDY #7          (* The offset to Function Identifier*)
  STA (_SP),Y  (* Put Boolean value in Identifier, XYZ*)
   .
#
END;
**************************************************************
```

COMMENTS

1. The FUNCTION XYZ is assigned a dummy value to conform to ISO requirement.

2. The offset from the Stack Pointer to B is 10:

+5 bytes	offset to first stack parameter
+2	local Integer Variable
+1	Boolean XYZ (the Function identifier)
+2	Integer Variable C
10 bytes	total offset to B

3. The program then reads the least and most significant bytes of the Value B, storing each in the temporary workspaces, _T and _T+1.

4. Finally, a Boolean value is loaded into the accumulator. That value is then loaded into the Stack location reserved for the Function Identifier, XYZ, which is offset from the Stack Pointer by 7 bytes.

+5 bytes	offset to first stack parameter
+2	local Integer Variable X
7 bytes	total offset to XYZ

Remember that the only difference between Functions and Procedures is that the Function Identifier is placed on the Parameter Stack. It is located after the passed parameters and before the local variables.

MISCELLANEOUS OPERATIONS

The following subprograms illustrate how to use assembly code to Peek and Poke memory locations. A Poke statement is a Procedure since it enters a value into a specific memory location. A Peek statement is a Function since it returns the value determined by the addressed memory location.

POKE

The following Procedure Pokes the value, **Val**, into memory location, **Loc**. The rules for locating parameters on the Stack indicate that **Loc** is offset from the Stack Pointer by 7 bytes and that **Val** is offset by 5 bytes. The Procedure first reads the Parameter List to determine the memory location to be poked. It then reads the value to be entered. Finally, after clearing the Y register, it stores the value to be poked into the address contained in _T.

```
************************************************************
PROCEDURE  Poke(Loc, Val: Integer);

BEGIN
#a
 LDY #7            ; Offset from _SP to Loc;
 LDA (_SP),Y       ; Get LSB of Loc;
 STA _T            ; Save LSB of Loc;
 INY
 LDA (_SP),Y       ; Get MSB of Loc;
```

```
    STA _T+1          ; Save MSB of Loc;
    LDY #5            ; Offset from _SP to Val;
    LDA (_SP),Y       ; Load Val into Accumulator;
    LDY #0                   ; Clear Y register;
    STA (_T),Y        ; Store the value in the Accumulator
                      ; in memory location _T;
  #
  END;
  ****************************************************************
```

PEEK

The Peek statement is actually a Function since it returns a single value that is based upon the Location parameter which indicates the memory address to be read.

That value is stored in the memory location reserved for the Function's identifier.

The following assembly code subroutine returns the value contained in the memory location, **Loc**.

```
  ****************************************************************
  FUNCTION  Peek(Loc: Integer): Integer;

  BEGIN
    Peek := 0;
  #a
   LDY #7             ; Offset to Loc ;
   LDA  (_SP),Y       ; Get LSB of Loc;
   STA  _T            ; Save LSB of Loc in workspace;
   INY
   LDA  (_SP),Y       ; Get MSB of Loc;
   STA  _T+1          ; Save MSB of Loc in workspace;
   LDY #0             ; Clear Y register;
   LDA  (_T),Y        ; Load Accumulator with the
                      ; Address being Peeked;
   LDY #5             ; Offset to Function Identifier
   STA  (_SP),Y       ; Store contents of Accumulator
                      ; in LSB of Function Identifier
    INY
    LDA  #0          ; Load Accumulator with 0 for MSB
                      of return integer
    STA  (_SP),Y     ; Store contents of Accumulator;
                     ; in MSB of Function Identifier;
  #
  END;
  *********************************************************************
```

COMMENTS

1. Peek is the reverse of Poke.

2. Read and store the memory location you want to examine in temporary workspace.

3. Read the memory location of the Function Identifier and write the value stored in the temporary workspace into the Function Identifier's location.

4. Note that Peek returns an integer and Poke writes to memory only a byte.

CONCLUSION

If you know Assembly Language programming, you should now be able to include Assembler routines in your Pascal programs. You should also have learned how to pass data items through parameter lists.

Since all data items are similar to Values or Variables, only those types of data were covered in this section. Values may be passed by any of the data types itemized in the list at the beginning of this section. Variables are always put on the Parameter Stack in terms of their pointers--i.e. they always occupy 2 bytes of memory allocation.

(This page left blank for your notes.)

VI WORKING WITH KIX

OVERVIEW

This manual has explained Kyan Pascal in terms of the file procedures used by the ProDOS operating system. We assumed that most users were familiar with it, and we felt that the fewer issues forced upon the beginning user, the easier it would be to learn Pascal. Actually, however, Kyan Pascal uses the KIX programming environment. KIX extends the standard ProDOS functions to include features like those supported by Berkeley UNIX.

KIX is a powerful disk and file management system that gives the programmer immediate and direct access to any file in the system. It also supports an extensive body of commands that let you find, move, copy, compare, and manipulate files.

When you boot Kyan Pascal, the KIX prompt % appears instead of the usual ProDOS prompt > . One of the powerful features of KIX is that you can issue KIX commands whenever you have the KIX prompt. This eliminates the need to access the ProDOS FILER when you want to manipulate files and disks.

This section presents an overview of the KIX system. It then explains the six groups of KIX commands.

* Directory Control
* Listing Directory and File contents
* Manipulating Files, Directories, and Volumes
* Comparing Files and Volumes
* Searching Files and Directories
* Controlling Date and Display attributes

Finally, it explains the configure utility and the use of wildcards, a feature that makes KIX extremely versatile and powerful.

NOTE: You can enter a KIX command whenever you have the % prompt.

THE KIX FILE STRUCTURE

To understand the use of KIX commands and the versatility of the KIX environment, you must understand the hierarchical structure of the KIX system.

KIX stores files in Directories or Subdirectories. Subdirectory information is stored in Directories. Directories are stored in Volumes. A Volume is actually the main Directory.

You can store related files in the same directory or subdirectory, and use directory names that are significant to the files stored in them.

The KIX storage scheme is similar to UNIX in that the command files are stored in a directory named /BIN on the master disk. The KIX.SYSTEM file, located in the volume directory, acts as a system shell or command interpreter. Every command you enter is interpreted by this program which then calls other files to perform the indicated task.

Study the following diagram carefully. It illustrates the structure of a hypothetical system that has two Volumes concurrently mounted on the system. NOTE: This hierarchical file structure is used in examples throughout this chapter.

```
                              / (Null)
          Volume1                              Volume2
             ^                                    ^
Directory1  Directory2                Directory1  Directory2
     ^           |                        |           ^
SubD1  SubD2     |                        |      SubD1  SubD2
  |      |       |                        |        |      |
File1  Filea   FileA                    FileX    Filea  File1
File2  Fileb   FileB                    FileY    Fileb  File2
File3          FileC                             Filec  File3
                                                        File4
```

The first / indicates the null directory. From here you can access all volumes in the system. A full pathname always includes this symbol.

The diagram illustrates two Volumes, Volume1 and Volume2. Each disk, of course, has its volume name. The Volume contains a list of all Directories written on the disk. Volume1 has two principal directories; Volume2 also has two main directories.

Directory1 on Volume1 has two subdirectories. The first subdirectory contains three files. The second subdirectory contains two files. Directory2 on Volume1 contains three files.

Directory1 on Volume2 contains two files. Directory2 on Volume2 contains two subdirectories. The first contains three files, the second contains four files.

PATHNAMES

To access a file, identify its Volume name, Directory name, Subdirectory name (if necessary), and Filename. KIX refers to this means of identifying a file as a Pathname. Note that files in different directories can have the same name without confusing the operating system. This is because the full pathnames are different.

Pathnames are written by placing a / before each element in the path. For example, **/Volume2/Directory1/FileX** accesses FileX. Locate this file on the diagram to make sure you understand its pathname.

For practice, locate the following files on the diagram:

/Volume1/Directory2/FileA

/Volume1/Directory1/SubD2/Fileb

/Volume2/Directory2/SubD1/Fileb

/Volume2/Directory2/SubD2/File3

MOVING AROUND THROUGH PATHS

Once you understand the concept of KIX pathnames and the structure of the system, you may wonder how to move from one file to another. A few rules govern this process.

1 A directory that contains other directories or files is called a PARENT Directory. The term is relative, i.e., a directory may be the parent directory of a group of files, yet itself have a parent directory. SubD1 on Volume1 illustrates this principle. Directory1 is its parent directory.

2. You can call files within the same working directory without specifying a full pathname. For example, if the prefix for the working directory is /Volume1/Directory2/, you can call File A from File B by simply entering the filename.

3. To call a file from a file that has a different parent directory, you must use the full pathnames for that file. The following examples illustrate this principle.

   To move from File1 in SubD1 to Fileb in SubD2, you must indicate the full pathname of Fileb The pathname would be **/Volume1/Directory1/SubD2/Fileb.**

   To move from Fileb in SubD2 to FileA in Directory2, the pathname would be **/Volume1/Directory2/FileA.**

Once you understand pathnames, you can begin to use the many KIX commands that let you create directories, move and copy files within or between directories, compare files, or manipulate the system itself.

KIX COMMANDS

Every KIX command has the same format:

```
% command [-options] [/pathname]
```

At the KIX prompt % enter any KIX command. You may use either upper or lower case letters.

Follow the command with any options that the command allows. Each command has its own list of options. To indicate an option, precede the option character with a - (minus) and separate multiple options with a space.

After the list of options, indicate the pathname to the file you wish to access. As indicated in the previous section, the extent of the pathname depends upon your current location in the file structure and the function you want to perform.

The following example illustrates the command that lists the contents of a directory named Routines on a volume named Subprograms. It produces a list of all the files contained in the directory along with important information about each file. It doesn't matter if you don't understand the command, as long as you can follow the syntax.

```
LS -l /Subprograms/Routines
```

LS tells the KIX command interpreter to write a list. **-l** is a command option which indicates that all information known about the files should be displayed. **/Subprograms/Routines** identifies the directory whose contents will be displayed.

The many KIX commands are grouped into the six major categories listed in the Overview to this section.

DIRECTORY CONTROL

Directories are identified, created, removed, or changed using the following commands

PWD
MKDIR
RMDIR
CD

PWD: Print Working Directory

PWD, the Print Working Directory command, prints the current directory you are in. It is an extremely useful command that you can use whenever you can't remember where you are in the system's structure.

If you issue this command when you first boot the system, it will print

```
/Kyan.Pascal
```

If you are working on a file and can't remember which directory you have saved it in, save the file, get the % prompt, and issue the PWD command. The pathname to your directory will be displayed.

Once you are in a directory, any unprefixed filenames are assumed to reside within that directory.

MKDIR: Make Directory

The MKDIR command creates a directory with the name you specify. Two rules apply to making directories.

1. You cannot create a directory with the name of an existing directory.

2. If the directory you are creating is a subdirectory of another directory, the directory name must be a valid pathname.

MKDIR can not be used to name a Volume or Disk. To create a volume, see the KIX commands FORMAT and CPV in the "Manipulating Files" part of this chapter.

If the working directory is /MYDISK,

MKDIR XTRA creates the directory XTRA on MYDISK

MKDIR XTRA/MORE creates a subdirectory, MORE, in the directory XTRA on MYDISK

MKDIR /NEWVOL/D1 tries to create a directory, D1, on disk NEWVOL.

The initial / indicates a new volume since the first / means the Null position in the file hierarchy. If MKDIR can not find the volume for the directory, it displays the message

```
INSERT DISK FOR /NEWVOL/D1
PRESS RETURN TO CONTINUE; ESC TO SKIP
```

If you press <**RETURN**>, MKDIR tries to make the directory again. If it still does not find NEWVOL, it repeats the INSERT DISK message.

If you press <**ESC**>, MKDIR skips the indicated directory. If you are creating number of directories, it goes to the next one in the list.

MKDIR will create as many directories as you indicate. Each directory, however, must be separated by a space. MKDIR tries to create the directories in the order they are listed. All of the directories in the above examples could have been created with one MKDIR command:

```
MKDIR XTRA XTRA/MORE /NEWVOL/D1
```

MKDIR would first create the directory, XTRA. Then it would create the subdirectory of XTRA, MORE. Finally, it would search the entire system for the volume, NEWVOL -- note the initial / that indicates the Null location. If it finds the disk, it creates the directory, D1. If it does not find the volume, it prints the INSERT DISK message.

Once you have created a directory, store related files within that directory.

RMDIR: Remove Directory

RMDIR destroys the directory or directories listed. **Before a directory can be removed, it must be empty of all subdirectories and all files.** If the directory is not empty, RMDIR prints a "FILE ACCESS ERROR" message. If any other problems arise, (e.g., the correct volume cannot be located), the INSERT DISK message is displayed.

CD: Change Directory

The Change Directory command lets you move from directory to directory. You must use valid pathnames that clearly indicate all the information the system needs to access the directory.

For example, typing

% CD /MYDISK sets the system prefix to /MYDISK. IF CD cannot locate the volume you have requested, the INSERT DISK message is displayed.

% CD / sets the system prefix to the Null location. All files must be specified with complete pathnames, including the volume name. In multiple drive systems, this allows you to move from one disk to the next.

% CD sets the system prefix to the 'home' volume of the present working directory. If the working directory is /KIX/XTRA/SubD, typing CD sets the working directory to /KIX. In other words, CD by itself returns the name of the volume the working directory is on. This command is one level lower than the CD / statement.

As usual, when a CD command cannot locate the indicated volume, the INSERT DISK message is displayed and the system waits until you insert the appropriate disk and press <RETURN> or press <ESC> to abort the command.

LISTING DIRECTORY AND FILE CONTENTS

Whenever you have the KIX prompt (%), you can execute a number of commands that return information about the directory or file you are working with. They are

LS - List
LPR -- Line Print
CAT -- Concatenate

LS: List Directories or Files

The LS command lists information about the directories, subdirectories, or files contained in the pathname you indicate. If you list a file, only information about that file is displayed. The LS command has many options which allow you to control which files you want to select and what information you want to see.

LS has two basic forms: the long list, which displays everything the system knows about the directory or file; and, the short list, which displays only the filenames contained in a directory.

When you save a file in a directory, KIX also stores information about that file. In the directory, it saves

Length	(in blocks)
File Type	(Text, Binary, Directory, System, Other)
Date/Time	(created or modified)
File Length	(in bytes)
Subtype Value	
Protection	(Read, Write, Rename, or Delete allowed)

List Options

The LS command has several options that let you specify how much of this information you want to know.

-l List using the 'long' format. This includes all of the data items listed above except the protection status of the files.

-p List the protection status for each file. To use this option, you must also have selected the -l (long format) option. A file may be protected from reading, writing, renaming, or deleting. The symbols used in the display are

r	Read protection is set
w	Write protection is set
n	Rename protection is set
d	Deletion protection is set
-	Protection not active

The -p option will display the categories that are protected. For example, if the screen displays **-w-d**, you may read or rename the file. You may not, however, write to the file or delete it.

-f Prints the character code abbreviation for the file type. The file type abbreviations are:

&	Text file (txt)
*	Executable file (bin)
/	Directory (dir)
@	System File (sys)
?	Other

-n Do not sort the filenames into alphabetical order as they are printed.

The output produced by the LS command depends upon the options you select and the pathname -- or lack or pathname -- you indicate. The normal rules for pathnames apply.

If you enter the command:	The screen displays:
LS -l /	a long list of all volumes available to the system
LS -l /VOL1	a long list of all files contained in the volume directory VOL1
LS -l /VOL1/DIR1	a long list of all files contained in DIR1 on VOL1.

If LS cannot locate the specified volume, the INSERT DISK message is displayed and the system waits for you to insert the correct disk and press **<RETURN>**, or press **<ESC>** to abort the process.

You can be even more specific about the files you want to access by including the type of file you want listed. The LS command accepts an additional option which indicates the type of files you want listed. If you only want information about the text files stored in DIR1 on disk VOL1, for example, type the command

```
LS -l -p /VOL1/DIR1 ;txt
```

The file types are indicated by the codes

;txt	List only text files
;bin	List only binary files
;dir	List only directories
;sys	List only system files

OUTPUT

Usually LS lists all the files you specify for output in alphabetical order. Using the -n option disables the filename sort and causes LS to list each file in the order in which it is actually stored in its directory. The option works in both short and long listing modes.

Output is usually directed to the screen. When the screen becomes full, output stops and the message appears at the bottom of the screen: [More].

Redirecting List Output

Although you usually direct the output of a list command to the screen, you may sometimes want to print a hard copy and, occasionally, you will want to save the list in a disk file.

The list command has a final option which redirects the output of the LS command to another device -- either a printer or disk drive. To redirect output, conclude the LS statement with a ">" followed by the slot number of the printer or a pathname indicating the file which will store the output. If no destination is specified, output goes to the screen.

For example, the command *% LS -l -p /VOL1 ;bin > 1* prints a long listing of all binary files contained on Volume 1. This listing includes all protection status conditions of the binary files. It prints this information on the printer in slot 1.

Short Listings

If you omit the -l option, the list command defaults to the 'short' list format. A short list simply includes the filenames requested and their respective types. The -f option which prints the character code abbreviation can be used with the short list command. The -p option, which lists the protection status of each file, is ignored.

Listing The Root Volume

LS uses a special output format when you specify the 'root' volume, i.e. the Null position in the file hierarchy.

```
LS /
```

prints the slot and drive location, name, and number of blocks free for each volume that is currently on the system. Finally, it prints the total number of blocks available on all volumes and the number of volumes currently 'mounted' on the system.

LPR: Line Print

The line print command prints the files you name, using the margins, special printer codes, and other formatting options specified in the system-configuration program. See the CFG command at the end of this section for information about resetting the default configuration.

LPR prints the output of text files as text; all other file types are ignored. You can stop the printer output by pressing the <ESC> key. If LPR cannot find the file you specify, it displays the INSERT DISK message.

CAT: Concatenate

The CAT command performs a number of functions. It lets you print the contents of a file, copy many files into a single file, or print files using different format options. CAT supports the following options

-n	Number output lines, starting at line 1
-b	Do not number blank lines
-s	Remove adjacent blank lines
-v	Print control characters as ASCII equivalents (for example, print ASCII 3 as [^c])

For example, CAT -n -b -s /Vol1/Dir1/Myfile, displays the contents of Myfile on the screen. It numbers the lines that contain data and removes multiple blank lines.

CAT prints text if the file is a text file. Other files are printed as 'hex' dumps.

Output Control

Normally, CAT prints to the screen. It prints 22 lines and prompts: [more] . Press <RETURN> to continue output to the screen; press <CONTROL>-<RESET> to cancel CAT. The CAT command also supports the output redirection command at the end of the line. >1 redirects output to the printer in slot 1. When redirecting output to the printer, all CAT options are available.

Assuming that the working directory is Myfile,

```
    CAT -n Myfile >1
```

prints the file, with numbered lines, on the printer in slot 1. When printing hard copy, CAT uses an 80-column, 60 line-per-page format.

Output to another file

If you redirect output to a pathname, all the files listed are copied in the file indicated by the pathname. Each file is appended to the preceding file. The first source file determines the type and protection status of the resultant file. If one of the files being copied is not the same type as the first file, CAT closes the destination file and exits the command. (NOTE: The format options effect the copying of text files only).

Use the CAT command to merge a number of small files into a larger unit. The following example merges three chapter files into one large file named Section which is contained in a directory named Book.

```
        CAT Chapt1 Chapt2 Chapt >/Vol1/Book/Section
```

If CAT cannot find the files or volumes indicated, it displays the INSERT DISK message. Press <ESC> to skip the current file and continue. Otherwise, place the appropriate disk in the drive and press <RETURN> to continue.

MANIPULATING FILES, DIRECTORIES, AND VOLUMES

Once you have created directories and files, six commands let you manipulate them.

CP	Copies files
MV	Moves files
RM	Removes files

CHMOD	Changes the protection status of files
FORMAT	Formats a disk
CPV	Copies (Duplicates) a volume

The functions of these commands are obvious. The important point to remember is that you use valid pathnames to identify the source and destination files. Also note that a copy is an exact duplicate of the original so you have two versions of the same file. Move, on the other hand, produces an exact copy of the original but destroys the source file.

CP: Copy

The copy command produces a replica of the source file in the destination file. The syntax is: *CP Source Destination*

CP supports one option, **-i**. It prompts you with an inquiry if the destination file already exists. If you answer the inquiry with a **Y**, the existing file is overwritten.

If the destination of the copy command is a directory, CP writes the source file into the directory and retains the original file name. If you specify only a filename to copy, CP will put the file in the working directory with the source file's filename.

Examine the following copy commands.

CP File1 /Vol1/Dir1	Copies File1 into the directory Dir1 and retains the file name File1.
CP -i File1 Dir1/SFile	If the file SFile already exists, CP prints the message REMOVE EXISTING "DIR1/SFILE" (Y,N)

If you respond with Y, the existing file is overwritten. If you press N, the command is aborted.

CP permits a list of source files to be copied into a destination directory. If the last pathname specified is not a directory, CP automatically moves all of the files listed into the working directory.

CP will not copy subdirectories. If a source file listed in CP is a subdirectory, it will be ignored. No error message will be generated.

Note: Due to a bug in ProDOS, the CP command will not function properly in systems with only one disk drive. You must continue to use the Filer to copy files from one disk to another.

MV: Move

The move command allows you to reset the pathname of a file. This has the effect of moving the file or renaming it. When a file is moved, the original file is deleted. MV allows two options

-i	Prompts for the new name of each file listed
-f	Ignores the protection status of files

If the destination of the MV command is a directory, the files are simply copied in the directory and retain their original filenames. You cannot use the -i option when copying a file into a directory.

Note: If you specify more than two files without including the -i option, the destination must be a pathname to a directory. Without the query option, the system must have some other way to identify file names. By copying them into a directory, they can retain their original identifiers.

If the last pathname listed is not a directory, MV will move the files into the working directory.

When moving files using the -i option, MV will prompt you for a pathname for each file listed. Typing <RETURN> without a pathname skips the MV command for that file.

RM: Remove

The remove command deletes files and directories. Before it can delete a directory, however, the directory must be empty. RM supports 3 options

-i	Inquire before destruction of each file
-f	Destroy file regardless of protection
-r	Empty directory specified before deleting it

The -i option lets you make sure you want to delete the specified file. It is a final protection mechanism. If you decide not to destroy the file, RM moves to the next file in the list.

Deleting Directories

In general, you cannot delete a directory unless it is already empty. (See the RMDIR command.) RM lets you delete non-empty directories by using the -r option which automatically deletes all files in the source directory (including subdirectories and their files) and then the directory itself.

If you include the -i option while deleting a directory, the prompt only inquires about the destruction of the first directory. If it contains subdirectories with their own files, they will be destroyed without warning. In addition, the -f option, which ignores the file protection status of each file, affects the deletion of each file in the target directory. The following command would delete all files and subdirectories contained in the directory TESTDIR.

```
% RM -i -f -r TESTDIR
```

Before the command is executed, the screen displays the message:

```
WARNING: TESTDIR WILL BE EMPTIED
REMOVE TESTDIR? (Y/N)
```

If you select **Y**, RM deletes all the files in each subdirectory, then those in each directory, moving up through the file hierarchy until it deletes TESTDIR itself.

If you are deleting non-directory files, the -r option has no effect.

CHMOD: Change Protection Mode

Every file has 4 modes of protection which determine if the file can be read, written to, renamed, or deleted. The change protection mode command allows you to set those protections. Each option is indicated by a character:

r	Read access
w	Write access
d	Destruction access
n	Rename access

In addition to indicating the type of access, you must also indicate whether to allow or deny permission for that mode. A + (plus) before the mode character allows the mode selected. A - (minus) before the mode character denies permission to access that mode.

Examine the following uses of the CHMOD command.

CHMOD +r -w -d +n File1 — allows a user to read or rename the file, File1. But the file cannot be written to or destroyed.

CHMOD +r +w -n File1 — lets the user read or write to the file. But it cannot be renamed. The deletion status is unchanged.

FORMAT: Format A Blank Disk

The Format command lets you format a disk while still remaining in the system. It is an invaluable capability when you suddenly need a disk and don't want to lose what you are working on, or don't want to exit the system to enter the ProDOS filer. The syntax of the FORMAT command is:

```
% FORMAT (s,d) /VolumeName
```

where "s" represents the slot number, and "d" the drive.

To make formatting easier on single-drive systems, FORMAT tells you to insert the blank disk before formatting actually begins. Insert the disk and press <RETURN>; or press <ESC> to stop the process.

Once formatting begins, a check determines if the disk in slot "s", drive "d" is already formatted. If it is, a message asks you to verify its destruction. The disk will be reformatted and given the new volume name.

CPV: Copy Volume

The copy volume command duplicates the source volume in the destination volume. The syntax of the CPV command is:

```
CPV (ss,sd) (ds,dd)
```

ss	source slot of original disk
sd	source drive of original disk
ds	destination slot of new volume
dd	destination drive of new volume

CPV identifies the source disk and asks if the new volume should have the same volume name. A positive response initiates the copy sequence. A negative response results in a request for the new volume name. Press <RETURN> to terminate the CPV command.

If the CPV command determines that the destination disk is already named, it will ask you to verify the destruction of the existing destination disk. A **Y** response causes CPV to format the disk and copy the volume. A **N** response terminates the CPV command.

COMPARING FILES AND VOLUMES

KIX supports commands that let you compare files and volumes to determine if they match. The two commands are

CMP	Compare
SDIFF	List differences

CMP: Compare

The CMP command compares files. The syntax of CMP is

```
CMP File1 File2
```

When CMP finds the first difference between the two files, it reports the location of the difference as an offset from the start of the files in Block 1, Byte1. No further comparison is made after the first difference is discovered.

The following input and output

```
% CMP File1 File2
FILES DO NOT MATCH: BLOCK 2, BYTE 457
```

compares the two files and finds a difference in the second block of the files. It identifies the difference in the 457th byte. If the files are identical, the message **FILES MATCH** appears on the screen.

CMP also supports the comparison of volumes. It works in the same manner as a file comparison, comparing the two volumes byte by byte until a difference is located. The syntax of the command, however, is different for volumes. The command must include the slot and drive identification of both volumes being compared. The syntax is:

```
CMP (ss,sd) (ds,dd)
```

ss	source slot of original disk
sd	source drive of original disk
ds	destination slot of new volume
dd	destination drive of new volume

SDIFF: Source Difference

Source Difference is a version of the CMP command. It is used to compare two **text** files, and prints the lines which contain differences. After 3 differences are located, the comparison terminates. If the files are not text files, SDIFF aborts.

SEARCHING FILES AND DIRECTORIES

KIX supports two commands that allow you to search directories for a specific file and to search files for a specific string.

FIND	Locates a file
GREP	Locates a string within a file

These commands do not have any options.

FIND: Locate File

The find command locates the specified file within the volume or directory indicated. The extent of the search is determined by the pathname of the directory to be searched. The command returns the pathname of all files that match the filename to be located. The syntax of the command is

```
FIND PathToDirectory  -Filename
```

Note the "-" (minus) sign preceding the filename; this is an essential part of the command syntax.

The path to the directory determines the extent of the search.

For example,

FIND /-File1 searches the entire system, including all volumes currently mounted on the system, for File1. It prints the pathname to the file everytime it finds File1. The / indicates that the search should begin with the root or null location in the system which includes all volumes.

FIND/VOL1-FILE1 searches the volume, VOL1, for any directories that contain File1. It prints the pathname to the file whenever it locates a file with that name.

For example, assume /AnyVol/Dir1 and /AnyVol/Dir2/SubD1 both contain a file named File1. The command

```
FIND /AnyVol-File1
```

returns the following lines of output:

```
/AnyVol/Dir1/File1
/AnyVol/Dir2/SubD1/File1
```

If FIND cannot locate the specified volume name, it displays the INSERT DISK message. Press <ESC> to abort the command; or insert the appropriate disk and press <RETURN>.

GREP: Locating Strings in Text Files

The GREP command allows you to locate a string of characters in any text file. You indicate a string and a list of files or a pathname to be searched. When GREP locates the string in a file, it prints the Filename and the line that contains the string. The search continues until all the indicated files have been examined.

If the string being sought contains spaces, enclose the string in single quotes.

GREP uses the [more] convention described previously in those cases where the screen fills with text. Press **<RETURN>** to continue. To exit the listing and return to the system prompt, press **<CONTROL>-<RESET>**.

The following command searches two files, MasterFile and TestFile for the string, 'Copyright (c)'. The resulting output obviously assumes that such a string exists within the files listed.

```
% GREP 'Copyright (c)' MasterFile TestFile

MASTERFILE: Copyright (c) 1986 KYAN SOFTWARE, INC.
MASTERFILE: COPYRIGHT (C) 1983 APPLE COMPUTER, INC.
TESTFILE:  Sample Program Copyright (c) 1985
```

Note that the search is not restricted by case. Any version of the string will be reported.

DATE, DISPLAY, and SYSTEM COMMANDS

KIX supports a number of commands that let you control the display and output of system data and files.

DATE	Reads or Sets the calendar and clock
C40	Sets video output to 40 columns
C80	Sets Video output to 80 columns
SD	Sends the screen display to the printer
MENU	Display the startup menu of system programs
INTRO	Display an introduction to Kyan Pascal
KIX	Display a summary of KIX commands
QUIT	Exit to another ProDOS interpreter
CFG	Configure the KIX system

DATE

The DATE command lets you read or set the ProDOS system date and time files -- even if you do not have a clock/calendar card.

To display the current date and time, enter **DATE** and press **<RETURN>**. For example,

```
% DATE
17-JAN-86 07:56
```

To set the date or time, enter **DATE** followed by the year, month, day, hour, and minute two-digit values. The syntax is

```
DATE yymmddhhmm
```

yy	year	(85 through 99)
mm	month	(01 through 12)
dd	day	(01 through 28-31)
hh	hour	(00 through 23)
mm	minute	(00 through 59)

No spaces are allowed between the values to be set, although you may include any non-space, non-numeric character to make it easier for yourself to read the entry. The following examples set the same date-time values:

```
% DATE 8603241500

% DATE 86/03/24/15-00
```

Both commands set the calendar/clock to

```
24-MAR-86 15:00
```

If you set the day, DD, value to zero, the DATE command returns **NO DATE.** The system time, however, is always displayed.

The DATE command is compatible with the ThunderClock and other clock/calendar peripheral cards. If your system has one of these cards installed, the DATE command lets you set it. When you then execute the DATE command, the system returns the DAY field along with the other information.

Using DATE to set the system also sets the clock card -- if the card is set-enabled. You may also set the day-of-the-week feature on the card by including the 3-letter abbreviation for the day of the week after you have described the set-date command line. For example,

```
% DATE 8603241500 MON
```

sets the system calendar/clock to

```
MON 24-MAR-96 15:00
```

When you set the DATE command, the system does not check to see if the current day-of-the-week is the correct. If the system does not have a clock card, the day argument is simply ignored.

The day of month value is checked against the month passed to DATE. If the specified month does not contain the number of days indicated, the value is not allowed. For example, 31-SEP is rejected. Also, 29-FEB is allowed only if the year is calculated to be a leap year.

C40: Set Monitor to 40 Column Display

This command sets the 40 column monitor display. If the system has an 80 column card, it is disabled. The command clears the screen.

C80: Set Monitor to 80 Column Display

This command sets the 80 column monitor display. If your computer does not have an 80-column card, the command is ignored.

SD: Screen Dump

The screen dump command outputs the current screen display to the printer. The SD command defaults to the printer slot specified in the current configuration files. If no printer slot is specified in the configuration files, the command is ignored.

INTRO: Introduction

The INTRO command displays several pages of text which introduce the novice to Kyan Pascal and the general programming environment.

MENU: Main System Menus

The MENU command displays the Kyan Pascal Main System Menu. This menu lists names and descriptions of the primary Kyan Pascal system facilities.

KIX: KIX Command Menu

The KIX command displays a summary of each KIX command. It also lists the options and pathname formats.

QUIT: Exit KIX

The QUIT command calls the ProDOS quit routine and allows you to invoke a different ProDOS interpreter (e.g., call another application program). Entering QUIT at the system prompt invokes the prompt: EXIT KIX? (Y/N). Typing "N" voids the command and returns you to the system prompt. Typing "Y" calls the ProDOS Quit Routine.

CFG: Configure the KIX System

The CFG (configure) commands let you modify the configuration of KIX which is installed on the system disk. To modify the default configuration values, type CFG at the system prompt (%) and the Master Configuration Menu will appear.

```
****************************************************
          KIX SYSTEM CONFIGURATION

     OPTION                 DESCRIPTION
       1              Select Startup Options
       2              Select Printer Options
       3              Save Changes and Quit
       4              Exit Without Updating Files

  TYPE OPTION NUMBER __
****************************************************
```

The configuration program is divided into two phases -- startup options and printer options. Select Option 1 from the main configuration menu to set the startup options. Select Option 2 from the menu to select your system printer specifications. Select Option 3 to install the changes you have made. Select Option 4 to cancel the configuration program and return to the system prompt.

Option 1: Startup Options

The startup options and their defaults are

1. Automatically load KIX command files into RAMdisk ....YES
2. Enable 80 column card at startup ....YES

Kyan Pascal and the KIX operating environment are compatible with most RAM expansion cards available for the Apple II. To make it more convenient to use the RAMdisk capability which these cards offer, the Kyan Pascal has an optional utility which automatically loads the files in Kyan.Pascal/BIN directory into RAM when the disk is booted.

To use this option, your computer must have at least 128K of memory. Check your Apple Owner's Manual for information about creating and using a RAMdisk. Since the Ram disk is actually a portion of memory that is set up to act like a disk drive, it performs as if you had another system disk. Since it is also an electronic disk, it operates much faster than any external disk. If you will be using the RAMdisk for another application, or if you will be compiling large programs that include other Pascal files, you will probably want to keep the system files on the boot disk.

If the RAMDISK auto-load option is enabled, the KIX command interpreter will automatically make a BIN directory in /RAM and move the files stored in the boot disk's BIN directory into the one in /RAM. The KIX command interpreter will continue to move files into /RAM/BIN until it can find no more KIX commands small enough to fit into the remaining space in RAM or all KIX commands are loaded. Then, when you execute a KIX command, the KIX command interpreter will search for the command first in /KIX/BIN, then in the system disk's BIN directory, and, finally, in the directory specified by the System Prefix. A 256K RAM card is needed to load all of the Kyan Pascal files into /RAM. (**Note:** ProDOS only recognizes 64K of RAMdisk. If you have more RAM available and want to use it as RAMdisk, you must use the install program furnished by the manufacturer of your RAM card.)

Option 2: Printer Options

The system printer options and their defaults are

1.	Printer Slot Number	1
2.	Characters per inch	10
3.	Top Margin (inches)	1.0
4.	Bottom Margin (inches)	1.0
5.	Left Margin (inches)	1.0
6.	Right Margin (inches)	1.0
7.	Line Spacing	Single

To change a default value, type the option number. You will be then be prompted for a new value. Note that margin settings are limited to half-inch increments and to a maximum of 2.0 inches. **NOTE: Before changing any values, check your owners manual to make certain that your printer supports the changes you want.**

If you don't have a printer, specify slot 0 (zero) for the Printer slot number. Slot 0 tells the KIX command interpreter that you don't have a printer and to ignore SD and LPR commands.

If you select the Printer Configuration Menu, you can change the output slot number, the margins, the type size, and the line spacing of printer output.

Option 3: Installation

When you are ready to save the configuration update and quit CFG, select option 3 from the Main Configuration Menu. CFG will write the changes to your KIX system disk and give you instructions for powering down the computer and rebooting the system. By resetting the system in this manner, you are assured that the new configuration options will be properly installed on all copies of the files (particularly those loaded in RAMdisk). (**Note**: Configuration changes are installed only on the volume which was booted to load KIX. To make sure your configuration changes are properly installed, boot the volume "/KIX" (Disk2, Side 1) and then make the desired changes).

Option 4: Exit Without Installation

If you change your mind after calling the CFG program and want to quit, select Option 4. This command will abort the configuration program and return you to the system prompt.

ABBREVIATIONS AND WILD CARDS

KIX supports the use of abbreviations and wildcards when writing pathnames and strings. Directory abbreviations decrease the amount of typing required to enter KIX commands. Wildcards can be used to replace elements of filenames.

Directory Abbreviations

The two directory abbreviations are used to refer to the current working directory and the parent directory.

.	refers to the working directory
..	refers to the parent directory

Note that each dot represents one position up the path hierarchy from the current file.

The "." abbreviation may be used wherever a directory name is used. Assume that the working directory is /Vol1/Dir1. Examine the following commands and the actions taken.

CP /Vol2/File1 .   copies File1 from Vol2 to the present working directory, Dir1 on Vol1. The pathname of the new file is /Vol1/Dir1/File1.

MV ./File1 /Vol2   moves File1 in the present working directory to Vol2. The pathname of the new file is /Vol2/File1. The original file, /Vol1/Dir1/File1, is destroyed by the move.

The ".." abbreviation represents the parent, or source, directory of the Working Directory. In effect, it moves the system 1 step up the path hierarchy from the Working Directory -- or 2 steps up from the current file. If the working directory is /Vol1/Dir1/MyDir, ".." represents the path /Vol1/Dir1.

You can use the ".." abbreviation wherever a directory name is appropriate. Assume that the Working Directory is /Vol1/Dir1/MyDir. Examine the following command and the action taken.

CD ../LastDir   changes the Working Directory from /Vol1/Dir1/MyDir to /Vol/Dir1/LastDir.

If the Working Directory is the Volume itself, the .. command sets the system to the Root or Null volume. If the Working Directory is /Vol1, the command

```
  LS ..
```

sets the source directory to the Root volume and produces the special listing for the Root volume which includes all volumes currently mounted on the system, the size of blocks, and the number of blocks available.

More examples of using directory abbreviations are provided in the next part of this section, "Useful KIX Command Lines."

Wildcards

KIX supports two Wildcards that can be used to replace strings and characters in filenames.

?	represents any character in a filename
*	represents any string in a filename

These two wildcards give you a great deal of power when you issue KIX commands. If you can't remember the exact name of a file, use a wildcard in the filename before searching for it. If you want to locate variations of a string in files, use wildcards to identify parts of the string.

? (Question Mark)

The character wildcard, **?**, represents a single character. An example illustrates the use of wildcards more clearly than an explanation. The command

```
LS File?
```

produces a short list of all files in the present Working Directory that are named File followed by another character. If the directory contains File, File1, File2,...File9, the command

```
LS File?
```

lists all of these files. If you wanted to copy all these files into another directory, the wildcard lets you transfer all the files with a single command that replaces ten copy statements.

```
CP .NewDirectory/File?
```

copies files 1 through 9 from NewDirectory into the Working Directory. The names of the files remain the same.

In another use of **?**, the statement

```
CAT ?AT
```

lists the files contained in the present Working Directory that begin with any letter followed by the letters "AT."

* (Asterisk)

The * wildcard is a more powerful character substitute. It can represent any string of characters, including a null or empty string. Using this wildcard, you can quickly search through volumes and directories for files about which you know very little. For example, the command

```
CAT -n M*.S
```

prints the contents, with lines numbered, of any file in the present working directory whose name begins with the letter "M" and ends with ".S".

As another example, the command

```
RM /AnyVol/AnyDIR/M*.S
```

deletes any file located on the volume, AnyVol, in the directory, ANYDIR, whose filename begins with an "M" and ends with a ".S". If the following files existed on the disk, they would be removed.

```
/AnyVol/AnyDir/MouseFile.S
/AnyVol/AnyDir/ModelTest.S
/AnyVol/AnyDir/Memo.S
```

See the next section, "Useful KIX Command Lines," for more examples using wildcards.

Using Wildcards

The use of wildcards is very logical. They may be used in the source filenames of the pathnames you specify in the KIX command. They cannot be used if the system needs to know the exact file or path.

They cannot be used in destination pathnames since the system would not know what character or string to substitute for the wildcard. Wildcards are also not valid in the source pathname of some KIX commands. For example, the make directory command, MKDIR, can not use wildcards in the directory name since the system would not know what the name should actually be.

KIX provides another command which is intended to let you see exactly which files match the wildcard usage you are entering before any actions are actually performed on those files. **ECHO** lets you specify a pathname which contains a wildcard character. ECHO then prints the complete pathnames of all files which match the wildcard specification. Practice with this command so that you become familiar with wildcards before actually using them in KIX commands.

In general, any KIX command which accepts a list of pathnames as valid source file identifiers will also allow wildcards in the source filename. The following lists these commands.

Command	Wildcard Usage
RMDIR	Allowed
LS	Allowed
LPR	Allowed
CAT	Allowed
ECHO	Allowed
CP	See Note 1
MV	See Notes 1 and 2
RM	Allowed
CHMOD	Allowed
FIND	Allowed
GREP	See Note 3

Note 1: Wildcards are valid in the source pathname if the destination pathname is a directory.

Note 2: Wildcards are valid in the source pathname when the -i option is also selected.

Note 3: Wildcards are not permitted in the search pattern string. They are allowed, however, in the list of filenames to be searched.

One final note on wildcards. You may be tempted to try to combine the two wildcards. Don't bother. The * string wildcard will always include the ? character wildcard.

The next part of this section contains examples of common KIX command lines.

USEFUL KIX COMMAND LINES

This section provides many examples that will further your understanding of the KIX environment. You will find most of these commands immediately relevant to your programming needs.

KIX Command Line	Action Taken
RMDIR *	Delete all empty directories in the working directory
RMDIR FILE.?	Delete all empty directories in the working directory whose filenames begin with FILE. and have no extensions or a single letter extension.
LS -l*.S >LSFILE	List, using the long format, all files in the Working Directory which end in .S

	Write the output to the text file, LSFILE in the Working Directory.
LS / >1	List the names, slot and drive locations, and blocks free along with the total number of blocks free to the system and number of volumes found. The list is printed on the printer in slot 1.
LS /ANYVOL	List the names of all files stored in volume directory /ANYVOL.
LS /ANYVOL/*	List all the files stored in volume /ANYVOL, expanding all subdirectories stored in the volume directory ANYVOL.
LS /ANYVOL/* ;SYS	List the names of all system files stored on the volume /ANYVOL, expanding all subdirectories stored in the volume directory ANYVOL, but listing only system files in those subdirectories.
LPR /AVOL/APL*.TXT	Search the directory volume whose name is AVOL. Print those files which have names beginning with "APL" and ending with ".TXT".
CAT WORK/*.O > FINAL.O	Concatenate all files in the directory "WORK" whose names end in ".O" to the file FINAL.O in that working directory.
CP /AVOL/* .	Copy all files in volume "AVOL" into the present Working Directory. (Same as CP /AVOL/*).
CP DIR1/*. S DIR2/*.O	Copy all files with names ending in ".S" in directory DIR1 and files with names ending in ".O" in directory DIR2 into the present Working Directory.

CP /VOL1/F*

Copy all files in /VOL1 with filenames beginning with the letter "F" into the working directory using the same filename.

CP /VOL1/FILENAME

Copy /VOL1/FILENAME into the Working Directory using FILENAME as the name in the destination directory.

MV /VOL1/* .

Copy all files in volume "AVOL" into the present Working Directory, destroying each source file.

MV -i *

Rename each file the Working Directory, and request the new name to be entered at the keyboard.

MV FILE1 FILE2

Rename FILE1, FILE2.

MV D1/F1 D2/F2 D3/F3 .

Move files F1, F2, F3 from their separate directories into the present Working Directory keeping the same filenames.

RM -r *

Empty then destroy any directory in the present Working Directory.

CHMOD -w -n /V1/DIR1/*

Deny writing and renaming permission to all files in the directory /V1/DIR1.

FORMAT (6,1) /NEWVOL

Format the disk in slot 6, drive 1, and name the volume NEWVOL. FORMAT checks the disk to see if it has recognizable contents. If it has, FORMAT requests verification before destruction.

CPV (6,1) (6,2)

Copy the volume in slot 6, drive1 to the disk in slot 6, drive 2. If the target disk is recognizable, request verification of the destruction of its contents. CPV also verifies the use of the source disk's volume name, permitting another name to be entered in its place on the new volume.

FIND ./VOL1 -MYFILE

Search the present Working Directory and VOL1 for a file named "MYFILE."

FIND / -MYFILE

Search all mounted volumes for the file, "MYFILE". Unlike other volume arguments, / used with the FIND command extends the search to every directory in every mounted volume.

FIND /V* -LOSTFILE

Search all volumes whose name begins with "V" for the file named "LOSTFILE". The search extends to all directories in those volumes.

GREP apple *

Search all text files in the present Working Directory for the string "apple" or "APPLE" or any upper/lowercase combination of those characters. Print the filename and line of text containing this string.

GREP 'apple computer' /

Search all text files on the top levels of all volumes for the string "apple computer" in any combination of lower/upper case letters. Print the filename and line of text.

NOTES FOR KIX USERS

The following notes will help KIX users use the Kyan Pascal system more efficiently.

Moving KIX To A Hard Disk Drive

There is an alternative to moving the BIN directory into a RAM disk. If you have a hard disk drive, you can load the system files to that disk. This method optimizes KIX because it leaves all of /RAM free yet still allows very fast access to the KIX system.

To move KIX to your ProFile, boot the system using the Kyan Pascal Disk. Then enter the following commands:

% MKDIR /PROFILE/BIN	(creates the command directory)
% CP KIX.SYSTEM /PROFILE	(moves the KIX shell program to the hard disk)
% CP BIN/* /PROFILE/BIN	(copies all files in /KIX/BIN to the hard disk's BIN directory)

To move KIX to a SIDER, perform the same operations, but use the volume name **HARD1** in place of **PROFILE**. If **KIX.SYSTEM** is the first **.SYSTEM** file in the hard disk's directory, you will boot directly into KIX in the ProDOS subsystem.

One Drive Users

Due to the amount of code used to create the the KIX environment, it would be a good idea to remove the less frequently used KIX commands from the BIN library on your Kyan Pascal disk. The most useful commands are: PWD, CD, LS, CP, RM, and FORMAT. The other commands can be stored on a second disk (use the same Volume Name) and loaded when you need them.

The RESET Key

If you press the <**CONTROL**>-<**RESET**> keys during the execution of any KIX command, you will be returned to the KIX prompt. The message "RESET KEY PRESSED" and the name of the aborted command will be printed in the upper-left corner of the screen.

If you press the <**CONTROL**>-<**RESET**> keys at the KIX prompt, you will cause a warm-restart of the system.

Short-cuts for Advanced Users

1. Advanced users can call the Editor, Compiler and Assembler directly from the KIX system prompt using the following command syntax.

Editor:	ED pathname
Compiler:	PC pathname -options > redirection
Assembler:	AS pathname -options > redirection

2. Running KIX on an Apple IIe or IIc gives you an additional command line editing feature. When you are entering a command line argument, you can type <**CONTROL**>-X to erase the entry and start over again at the system prompt.

3. Advanced users can pick up some space on the disk and streamline the KIX process by removing MENU, INTRO, KIX, and CFG files from the BIN directory. The KIX.SYSTEM program will then boot directly into KIX instead of executing the MENU program.

4. KIX command options and output redirection characters can be placed anywhere on the input line (just make sure that pathnames are properly located). In addition, options can be specified in groups following a single - (minus) character . For example,

```
% LS >1 /VOLUME1/DIR1 -LP
```

prints a list of files in directory /VOLUME1/DIR1 to the printer in slot 1 using the long listing format with protection printed. The + (plus) character can be used with the CHMOD command.

5. KIX works best when moved to a hard-disk volume, leaving the RAMdisk free for other uses. However, a 256K (or larger) RAMdisk is big enough to load the KIX environment and all the Kyan system files. ProDOS only recognizes 64K of RAMdisk. If your card has more RAM available and you want to use it as RAMdisk, you must use a RAM install program which is furnished by the manufacturer of your RAM card.

CONCLUSION

Once you begin to master KIX, you will appreciate just how powerful and efficient it is. Since all of the Kyan Pascal files are KIX files, you will soon bypass the menus and, using KIX commands, control the system directly.

VII REFERENCE GUIDE

The Reference Section lists all the Reserved Words, Predefined Functions, Procedures, and Routines that are used in this implementation of Pascal.

The items are listed in alphabetical order to provide easy reference to any word or command. They are not grouped by types of words or routines since someone using this section might not know in advance the type of word or command.

Each entry contains a brief description of the purpose and syntax of the particular item. A series of comments then provide useful information about that item. Finally, it cross references related items.

This section is by no means exhaustive and is not a substitute for a complete, technical guide to Pascal. The entries provide general information that would be useful in normal programming situations. See the bibliography at the end of the Introduction for a list of more technical guides to the Pascal programming language.

ABS

PURPOSE Returns the absolute value of an expression

SYNTAX ABS (*expression*)

COMMENTS A predefined function.

The expression can be real or integer.

The absolute value of the expression is the same as the value of the expression if that value is positive. If the value of the expression is negative, the absolute value is the opposite of the value of the expression.

```
e.g.,   ABS(2) = 2
        ABS(-2) = 2
```

ADDRESS

PURPOSE

Allows the programmer to determine the address of a variable by referencing its identifier.

SYNTAX

IntegerVariable := ADDRESS(VariableIdentifier);

COMMENTS

This function returns the address of the first (lowest) memory location allocated to the data type which corresponds to the identifier specified.

For example, to find the starting address of an array of Integers, you would declare the array as usual and declare an integer VARiable for the array's memory location. Then, the statement IntForMemLoc := ADDRESS(Array of Integers) would put the 2 byte address of the array into the IntForMemLoc integer variable.

SEE ALSO

POINTER

AND

PURPOSE Returns the value TRUE if both expressions joined by AND are TRUE

SYNTAX A AND B

COMMENTS A Boolean operator.

AND evaluates two or more BOOLEAN expressions. All expressions evaluated by AND must be true for the results to be TRUE. If any expression evaluated by AND is FALSE, the evaluation returns a FALSE condition.

SEE ALSO OR

NOT

ARCTAN

PURPOSE Returns the arctangent of an expression.

SYNTAX ARCTAN(*expression*)

COMMENTS A predefined function.

The expression can be an integer or a real number. ARCTAN returns a real value expressed in radians.

SEE ALSO COS

SIN

ARRAY

PURPOSE

Declares a structured data type or a variable consisting of a number of defined elements of the same data type which share the same identifier

SYNTAX

ARRAY [*index-type*] OF *component-type*

COMMENTS

A predefined type of data.

An ARRAY is defined under the TYPE or VAR declaration. In the declaration, the borders of the ARRAY and the type of elements contained in the ARRAY must be identified.

The index-type may be an Integer, Char, BOOLEAN, or Subrange data type.

The component-type may be any predefined or user-defined data type.

To refer to an individual component of an ARRAY, refer to its variable-identifier, i.e. its name, and its index component. For example, if the Variable, Table, is identified as a type, TableType, and if TableType is defined as an ARRAY [1 . . 20] OF Integer, the expression

```
Table [ 5 ]
```

refers to the 5th integer in the ARRAY.

Any element in the ARRAY may be used anywhere in a program as long as the data type of the element is the same as the variable that refers to that element.

ARRAY (cont.)

An ARRAY may contain several dimensions. For example, the declaration ARRAY[1..5,1..10] OF INTEGER defines a grid 5 rows long and 10 columns wide. Each position in the grid can hold an INTEGER value. For more on multidimensional ARRAYs, see the chapter on ARRAYS in "Programming Techniques."

SEE ALSO "Programming Techniques"

TYPE

This page is intentionally blank

BEGIN

PURPOSE

Marks the start of a program, subprogram, or sequence of statements controlled by a loop statement

SYNTAX

```
BEGIN
 statement;
 statement
END.

  or

BEGIN
 statement;
 statement
END;
```

COMMENTS

The BEGIN statement, a predefined word, is always associated with a corresponding END statement. The two words delimit a program, subprogram, or block of statements.

SEE ALSO

END

FUNCTION

PROCEDURE

BOOLEAN

PURPOSE

A data type which returns either a TRUE or a FALSE value

SYNTAX

Identifier : BOOLEAN

COMMENTS

This predefined data type is declared in the Variable section of the program.

BOOLEAN variables are either TRUE or FALSE.

A BOOLEAN variable may represent an expression that uses the BOOLEAN operators: AND;'OR, and NOT. For example, if "Correct" is a BOOLEAN variable, the expression

```
   Correct:= (X>10) AND (X<20)
```

will be TRUE only if X is an integer between 11 and 19.

SEE ALSO

AND

NOT

OR

CASE

PURPOSE

Controls the selection of one of several possible statements depending upon the value assigned to the case selector

SYNTAX

CASE *case-selector* OF
 case-indicator : *statement*;
 case-indicator : *statement*
END;

COMMENTS

The CASE statement defines a case-selector which can be an expression. Depending upon the value the case-selector equals, the program executes the statements identified by the corresponding case-indicator. Multiple statements governed by a case-indicator are enclosed between BEGIN/END braces.

For example, the following statements ask the user to enter a selection from a menu. The program then executes the appropriate PROCEDURE which must be defined elsewhere.

```
Writeln('Enter selection');
Writeln('1. Read');
Writeln('2. Write');
Writeln('3. Quit');
Readln(Choice);
CASE Choice OF
  1 : Read;
  2 : Write;
  3 : Quit
END;
```

CASE (cont.)

This sequence of statements asks the user to select a value from a menu. Each item in the case list of values is associated with a specific PROCEDURE that has already been defined. If the user enters "1," for example, the program executes the PROCEDURE named "Read."

SEE ALSO OF

CHAR

PURPOSE a predefined ordinal data type that consists of single character values.

SYNTAX *Variable* : CHAR

COMMENTS Chars can be defined by their ordinal values (e.g., the ASCII value).

A Variable that is defined as a CHAR type may represent any ASCII symbol as text. This includes all letters, symbols, numbers included as text, and non-printing control codes such as carriage-return.

CHAR constants must be enclosed in apostrophes (or single quotes).

To print an apostrophe character in text, type a double apostrophe and enclose it in apostrophes.

The following examples illustrate valid character constants:

```
'A'
'z'
'5'
'>'
''''
```

CHAR (cont.)

Strings of CHAR are similarly enclosed in apostrophes. The following statement writes a string of characters to the screen:

```
Writeln('Hello');
```

When the ORD function is used with a CHAR data type, the result is ASCII value of the character. ORD('Q'), for example, returns the integer 81, which is the ASCII value of the character, Q.

SEE ALSO

ORD

STRING

TYPE

CHR

PURPOSE Converts an integer expression into the ASCII character represented by that value

SYNTAX CHR(*x*)

COMMENTS A predefined function.

The parameter x must be an Integer between 0 and 255.

The CHR function can be used to indicate non-printable ASCII values.

SEE ALSO ORD

CONCAT

PURPOSE When used with Strings, joins or concatenates the strings into a single string

SYNTAX Concat(*String1*, *String2,String3*);

COMMENTS Concat is not a predefined procedure. It must be included in the Pascal program using the include feature. Concat combines multiple strings into a single string element. A String must be defined as an ARRAY OF Char before this command is used.

The File **Concat** must be "included" in the declaration section of the Program, Procedure, or Function that uses the Concat statement. The constant Maxstring must also be defined in the declaration section. Maxstring determines the absolute length of the string.

If a program contains a String identified as **String1** which contains the characters 'Any ' and a String identified as **String2** which contains the characters 'Body ', the statement: *Concat(String1,String2,String3);* adds String2 to String1 and stores the resultant string in the variable **String3.** String3 equals the string: *'AnyBody '*

SEE ALSO Index
Length
String
Substring

CONST

PURPOSE

Identifies data as a constant value throughout the program

SYNTAX

```
CONST
 identifier = value
```

COMMENTS

Like TYPE, the word CONST, a predefined data type, identifies a list of items. The value can be any data type, even character or string. The data items listed under CONST retain their assigned value throughout the program.

Constants are listed immediately after the program declaration, and they are defined by using the equal sign (=) to equate the identifier with its value.

FUNCTIONs and PROCEDUREs may declare their own constants, but those values are significant only within the FUNCTION or PROCEDURE in which they are declared.

SEE ALSO

TYPE

VAR

COS

PURPOSE Returns the cosine of the expression

SYNTAX COS(*expression*)

COMMENTS A predefined function.

The expression can be an integer or a real number which is expressed in radians.

COS returns a value that is a real number.

SEE ALSO ARCTAN

SIN

DISPOSE

PURPOSE

Frees the memory space assigned to hold the pointer variable which had been reserved by the NEW statement.

SYNTAX

DISPOSE(*PointerVariable*)

COMMENTS

When pointer variables are created by the NEW statement, they remain in memory even after the locations they point to are no longer relevant to the program.

The DISPOSE statement, a predefined pointer procedure, destroys the pointer variable and frees the memory locations it occupied.

SEE ALSO

NEW

DIV

PURPOSE Returns the whole number that results from dividing integer value A by integer value B

SYNTAX A DIV B

COMMENTS A predefined operator.

If A = 12 and B = 4, A DIV B returns 3. If A = 14 and B = 4, A DIV B also returns 3. DIV returns only the dividend of a division. It does not return fractional parts or remainders. Division by zero will return an error.

SEE ALSO MOD

DO

PURPOSE

Indicates the beginning of a series of statements that are regulated by a FOR or WHILE statement.

SYNTAX

```
FOR control-variable TO control variable DO
BEGIN
 statement 1;
 statement 2;
 statement 3
END;
```

COMMENTS

A syntactical part of conditional statements.

DO indicates the beginning of one or more statements. The sequence must be terminated with an "END;" statement.

SEE ALSO

FOR

WHILE

DOWNTO

PURPOSE Regulates a FOR loop countdown from a higher to a lower number.

SYNTAX FOR *control-variable* DOWNTO *control-variable* DO

COMMENTS A syntactical part of a FOR statement.

The value of the first control-variable must be greater than the value of the second variable.

DOWNTO indicates the value that terminates the loop.

SEE ALSO FOR

ELSE

PURPOSE

Indicates an alternate set of instructions that the program executes when the condition controlling an IF statement is FALSE.

SYNTAX

```
IF control-expression THEN
 BEGIN
   statement ;
   statement
 END
 ELSE
  BEGIN
   statement,
   statement
  END;
```

COMMENTS

The line preceding an ELSE statement requires no punctuation.

If the ELSE statement executes only 1 command, the BEGIN/END braces are not required.

SEE ALSO

IF..THEN

END

PURPOSE

Terminates a series of statements associated with a BEGIN, CASE .. OF, or Record statement.

SYNTAX

```
BEGIN
 statement;
 statement
END.
```

COMMENTS

Every BEGIN statement must have a corresponding END statement. The END statement that terminates the program is punctuated with a period.

The END statement that terminates a PROCEDURE, a FUNCTION, or a sequence of actions which form a block within the program is punctuated with a semicolon(;).

The line preceding an END statement requires no punctuation.

The rule for punctuating a statement that precedes an END statement holds true even if that line is itself an END statement. For example:

```
BEGIN
  statement,
  statement,
  BEGIN
   statement;
   statement
  END
 END.
```

SEE ALSO

BEGIN

EOF

PURPOSE A BOOLEAN function that remains FALSE until the file pointer reaches the End Of File marker

SYNTAX EOF

COMMENTS The EOF standard Boolean function is used when reading files. Since the program usually does not know how many items are in the file, the EOF function allows the program to continue reading data until it reaches the end of the file.

The following statements continue to retrieve information stored in the file, StudentName, until the EOF marker is read.

```
WHILE NOT EOF(StudentName) DO
      BEGIN
        statements
      END;
```

SEE ALSO EOF

GET

READ

EOLN

PURPOSE

A predefined BOOLEAN function that remains FALSE until the <**RETURN**> key is pressed or a return ASCII character is read (e.g., from a disk file).

SYNTAX

EOLN

COMMENTS

The EOLN variable is used to test entry of data from the keyboard. It remains FALSE until the <**RETURN**> key is pressed or a return character is read.

The READLN statement resets the EOLN variable to FALSE.

SEE ALSO

EOF

READLN

EXP

PURPOSE Calculates the value of the base of the natural logarithm raised to the power indicated by the parameter.

SYNTAX EXP (X)

COMMENTS A predefined function.

EXP computes the exponential of the parameter X; that is, e^X, where X can be an integer or real data type. The EXP function returns a value of type Real.

For example, EXP(3) raises the value of the natural logarithm to the power of 3. The result is 20.085537.

SEE ALSO LN

FALSE

PURPOSE Indicates a BOOLEAN logical state

SYNTAX FALSE

COMMENTS FALSE is a predefined Boolean constant. BOOLEAN expressions, yield either a TRUE or a FALSE state. The identifier FALSE may be used in expressions that evaluate that state.

For example, if the variable *STATUS* is defined as a BOOLEAN variable, the following expression evaluates the variable and takes the appropriate action:

```
CASE Status OF
  TRUE : X := 10;
  FALSE : X := 20
END;
```

If the BOOLEAN variable Status is FALSE, the program sets the value of X to 20.

SEE ALSO BOOLEAN

TRUE

FILE

PURPOSE

Creates a user-defined TYPE that consists of a sequence of elements of the same TYPE. A FILE redirects input and output from the keyboard and screen to an external source, usually a disk.

SYNTAX

VAR
  identifier : FILE OF *component-type*;

COMMENTS

A Reserved word.

A FILE may consist of any component-type, either predetermined or user-defined. That is, a FILE may consist of integers, real numbers, pointers, records, or arrays. A FILE, however, cannot consist of any structured type that itself contains a FILE.

When a FILE is declared, the compiler creates a FILE buffer. The buffer is referred to by the FILE identifier followed by a caret(^). The buffer can hold only one data item of the declared component-type at a time.

The PUT statement writes the FILE-buffer contents to the external device.

The GET statement reads the contents of the external device into the FILE-buffer.

SEE ALSO

GET

PUT

FOR

PURPOSE

Creates a loop that executes a series of statements until the conditions defined in the FOR statement are satisfied.

SYNTAX

FOR *control-variable* := *low-value* TO *high-value* DO
 BEGIN
  statement;
  statement
 END;

COMMENTS

The FOR statement increments the control-variable by 1 each time it executes the commands it controls. This process continues until the control-value equals the high-value of the control loop.

The lower and upper limits of the loop are defined by using the := assignment indicator.

If the FOR loop only executes one action, the BEGIN/END braces are not required.

The FOR statement can also use the DOWNTO statement. In this case, the first value in the control-loop is the high value and the second is the low value.

SEE ALSO

TO

DOWNTO

FUNCTION

PURPOSE

Declares the statements associated with it as a single subprogram. It receives data from the main program, performs calculations using that data, and returns a single value which is identified by the name assigned to the FUNCTION.

SYNTAX

FUNCTION *identifier*(*parameter list* : *parameter-type*) : *resultant-type* ;

CONST
VAR

BEGIN
  statements
END;

COMMENTS

This predefined identifier names the FUNCTION. The main program refers to the FUNCTION by this name.

The parameter list contains identifiers for the values that the main program passes to the FUNCTION. The data type of each element passed must be included and is separated from its identifier by a colon (:) . Multiple parameters are separated from each other by a semicolon (;) . The entire parameter list is contained within parentheses.

The data type of the value returned by the FUNCTION is declared by placing the data type after the parameter list and separated from it by a colon (:) .

FUNCTION (cont.)

Any constants and variables used by the FUNCTION must be declared before the body of the FUNCTION. Such constants and variables are considered to be local to the FUNCTION.

The body of the FUNCTION must include one statement that assigns the value determined by the FUNCTION to the FUNCTION-identifier.

The main program simply names the FUNCTION, including the parameters to be passed to it, and the FUNCTION immediately returns the value which is then associated with the FUNCTION's name/identifier.

SEE ALSO PROCEDURE

"Programming Techniques"

GET

PURPOSE

a standard procedure that transfers one element of a file to the associated buffer variable.

SYNTAX

GET(*FileIdentifier*);

COMMENTS

GET retrieves a component from the FILE indicated by the File Identifier. It stores that data in the file buffer and then advances the file pointer to the next position.

If another element of the FILE exists, the End Of File Variable remains FALSE. If no further component is found, the EOF variable is set to TRUE.

This command is effective only after the FILE has been RESET and the first item stored in a FILE buffer.

The following statements RESET an existing file and read its contents, which are a series of REAL numbers, printing them to the screen.

```
VAR
 List : FILE OF Integer;
 Number : REAL;

BEGIN
 RESET(List, 'Pathname');
 Number := List^;
 GET(List);
 WRITELN(List)
END.
```

GET (cont.)

This sample program will continue to read the FILE until it reaches the End Of File marker.

For a full discussion of commands used in file manipulation, see the chapter on FILEs in "Programming Techniques."

SEE ALSO

PUT

WRITE

WRITELN

GOTO

PURPOSE

Forces the program to make an unconditional branch to another statement. The statement that is branched to must be labeled by an unsigned integer.

SYNTAX

GOTO *label*

COMMENTS

A nonstandard Pascal statement.

Labels must be declared under a LABEL heading at the beginning of the program. The maximum size of the label is 4 digits.

The label that identifies the line must begin in the 1st or 2nd column of the program line. It is followed by a colon (:) and then the program line.

Labels that are used in PROCEDUREs and FUNCTIONs must be declared locally.

GOTO may make either forward or backward jumps to labels within a FUNCTION or PROCEDURE. GOTO may also be used to make an unconditional jump out of a FUNCTION or PROCEDURE. GOTO should not be used to enter a FUNCTION or PROCEDURE from the main program.

SEE ALSO

LABEL

HGR

PURPOSE Puts that the computer into the High Resolution graphics mode.

SYNTAX HGr;

COMMENTS HGr is not a predefined Pascal procedure. The routine must be included in the Pascal program using the file Hires.I. The high resolution graphics procedures enable the programmer to draw graphic illustrations on the screen.

To use high resolution graphics, the Pascal program must be relocated by a

```
#a
        ORG    $4000
#
```

declaration before the declaration of the main program.

The HGr procedure is contained in the file **HIRES.I** which must be "included" in the program declaration.

The command **Tx** exits the High Resolution Graphics mode.

SEE ALSO Tx

Section III, "Graphics"

IF

PURPOSE

Tests a conditional expression and performs specific actions if the condition is TRUE (an optional ELSE clause performs specific actions if the condition is FALSE)

SYNTAX

```
IF condition THEN
  statement
 ELSE
  statement;
```

COMMENTS

A predefined conditional statement.

IF the condition is TRUE, the statement following the word, THEN, is executed. If the condition is false, the statement following ELSE is executed.

If the THEN or the ELSE clause should execute several statements, they must be enclosed in BEGIN/END braces. Statements within the BEGIN/END brace are punctuated with semicolons.

Note that no punctuation is allowed between THEN and the statement it executes, or between the THEN clause and the ELSE clause.

IF (cont.)

Also, all rules governing the use of END apply if BEGIN/END braces are used. Note the punctuation of the following sample IF diagram:

```
IF condition THEN
 BEGIN
        statement;
        statement
 END
 ELSE
 BEGIN
        statement;
        statement
 END;
```

The ELSE clause always modifies the nearest IF-THEN statement. Consequently, if you nest IF-THEN-ELSE statements, you may inadvertently associate an ELSE clause with an inappropriate IF-THEN statement.

SEE ALSO ELSE

END

THEN

IN

PURPOSE Determines if an element is part of a set

SYNTAX *Condition variable* IN *set-name*

COMMENTS IN, a set operator, determines if the variable is included in the set. A conditional statement usually controls the statements that will be executed depending upon the results of the IN test.

For example, if Weekdays is a set of days from **Mon** to **Fri**, and the user enters the string, **Sun**, the following statements will execute the statement controlled by the ELSE statement.

```
Writeln('Enter a day');
Readln(Day);
IF Day IN Weekday THEN
  Writeln(Day, 'is a weekday')
ELSE
  Writeln(Day, 'is not a weekday')
```

SEE ALSO SET

INDEX

PURPOSE Returns the position of one String within another

SYNTAX Index(*String1,String2*);

COMMENT Index is a not predefined Pascal procedure. It must be included in the Pascal program using the **Index.I** file.

A data type of String must be declared in the program before this function can be used. The constant Maxstring must also declare the absolute length of any string entered.

The File **Index** must be "included" in the declaration section of the Program or routine that uses the Index function. Files are "included" by using the #i command.

The function returns the position of String2 within String1. For example, if String1 = 'Baby' and String2 = 'a   ', the statement: *Index(String1,String2);* returns the value 2 to the Function identifier **Index**. This is the location where the first string contains the character "a". If String2 is not found in String1, the value of **Index** equals 0.

SEE ALSO Concat, Length, String, Substring

INPUT

PURPOSE Indicates information will be retrieved from the default file, which is the keyboard

SYNTAX PROGRAM Identifier(INPUT, Output);

COMMENTS INPUT, a predefined file type, identifies a TEXT file that takes information in the form of ASCII values from the keyboard. The information is stored in memory until it is redirected to another device.

SEE ALSO FILE

OUTPUT

INTEGER

PURPOSE Denotes positive or negative whole numbers

SYNTAX X : INTEGER

COMMENTS Integer, a predefined data type, may identify any variable. X is identified as an INTEGER data type. No commas or decimal points may be used to indicate an INTEGER.

The maximum size of an INTEGER is known by the predefined CONSTANT, MAXINT.

Kyan Pascal supports integers from -32768 to +32767.

SEE ALSO MAXINT

LABEL

PURPOSE

Identifies a decimal integer which is used with the GOTO statement

SYNTAX

```
LABEL
 X, Y;
```

COMMENTS

A reserved Identifier.

A LABEL must be declared at the beginning of the program. The values X and Y may be any number containing up to 4 digits.

The number identified as a LABEL is placed at the beginning of the line that includes the statement to be labeled. The LABEL is followed by a colon (:) .

For example, if 33 is declared as a label, a GOTO 33 statement would execute the following statement:

```
33: Writeln('Jump here.');
```

SEE ALSO

GOTO

LENGTH

PURPOSE	Returns the actual length of a String.
SYNTAX	Length(*String*);
COMMENTS	Length is not a predefined Pascal procedure. The String must be declared as an ARRAY of CHAR, and the file **Length.I** must be "included" in the declaration of the Program or routine that uses the Length command. Files are included by using the **#i** command. The value of **Maxstring**, the maximum length of the string must also be declared. If String1 = 'letters ', the statement *Length(String1);* returns the value 7. Use the Length statement to eliminate trailing spaces in writing strings to output. The following statement writes only the actual characters contained in the string to the screen: *Writeln(String1: Length(String1));*

SEE ALSO

Concat	Index
String	Substring

LN

PURPOSE Calculates the natural logarithm of the parameter

SYNTAX LN(X)

COMMENTS A predefined function.

The value of X must be greater than 0.

X may be either an integer or a real number.

The LN function returns a real number.

SEE ALSO EXP

MAXINT

PURPOSE Equals the largest integer the compiler can accommodate

SYNTAX MAXINT

COMMENTS A predefined constant.

In this implementation of KYAN PASCAL, MAXINT = 32767.

MOD

PURPOSE Returns the value of the Remainder that results from the division of one integer by another.

SYNTAX A MOD B

COMMENTS A reserved arithmetic operator.

If A = 12 and B = 4, A MOD B equals 0. If A = 14 and B = 4, A MOD B equals 2.
B cannot be equal to 0.

SEE ALSO DIV

NEW

PURPOSE Allocates space in memory for a dynamic variable which is referred to by a pointer variable.

SYNTAX NEW (*PointerVariable*)

COMMENTS NEW, a predefined Pointer Procedure, is used in dynamic allocation routines which allow the programmer to have direct control over memory locations.

Dynamic allocation routines require that space in memory be set aside to hold an address of another area in memory. The first area is called a **Pointer** because it points to the other address.

Since you cannot anticipate where the computer will store the Pointer, you must identify the area by a variable. You must also make sure that the area contains no residue from other programs. NEW insures that whatever area used to store the pointer is initialized before a value is stored in it.

For example, if you declare a pointer variable as

```
VAR
  Pointer : ^INTEGER
```

you must clear the area the will contain the value stored in **POINTER.** The body of the program must issue a

```
    NEW(Pointer);
```

command.

NEW (cont.)

After execution of **NEW(Pointer)**, the pointer contains the address of the newly-created dynamic variable.

SEE ALSO

DISPOSE

"Kyan Pascal Programming"

NIL

PURPOSE Indicates the last element in a linked list of dynamic variables.

SYNTAX Pointer^.Lint := NIL;

COMMENTS NIL is a predeclared identifier used in linked records to indicate which is the last record in the list.

REMEMBER: records are linked backwards -- the first entered is the last retrieved.

Consequently, declare the pointer for the first record as NIL. When you want to read the list of records use

WHILE *RecordIdentifier* <> NIL DO

Then WRITELN the record buffer, update the pointer, and retrieve the next record.

The program reads through the list of records until it finds a pointer equal to NIL.

SEE ALSO DISPOSE

NEW

NOT

PURPOSE Returns the opposite value.

SYNTAX NOT *expression*

COMMENTS A Boolean operator.

If A is an expression that is TRUE, NOT A returns FALSE.

If A is an expression that is FALSE, NOT A returns TRUE.

SEE ALSO AND

BOOLEAN

OR

ODD

PURPOSE Tests an expression to determine if it equals and odd integer

SYNTAX ODD(*expression*)

COMMENTS The ODD function is a predefined Boolean function. It returns TRUE if the expression has an odd value and FALSE if it has an even value.

The expression must either be or yield an integer value.

SEE ALSO BOOLEAN

OF

PURPOSE

When used with the CASE statement, selects an action from a list of statements. The action selected depends upon the CASE-selector variable.

SYNTAX

CASE *variable* OF
variable 1: *statement 1*;
variable 2: *statement 2;*
. . .
variable n: *statement n*
END;

COMMENTS

CASE ... OF Statements list a series of possible actions. The action selected by the program depends upon the variable.

Each possible variable is listed along with the statement of the action to be taken. The variable and the statement are separated by a colon (:) .

OF is also used in declaring arrays and sets.

SEE ALSO

CASE

OR

PURPOSE A Boolean operator that returns TRUE if either of the expressions associated by OR is TRUE.

SYNTAX A OR B

COMMENTS The expression A OR B returns the Boolean value TRUE if either the A expression or the B expression is TRUE. It returns a FALSE value only if both A and B are FALSE.

SEE ALSO AND

BOOLEAN

NOT

ORD

PURPOSE

Returns the numerical position of a parameter X as an integer in the ordered sequence of values.

SYNTAX

ORD(*expression*)

COMMENTS

A predefined Function. The parameter must be of an ordinal type.

If a set, MONTHS, is declared that contains the months of the year, the expression

```
ORD(JAN)
```

returns the value 0. Positioning in a set always begins with the zero position.

Since integers are a subset, the ORD value of an integer is the integer itself.

ORD can be used to return the ASCII value of a character.

SEE ALSO

PRED

SUCC

OUTPUT

PURPOSE Indicates data will be printed to the default file, i.e. the screen

SYNTAX PROGRAM Identifier(Input, OUTPUT);

COMMENTS OUTPUT is a TEXT file that takes data stored in memory and directs it to the output device.

SEE ALSO FILE

INPUT

TEXT

PACK

PURPOSE

Commonly used in Pascal to copy components of an unpacked array into a packed array

SYNTAX

PACK(*UnpackedArrayIdentifier, Index, PackedArrayIdentifier*)

COMMENTS

Large computers often reserve a group of memory locations for each element in an Array. The PACK command is used to compress the data and free any unused memory space.

The PACK procedure requires the identifier of the unpacked array, the starting index value of the section of the array where packing should begin, and the identifier of the PACKED ARRAY.

Kyan Pascal automatically PACKs all arrays. This command is useful for copying components of arrays.

SEE ALSO

PACKED

UNPACK

PACKED

PURPOSE Indicates that an ARRAY consists of packed elements

SYNTAX PACKED ARRAY [1 . . 6] OF Char;

COMMENTS Mainframe computers often allocate a number of memory locations for each element of an ARRAY. A character ARRAY, however, needs only one position for each character. PACKED forces each element of the character ARRAY into 1 byte so that the extra memory space is not wasted.Kyan Pascal automatically packs ARRAYs so this command is not needed by the programmer.

SEE ALSO UNPACK

PAGE

PURPOSE Skips from the current page to the next page of a TEXT file.

SYNTAX PAGE(FileIdentifier)

COMMENTS The PAGE procedure causes the system to clear the file buffer by executing a WRITELN statement. It then advances the output to a new page of the specified text file or clears the screen and moves the cursor to home.

This procedure can be used only on TEXT files.

SEE ALSO TEXT

POINTER

PURPOSE

Allows the programmer to write to or read from specific memory locations.

SYNTAX

PointerVariable := Pointer(MemLocation);

COMMENTS

POINTER, a non-standard command, allows the programmer to write data directly into a specific memory location. It can also be used to read the contents of memory locations.

POINTER can be used to PEEK and POKE memory locations.

For example, if you declare a pointer variable, assign a location in memory to the variable and then write the dynamic variable. The result is equivalent to a BASIC PEEK command. The value stored in that memory location will be written to the screen.

The reverse process is used to POKE data into a specific memory location. First declare the pointer variable and assign it a specific location in memory. Then equate the area referred to by the pointer with a value.

SEE ALSO

ADDRESS

PRED

PURPOSE

Returns the data item that precedes the parameter

SYNTAX

PRED(*parameter*)

COMMENTS

A predefined Function.

The parameter must be part of an ordered sequence of items. It cannot be the first item in the sequence.

If a set is defined as {sun, mon, tues}, PRED(mon) returns sun. There can be no PRED(sun).

SEE ALSO

ORD

SUCC

PROCEDURE

PURPOSE

Declares that the statements associated with the name of the PROCEDURE are a single subprogram. It may receive data from the main program in the form of parameters. These parameters may or may not be transmitted back to the main program.

SYNTAX

PROCEDURE Identifier(*Parameter list*: *Parameter Type*);

CONST
Constant Identifiers;

VAR
Variable Identifiers;

BEGIN
statements
END;

COMMENTS

A reserved word.

The Identifier names the PROCEDURE. The main program refers to this PROCEDURE by the name. The PROCEDURE declaration may contain a list of parameters, or values that the main program passes to the PROCEDURE.

The main program may pass any type of data to the PROCEDURE with the exception of FILE data types, but all parameters must be declared in the parameter list. The list contains the identifiers of the values to be passed as well as the data type of each parameter.

PROCEDURE (cont.)

If a parameter is declared to be a Variable, the variable identifier will represent the value that results from the action of the PROCEDURE when control returns to the main body of the program.

This is an important difference between FUNCTIONs and PROCEDUREs.

A FUNCTION returns one value to the main program, and that value is represented by the FUNCTION's identifier or name.

A PROCEDURE may affect the value of any Variable identifiers that are part of its parameter list.

Identifiers declared within the PROCEDURE are local to that PROCEDURE and are not accessible to other parts of the program.

The body of the PROCEDURE is marked by a BEGIN and END bracket. The END statement is terminated with a semicolon to differentiate it from the END of the program.

The main program calls the PROCEDURE by declaring its name and including the parameters to be passed to it within parentheses. The list of parameters must be identical to the parameter list declared in the PROCEDURE.

SEE ALSO FUNCTION

"Programming Techniques"

PROGRAM

PURPOSE Identifies the entire program to the compiler

SYNTAX PROGRAM Identifier(Input,Output);

COMMENTS A PASCAL program must first be identified as a PROGRAM and given a name that identifies it. The Input and Output statements are optional, but they indicate that input will come from the keyboard and that output will be directed to the screen.

If any other device files are used, their identifiers must be included in the Input, Output list.

The structure of a program is outlined below:

1. PROGRAM declaration
2. Label declaration
3. Constants
4. Variables
5. Subprograms: Functions and Procedures
6. Main BODY of the program

PUT

PURPOSE Writes the contents of a file buffer to a FILE.

SYNTAX PUT(*file-identifier*);

COMMENTS PUT, a predefined file procedure, writes the contents of a file buffer, identified by the FileName followed by the Caret sign, e. g. **FileName^**, to the file associated with the File Identifier.

Before executing the first PUT statement, however, the file must be opened for writing by a REWRITE command and the data must be stored in the FILE buffer variable -- which is usually the File Name followed by a caret sign ^. The following statements declare a FILE and input data to that FILE.

```
VAR
  List : FILE OF Integer;
  Number : Real;

BEGIN
  REWRITE(List,'Pathname');
  WRITELN('Enter a Number.');
  READLN(Number);
  List^ := Number;
  PUT(List)
END.
```

Note that the *Pathname* is the identifier that indicates the pathname of the file that will hold the data on the storage device.

SEE ALSO GET

READ

PURPOSE

Retrieves data from a FILE without appending an End Of Line marker to the data retrieved

SYNTAX

READ(*FileIdentifier, variable- identifiers*)

COMMENTS

The READ procedure gets data from an input file and stores the data in a variable.

If the File Identifier is omitted, the default is the INPUT file (i.e., the keyboard).

Multiple variable identifiers may be included, but they must be separated by commas.

The READ procedure retrieves data from the current file until has found a value for each variable in the variable list.

When READ is used with a TEXT file, it performs an assignment statement, and executes a GET command. For example, *READ(Text1,Identifier);* equals

```
        Identifier := Text1^;
        GET(Text1);
```

SEE ALSO

READLN
WRITE
WRITELN

READLN

PURPOSE Reads a line of data from a text file

SYNTAX READLN(*FileIdentifier,*
VariableIdentifier)

COMMENTS READLN, a predefined file procedure, reads a line of data; that is, it gets the contents of the current line of a TEXT file until it reaches an End Of Line marker.

If the File Identifier is not specified, READLN assumes the INPUT file, which is the keyboard.

The Variable Identifier indicates the name of a place in memory that holds the line until the program does something with it.

The following statement retrieves the entire line entered by the user before he presses the **<RETURN>** key.

```
READLN(Entry);
```

It stores the data entered in a variable named **Entry.**

READLN (cont.)

The following statements read a line from a TEXT file named **Manuscript** and then print that line on the screen.

```
READLN(Manuscript,Sentence);
WRITELN(Sentence);
```

The READLN statement reads the values of all listed variables that occur within the line. If necessary, it will read all the lines it needs to assign values to the variables contained in the current line.

SEE ALSO READ

WRITE

WRITELN

REAL

PURPOSE Indicates decimal or fractional numbers

SYNTAX Variable : REAL

COMMENTS Real, a predefined data type indicates that a value contains a fractional or decimal component. They may be expressed in decimal or exponential notation. The following are valid expressions of REAL number:

```
5.78                1.0
999.87654           4.1E2
```

Exponential notation is used with numbers that are too big or too small to write conveniently in decimal notation. The expression "4.1E2" in the above example represents 4.1 multiplied by 10 raised to the power of 2, i.e. 4.1 x 100, or 410. The number after E indicates the number of decimal places that follow the number that precedes E.

Positive or negative values after E determine how the decimal point is moved.

```
2.1E2  = 210
2.1E-2 = .021
```

Negative values that precede the E can yield strange results since such values are actually expressions consisting of a negative operator and a real number.

SEE ALSO INTEGER

RECORD

PURPOSE

A structured data type defined by the user that consists of fields that may themselves be different data types

SYNTAX

```
TYPE
 record-type-identifier = RECORD
  field-name  : data-type;
  field-name  : data-type
 END;
```

COMMENTS

A RECORD is defined under the TYPE label because it is a user-defined TYPE of data. The record-type identifier is followed by an equal sign and the declaration, RECORD.

The fields of the RECORD are then named. The name is followed by a colon and the data type of the field. A field may itself be any data type, including other records.

To refer to a field in a RECORD, specify the name of the record and the name of the field, separating them with a period. Names of fields must be unique within each record, but other RECORD types may uses the same field name.

For example, The RECORD, Students, contains the fields, Name, ID, Grades. Name is a STRING data type, ID is also a STRING, Grades is an ARRAY of real numbers. The expression *Students.Name*

RECORD (cont.)

returns the STRING of characters that represent the name of the student. Similarly, Student.Grades, returns all of the grades contained in the ARRAY. A RECORD, Classes, may also contain a field called Name.

A RECORD may contain a RECORD in one of its fields. The RECORD that is a field in another RECORD will itself contain fields. To access the fields of a RECORD within a RECORD, use a 3 part identifier:

```
RECORD.FIELD(itself a record).FIELD(in
second record).
```

The WITH statement allows the program to identify a RECORD. Subsequent references to the fields within the RECORD may be made directly. The following statements print the student's name in the RECORD, Students.

```
WITH Students DO
  Writeln(Name);
```

RECORDS may also contain Variant fields. See the section on Records in "Programming Techniques."

SEE ALSO TYPE

WITH

"Programming Techniques"

REPEAT

PURPOSE

Executes one or more commands until a condition becomes true

SYNTAX

REPEAT
  statement;
  statement;
UNTIL *conditional-expression*;

COMMENTS

REPEAT, a predefined instruction, tests for a TRUE condition only after it has already executed the sequence of commands. It continues to execute and then test the condition until the condition is FALSE. Consequently, the statements contained in the loop are always executed at least once.

The REPEAT statement is terminated by UNTIL and the conditional expression. Any number of statements, therefore, can be included within the loop without framing them in BEGIN/END braces.

SEE ALSO

UNTIL

RESET

PURPOSE Opens a FILE for reading data and sets the file marker to the beginning of the FILE

SYNTAX RESET(*file-identifier*);

COMMENTS If the FILE is not already opened for reading, RESET automatically OPENS the file and places the file marker at the beginning of the FILE. If the FILE is not empty, the value EOF, the End Of File marker, is FALSE. Otherwise, the value of EOF is TRUE.

Before the GET statement can be used to read the data in the FILE, the first element of the file must be entered into the file buffer, which is designated by **FileName^**.

Always use the RESET statement before trying to read a FILE. The following statements RESET a pre-existing file for reading and print its contents to the screen.

```
VAR
  List : FILE OF Integer;

BEGIN
  RESET(List);
  WHILE NOT EOF(List) DO
   BEGIN
     Number := List^;
     WRITELN(Number);
     GET(List)
   END
END.
```

RESET (cont.)

Note that the NOT EOF condition keeps checking the file until the End Of File marker is found.

SEE ALSO EOF

GET

REWRITE

PURPOSE Opens a FILE for writing data and sets the file marker to the beginning of the file

SYNTAX REWRITE(*file-identifier,'PATHNAME'*);

COMMENTS The REWRITE procedure readies a file for input. It sets the file marker to zero and the End Of File marker to TRUE. Any residual data in the file is cleared.

In Kyan Pascal, the File Identifier must be equated with a Pathname which locates the file on the storage device. The name of the file on the disk need not be the same as the file identifier used by the program.

SEE ALSO RESET

ROUND

PURPOSE Returns the closest Integer value of a Real number

SYNTAX ROUND(number)

COMMENTS The ROUND predefined function always returns an integer:

ROUND(5.3) returns the value 5.
ROUND(5.6) returns the value 6.

SEE ALSO MAXINT

TRUNC

SET

PURPOSE Identifies a user-defined data TYPE consisting of elements all of which must be the same data type

SYNTAX SetIdentifier = SET OF base type

COMMENTS A set is merely a collection of items that are all the same type of data. The base type identifies the kind of elements that may be included in the set.

Define a SET by listing its elements in brackets. If a SET is defined as a SET OF INTEGERS, it may contain up to 256 elements. A set can not contain REAL numbers. Consecutive items in a SET may be indicated by the subrange symbol (..).

All of the following are valid sets:

```
[Sun,Mon,Tues,Wed,Thurs,Fri,Sat]
[1,5,8,9,15]
[32,65,70..75]
```

The last set above contains the Integers 32, 65, 70, 71, 72, 73, 74, and 75.

The SET operators, =, <>, <=, =>, and IN are used to manipulate sets.

SEE ALSO IN TYPE

"Programming Techniques"

SIN

PURPOSE Returns the sine of the expression

SYNTAX SIN(expression)

COMMENTS A predefined Function.

The expression can be either an integer or a real number which is in radians.

SIN returns a real number.

SEE ALSO ARCTAN

COS

SQR

PURPOSE Returns the square of the expression

SYNTAX SQR(*expression*)

COMMENTS A predefined function.

The expression can be any arithmetic type.

SQR returns a value that is the same data type as the expression.

SEE ALSO SQRT

SQRT

PURPOSE Returns the square root of the expression

SYNTAX SQRT(*expression*)

COMMENTS A predefined function.

The expression can be either an integer or a real number.

SQRT returns a value that is a real number.

If the value of the expression is less than 0, an error results.

SEE ALSO SQR

STRING

PURPOSE

Indicates a user-defined data type that consists of an ARRAY OF Characters

SYNTAX

String = ARRAY[1..Maxstring} OF Char

COMMENTS

String is not a predefined Pascal procedure. You must declare it to be an ARRAY OF CHAR. Once a String has been declared, you must define **Maxstring** as the absolute length of the String that can be entered.

To use any of the String Functions (Length, Index, or Substring) or the String Procedure (Concat), you must "include" the appropriate File in the declaration section by using the **#i** command and the name of the Function.

SEE ALSO

Concat

Index

Length

Substring

SUBSTRING

PURPOSE

Extracts part of a String from that String

SYNTAX

Substring(*String1 String2, Begin Position, length of String extracted*)

COMMENTS

Substring is not a predefined procedure. The String must be declared as an ARRAY OF Char. The File **Substring.I** must be "included" using the **#i** command.

The function uses 4 parameters. The first parameter is the Source String Identifier (String1). The second parameter is the destination string variable (String2). The third parameter is the first position of the String to be isolated. The fourth parameter is the length of the string to be extracted from the first string.

For example, if String1 equals 'Extraction', the statement

```
Substring(String1, String2, 1, 7)
```

yields String2 equal to **Extract.**

SEE ALSO

Concat	Length
Index	String

SUCC

PURPOSE Retums the next item in a list of items

SYNTAX SUCC(*parameter*)

COMMENTS The parameter must be part of a list of items. SUCC, a predefined function, returns the succeeding value.

The parameter can not be the last item in the set. In the set{sun, mon, tues}, SUCC(mon) yields tues. SUCC(tues) is an invalid expression.

SEE ALSO PRED

ORD

TEXT

PURPOSE

Identifies a predefined FILE TYPE that consists of characters.

SYNTAX

VAR
 identifier : TEXT

COMMENTS

TEXT creates a File of characters. It is different from a user-defined file of characters in that it is divided into lines. Each line in the TEXT file is a series of characters that is terminated by an End Of Line (EOL) marker.

INPUT and OUTPUT are TEXT files because they simply transmit text between the keyboard, the monitor, and any external devices.

TEXT Files are written by using the statement

```
Write(text-file-name, identifier)
```

TEXT Files are read by using the similar statement

```
Read(text-file-name, identifier)
```

The identifier is the variable name that holds the characters while they are being written or read.

SEE ALSO

TYPE

READ

WRITE

THEN

PURPOSE A reserved word that governs the execution of statements when an IF conditional test returns TRUE

SYNTAX IF *condition* THEN
    statement;

COMMENTS For a complete discussion of THEN as part of an IF statement, see IF

SEE ALSO IF

TO

PURPOSE A reserved word that indicates the upper boundary of a FOR loop

SYNTAX FOR *variable-identifier* := *X* TO *Y* DO
statement

COMMENTS X is the low value and Y the high value.

SEE ALSO DOWNTO

FOR

TRUE

PURPOSE Indicates a BOOLEAN logical state

SYNTAX TRUE

COMMENTS BOOLEAN expressions yield either a TRUE or a FALSE state. The identifier TRUE may be used in expressions that evaluate that state.

For example, if the variable STATUS is defined as a BOOLEAN variable, the following expression evaluates the variable and takes the appropriate action:

```
CASE Status OF
  TRUE : X := 10;
  FALSE : X := 20
END;
```

If the BOOLEAN variable Status is TRUE, the program sets the value of X to 10.

SEE ALSO BOOLEAN

FALSE

TRUNC

PURPOSE Converts a value to an Integer by truncating any fractional part of the value

SYNTAX TRUNC(*x*)

COMMENTS TRUNC, a predefined function, drops any decimal part of a value.

TRUNC(5.2) returns the value 5.

TRUNC(5.9) also returns the value 5.

SEE ALSO MAXINT

ROUND

TYPE

PURPOSE

Indicates a user-defined data element

SYNTAX

```
TYPE
  type-identifier = ARRAY
    [lower-limit..upper
    limit] OF data-type;
  type-identifier = RECORD
```

COMMENTS

TYPE indicates that you are defining a data element that is not one of PASCAL's predefined data types: CHAR, INTEGER, REAL, or BOOLEAN.

Data types that the programmer defines are called structured data types since they are organized to fit each particular situation.

A user-defined TYPE must be declared after the list of constants but before the list of variables. This is because the TYPE may include constants, but it must then be identified by a Variable identifier.

SEE ALSO

ARRAY

RECORD

VARIABLE

FILE

TX

PURPOSE Cancels the high resolution graphics mode

SYNTAX TX;

COMMENTS A nonstandard Pascal procedure. The program must first have entered the high resolution graphics mode by a HGr statement.

The procedure HIRES.I must also be "included" in the program using the #i command.

SEE ALSO HGR

UNPACK

PURPOSE Copies components of a PACKED array variable into an UNPACKed array variable

SYNTAX UNPACK(*PackedArray,Index, UnpackedArray)*

COMMENTS Kyan Pascal automatically packs and unpacks arrays as needed by the program. Consequently, this command is usually not needed.

To unpack a packed array, indicate the Identifier of the PACKED array, the index value of the Packed array where the unpacking begins, and the identifier of the Unpacked array.

SEE ALSO PACK

PACKED

UNTIL

PURPOSE

Indicates the condition which ends a REPEAT loop

SYNTAX

REPEAT
  statement;
UNTIL *conditional-expression;*

COMMENTS

The REPEAT loop continues to execute until the condition defined by the predefined word UNTIL becomes TRUE.

Note the difference between the REPEAT .. UNTIL loop, which tests for the condition at the end of the first and subsequent executions, and the WHILE .. DO loop, which performs the conditional test before it executes the loop.

SEE ALSO

REPEAT

WHILE

VAR

PURPOSE	Indicates variables used in the program
SYNTAX	VAR *identifier* : *data-type*;
COMMENTS	A Variable declaration identifies a name that will represent a value. When declaring the variable, you must indicate the type of data it will represent. The Variable declaration is punctuated with a semicolon(;). A Variable may represent any type of data. When using a Variable in a program, refer to it by its name-identifier.
SEE ALSO	CONST TYPE

WHILE

PURPOSE

A predefined statement that executes one or more commands as long as the control conditions remain TRUE

SYNTAX

```
WHILE conditional-expression DO
  BEGIN
    statement;
    statement
  END;
```

COMMENTS

The WHILE loop evaluates the conditional expression and executes the statement following the word DO only if the conditional expression is TRUE. If the conditional expression is FALSE, the statement is skipped.

Ordinarily, the WHILE statement executes only the statement following the word DO. If more than one action should be taken, the statements must be enclosed in BEGIN/END braces.

Unlike the REPEAT statement, the WHILE statement tests the conditional expression before it executes the commands controlled by the loop.

SEE ALSO

DO

REPEAT

WITH

PURPOSE Accesses fields within a record

SYNTAX WITH (*record-identifier*) DO
statement referring to a field.

COMMENTS The record identifier is the name of the record variable. The WITH statement declares the record that will be used.

Statements that follow the word DO can refer directly to fields within the record without the usual Record.Field syntax. These statements continue to refer to fields within the current record until the WITH statement is ENDed.

Nested records, i.e. records that contain fields which are themselves records, can be accessed by identifying more than one record variable in the WITH .. DO statement. If the Record, Student, contains a field, Scores, which is itself a record containing the field, SAT, the following WITH statement allows direct access to SAT:

```
     WITH Student, Scores DO
     Writeln(SAT)
    END;
```

The record identifiers must be listed in the order in which they are nested, from the largest nest to the smallest.

If DO should control more than one statement, the statements must be enclosed in BEGIN/END braces.

SEE ALSO DO

WRITE

PURPOSE

Directs data to a storage or display device

SYNTAX

WRITE(*file-identifier*,
  data-identifier);

COMMENTS

The WRITE statement directs data to a device indicated by the file identifier. The file identifier is optional. If it is omitted, WRITE directs the data to the screen by default. If an external device is indicated by the file identifier, data is directed to that device. For a detailed discussion of writing to a TEXT file, see the chapter on FILES in "Programming Techniques."

The data identifier may also be a literal series of characters. If this is so, simply enclose the items in single quotes or apostrophes. For example, WRITE('Hello') prints Hello on the monitor.

The WRITE statement does not append an End Of Line (EOLN) marker to the data. This means that if you WRITE data to the screen, the cursor remains at the position immediately following the data printed on the monitor.

The data identifier is a Variable that holds the data which is being redirected.

WRITE (cont.)

The following examples illustrate different uses of the WRITE statement:

```
WRITE('Hello');
```

prints "Hello" on the screen

```
WRITE(NAME);
```

prints the characters represented by the Variable "Name" on the screen

```
WRITE(DiskFile, Sentence);
```

writes the characters contained in the Variable "Sentence" to the disk file "DiskFile."

REMEMBER: The WRITE statement does not append an EOL marker.

SEE ALSO

EOLN

WRITELN

FILE

WRITELN

PURPOSE Directs the output of data to a monitor or external device and appends the End Of Line marker

SYNTAX WRITELN(*FileName,item*);

COMMENTS The WRITELN statement writes data to a TEXT file. If no file is specified, the default is OUTPUT. The item written may be a literal string or any data represented by an identifier.

WRITELN inserts an End Of Line marker at the end of the line that is written and advances the file marker to the next position in the file.

The following examples illustrate some of the uses of WRITELN.

WRITELN('Enter a value:');
Writes the literal string **Enter a value:** to the screen and advances the cursor to the next line.

WRITELN(Total);
Writes the value represented by the variable **Total** to the screen and advances the cursor to the next line.

WRITELN(Chapter1, 'Introduction');
Writes the literal string **Introduction** to a text file named **Chapter1** and advances the file pointer to the next line

SEE ALSO

WRITE	READLN
READ	FILE

APPENDIX A

GUIDE TO ISO STANDARD PASCAL

DATA TYPES

STRUCTURED:	Array, File, Set, Record
POINTERS	
SIMPLE:	Real Ordinal ..Enumerated ..Predefined (Boolean, Integer, Char) ..Subrange

STANDARD IDENTIFIERS

CONSTANTS:	False, MaxInt, True
TYPES:	Boolean, Char, Integer, Real, Text
VARIABLES:	Input, Output
FUNCTIONS:	Abs, ArcTan, Chr, Cos, Eof, Eoln, Exp, Ln, Odd, Ord, Pred, Round, Sin, Sqr, Sqrt, Succ, Trunc
PROCEDURES:	Dispose, Get, New, Pack, Page, Put, Read, Readln, Reset, Rewrite, Unpack, Write, Writeln

TABLE OF SYMBOLS

SPECIAL SYMBOLS

```
+     -     *     /     =
<     >     <=    >=    <>
.     ,     :     ;     :=
(     )     [     ]     |
..    (*    *)    {     }
```

WORD SYMBOLS (RESERVED WORDS)

and
array
begin
case
const
div
do
downto
else
end
file
for
function
goto
if
in
label
mod
nil
not
of
or
packed
procedure
program
record
repeat
set
then
to
type
until
var
while
with

DIRECTIVE forward

APPENDIX B

TECHNICAL SPECIFICATIONS

RUNTIME MEMORY MAP

_SystemFiles are loaded at $2000 and relocated depending on the setting of the HiRES graphics relocation utility, "_UsesHires" (on or off). If the utility is active, the BIN image of the object program is loaded at location $4000 allowing the heap to occupy from $800 to $2000. Locations $2000 - $3FFF are thereby left clear for the HiRES routines.

Runtime Memory Map with "_UsesHires" **ACTIVE**

0 — $ 7FF	Apple system overhead
$ 800 — $1FFF	System variable space
$2000 — $3FFF	HiRES Graphics (page 1)
$4000 — $8FFF	Program
$9000 — $BEFF	_LIB (Kyan Runtime Library)
$BF00 — $BFFF	ProDOS primary access page
$C000 — $CFFF	Soft switches/peripheral ROM space
$D000 — $F7FF	Applesoft BASIC
$F800 — $FFFF	System Monitor

Runtime Memory Map with " _UsesHires" **INACTIVE**

0	-	$ 7FF	Apple system overhead
$ 800	-	_LoMem	Program
_LoMem	-	$9000	System variable space
$9000	-	$BEFF	_LIB
$BF00	-	$BFFF	ProDOS primary access page
$C000	-	$CFFF	Soft switches/peripheral ROM space
$D000	-	$F7FF	Applesoft BASIC
$F800	-	$FFFF	System Monitor

TECHNICAL DATA

OPERATING SYSTEM:	ProDOS
INTEGER RANGE:	-32768 to +32767
REAL RANGE:	-1.00E+99 to +1.00E+99
CHARACTERS:	ASCII character set with corresponding ordinal values
SET:	Maximum 256 members
POINTER:	Represented by 16-bit integers
CUT BUFFER SIZE:	2K
BCD MATH PRECISION:	13 digits
MIN. RAM REQUIRED:	64 K
MAX IDENTIFIER LENGTH:	256 characters

APPENDIX C

COMPILER ERROR MESSAGES

NUMBER	DESCRIPTION
1	error in simple type
2	identifier expected
3	'Program' expected
4	')' expected
5	':' expected
6	illegal symbol
7	error in parameter list
8	'of' expected
9	'(' expected
10	error in type
11	'[' expected
12	']' expected
13	'end' expected
14	';' expected
15	integer expected
16	'=' expected
17	'begin' expected
18	error in declaration part
19	error in field-list
20	'.' expected
21	'*' expected
50	error in constant
51	':=' expected
52	'then' expected
53	'until' expected
54	'do' expected
55	'to'/'downto' expected
56	'if' expected
57	'file' expected

58	error in factor
59	error in variable
101	identifier declared twice
102	low bound exceeds high bound
103	identifier is not of appropriate class
104	identifier not declared
105	sign not allowed
106	number expected
107	incompatible subrange types
108	file not allowed here
109	type must not be real
110	tagfield type must be scalar or subrange
111	incompatible with tagfield type
112	index type must not be real
113	index type must be scalar or subrange
114	base type must not be real
115	base type must be scalar or subrange
116	error in type of standard procedure parameter
117	unsatisfied forward reference
118	forward reference type identifier in variable declaration
119	forward declared; repetition of parameter list not allowed
120	function result type must be scalar, subrange, or pointer
121	file value parameter not allowed
122	forward declared function; repetition of result type not allowed
123	missing result type in function declaration
124	F-format for real only
125	error in type of standard function parameter
126	number of parameters does not agree with declaration
127	illegal parameter substitution
128	result type of parameter function does not agree with declaration
129	type conflict of operands
130	expression is not of set type
131	test on equality allowed only
132	strict inclusion not allowed
133	file comparison not allowed

134	illegal type operands
135	type of operand must be Boolean
136	set element type must be scalar or subrange
137	set element types not compatible
138	type of variable is not array
139	index type is not compatible with declaration
140	type of variable is not record
141	type of variable must be file or pointer
142	illegal parameter substitution
143	illegal type of loop control variable
144	illegal type of expression
145	type conflict
146	assignment of files not allowed
147	label type incompatible with selecting expression
148	subrange bounds must be scalar
149	index type must not be integer
150	assignment to standard function is not allowed
151	assignment to formal function is not allowed
152	no such field in this record
153	type error in read
154	actual parameter must be a variable
155	control variable must not be declared on intermediate level
156	multidefined case label
157	too many cases in case statement
158	missing corresponding variant declaration
159	real or string tagfields not allowed
160	previous declaration was not forward
161	again forward declared
162	parameter size must be constant
163	missing variant in declaration
164	substitution of standard proc/func not allowed
165	multidefined label
166	multideclared label
167	undelared label
168	undefined label
169	error in base set
170	value parameter expected
171	standard file was redeclared
172	undeclared external file
173	Fortan procedure or function expected
174	Pascal procedure or function expected

175	missing file "input" in program heading
176	missing file "output" in program heading
177	assignment to function identifier not allowed here
178	multidefined record variant
179	X-opt of actual proc/func does not match formal declaration
180	control variable must not be formal
181	constant part of address out of range
201	error in real constant : digit expected
202	string constant must not exceed source line
203	integer constant exceeds range
204	8 or 9 in octal number
205	zero string not allowed
206	integer part of real constant exceeds range
250	too many nested scopes of identifiers
251	too many nested procedures and/or functions
252	too many forward references of procedure entries
253	procedure too long
254	too many long constants in this procedure
255	too many errors on this source line
256	too many external references
257	too many externals
258	too many local files
259	expression too complicated
260	too many exit labels
300	division by zero
301	no case provided for this value
302	index expression out of bounds
303	value to be assigned is out of bounds
304	element expression out of range
398	implementation restriction
399	variable dimension arrays not implemented

APPENDIX D

ProDOS (MLI) ERROR MESSAGES

NUMBER	DESCRIPTION
$00	No error
$01	Bad system call number
$04	Bad system call parameter count
$25	ProDOS Interrupt table full
$27	I/O error
$28	No device connected
$2B	Disk write protected
$2E	Disk switched
$40	Invalid pathname
$42	Max number of files open (Max = 8 files)
$43	Invalid reference number
$44	Directory not found
$45	Volume not found
$46	File not found
$47	Duplicate filename
$48	Volume full (No blocks free)
$49	Volume directory full (Max = 51 entries)
$4A	Incompatible file format or a ProDOS directory
$4B	Unsupported storage type
$4C	End of file encountered. No data was read
$4D	Position out of range
$4E	File access error or a locked file
$50	File is open (Multiple opens not allowed)
$51	Directory structure is damaged
$52	Not a ProDOS volume
$53	Invalid system call parameter
$55	Volume control block table full (Max = 8 volumes)
$56	Bad buffer address (Already in use)
$57	Duplicate volume
$5A	File structure damaged

ProDOS File Types

NUMBER	DESCRIPTION
$00	Typeless file
$04	ASCII text file
$06	General Binary file
$0F	Directory file
$C0 - $EF	ProDOS reserved
$F0	ProDOS added command line
$F1 - $F8	ProDOS user defined files 1 - 8
$F9	ProDOS reserved
$FA	Integer BASIC program file
$FB	Integer BASIC variable file
$FC	Applesoft program file
$FD	Applesoft variables file
$FE	Relocatable code file (EDASM)
$FF	ProDOS system file

APPENDIX E

ASSEMBLER ERROR MESSAGES

NUMBER	DESCRIPTION
	Syntax Errors
1	Address Error
2	Cannot Include File
3	Format Error
4	Forward Reference in Expression
5	Illegal Use of Conditional Assembly Directive before or
6	Misplaced Else Operator
7	Identifier Expected as Operand
8	Label Required
9	Multiply Defined Symbol
10	Nesting Error
11	Invalid Op-Code
12	Phase Error
13	Questionable Syntax
14	Undefined Symbol
15	Illegal Argument for Conditional Assembly
16	Symbol not in Macro Call Parameter List
17	Directive Requires "on" or "off"
	Fatal Assembler Errors
20	Unknown Error
21	Symbol Table Overflow
22	Lost Label
23	End of File During Macro Definition
24	End of File During Conditional Assembly

APPENDIX F

RUNTIME ERROR MESSAGES

NUMBER	DESCRIPTION
1	Case Index Error
2	Array Index Error
3	Input Error
4	ProDOS Error
5	Range Error
6	Arithmetic Overflow
7	End of File

APPENDIX G

ASCII CHARACTER SET

ASCII Control Characters, High Bit Off

#	CHAR	MEANING	WHAT TO TYPE
0	NUL	Null	CNTL -@
1	SOH	Start of Heading	CNTL -A
2	STX	Start of text	CNTL -B
3	ETX	End of text	CNTL -C
4	EOT	End of transmission	CNTL -D
5	ENQ	Enquiry	CNTL -E
6	ACK	Acknowledgement	CNTL -F
7	BEL	Bell	CNTL -G
8	BS	Backspace	CNTL -H or <–
9	HT	Horizontal Tab	CNTL -I or TAB
10	LF	Line Feed	CNTL -J or↓
11	VT	Vertical Tab	CNTL -K or↑
12	FF	Form Feed	CNTL -L
13	CR	Carriage Return	CNTL -M or RETURN
14	SO	Shift Out	CNTL -N
15	SI	Shift In	CNTL -O
16	DLE	Data Link Escape	CNTL -P
17	DC1	Device Control 1	CNTL -Q
18	DC2	Device Control 2	CNTL -R
19	DC3	Device Control 3	CNTL -S
20	DC4	Device Control 4	CNTL -T
21	NAK	Neg. Acknowledge	CNTL -U or –>
22	SYN	Synchronization	CNTL -V
23	ETB	End of Text Block	CNTL -W
24	CAN	Cancel	CNTL -X
25	EM	End of Medium	CNTL -Y
26	SUB	Substitute	CNTL -Z
27	ESC	Escape	CNTL -[or ESC
28	FS	File Separator	CNTL -\
29	GS	Group Separator	CNTL -]
30	RS	Record Separator	CNTL -^
31	US	Unit Separator	CNTL -_

ASCII Special Characters, High Bit Off

#	CHAR	MEANING	WHAT TO TYPE
32	SP	Space	SPACE BAR
33	!	Exclamation	
34	"	Quote Mark	
35	#	Pound	
36	$	Dollar	
37	%	Percent	
38	&	Ampersand	
39	'	Closing Quote	
40	(	Open Parenthesis	
41	)	Close Parenthesis	
42	*	Asterisk	
43	+	Plus	
44	,	Comma	
45	-	Hyphen	
46	.	Period	
47	/	Slant	
48	0	zero	
49	1		
50	2		
51	3		
52	4		
53	5		
54	6		
55	7		
56	8		
57	9		
58	:	Colon	
59	;	Semi-colon	
60	<		
61	=	Equal	
62	>		
63	?		

ASCII Upper Case Characters, High Bit Off

#	CHAR	MEANING	WHAT TO TYPE
64	@		
65	A		
66	B		
67	C		
68	D		
69	E		
70	F		
71	G		
72	H		
73	I		
74	J		
75	K		
76	L		
77	M		
78	N		
79	O		
80	P		
81	Q		
82	R		
83	S		
84	T		
85	U		
86	V		
87	W		
88	X		
89	Y		
90	Z		
91	[	Opening Bracket	
92	\	Reverse Slant	
93	]	Closing Bracket	
94	^	Caret	
95	_	Underline	

ASCII Lowercase Characters, High Bit Off

#	CHAR	MEANING	WHAT TO TYPE
96	‘	Opening Quote	
97	a		
98	b		
99	c		
100	d		
101	e		
102	f		
103	g		
104	h		
105	i		
106	j		
107	k		
108	l		
109	m		
110	n		
111	o		
112	p		
113	q		
114	r		
115	s		
116	t		
117	u		
118	v		
119	w		
120	x		
121	y		
122	z		
123	{	Opening Brace	
124	|	Vertical Line	
125	}	Closing Brace	
126	~	Overline (Tilde)	
127	DEL	Delete/Rubout	

NOTE: The ASCII character set repeats with numbers 128 through 255 for the case with the High Bit On.

INDEX

A

ABS VII-2
Actual Parameter IV-74
Addition IV-27
Address IV-156 VII-3
AND VII-4
ARCTAN VII-5
Argument IV-77
ARRAY IV-95, VII-6
..Copy IV-104
..Multidimensional IV-98
..OF Records IV-118
ASCII codes Appendix.- G
Assembler V-1
Assembler Directives V-4
Assembly Routine V-4, 20
Assignment operator IV-25
AutoBoot III-22

B

BEGIN/END VII-9, IV-14
Block Commands II-6
Body IV-14
Boolean IV-49, VII-10
Bracket II-8
Byte IV-12

C

C40 VI-25
C80 VI-25
CASE selector VII-11
CASE statement IV-62
CAT VI-13
CD VI-8
CFG VI-26
Character data type IV-40
..ARRAYS OF IV-41
..Strings IV-42
Chain III-18
Char VII-12, V-37
CHMOD VI-18
CHR IV-46, VII-15
CMP VI-20
Color III-17
Comment IV-9
Compiler III-1
Concat III-14, VII-16
Conditional IV-26
Configuration I-12, VI-26
Constant IV-15
CONST VII-17, IV-14
COS VII-18
CP VI-15
CPV VI-19
Cursor control II-5
Cut Buffer II-6

D

Data types, Predefined
..ARRAY IV-40
..BOOLEAN IV-49
..CHAR IV-40
..INTEGER IV-31
..REAL IV-31
..VAR IV-16
Data types, User-defined
..Record IV-113
..Scalar IV-59
..String IV-37
Date VI-23
Declaration IV-8,67,77,95
Decrement IV-34
Difference IV-130
Directory Control VI-6
DISPOSE IV-166, VII-19
DIV IV-53, VII-20
Division IV-27
DO IV-34, VII-21
DOWNTO VII-22

INDEX

DRAW III-16

E

Echo VI-33
EDITOR I-12, II-1
Element
..OF ARRAY IV-96
..OF SET IV-127
ELSE VII-23
END VII-24
EOF IV-139, VII-25
EOLN IV-109, VII-26
Error Messages III-35
..Compiler Appendix - C
..ProDOS Appendix - D
..Assembler Appendix - E
..Runtime Appendix - F
EXP VII-27

F

False VII-28
File IV-135
..defined VII-29
..management IV-146
..names V-3
..of records IV-142
..random access IV-148
..reading IV-138, IV-146
..text IV-152
..writing IV-136
Find VI-21
FOR IV-29, VII-30
Format VI-19
Formatting output VI-27
Formulas IV-11
Forward reference IV-91
Functions IV-77, VII-31

G

Graphics III-14
GET IV-149, VII-33
Global IV-83
GOTO IV-93, VII-35
GREP VI-22

H

Heap IV-162
Hierarchy
In KIX files VI-2
HGr III-16, VII-36
HIRES III-16

I

Identifier IV-14, IV-142
IF VII-37
conditions IV-26
IN IV-129, VII-38
INCLUDE III-9
Index III-13, IV-100 VII-40
INPUT IV-16, VII-41
INTEGER IV-32 VII-42
Intersection IV-130
INTRO VI-26
ISO PASCAL i., Appendix - A

J

K

KIX Commands VI-5
..Advanced Topics VI-38
..Command Menu VI-26
..File Structure VI-2
..Useful Commands VI-34
..Wildcards VI-29

L

Label V-14 VII-43
LENGTH III-12 VII-44
LIBRARY (LIB) iv, I-9
Linked Lists IV-162
Linking Programs III-18
LINK iii
Literal IV-8
LN VII-45
Local V-17
..in Assembler V-14
..Variables IV-83
LS VI-9
LPR VI-13

M

MAXINT VII-46, IV-29,162
Memory Map Appendix - B
Memory Usage IV-156, V-11
MENU VI-26
MKDIR VI-6
MOD IV-53, VII-47
Multidimension Arrays IV-98
Multiple Parameters IV-73
Multiplication IV-27
MV VI-16

N

Nests IV-28, 86
NEW IV-158, VII-48
NIBBLE IV-12
NIL VII-50
Node IV-155
NOT VII-51
Numbers
..Integer IV-32, VII-42
..Real IV-32, VII-68

O

Object Code VI-2
ODD VII-52, IV-81
OF VII-53
Operating System I-3
Operator IV-27
..Arithmetic IV-27
..Precedence IV-54
..Relational IV-27
..Set IV-130
Options, Compiler III-3
OR VII-54
ORD IV-63, VII-55
OUTPUT III-7, IV-16, VI-11
..printer III-7, IV-11,17
..screen IV-6

P

PACK VII-57
PACKED VII-58
PAGE VII-58
Parameter IV-70
..Multiple IV-73
..Used in Arrays IV-105
..Used in Assembler V-14
..Value IV-72, V-14
..Variable IV-72, V-18
Pascal i
Pathname I-5, VI-3
Peek V-24
Pointer IV-155 VII-59

INDEX

Poke V-23
Precision iii
PRED IV-63, VII-60
Predefined Funcs. IV-34, 80
Predefined Words IV-7
PR.I III-8, IV-17
PRINT III-7
Printer IV-11, 17
Procedure IV-67,V-18, VII-61
ProDOS I-3
PROGRAM VII-64
Programs, Sample
..AddMatrix IV-103
..AddressBook IV-120
..AddStrings IV-105
..AlphaScreen IV-161
..Appointments IV-163
..Average IV-30
..Calc IV-89
..CallMenu IV-69
..Compute IV-92
..Construction IV-12
..DivLesn IV-50
..Ego IV-4
..Elapsed IV-116
..Exchange IV-84
..Finals IV-131
..FirstWord IV-38
..FormalParameter IV-74
..Format IV-6
..GetWord IV-110
..GoExample IV-94
..HexToDec IV-56
..Locate IV-97
..Math IV-79
..Matrix IV-100
..ParamArray IV-107
..RecDemo IV-144
..RecRead IV-147
..SeekDemo IV-149
..SetDemo IV-128
..SocialSecurity IV-22
..StoreData IV-140
..TestGrades IV-132
..VariantRec IV-123
..WordProc IV-153
PUT IV-149, VII-64
PWD VI-6

Q

Quit VI-26

R

Random Access Files IV-148
Random Numbers III-24
READ IV-16, VII-65
READLN IV-16, VII-66
Real IV-32, VII-68
Record IV-113, VII-69
..ARRAY OF IV-118
..Copy IV-116
..FILE OF IV-142
..Variant IV-121
Recursion IV-111
Redirection VI-12
Relational Operator IV-27
REPEAT IV-61, VII-71
Reserved word IV-7
RESET VI-149,VII-72
REWRITE IV-136,VII-74
RM VI-16
RMDIR VI-8
ROUND IV-29, VII-75
Run Program III-24
..Compiler III-2
..Editor II-2
..Assembler V-2

S

Scalar IV-55,129
Scientific Notation IV-32
Scope IV-83
Screen config. III-16
SD VI-25
SDIFF VI-21
Seek IV-148
Sequential File IV-148
Set IV-127, VII-76
SIN VII-77
Source Code IV-2
SQR VII-78
SQRT VII-79
Stack V-9
Standalone Disks III-22
String III-10, IV-96, VII-80
Subrange IV-60
Subscript IV-96, IV-100
Substring III-13, VII-81
Subtraction IV-27
SUCC IV-63, VII-82
Syntax Errors III-5

T

Tab II-8
TEXT VII-83
 Editor II-1
 files IV-152
THEN VII-84
TO VII-85
TRUE VII-86
TRUNC IV-29, VII-87
TYPE IV-91, VII-88
Tx III-16, VII-89

U

Unconditional Branch IV-93
Underscore II-8
Union IV-130
UNPACK VII-90
UNTIL VII-91

V

Value Parameters IV-21,72
VAR IV-14, VII-92
Variable IV-16
..Assigning IV-24
..Boolean IV-49
..Declaring IV-26
..Global IV-83
..Local IV-83
..Parameters IV-72, V-18
..Relative IV-87
..Scalar IV-59, IV-129
Variant records IV-121

W

WHILE IV-45, VII-93
Wildcard VI-29
WITH IV-115, VII-94
WRITE IV-16, VII-95
WRITELN IV-16, VII-97

X

X Register V-5

Suggestion Box

We do our best to provide you with complete, bug-free software and documentation. With products as complex as compilers and programming utilities, this is difficult to do. If you find any bugs or areas where the documentation is unclear, please let us know. We will do our best to correct the problem in the next revision. We would also like to hear from you if have any comments or suggestions regarding our product.

To help us better understand your comments please use the following form in your correspondence and mail it to: Kyan Software Inc.,
1850 Union Street #183, San Francisco, CA 94123.

Name____
Address____
City______________________________State______ZIP__________
Telephone:
(day)____________________ (evening)__________________

Kind of Problem

__ Software Bug
__ Documentation Error
__ Suggestions
__ Other ____________________

Software Description

Product Name ______________
Version No. ________________
Date Purchased _____________

Kyan Software Products You Use

__ Kyan Pascal
__ Program. Utilities Toolki
__ MouseText Toolkit
__ Kyan Macro Assembler/Linker
__ Advanced Graphics Toolkit
__ Other ________________

Your Hardware Configuration

Type/Model of Computer ______________________________
How many and what kind of disk drives ______________________
What is your screen capability: ___40 Column ___80 Column
How much RAM?____K (what kind of RAM Board?______________)
What kind of printer and interface card do you use?______________
____
What kind of modem?________________________________
Other information about your computer system: ______________
____

What do you use this software for?

___ Education (I am a __ teacher __ student)
___ Hobby
___ Professional Software Development
___ Other ____

Problem Description (if appropriate, please include a disk or program listing).

Suggestions

OWNER REGISTRATION

Thank you for purchasing this product from Kyan Software. Please fill in this Registration form and mail it to Kyan Software. As a registered owner, you are entitled to receive technical support, product upgrades, and complimentary copy of "**Update .... Kyan**", the Kyan newsletter.

Name ____
Address ____
City ______________________ State_______ ZIP__________
Telephone (day) _______________ (evening) _____________

Software Description
Product Name ____________________ Version No. _______
Software Dealer ___________________ Purch. Date _______

What Kyan Software products do you use?
__ Kyan Pascal
__ System Utilities Toolkit
__ TurtleGraphics Toolkit
__ MouseText Toolkit
__ Kyan Macro Assembler
__ Advanced Graphics Toolkit
__ Other _________________
__ Other _________________

Hardware Configuration?
Type/Model of Computer ______________________________
Disk drive(s) ______________________________________
How much RAM ____K (what kind of RAM Board _____________)
What kind of printer and interface card do you use ___________
____
What kind of modem _________________________________
Other information or accessories ________________________
____

How will you use this software?
___ Education (I am a __ teacher __ student) ___ Hobby
___ Software Development ___ Other

How did you hear about this product?
___ Magazine advertisement
___ From a friend or User Group
___ Software dealer
___ Article or Product Review
___ At school
___ Other

What computer magazines do you read on a regular basis?
____
____

Kyan Software Inc.
1850 Union Street #183
San Francisco, CA 94123

Toolkit I

SYSTEM UTILITIES

USERS MANUAL

KYAN SOFTWARE INC.
SAN FRANCISCO, CALIFORNIA

TOOLKIT I

SYSTEM UTILITIES

Requires
Kyan Pascal (Version 2.0)
and
an Apple II with 64K of memory

Copyright 1986
Kyan Software Inc.
San Francisco, California

TABLE OF CONTENTS

SECTION **PAGE**

PREFACE I-3

A. INTRODUCTION

Overview I-7
How to Use the System Utilities I-9
Demonstration Programs I-10

B. ProDOS UTILITIES LIBRARY

Overview and Routine Summary I-11
Using the ProDOS Utility Library I-12
Declaring Global Types I-12
Routine Descriptions I-14

C. DEVICE DRIVER LIBRARY

Overview and Routine Summary I-25
Using the Device Driver Library I-26
Routine Descriptions I-27

D. SCREEN MANAGEMENT LIBRARY

Overview and Routine Summary I-33
Using the Screen Management Library I-34
Routine Descriptions I-35

E. OTHER SYSTEM UTILITIES

Overview I-41
Random Number Routines I-42
Conversion Routines I-44
Line Parse Routine I-47
Sort/Merge Routine I-48

F. APPENDIX (Disk Directory) I-53

Notice

Kyan Software reserves the right to make improvements to the products described in this manual at any time and without notice. Kyan Software cannot guarantee that you will receive notice of such revisions, even if you are a registered owner. You should periodically check with Kyan Software or your authorized Kyan Software dealer.

Although we have thoroughly tested the software and reviewed the documentation, Kyan Software makes no warranty, either express or implied, with respect to the software described in this manual, its quality, performance, merchantability, or fitness for any particular purpose. This software is licensed "as is".

In no event will Kyan Software be liable for direct, indirect, incidental or consequential damages resulting from any defect in the software or documentation even if it has been advised of the possibility of such damages.

Some states do not allow the exclusion or limitation of implied warranties or liabilities or consequential damages, so the above limitation or exclusion may not apply to you.

Use of Routines in this Toolkit

Kyan Software hereby grants you a non-exclusive license to merge or use the routines in this Toolkit in conjunction with your own programs for either private or commercial purposes.

Copyright

This users manual and the computer software (programs) described in it are copyrighted by Kyan Software Inc. with all rights reserved. Under the copyright laws, neither this manual nor the programs may be copied, in whole or part, without the written consent of Kyan Software Inc. The only legal copies are those required in the normal use of the software or as backup copies. This exception does not allow copies to be made for others, whether or not sold. Under the law, copying includes translations into another language or format.

This restriction on copies does not apply to copies of individual routines copied and distributed as an integral part of programs developed by the purchaser of this Toolkit.

Backup Copies

We strongly recommend that you make and use backup copies of the Toolkit diskette. Keep your original Kyan diskettes in a safe location in case something happens to your copies. (Remember ..... Murphy is alive and well, and he loves to mess with computers!)

Copy Protection

Kyan Software products are not copy-protected. As a result, you are able to make backup copies and load your software onto a hard disk or into a RAM expansion card. We trust you. Please do not violate our trust by making or distributing illegal copies.

Limited Warranty

Kyan Software warrants the diskette(s) on which the Kyan software is furnished to be free from defects in materials and workmanship under normal use for a period of ninety (90) days from the date of delivery to you as evidenced by your proof of purchase.

Disclaimer of Warranty -- Kyan Software Inc.

Except for the limited warranty described in the preceding paragraph, Kyan Software makes no warranties, either express or implied, with respect to the software, its quality, performance, merchantability or fitness for any particular purpose. This software is licensed "as is". The entire risk as to its quality and performance is with the Buyer. Should the software prove defective following its purchase, the buyer (and not Kyan Software, its distributors, or its retailers) assumes the entire cost of all necessary servicing, repair, or correction and any incidental, or consequential damages.

In no event, will Kyan Software be liable for direct, or indirect, incidental, or consequential damages resulting from any defect in the software even if it has been advised of the possibility of such damages. The sole obligation of Kyan Software Inc. shall be to make available for purchase, modifications or updates made by Kyan Software to the software which are published within one year from date of purchase, provided the customer has returned the registration card delivered with the software.

Some states do not allow the exclusion or limitation of implied warranties or liabilities for incidental or consequential damages, so the above limitations or exclusions may not apply to you.

If any provisions or portions of this Agreement shall be held by a court of competent jurisdiction to be contrary to law, the remaining provisions of this Agreement shall remain in full force and effect. The validity, construction and performance of this Agreement shall be governed by the substantive law of the State of California.

This Agreement constitutes the entire agreement between the parties concerning the subject matter hereof.

Technical Support

Kyan Software has a technical support staff ready to assist you with any problems you might encounter. If you have a problem, we request that you first consult this users manual.

If you have a problem which is not covered in the manual, our support staff is ready to help. If the problem is a program which won't compile or run, we can best help if you send us a description of the problem and a listing of your program (better yet, send us a disk with the listing on it). We will do our best to get back to you with an answer as quickly as possible.

If you question can be answered on the phone, then give us a call. Our technical staff is available to assist on Monday through Friday between the hours of 9 AM and 5 PM, West Coast Time. You may reach them by calling:

Technical Support: (415) 626-2080

Suggestion Box

Kyan Software likes to hear from you. Please write if you have sugges-tions, comments and, yes, even criticisms of our products. We do listen. It is your suggestions and comments that frequently lead to new products and/or product modifications.

We encourage you to write. To make it easier, we have included a form in the back of this manual. This form makes it easier for you to write and easier for us to understand and respond to your comments. Please let us hear from you.

Mailing Address: **Kyan Software Inc.**
1850 Union Street #183
San Francisco, CA 94123

A. Introduction

Thank you for purchasing this System Utilities Toolkit. It is designed for use with Kyan Pascal (Version 2.0 or later) and an Apple // with at least 64K of memory (RAM).

Overview

The Toolkit contains many useful and powerful routines which can be merged directly into your Kyan Pascal programs. These routines are grouped into four libraries or directories.

I. ProDOS Utility Library

This library contains routines which provide support for various ProDOS functions and procedures from within Pascal programs.

- Delete
- Append
- RemDir
- GetDir
- GetDate
- BSave
- PrintFile
- Rename
- Lock
- Find
- GetPrefix
- FindClock
- BLoad
- Copy
- Unlock
- ScanFile
- Format
- SetDate
- SetTime
- SetPrefix
- MakeDir
- FileType
- GetTime
- PrtMLIerror
- GetTimeM

II. Device Driver Library

This library contains functions and procedures which establish communication between your application programs and an external device (i.e., mouse, joystick or trackball).

- FindMouse
- MouseMoved
- SetMouseXY
- EndMouse
- JoyStX
- InitMouse
- MouseX
- SetXBounds
- PrtMouseChar
- JoyStY
- MouseClick
- MouseY
- SetYBounds
- Button0
- MouseHeld
- ZerMouse
- HomeMouse
- Button1

III. Screen Management Library

This library contains routines used to control screen functions.

- CLS
- Normal
- ClrEOLN
- CursorY
- ScrnFull
- GoToXY
- ScrollUp
- ClrEOP
- GetChar
- IDMachine
- TAB
- ScrollDown
- Col80
- ScrnTop
- ON40
- Inverse
- ClrLine
- CursorX
- ScrnBottom
- ON80

IV. Other System Utilities

- Random Number Routines

  -- Seed ("seed" the random number generator routine)
  -- Rnd (return a random number between 0 and 1)
  -- Random (return a random number in range [min .. max])

- Conversion Routines

  -- Integer to String
  -- String to Real Number
  -- Real Number to String
  -- String to Integer

- Line Parsing Routine

- Sort/Merge Routine

How to Use the System Utilities

The routines in each Library are text files and are structured to be used as "include" files in your Pascal programs. To use them:

1. Copy the desired Toolkit routine(s) into your current working directory.
2. Declare the "included" file(s) in the declarations portion of your program.
3. Call the routine(s) as required in the body of your program.

Some libraries require global types to be separately declared. The steps for declaring these global types are described later in this Manual.

While most of the Toolkit routines are independent of all others, some routines incorporate others in the body of their programs. In these circumstances, it is necessary to include both Toolkit routines in your Pascal program. If a routine is dependent on some other routine, the dependency is noted in the application notes for the routine.

It is a good idea to review the section in Chapter III of your Kyan Pascal manual which describes the use of "include" files in your Pascal programs. You should also look at Chapter V which describes assembly language programming and Appendices C-F which list the meaning of MLI and other error messages.

You are encouraged to examine the source code of the Toolkit routines. To do so, simply load the routine's include file using the Kyan Text Editor. The source files are fully commented, and so you should be able to easily follow the logic and flow of the program. You can also modify any of the routines, if desired, and customize them for your particular application.

The Appendix illustrates the directory and file organization of the System Utilities Toolkit disk. Always be sure to specify the complete pathname of the include file when you are copying routines into your working directory or running the demonstration programs. Also, when running the demo programs, be sure there is a copy of the Kyan Pascal Runtime Library (LIB) in the working directory.

Demonstration Programs

The System Utilities Toolkit contains a number of demonstration programs which illustrate the use of Toolkit routines. Most of these programs are included in both source and object code formats.

TITLE	DESCRIPTION
CATALOG.P	This program will print a short catalog of the directory you indicate. It will list each file and its type and size (in blocks). This program demonstrates the ProDOS utilities.
MOUSE.DEMO.P	Uses the Mouse routines in conjunction with a simple menu application interfaced to the mouse. This program demonstrates Device utilities.
RANDOM.DEMO.P	Play a simple random number game. This program demonstrates seeding of the random number generator.
ESORT.DEMO.P	Demonstrates the Sort routine.
MERGE.DEMO.P	Demonstrates the Merge routine.
MOUSETEXT.DEMO	(Object code only). Displays the procedure and complete table of mouse text characters.
TURTLE.DEMO	(Object code only). Illustrates the power of Kyan's TurtleGraphics Toolkit.

B. ProDOS Utility Library

Overview

The ProDOS Utility Library contains 26 different routines. They include a mix of functions and procedures which can be incorporated into your Pascal programs. Each ProDOS utility routine is described on following pages.

The ProDOS routines included in this Library are:

Append
BLoad
BSave
Copy
Delete
Filesize
Filetype
Find
FindClock
Format
GetClock
GetDate
GetDir
GetPrefix
GetTime
Lock
MakeDir
PrintFile
PrtMLIerror
RemDir
Rename
ScanFile
SetClock
SetDate
SetPrefix
SetTime
Unlock

The ProDOS routines are listed in alphabetical order.

Using the ProDOS Utility Library

To use the ProDOS Utility routines, you must first declare a set of global types and then "include" the desired routine after the variable and type declarations in your Pascal program. (Please refer to Chapter III of the Kyan Pascal User Manual for more information about the use of Include files in Pascal programs.) Once a routine is included, it can be called as often as needed in your program.

Declaring Global Types

You can declare the global variables in either of two ways. First, you can simply type the following global declarations into your Pascal program:

```
Type
  FileString = ARRAY [1..16] of CHAR;
  PathString = ARRAY [1..65] of CHAR;
  FilePointer = ^FileRecord;
  FileRecord = RECORD
         Filename : FileString;
         NextFile : FilePointer
         END;
```

Alternately, you can "include" the file on the program disk called ***PRODOS.TYPES.I*** in your Pascal program using the following format:

```
Type
#i PRODOS.TYPES.I
```

Both methods acheive the objective of declaring the global types used in the ProDOS Utility Library routines.

Notes

1. Don't forget to place a copy of all the files "included" in your Pascal program in the same working directory as the main program. If you forget, the compiler will not be able to find the file and a "File Not Found" compiler error will occur.

2. There are no global types to be declared with Device Driver routines.

3. The Utility program disk contains a set of sample programs. The program file ***CATALOG.P*** demonstrates the ProDOS utilities.

4. All of the ProDOS Utility Library routines are similar in that: (1) each ProDOS Function returns the MLI error code of its operation; and, (2) all pathnames passed must be padded with spaces to the right.

Command Name: *Append*

Syntax: FUNCTION APPEND(VAR sourcepath, addpath :
PathString) : INTEGER;

Description: Append the contents of "sourcepath" to "addpath". Type checks are not made; the user must do this first using the FILETYPE function included in this toolkit. A 512-byte local buffer is used for the data transfer area.

****

Command Name: *BinaryLoad*

Syntax: FUNCTION BLOAD (VAR pathname : PathString;
len, dest : INTEGER) : INTEGER;

Description: Load the first "len" bytes of a BINary image named "pathname" starting at "dest". If "len" is zero, the entire file is loaded. **For Example:** To load a hi-resolution graphics image into page 1 of hi-res memory, the command BLOAD(name, 8192, 8192) would be used since the image length is 8192 bytes long ('len') and hi-res page 1` starts at memory location 8192 ($2000) ('dest'). If the image had been saved with a length of 8192 previously, the command BLOAD(name,0,8192) would perform the same opereation since a lentgth specification of zero loads the entire binary file into memory.

****

Command Name: *BinarySave*

Syntax: FUNCTION BSAVE (VAR pathname : PathString;
len, dest : INTEGER) : INTEGER;

Description: Save a BINary image named "pathname" starting at "dest" of "len" bytes in length. **For example:** To save a hi-resolution graphics image stored in hi-res page 1, the command BSAVE(name, 8192, 8192) would store a binary image of the first 8192 bytes it found (len of 8192) starting at location 8192 ($2000) (start) into file 'name'.

Command Name: *Copy*

Syntax: FUNCTION COPY(VAR sourcepath, destpath : PathString):
INTEGER;

Description: Copy the file designated by "sourcepath" to the file designeated by "destpath". The destination filename already exists, it is destroyed before the copy begins. COPY uses a 512-byte buffer in which to perform the data transfer. The volumes involved in the copy function must both be on-line at the time of the COPY. COPY will not ask for volumes to be inserted and removed.

****

Command Name: *Delete*

Syntax: FUNCTION DELETE(VAR pathname:PathString):
INTEGER;

Description: Delete the file designated in "pathname". Error codes are returned as Integers

****

Command Name: *FileSize*

Syntax: FUNCTION FILESIZE (VAR pathname : PathString;
VAR fsize : INTEGER) : INTEGER;

Description: Set FSIZE to the number of disk blocks occupiedby "pathname" (i.e. return size of file in blocks).

Command Name: *FileType*

Syntax: FUNCTION FILETYPE (VAR pathname : PathString;
VAR ftype : INTEGER) : INTEGER;

Description: Set FTYPE to the file type of "pathname" (i.e. return type of file).

****

Command Name: *Find*

Syntax: FUNCTION FIND (VAR filename : PathString;
VAR found : BOOLEAN) : INTEGER;

Description: Returns "found" TRUE if the file name passed is located in the system prefix (Working Directory). Only the Working Directory is searched.

****

Command Name: *FindClock*

Syntax: FUNCTION FINDCLOCK : INTEGER;

Description: Returns the slot number of a ThunderClock or compatible Apple II clock card. If there is no compatible clock in the system, the number returned is zero. A ThunderClock is recognized by the following identification bytes:

LOCATION	VALUE
Cx00	8
Cx01	$27
Cx02	$28

where "x" is the slot number of the card.

Command Name: ***Format***

Syntax: FUNCTION FORMAT (slot, drive : INTEGER; volname : FileString; VAR fmterror : INTEGER) : INTEGER;

Description: Format the volume in (slot#, drive#) and name the new volume "volname". If a format error occurs, an MLI error code or one of the following code numbers (fmterror) will be returned

$27 - Disk access error
$33 - Drive too slow
$34 - Drive too fast

If an error occurs during the writing of a boot block or construction of the Volume Bit Map, it is returned as the function value. The volume name specified must conform to pathname guidelines and begin with a '/', or an "INVALID PATHNAME" will be returned as the function value.

Due to the size of the FORMAT routine, it must be loaded from disk each time it is used. A file named FORMAT.OBJ is included in the ProDOS utilities directory on your diskette. This file must be located in the Working Directory when the FORMAT routine is called.

FORMAT.OBJ is BLOADed into memory at location $2000 and requires the use of HiRes graphics page 1. For this reason, any program using the FORMAT function must begin with the following code:

```
#A
_UsesHires
#
```

The "_UsesHires" declaration tells the compiler not to allocate any memory between locations $2000 and $3FFF to any part of the Pascal program or runtime environment. (Please refer to your Kyan Pascal manual for more information).

Declaring a Pascal program to be a "_SystemFile" does not effect the FORMAT routine.

Command Name: ***GetClock***

Syntax: PROCEDURE GETCLOCK (VAR mon, day, yr, hr, min,
weekday, sec, millisec: INTEGER);

Description: Returns readings from Thunderclock. If no clock is present, the values returned are undefined. The FINDCLOCK function must be called previous to this procedure.

****

Command Name: ***GetDate***

Syntax: PROCEDURE GETDATE (VAR day, month, year
: INTEGER);

Description: Returns the day (0..31), month (0..12) and year (00..99) as integers. (**NOTE:** The function FindClock must always be called before a GetDate or SetDate procedure is used. Without this call, these routines will not know there is a clock card in the system.)

****

Command Name: ***GetDirectory***

Syntax: FUNCTION GETDIR (VAR dirname : PathString;
VAR ListPtr : FilePtr) : INTEGER;

Description: Returns a linked list of filenames in "dirname". The list is terminated by a NIL. If the directory is located but empty, LISTPTR returns pointing at NIL. If the MLI return code is non-zero, LISTPTR will be undefined.

Command Name: *GetPrefix*

Syntax: PROCEDURE GETPREFIX (VAR prefix : PathString) ;

Description: Returns the current system prefix (Working Directory). "Prefix" returns with the first 64 characters the system prefix; the remainder is buffered by spaces. If the system prefix is null, the returned array contains only blanks.

****

Command Name: *GetTime*

Syntax: PROCEDURE GETTIME (VAR hour, minute: INTEGER);

Description: Returns the system time in hour/minute military format

****

Command Name: *Lock*

Syntax: FUNCTION LOCK (VAR pathname : PathString) :
INTEGER;

Description: Deny Write, Delete, and Rename access to the file designated in the "pathname".

Command Name: ***MakeDirectory***

Syntax: FUNCTION MAKEDIR (VAR dirname : PathString) : INTEGER;

Description: Create a directory named "dirname". The directory is created as a linked subdirectory type.

****

Command Name: ***PrintFile***

Syntax: PROCEDURE PRINTFILE (pathname : PathString;
Slot, LeftMargin, RightMargin, TopMargin, BottomMargin, CPI:
INTEGER; header : FileString) : INTEGER;

Description: Print the text file designated by "pathname" to the printer in "slot" using margins listed. Header is the printer conditioning control codes. If an "include" file is encountered in the text file, an attempt is made to find and print the included file. If it fails, it is ignored and an MLI code is returned which corresponds to an error caused by the pathname specified and NOT the include files. The printer must be on-line and at top of form when this procedure is called.

****

Command Name: ***PrintMLIerror***

Syntax: PROCEDURE PRTMLIERROR (errorcode : INTEGER);

Description: Print the MLI error code passed at current cursor position. Unknown error codes are printed as "*PRODOS ERROR $xx*" where *xx* is the hexidecimal error code. The error text is printed from the current position of the cursor and NO carriage return is generated by the output routine.

Command Name: ***RemoveDirectory***

Syntax: FUNCTION REMDIR (VAR dirname : PathString) : INTEGER;

Description: DELETE the directory "dirname", first checking to make sure that "dirname" is an empty directory. If "dirname" is not empty, a "File Access Error" will be returned.

****

Command Name: ***Rename***

Syntax: FUNCTION RENAME (VAR oldpath, newpath : PathString) : INTEGER;

Description: Rename the file defined by "oldpath" with the name defined by "newpath". The files must be in the same directory for the rename to succeed.

****

Command Name: ***ScanFile***

Syntax: FUNCTION SCANFILE (VAR pathname, string : PathString; VAR position : INTEGER) : INTEGER;

Description: Scan the TEXT file designated in "pathname" for the string designated in "string". If the string is found, "position" returns the byte number of the file position of the first character in the string which matchs. If no match is found, "position" returns a value of-1. The SCANFILE search is NOT case sensitive. This command is useful for searching identification fields stored in text files (e.g., high score files).

Command Name: *SetClock*

Syntax: FUNCTION SETCLOCK(Month, Day, Year, Hours, Minutes,
Dayofweek: INTEGER):INTEGER;

Description: Set the ThunderClock peripheral card to the values passed. The ThunderClock or compatible card must be write enabled for the setting to succeed. The function values returned are:

Code	Error Description
0	No error
1	Month not in 0..59
2	Day not in range according to month
3	Year not 86..99
4	Hour4 not in range 0..59

If an error occurs no time change takes place. The FINDCLOCK function must be called previous to this function.

****

Command Name: *SetDate*

Syntax: FUNCTION SETDATE (day, month, year:
INTEGER): INTEGER;

Description: Set the system date to the value specified by the user. Error codes may be returned which correspond to an out-of-bounds value:

Code	Error Description
0	No error
1	Month is not between 1 and 12
2	Day is out of range (according to month passed)
3	Year is not between 86 and 99

If an error occurs, no date change takes place. (**NOTE:** The function FindClock must always be called before either the SetDate or GetDate procedure is used. Without this call, the these routines will not know there is a clock card in the system.)

Command Name: ***SetPrefix***

Syntax: FUNCTION SETPREFIX (VAR newprefix : PathString) : INTEGER;

Description: Set the system prefix (or Working Directory) to the pathname specified. If the pathname is all blanks, the system prefix is set to the root volume (null)

****

Command Name: ***SetTime***

Syntax: FUNCTION SETTIME (hours, minutes : INTEGER) : INTEGER;

Description: Set the system time to the values passed. If an error occurs, the function values returned are:

Code	Error Description
0	No error
1	Hour is not in range 0..23
2	Minute is not in range 0..59

If an error occurs, no time change takes place.

****

Command Name: ***Unlock***

Syntax: FUNCTION UNLOCK(VAR pathname : PathString) : INTEGER;

Description: Reverse the effects of the LOCK function.

C. Device Driver Library

Overview

The routines in the Device Driver Library allow you to link external devices to your Pascal programs. The routines are intended for use with a mouse, trackball (which behave exactly like a mouse), or joystick.

If you are planning a major project using mouse routines, you should look at Kyan's **MouseText Toolkit**. The routines in this **System Utility Toolkit** are quite primitive in comparison. The MouseText Toolkit provides all of the macros needed for windows, pull-down menus, option selection via mouse, and icon manipulation.

A mouse interface works best when it is "interrupt driven", that is, when the programmer sets up a series of routines which the mouse firmware (i.e., the ROM's on the Mouse card) calls as events occur. The routines in this Toolkit are not interrupt driven. Instead, the program must continuously poll the device status to determine when a condition changes. This method is less efficient than the interrupt technique but is also much easier to use in small Pascal programs. You can look at the Mouse Demo program (described on the next page) to see a practical method for interfacing the mouse with your program.

For a complete discussion of mouse firmware and the interface between mouse firmware and your Apple II, please refer to the documentation which you received with your mouse.

The routines in the Device Driver Library are:

Button0
Button1
EndMouse
FindMouse
HomeMouse
InitMouse
JoyStXX
JoyStXY
MouseClick
MouseHeld
MouseMoved
MouseX
MouseY
PrtMouseChar
SetMouseXY
SetXBounds
SetYBounds
ZerMouse
Zero Mouse

Using the Device Drive Library

To use the Device Driver routines, you must first "include" the desired routine after the variable and type declarations in your Pascal program. (Please refer to Chapter III of the Kyan Pascal User Manual for more information about the use of Include files in Pascal programs.) Once the routine is included, you can call it as often as needed in your program.

Notes

1. Don't forget to place a copy of all the files "included" in your Pascal program in the same working directory as the main program. If you forget, the compiler will not be able to find the file and a "File Not Found" compiler error will occur.

2. There are no global types to be declared with Device Driver routines.

3. The program disk contains a set of sample programs. The program file Mouse.Demo.P demonstrates the use of mouse routines in conjunction with TurtleGraphics. You can use the Kyan Pascal editor to examine the source code and to see how the device routines can be utilized in your own programs.

4. The use of any mouse routine requires FUNCTION FINDMOUSE to be in the host program. Also, PROCEDURE INITMOUSE must be loaded and called before any other mouse routines are used.

Command Name: ***Button0***

Syntax: FUNCTION BUTTON0 : BOOLEAN;

Description: This function returns the value TRUE if button 0 or the Open-Apple Key is pressed.

****

Command Name: ***Button1***

Syntax: FUNCTION BUTTON1 : BOOLEAN;

Description: This function returns the value TRUE if button 1 or the Closed-Apple Key is pressed.

****

Command Name: ***EndMouse***

Syntax: PROCEDURE ENDMOUSE;

Description: This procedure disables the mouse system interrupts.

Command Name: ***FindMouse***

Syntax: FUNCTION FINDMOUSE : INTEGER;

Description: This function returns the slot number of the mouse card. If none is present, FINDMOUSE will return a zero and all other calls to mouse routines will be ignored. The FINDMOUSE function must be called in every program using mouse routines (along with the INITMOUSE procedure).

****

Command Name: ***HomeMouse***

Syntax: PROCEDURE HOMEMOUSE;

Description: This procedure moves the mouse X,Y coordinates to their lowest boundaries as defined by the SetXBounds and SetYBounds routines.

****

Command Name: ***InitializeMouse***

Syntax: PROCEDURE INITMOUSE;

Description: This procedure prepares the mouse firmware for use. It sets the X, Y lower and upper bounds to 0 and 1023 and disables the mouse firmware interrupts. INITMOUSE must be called before any other mouse routines are used (the FINDMOUSE procedure must also be called as part of initializing the mouse routines).

Command Name: ***JoyStickX***

Syntax: FUNCTION JOYSTX : INTEGER;

Description: This function returns a value between 0 and 255 for the joystick X coordinate (paddle 0).

****

Command Name: ***JoyStickY***

Syntax: FUNCTION JOYSTY : INTEGER;

Description: This function returns a value between 0 and 255 for the joystick Y coordinate (paddle 1).

****

Command Name: ***MouseClick***

Syntax: FUNCTION MOUSECLICK : BOOLEAN;

Description: This function returns the value TRUE is the mouse button is down.

Command Name: ***MouseHeld***

Syntax: FUNCTION MOUSEHELD: BOOLEAN;

Description: This function returns the value TRUE if the mouse button has been down since the last reading of its status.

****

Command Name: ***MouseMoved***

Syntax: FUNCTION MOUSEMOVED : BOOLEAN;

Description: This function returns the value TRUE if the mouse's position has changed since the last reading of its status.

****

Command Name: ***MouseX***

Syntax: FUNCTION MOUSEX : INTEGER;

Description: This function returns the value of the mouse's X coordinate (0 thru 1023).

Command Name: ***MouseY***

Syntax: FUNCTION MOUSEY : INTEGER;

Description: This function returns the value of the mouse's Y coordinate (0 thru 1023).

****

Command Name: ***PrintMouseCharacter***

Syntax: PROCEDURE PRTMOUSECHAR (ch:CHAR);

Description: This procedure prints the character passed as a mouse character at the current cursor position.

****

Command Name: ***SetMouseXY***

Syntax: PROCEDURE SETMOUSEXY (x,y:INTEGER);

Description: This procedure sets the mouse firmware coordinates to the values x and y.

Command Name: ***SetXBounds***

Syntax: PROCEDURE SETXBOUNDS (xmin, xmax : INTEGER);

Description: This procedure sets the upper and lower bounds for the X coordinate.

****

Command Name: ***SetYBounds***

Syntax: PROCEDURE SETYBOUNDS (ymin, ymax : INTEGER);

Description: This procedure sets the upper and lower bounds for the Y coordinate.

****

Command Name: ***ZeroMouse***

Syntax: PROCEDURE ZERMOUSE;

Description: This procedure zeroes the X,Y mouse coordinates on firmware.

D. Screen Management Library

Overview

The Screen Management Library contains 20 different routines. They include a mix of functions and procedures which can be incorporated into your Pascal programs. Each Screen Management routine is described on following pages.

The Screen Management routines included in this Library are:

CLREOLN	(Clear to end of line)
CLREOP	(Clear to end of page)
CLRLINE	(Clear line)
CLS	(Clear screen)
COL80	(Check for 80 column card)
CURSORX	(Return X position of cursor)
CURSORY	(Return Y position of cursor)
GETCHAR	(Return keypress character)
GOTOXY	(Move cursor to coordinates X,Y)
IDMACHINE	(Return machine ID information)
INVERSE	(Set inverse mode)
NORMAL	(Set video to normal)
ON40	(Enable 40 column display)
ON80	(Enable 80 column display)
SCROLLDOWN	(Scroll down 1 line)
SCROLLUP	(Scroll up 1 line)
SCRNBOTTOM	(Set bottom screen margin)
SCRNFULL	(Return display to full size)
SCRNTOP	(Set top screen margin)
TAB	(Move cursor to position X)

Using the Screen Management Library

To use the Screen Management routines, you must first "include" the desired routine after the variable and type declarations in your Pascal program. (Please refer to Chapter III of the Kyan Pascal User Manual for more information about the use of Include files in Pascal programs.) Once the routine is included, you can call the routines as often as needed in your program.

Notes

1. Don't forget to place a copy of all the files "included" in your Pascal program in the same working directory as the main program. If you forget, the compiler will not be able to find the file and a "File Not Found" compiler error will occur.

2. There are no global types to be declared with Screen Management routines.

3. The Screen Management routines use the following convention:

    Cursor X position: 0 thru 39 (0 thru 79 in 80 column mode)
    Cursor Y position: 0 thru 23

Command Name: ***Clear to End of Line***

Syntax: Procedure CLREOLN;

Description: Clear from the cursor to the end of the line.

****

Command Name: ***Clear to End of Page***

Syntax: Procedure CLREOP;

Description: Clear from the cursor to the end of the page.

****

Command Name: ***Clear Line***

Syntax: Procedure CLRLINE;

Description: Clear horizontal line y. Cursor does not move.

****

Command Name: ***Clear Screen***

Syntax: Procedure CLS;

Description: Clear the current text display.

Command Name: ***Column 80***

Syntax: Function COL80 : Boolean;

Description: Returns the value TRUE if the 80 column firmware is active.

****

Command Name: ***Cursor X Position***

Syntax: Function CURSORX : Integer;

Description: Return the X (horizontal) position of the cursor.

****

Command Name: ***Cursor Y Position***

Syntax: Function CURSORY : Integer;

Description: Returns the Y (vertical) position of the cursor.

****

Command Name: ***Get Character***

Syntax: Function GETCHAR : Char;

Description: Wait for a keypress and then return it as a character.

Command Name: ***Go To Position X,Y***

Syntax: Procedure GOTOXY (x,y : integer);

Description: Move the cursor to screen coordinates (X,Y). If the values passed are out of the range of the current screen (for example, an x coordinate of 65 while in forty column mode), the command is ignored.

Notes:

1. Be sure to use this GOTOXY routine and not any previously published versions. This routine automatically recognizes the four different versions of the Apple II and treats the firmware accordingly.

2. GOTOXY (0,0) moves the cursor to the top left corner of the screen.

****

Command Name: ***Identify Hardware Configuration***

Syntax: Procedure IDMACHINE (VAR version : Char; VAR card80, extend80 : Boolean);

Description: Returns information regarding the hardware configuration as follows:

version:	' ' - Apple]['+' - Apple][+ 'E' - Apple //e 'e' - Apple //e with 65c02 chips 'c' - Apple //c
card80	Returns TRUE if an 80 column card is found
extended80	Returns TRUE if the system has more than 64K of memory.

Command Name: ***Inverse Screen Mode***

Syntax: Procedure INVERSE;

Description: Set the screen to inverse mode. Note that lower case characters do not appear correctly in the 40 column mode.

****

Command Name: ***Normal Screen Mode***

Syntax: Procedure NORMAL;

Description: Return the screen mode to normal.

****

Command Name: ***Enable 40 Column Display***

Syntax: Procedure ON40;

Description: Enable the 40 column display (this procedure disables the firmware in the 80 column card).

****

Command Name: ***Enable 80 Column Display***

Syntax: Procedure ON80;

Description: Enable the 80 column display. If the system does not contain an 80 column card, this command is ignored.

Command Name: ***Scroll Down***

Syntax: Procedure SCROLLDOWN;

Description: Scroll the 80 column display down one line. The cursor position remains unchanged.

****

Command Name: ***Scroll Up***

Syntax: Procedure SCROLLUP;

Description: Scroll the 80 column display up one line. The cursor position remains unchanged.

****

Command Name: ***Screen Bottom***

Syntax: Procedure SCRNBOTTOM (x : INTEGER);

Description: Set the bottom margin of the video screen to a value between 0 and 23. This command effects the scope of the CLS command. The SCRNTOP and SCRNBOTTOM procedures are used to limit video output. Setting SCRNTOP and SCRNBOTTOM to the same value will limit the screen to single vertical line of text. The position of this line is determined by the value passed to the procedures.

Command Name: ***Screen Full***

Syntax: Procedure SCRNFULL;

Description: This procedure cancels the SCRNTOP and SCRNBOTTOM commands and returns the screen to full size.

****

Command Name: ***Screen Top***

Syntax: Procedure SCRNTOP (x : Integer);

Description: Set the top margin of the video screen to a value between 0 and 23. This command effects the scope of the CLS command. The SCRNTOP and SCRNBOTTOM procedures are used to limit video output. Setting SCRNTOP and SCRNBOTTOM to the same value will limit the screen to single vertical line of text. The position of this line is determined by the value passed to the procedures.

****

Command Name: ***Tab***

Syntax: Procedure TAB (x : Integer);

Description: Move the cursor to position X in the current horizontal line. Out of range values are ignored.

E. Other System Utilities

Overview

The Other System Utilities Directory contains the following routines.

- o Random Number Routines

REAL	(Generates a random number between 0 and 1)
RANDOM	(Generates a random integer in range, min..max)
SEED	("Seeds" the random number generator)

- o Conversion Routines

  REAL NUMBER TO STRING
  INTEGER TO STRING
  STRING TO REAL NUMBER
  STRING TO INTEGER

- o Line Parse Routine

- o Sort/Merge Routine

Using Other System Utilities

To use the Other System Utilities, you must first "include" the global type declarations (if any) and the desired routines after the variable and type declarations in your Pascal program. (Please refer to Chapter III of the Kyan Pascal User Manual for more information about the use of Include files in Pascal programs.) Once the routine is included, you can call the routine as often as needed in your program.

Note: Don't forget to place a copy of all the files "included" in your Pascal program in the same working directory as the main program. If you forget, the compiler will not be able to find the file and a "File Not Found" error will occur.

Random Number Routines

There are three routines in this group. They can be used in your Pascal programs to generate random numbers.

There are no global types associated with these routines.

****

Command Name: ***Random Number 1***

Syntax: FUNCTION RND: REAL;

Description: Generates a real random number between 0 and 1.

****

Command Name: ***Random Number 2***

Syntax: FUNCTION RANDOM (min, max : INTEGER) : INTEGER;

Description: Returns a random integer between min and max.

Note: *Random Number 2* utilizes *Random Numbner 1* (FUNCTION RND) in its source code. As a result, you must be certain to include a copy of the *Random Number 1* routine in any programs which use *Random Number 2.*

Command Name: ***Seed Random Number***

Syntax: PROCEDURE SEED (seed1 seed2, seed3 seed4:
INTEGER);

Description: This routine is used in conjunction with either of the random number generators to "seed" the string of random numbers generated. Using this routine, it is possible to fix the starting value of the random number sequence.

To "seed" the Random Number Generator, you first include the Seed and Random Number procedures in your Pascal program. Then, you specify four integers of your choosing (i.e., seed1, seed2, seed3, seed4). When the program runs, the Random Number Generator takes these four values, inputs them into its polynomial equation, and generates a sequence of random numbers. Everytime the program is run, the Random Number Generator produces the same sequence of random numbers. To change the sequence, you simply change one or more of the seed values.

Conversion Routines

This group contains four conversion routines and one global type file.

The global types can be declared by adding the following lines of code to the declarations portion of your Pascal program or by including the file ***CONV.TYPES.I*** found on the System Utilities disk.

```
STRING6  = ARRAY[1.. 6] OF CHAR;
STRING20 = ARRAY[1..20] OF CHAR;
```

****

Command Name: ***Real to String Conversion***

Syntax: PROCEDURE REALTOSTR (VAR number:REAL; leading,
decpt: INTEGER; VAR result: String20);

Description: This routine returns a STRING20 type in the format indicated by "leading" and "decpt". "Leading" is the number of characters to use for the leading digits in the resulting string. "Decpt" is the number of decimal places allowed for expansion. For example:

```
REALNUM:= 35932.382;
REALTOSTR (REALNUM, 10, 5, ANSWER);
WRITELN (ANSWER);
```

will output: 35932.38200   (note the five leading spaces)

Notes:

1. Extra space must be left for negative signs in the "leading" specification. Also, if you specify "leading" to be zero, scientific notation will be used for output.

2. If the real number is too large to fit into the string, the string returned will be filled with # symbols. Also, if the leading characters fit but the number of decimal places do not, then as many numbers to the right of the decimal point that will fit will be used.

Command Name: ***Integer to String Conversion***

Syntax: PROCEDURE INTTOSTR (number: INTEGER; justify: CHAR; VAR result: String6);

Description: This routine converts the integer passed into string. A leading minus is used when the value is negative. Justification characters are:

R	Right justify number, buffering to left with spaces
Z	Right justify number, buffering to left with zeros
L	Left justify with spaces to right (default).

Notes:

1. Any unrecognizable justify characters are treated as Left.

2. The justify character passed must be a capital letter.

****

Command Name: ***String to Real Conversion***

Syntax: FUNCTION STRTOREAL (VAR number: String20) : REAL;

Description: This routine converts a string passed to a real number.

Notes:

1. Non-numeric characters are ignored.

2. The first decimal point encountered is used for conversion.

3. Negative numbers are valid if the first character in the string is a "-".

Command Name: ***String to Integer Conversion***

Syntax: FUNCTION STRTOINT (VAR number: String6) : INTEGER;

Description: This routine converts a string passed to an integer.

Notes:

1. All non-numeric characters are treated as zeroes.
2. A leading minus will give the INTEGER a negative value.

Line Parsing Routine

The Line Parsing routine gives you a method by which to read and "parse" parameter inputs to a program. The source of this input can be the keyboard or another Pascal program. The line parser reads the input string in the Apple input buffer (location $200 to $2FF); it then looks for spaces and breaks the string into records (words); next, it puts these records into a linked list; and, finally, it returns a pointer which identifies the location of the first record in the linked list.

To use the line parsing routine, you must first declare certain global types in your Pascal program. You can declare these global types by adding the following code to the global declarations portion of your Pascal program or by including the ***Parse.Types.I*** file found in the Other Utilities directory.

```
String127 = ARRAY [1..127] of CHAR;
StrPointer = ^StrRecord;
StrRecord = RECORD
                StrFound : String127;
                NextStr : StrPointer
            END;
```

****

Command Name: ***Line Parse Routine***

Syntax: FUNCTION PARSELINE : StrPointer;

Description: This routine returns a pointer to a linked list containing the records or words found in the line passed. The records are considered terminated when they are followed by at least one space (blank). If a blank line is passed to PARSELINE, the pointer 'ParseLine' will point to NIL.

Merge and Sort Routines

The merge and sort routines are very handy for organizing your files. The MERGE procedure will combine up to five presorted files into a single file that is in alphabetically and/or numerically ascending or descending order. The SORT procedure will arrange a file of any type of record into alphabetical or numerical order.

Global Declarations

To use one, or both, of the routines in a Pascal program, you must first Include the file "SRTMERG.TYPES.I", which declares the data types for both of the procedures. This include file declares the following:

```
PATHSTRING = ARRAY [1..65] OF CHAR;
NAMEARRAY = ARRAY [1..7] OF PATHSTRING;
FIELD_TYPE  = (ALPHA_FIELD, INTEGER_FIELD,
               REAL_FIELD);
```

MERGE also requires the declaration of a VARiable of type NAMEARRAY in which pathnames will be stored. You can declare your own or include the file "SRTMERG.VARS.I" into the global VAR section of your program. For convenience sake, we will assume you have included the SRTMERG.VARS.I file and are using MERGENAMES as your global VARiable of type NAMEARRAY.

Using the MERGE Routine

The MERGE procedure takes between two and five ordered data files, sorts records as they are encountered, and produces one large resultant file containing those merged records. You must specify which files are to be used as source, and the name of the 'intermediate' (or temporary) file for the completely merged image. You even have the option to specify a second destination file.

The MERGE procedure is stored in file "MERGE.I." The procedure is declared as follows:

```
PROCEDURE MERGE (VAR MERGENAMES: NAMEARRAY;
      SELECT, FNUM, RLEN, KLEN, OSET, ORDER: INTEGER;
      KTYPE: FIELD_TYPE);
```

The Parameters are:

MERGENAMES: The MERGE procedure permits you to sort/merge up to five data files. The names of these files must be stored sequentially in MERGENAMES[1..5], starting at element 1. MERGENAMES[6] must contain the pathname of a temporary file to which the MERGE procedure will write out the merged file. The pathname in MERGENAMES[6] must be a valid pathname (if it is not, MERGE will fail immediately). MERGENAMES[7] may contain a different name for the resulting data file if you wish, or be left blank, depending on the value of SELECT below.

SELECT: SELECT is an index to array MERGENAMES; it must be in the range of 1 to 7 inclusive. SELECT indicates what filename to use as a destination file for the resulting merge output file. If SELECT has a value between 1 and 5, the corresponding data file pathname will be used to write the merged file to, thus replacing the data file contents. If SELECT is 6, the temporary filename indicated by MERGENAMES will be left the only output as a result of the MERGE call. If SELECT is 7, the pathname stored in MERGENAMES[7] will be used as a final output destination by MERGE.

FNUM: FNUM is the number of data files to be merged together by the MERGE procedure. Think of this number as an index to the last pathname in the MERGENAMES array you want merged. FNUM must have a value between 2 and 5 inclusive.

RLEN: RLEN is the record length in bytes. In general, record length is fairly easy to calculate. For more information on calculating record lengths and storage sizes by types please consult the Assembly Language programming section of your Kyan Pascal User Manual.

KLEN: KLEN is the length of the key record field in bytes. If you are sorting with a key field of either REALs or INTEGERs, KLEN is automatically set according to type (8 for REAL or 2 for INTEGER). However, using a key made up of CHARacters (alpha_field) will cause the comparison of the keys to take place against KLEN number of characters. KLEN cannot be longer than 255 bytes.

OSET: OSET is the byte offset of the first byte of the key field in the record. OSET can be thought of as the number of bytes found in the record before the first byte of the key field. Therefore, if the key field in your record was the first field declared, OSET would be passed as a 0, since there are no bytes before the key field in that record layout.

ORDER: ORDER determines in what fashion the resulting sorted file's records will be stored. If ORDER is negative, the records will be sorted in descending (highest first) order. If ORDER is non-negative (zero or positive), the records will be sorted in ascending order (lowest first).

KTYPE: KTYPE indicates the type of key field you have specified. If you are using a key that is a character or an array of characters, specify ALPHA_FIELD as KTYPE. If you are using INTEGERs, specify INTEGER_FIELD; if sorting against real numbers use REAL_FIELD as KTYPE.

Using the ESORT routine

The ESORT routine requires the following:

1. The SRTMERG.TYPES.I file be included as global types
2. The SRTMERG.VARS.I file be included as global variables
3. The MERGE.I file be included in the Pascal host program previous to the ESORT procedure.

The global VARiables declared in SRTMERG.VAR.I are:

```
FYLE                : PATHSTRING;
MERGENAMES          : NAMEARRAY;
KTYPE               : FIELD_TYPE;
RLEN, OSET,
ORDER, KLEN,
FNUM, SELECT        :INTEGER;
```

Each variable listed must be conditioned before calling the ESORT routine.

ESORT should be used when one data file containing records with key fields must be sorted. This is accomplished by following these steps:

1. Assign the name of the file to be sorted to the global variable FYLE. Note that the name of the file to be sorted cannot be longer than 62 characters (ESORT needs the remaining bytes in order to append file suffixes to the pathname specified)

2. The global variables RLEN, KLEN, OSET, ORDER, and KTYPE must be assigned values corresponding to those explained in the MERGE documentation.

3. Call ESORT (remember - ESORT has NO PARAMETERS!) The parameters in MERGE and the global variables used by ESORT have the same name. However, you should always remember to assign the global variables their correct values before calling ESORT.

F. APPENDIX

UTILITIES DISK DIRECTORY

Volume Name: UTILITY.TOOLKIT

Directory: ProDOS.LIB

Include Files:

PRODOS.TYPES.I (Global Types)
DELETE.I
RENAME.I
COPY.I
SETPREFIX.I
GETPREFIX.I
APPEND.I
LOCK.I
UNLOCK.I
MAKEDIR.I
REMDIR.I
GETDIR.I
FIND.I
SCANFILE.I
FILETYPE.I
BSAVE.I
BLOAD.I
FORMAT.I
GETCLOCK.I
GETTIME.I
GETDATE.I
SETCLOCK.I
SETTIME.I
SETDATE.I
FINDCLOCK.I
PRTMLIERROR.I
PRINTFILE.I

Directory: DEVICE.LIB

Include Files:
FINDMOUSE.I
INITMOUSE.I
MOUSECLICK.I
MOUSEHELD.I
MOUSEMOVED.I
MOUSEX.I
MOUSEY.I
ZEROMOUSE.I
SETMOUSEXY.I
SETXBOUNDS.I
SETYBOUNDS.I
HOMEMOUSE.I
ENDMOUSE.I
PRTMOUSECHAR.I
BUTTON0.I
BUTTON1.I
JOYSTX.I
JOYSTY.I

DIRECTORY: SCREEN.LIB

Include Files:
CLS.I
GOTOXY.I
TAB.I
INVERSE.I
NORMAL.I
SCROLLUP.I
SCROLLDOWN.I
CLRLINE.I
CLREOLN.I
CLREOP.I
COL80.I
CURSORX.I
CURSORY.I
GETCHAR.I
SCRNTOP.I
SCRNBOTTOM.I
SCRNFULL.I
IDMACHINE.I
ON40.I
ON80.I

Directory: OTHER.LIB

Include Files:

CONV.TYPES.I (Global Types)
REALTOSTR.I
STRTOREAL.I
INTTOSTR.I
STRTOINT.I

SEED.I
RND.I
RANDOM.I

SRTMERG.TYPES.I (Global Types)
SRTMERG.VARS.I (Global Variables)
ESORT.I
MERGE.I

PARSE.TYPES.I
PARSELINE.I

Directory: DEMO.LIB

CATALOG.P (Source Code)
CATALOG (Object Code)

MOUSE.DEMO.P (Source)
MOUSE.DEMO (Object)

RANDOM.DEMO.P (Source)
RANDOM.DEMO (Object)

ESORT.DEMO.P (Source)
ESORT.DEMO (Object)

MERGE.DEMO.P (Source)
MERGE.DEMO (Object)

MOUSETEXT.DEMO (Object)

TURTLE.DEMO (Object)

Suggestion Box

We do our best to provide you with complete, bug-free software and documentation. With products as complex as compilers and programming utilities, this is difficult to do. If you find any bugs or areas where the documentation is unclear, please let us know. We will do our best to correct the problem in the next revision. We would also like to hear from you if have any comments or suggestions regarding our product.

To help us better understand your comments please use the following form in your correspondence and mail it to: Kyan Software Inc.,
1850 Union Street #183, San Francisco, CA 94123.

Name____
Address______________________________________
City______________________State______ZIP__________
Telephone:
(day)____________________ (evening)______________

Kind of Problem

__ Software Bug
__ Documentation Error
__ Suggestions
__ Other ____________________

Software Description

Product Name ______________
Version No. ______________
Date Purchased ______________

Kyan Software Products You Use

__ Kyan Pascal
__ System. Utilities Toolkit
__ MouseText Toolkit
__ TurtleGraphics Toolkit
__ Kyan Macro Assembler/Linker
__ Advanced Graphics Toolkit
__ MouseGraphics Toolkit
__ Other ______________

Your Hardware Configuration

Type/Model of Computer ____________________________
How many and what kind of disk drives __________________
What is your screen capability: ___40 Column ___80 Column
How much RAM?____K (what kind of RAM Board?____________)
What kind of printer and interface card do you use?____________
____
What kind of modem?_____________________________________
Other information about your computer system: ______________
____

What do you use this software for?
___ Education (I am a __ teacher __ student)
___ Hobby
___ Professional Software Development
___ Other ____

Problem Description (if appropriate, please include a disk or program listing).

____
____
____
____
____
____
____
____
____

Suggestions

____
____
____
____
____
____
____
____
____
____
____
____
____
____

kyan

Quick Guide to Kyan Pascal

KIX™ Command Summary

Command	Description	Options/Arguments	
C40	Set 40 Column Mode		
C80	Set 80 Column Mode		
CAT [ ]	Display File Contents	-n	Number Output Lines
		-b	Skip Blank Lines
		-s	Remove Blank Lines
		-v	Print ASCII Equiv. of Control Chars.
CD [ ]	Change Directory		
CFG	System Configuration Program		
CHMOD [ ]	Add/remove File Access (+ add / - remove)	+/- r	Read
		+/- w	Write
		+/- d	Deletion
		+/- n	Rename
CMP f1 f2	Compare Files	f1,f2 =	File1,File2
CMP (s,d) (s,d)	Compare Volumes	(s,d) =	(Slot,Drive)
CP [ ] [ ]	Copy Files	-i	Destination Replace
CPV (s,d) (s,d)	Copy Volume	(s,d) =	(Slot,Drive)
DATE [ ]	Set System Date/Time		
ECHO [ ]	Print Pathname Evaluations.		
FIND d-f	Print File Pathnames	d	Directory List
		-f	File to Be Found
FORMAT (s,d)[ ]	Format Disk, Volume Name	(s,d) =	(Slot,Drive)
GREP a b	Search for Pattern	a	Pattern or String
		b	List to Be Searched
KIX	KIX Command Summary		
LPR [ ]	Print Files Listed		
LS /	List Volumes		
LS [ ]	List Contents of File or Directory	-l	Extended Directory
		-f	List File Types
		-p	List Protection Status
		-n	Don't Sort Filenames
		;type	List Only Files of Type Specified
MENU	System Main Menu		
MKDIR [ ]	Make Directory		
MV [ ] [ ]	Rename or Move Files	-i	Prompt for Rename
		-f	Rename Locked Files

KIX™ Command Summary Continued

Command	Description	Options/Arguments	
PWD	Print Working Directory		
QUIT	Exit KIX to ProDOS Quit		
RM [ ]	Remove Files	-i	Prompt for Delete
		-f	Delete Locked Files
		-r	Empty Directories
RMDIR [ ]	Remove Directory		
SD	Screen Dump to Printer		
SDIFF f1 f2	Compare Files & Display Differences	f1,f2 =	File1,File2

Kyan Text Editor

Cursor Movement Commands

<CNTL>-S	Move Cursor 1 Space to Left
<CNTL>-D	Move Cursor 1 Space to Right
<CNTL>-A	Move Cursor 1 Word to Left
<CNTL>-F	Move Cursor 1 Word to Right
<CNTL>-E	Move Cursor 1 Line Up
<CNTL>-X	Move Cursor 1 Line Down
<CNTL>-R	Move Cursor 20 Lines Up
<CNTL>-C	Move Cursor 20 Lines Down
<CNTL>-T	Move Cursor to TOP of File
<CNTL>-V	Move Cursor to BOTTOM of File

Character Deletion Commands

<CNTL>-G	Delete Character Cursor is On
<CNTL>-Q	Delete Character to Left of Cursor (EQUIVALENT TO THE APPLE II DELETE KEY).
<CNTL>-Y	Delete Line Cursor is On

Cut and Paste Commands

<CNTL>-O	Start or Finish Cut.
<CNTL>-P	Paste Copy.
<CNTL>-L	Save Cut Text in New File

Search Commands

<CNTL>-Z	Move Cursor *Forward* to Next Occurrence.
<CNTL>-W	Move Cursor *Backward* to Next Occurrence.

Command Syntax: ED Pathname

KIX is a Trademark of Kyan Software, Inc.

Kyan Pascal Compiler

Compiler Options

-O pathname	Assign New Pathname to Compiled File
-S	Generate Assembly Source Code File Only
-D	Print Line Number and Filename On Runtime Error.

Redirection Option

> n	Direct Output to Slot n (n=1..7)
> pathname	Send Output to File

Command Syntax: PC Pathname - Options

Kyan Assembler

Assembler Options

-l	Print a Listing.
-0 pathname	Assign New Pathname to Output File.

Redirection Options

> pathname	Direct Output to a File
>n	Direct Output to Slot n (n=1..7)

Command Syntax: AS Pathname - Options

Assembler Directives

Symbol	Description
ORG	**Origin:** Assembled code should start at the specified location in memory
EQU	**Equate:** Assign a value to the label whenever it appears in the program
DB/DW	**Define Byte** and **Define Word:** Indicates the location in memory of a string or table.
>	**Least Significant Byte (LSB):** Indicates least significant byte of a 2-byte hex number.
<	**Most Significant Byte (MSB):** Indicates most significant byte of a 2-byte hex number
DS	**Define Storage:** Saves space for number of bytes in the expression field.
STR	**String:** Counts characters in an expression field and puts number in first byte followed by the ASCII values of each character.
IFDEF	**If Defined:** Assembles code if identifier in the expression field is defined
IFNDEF	**If Not Defined:** Assembles code if identifier in the expression is not defined

Assembler Directives *Continued*

Symbol	Description
IFEQ	**If Equal:** Assembles code if the expression is equal to zero
IFNE	**If Not Equal:** Assembies code if expression is not equal to zero
ELSE	**Else:** Follows one of the IF- directives and reverses the conditional assembly
ENDIF	**End If:** Ends the conditional assembly associated with IF- or IF- ELSE directives
INCLUDE	**Include** file in expression field
LST ON	**Listing On:** Turns on listing at that point.
LST OFF	**Listing Off:** Turns off the listing.
DSECT	**Data Section:** Defines memory for data only.
DEND	**Data End:** Ends memory reserved for data only.
MACRO	**Macro** definition follows.
ENDM	**Macro** definition ends
MEX ON	**Macro EXpansion ON** for listing.
MEX OFF	**Macro EXpansion OFF** for listing.
SYS	**SYStem:** Makes executable file a system file.
ASC	**ASCii:** Puts ASCII values of strings in expression field following the directive
DFLAG	**Define FLAG:** Used with IFDEF and IFNDEF to assemble code required by already assembled macros or code segments.

Reference to ISO Standard Pascal

Data Types

STRUCTURED:	Array, File, Set, Record
POINTERS	
SIMPLE:	Real Ordinal ..Enumerated ..Predefined (Boolean, Integer, Char) ..Subrange

Standard Identifiers

CONSTANTS:	False, MaxInt, True
TYPES:	Boolean, Char, Integer, Real, Text
VARIABLES:	Input, Output
FUNCTIONS:	Abs, ArcTan, Chr, Cos, Eof, Eoln, Exp, Ln, Odd, Ord, Pred, Round, Sin, Sqr, Sqrt, Succ, Trunc
PROCEDURES:	Dispose, Get, New, Pack, Page, Put, Read, Readln, Reset, Rewrite, Unpack, Write, Writeln

Table of Symbols

SPECIAL SYMBOLS

+	-	*	/	=	(*	*)
<	>	<=	<=	<>		
.	,	:	;	:=		
(	)	[	]	↑	{	}

WORD SYMBOLS (RESERVED WORDS)

and
array
begin
case
const
div
do
downto
else
end
file
for
function
goto
if
in
label
mod
nil
not
of
or
packed
procedure
program
record
repeat
set
then
to
type
until
var
while
with