CLSN Pascal

for the 130XE

Table of Contents

Introduction

Congratulations for purchasing CLSN Pascal Version 1.0. With CLSN Pascal you will step into the world of higher level structured languages and discover many of the advantages they have over BASIC and ACTION!

At the same time you will not be subject to constantly swapping disks and reloading a compiler and editor in order to get a program running. CLSN Pascal is designed to circumvent all those difficulties by giving you a complete, integrated programming environment.

This manual, however, is not meant to be a Pascal tutorial, but rather a point of reference for this particular dialect of Pascal along with the inside technical information of CLSN Pascal.

We at CLSN Software hope CLSN Pascal continues to make you realize the power still available on the Atari line of computers.

Copyright

The compiler, editor, and manual are Copyright 1989 CLSN Pascal and are protected by United States copyright law. A back-up copy of CLSN Pascal should be made, but under no other circumstances should this program be copied.

Programs written in CLSN Pascal require no licensing and may be distributed or sold.

Warranty

CLSN Software warrants that the physical diskette enclosed shall be free of defects in materials and workmanship for a period of 30 days from the date of purchase. In the event of notification within the warranty period of defects in material or workmanship, CLSN Software will replace the defective diskette.

CLSN Software specifically disclaims all other warranties, expressed or implied. In no event shall CLSN Software be liable for any loss of profit or any other commercial damage, including but not limited to special, incidental, consequential or other damages.

System Configuration

CLSN Pascal is designed to run on an Atari 130XE computer with 128k memory along with at least one single density disk drive and DOS 2.5.

The Editor

CLSN Pascal's editor is designed specifically for programming pascal, but it may be used on its own as a rudimentary word processor. Up to 16k of text (about 16000 characters) may be entered into the editor at once.

If you wish, you may write your programs in another editor and INCLUDE them during compilation. The only requirement is that each source line is delimited by a carriage return (#155) and that the lines themselves do not exceed 128 characters.

The editor is divided into three parts, the command line, the status line and the text window. The command line contains various editor and compiler commands. The status line reveals the current file and the current column and row of the cursor. The text window is where the source code of the program is created.

Getting around the Editor

Entering text is accomplished much like a typewriter; the text is typed followed by a carriage return.

Many of the Atari's basic editing commands are available. The following table summarizes the keys and their operations.

Key	Operation
ESC	Move to command line
CTRL -	Move cursor up one line
CTRL =	Move cursor down one line
CTRL +	Move cursor left one column
CTRL *	Move cursor right one column
TAB	Move cursor right four spaces
BKSP	Delete character left of the cursor
RETURN	Move to beginning of next line
SHIFT RETURN	Split the line at the cursor
SHIFT >	Insert a line at the cursor
SHIFT BKSP	Delete a line at the cursor
OPTION -	Move up a page
OPTION =	Move down a page
OPTION +	Move to beginning of line
OPTION *	Move to end of line
OPTION >	Insert block
SELECT -	Move to top of text
SELECT =	Move to bottom of text
START 1	Toggle screen during compilation

Special Keys

[SHIFT RETURN] splits the current line at the cursor position.

Pressing [BKSP] while in the first column will concatenate the current line with the one above it.

Moving/Copying/Deleting Blocks of Text

These three facilities, essential to any editor, are accomplished by the same procedure. The method is to move above the block to be affected, then using [SHIFT BKSP] (and staying on the same line), delete the entire block up into the cursor. This process removes the block from the text, but at the same time copies the data into a paste buffer where it can be copied or moved using [OPTION >]. Successive copies can also be made by using [OPTION >]; the paste buffer will remain intact until the next [SHIFT BKSP] takes place. The past buffer will even survive the NEW command.

The block operations are summarized below:

Block Operation	Method
Delete	Move cursor above block and use [SHIFT BKSP] to delete each line of the block.
Move	Move cursor above block and use [SHIFT BKSP] to delete each line and move it into the paste buffer. Move cursor to new position for block and press [OPTION >].
Copy	Move cursor above block and use [SHIFT BKSP] to delete each line and move it into the paste buffer. Press [OPTION >] to reinsert the block. Move to the new position and press [OPTION >] to make a copy.

There are two more commands which will help you when creating your source code, FIND and REPLACE, both of which are located on the menu bar. These commands are detailed below.

The Command Line

The command line consists of various commands for editing and compiling programs. To get to the command line from within the editor, simply press [ESC]. A command may be chosen by moving the cursor around the command line with the direction keys, and by pressing [RETURN] when above the desired option. Pressing the first letter of the command will also execute the command.

The commands available are summarized below. More detailed information concerning the commands are given following the table.

Command	Operation
Edit	Return to editor
Load	Load a file from disk
Save	Save a file to disk (or device)
New	Clear editor
Quit	Exit to DOS
Compile	Compile program in memory
Run	Run program in memory
Find	Find a specified string in the text
Replace	Find and replace a specified string in the text
Write	Write the compiled form of the program to disk

Edit

Edit simply returns you to the editor at the place you left off. Pressing [ESC] will do the same.

Load

Load moves a file from disk to memory.

After choosing this command, you will be prompted for a filename which to load; it should follow standard DOS convention. If no device (D:, D2:) is specified, D1: is assumed. If no extender is given, .PAS is assumed. Pressing [RETURN] will load the file.

If the file to load is not known, a directory may be requested by pressing [SHIFT 1] or [SHIFT 2] (for drive 1 or 2) when asked for a filename. This will clear the text window and display all the files (up to 50) in the text window. The appropriate file may then be chosen by moving the cursor atop it with the cursor keys and pressing [RETURN].

The specified file will then be loaded.

NOTE: This command is actually more like MERGE because it loads the text at the position of the cursor and deletes everything else below. This is convenient to concatenate two files, or perhaps merge a temporary text file for viewing. If this is not desired, precede this command by NEW, or move to the top of the text by pressing [SELECT -].

Save

Save moves the text in memory to disk.

After choosing this command you will be prompted for a filename which to save; it should follow standard DOS convention. If no device (D:, D2:) is specified, D1: is assumed. If no extender is given, .PAS is assumed. Pressing [RETURN] will save the file.

If the file to save is not known, a directory may be requested by pressing [SHIFT 1] or [SHIFT 2] (for drive 1 or 2) when asked for a filename. This will clear the text window and display all the files

(up to 50) in the text window. The appropriate file may then be chosen by moving the cursor atop it with the cursor keys and pressing *[RETURN]*.

The file will then be saved under the given name.

New

New clears the editor of any text.

After choosing this command, you will be asked, "Are you sure?" in order to verify the command. Pressing "Y" will clear the text; pressing any other key will abort the instruction.

Quit

Quit leaves CLSN Pascal and calls DOS.

After choosing this command, you will be asked, "Are you sure?" in order to verify the command. Pressing "Y" will exit to DOS; pressing any other key will abort the instruction.

Compile

Compiles the program in memory.

The compiler will attempt to translate the program in memory into executable code. As it does so, the current line it is working on is displayed on the status line. If the screen has been set off using [START 1], the screen will go black during compilation.

If compilation is successful, it will simply return to the command line. If an error does occur, a message stating the error will appear on the status line, and the compiler will wait for a key to be pressed. It will then reveal the location of the error.

Turning the screen off while compiling increases the speed of the compiler 30%. This switch can be toggled by pressing [START 1] while in the editor. A message will state whether the screen will be on or off during the next compilation.

Run

Executes the compiled program in memory.

Run transfers control to the compiled program in memory. If the program has not been compiled a message stating so will appear.

NOTE: Modifying the source code does not affect the previously compiled code. This means that if the program is not recompiled, the code from the previous compilation will be executed.

Find

Find locates a specified string in the source text.

After choosing the command, the editor will request a string to find. If this command was used before the string entered then will be present. Pressing [RETURN] will choose the old search string, otherwise, typing a new string will remove the old one. Pressing [ESC] will abort the command.

Pressing [RETURN] will initiate the search starting at the current cursor position and moving downward.

Find only finds exact matches (i.e. "VAR" will not match "var")

If the string is found, it will be highlighted within the text, and you will be asked, "Search Again?" A "Y" will search again, any other key will abort the search.

If the string is not found, a message stating so will be displayed on the status line.

Replace

Replaces a specified string in the text with another.

This command will first ask for the string to find in the same manner as FIND above. Next, the replacement string will be requested. [ESC] will abort this command.

Once [RETURN] is pressed, the editor will search for the string. If it is found, you will be asked, "Replace?" A "Y" will replace the string, any other key will not. The editor will then ask, "Search Again?" A "Y" will repeat the search, any other key will stop the find/replace.

If the search string is not found, a message stating so will appear on the status line.

Write

Stores the compiled form of the program in memory on disk.

Write uses the name of the current file, but adds the extender ".OBJ", then writes the compiled code to the disk. The resulting file can then be run from DOS using the binary load command, or may be renamed AUTORUN.SYS so it may be automatically executed when the disk with the file is booted.

A compiled program generated with CLSN Pascal can be run completely on its own.

If the program has not been compiled, a message stating so will appear on the status line.

The Compiler

Syntax

CLSN Pascal is slightly more stringent on syntax than Standard Pascal. The result is that wherever a semicolon is optional in Standard Pascal, it is required in CLSN Pascal (i.e. a semicolon must follow the statement before an END). This does not apply to the IF and CASE statements where an optional semicolon determines if there is an ELSE clause.

Identifiers

Identifiers can be 127 characters long, all of which are significant. The first character of an identifier must be a letter or an underscore but the following characters can be letters, digits, or underscores.

CLSN Pascal is not case sensitive, so the identifiers "SCORE", "score" and "ScOrE" are all equivalent.

Labels

Labels (for use with the GOTO statement) can be a sequence of digits (with the leading zeros significant) or an identifier.

Reserved Words

What follows is a list of words that are reserved by CLSN Pascal and may not be used as identifiers:

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

Comments

Comments may be embedded anywhere within the text in order to better describe the code; they are ignored completely by the compiler. A comment must begin with "(*" and end with "*)". Standard Pascal allows comments to be enclosed between two braces but since the Atari character set does not have braces the previously mentioned format must be used.

Program Heading

The program heading, although it must follow the correct syntax, does absolutely nothing. The program name and its parameters do not even need to be unique.

Types

There are six different types in CLSN Pascal, ordinal, real, string, pointer, structured. These are discussed below.

Ordinal

Ordinal types may be of a subrange type or an enumerated type. There are seven predefined ordinal types in CLSN Pascal. The basic ordinal types are CHAR and LONGINT. Except for CHAR and BOOLEAN, all other predefined ordinal types are a subrange of LONGINT. The types and their definition follow.

Identifier	Type	Definition
char	Ordinal	Set of all ASCII characters
boolean	Enumerated	(FALSE, TRUE)
byte	Subrange	0..255
shortint	Subrange	-128..127
word	Subrange	0..65535
integer	Subrange	-32768..32767
longint	Ordinal	-8388608..8388607

Examples

```
type
    upper = 'A'..'Z';
    numbers = (one, two, three);
```

Real

The REAL type is a subrange of the mathematical set of real numbers. It has a range of approximately -9.99999999E-98 to 9.99999999E+98 with 9 or 10 significant figures.

String

The STRING type is a structured type that refers to a group of characters that have a variable length. The length of the string is contained in the zeroeth element of the string. The memory size and maximum length of a string is set with its type declaration. If no size is specified the largest string available, 255 characters long, is assumed.

Example

```
type
    short_str = string[16];
    long_str  = string;      (* 255 Characters *)
```

Pointer

The pointer type contains a index to a dynamic variable. The procedures NEW, GETMEM, DISPOSE, and FREEMEM are used in conjunction with pointers to create dynamic variables. A pointer may forward reference its type, but only within the current block.

Nil is a pointer value compatible with any pointer type and has the physical value of zero. It is an error to dereference a pointer variable with the value Nil.

CLSN Pascal has a predefined pointer type, namely POINTER, which is compatible with any pointer type.

Examples

```
type
    bptr = ^byte;
    sptr = ^string;
```

Structured

CLSN Pascal has four different structured types, array, set, record, and file.

Array

Arrays are linear structures that allow a variable to take on more than one value by having the different values accessed by an index. Except for memory, there is no limit on the dimension of array structures.

CLSN Pascal has restrictions on how a multidimensional array is declared. Consider the following type declaration:

```
type
  maze = array [1..10, 1..10] of char;
```

Although legal in Standard Pascal, it is not legal in CLSN Pascal. The above example should be declared as:

```
type
  maze = array [1..10] of array [1..10] of char;
```

Although it is a bit more wordy, it is still completely compatible with Standard Pascal.

An ARRAY of CHAR can be considered to be a a string of a fixed length and is compatible with the STRING type with the exceptions that a STRING type cannot be assigned to an ARRAY of CHAR.

Example

```
type
    list_type = array [1..10] of real;
    scr_type = array [1..24] of array [1..40];
    initial = array [1..3] of char;
```

Set

A set is a structure that can hold up to 256 values of an ordinal type. An empty set is defined as an open-close bracket sequence, [], and is compatible with any set type.

Example

```
type
    ascii = set of char;

    dir_type = (up,down,left,right)
    dir_list = set of dir_type;
```

Record

The record structure defines a group of variables of different types under one large structure. CLSN Pascal does not support a record with a variant part.

Example

```
type
    id_rec = record
               name: string[20];
               code: longint;
             end;

    time_rec = record
                  second: 0..59;
                  minute: 0..59;
                    hour: 1..12;
                    when: (am, pm);
               end;
```

File

Files may be of any type except FILE. CLSN Pascal does not support untyped files.

Example

```
type
    id_file    = file of id_rec;
    score_file = file of integer;

var
    scores: score_file; (* File of Integers *)
   id_list: id_file;    (* File of ID Records *)
    words: text;        (* Text File *)
```

Type Matching

CLSN Pascal is also stringent on type matching. Consider the following constructs:

```
var
  a: array [1..10] of byte;
  b: array [1..10] of byte;
```

These two variables, although their declaration is equivalent, are completely incompatible. In order to make them compatible, the variables should be declared in the same statement, as in the following example:

```
var
  a,b: array [1..10] of byte;
```

The variables are now compatible, but not much else is. The proper way to declare these variables follows:

```
type
  list = array [1..10] of byte;

var
  a, b: list;
```

These variables are now compatible, their type is explicitly known, and it is overall much nicer to read.

Variables

Variables may be defined to reside at a specific location in memory, (i.e. a player missile register) or atop another variable by use of the ABSOLUTE clause.

Example

```
var
    hposp0: byte absolute 53248; (* Column of Player 0 *)

       str: string;
    strlen: byte absolute str;   (* Length of Str *)
```

Standard Pascal references elements of an array by separating the indices by commas or brackets. CLSN Pascal does not support the former syntax. If, for example, MAZE is a two dimensional array, in order to access an element of the array, the syntax must be MAZE[y][x] not MAZE[y, x].

NOTE: Because of speed considerations, CLSN Pascal does not perform range checking on array indices, variable assignments, or string lengths.

Typed Constants

Typed constants are variable declarations with initialized values. The values are initialized once at the start of the program. CLSN Pascal allows any structure to be filled with values except for the FILE type.

The length of an initialized string does not have to match its declared maximum length.

To initialize an array, specify the values of the initial values between parenthesis and separated by commas. There must be a value for each array element.

When initializing a record, all fields must be initialized in the order of the fields in the record. A field is initialized by citing its name followed by a colon and its initial value. All the fields should be enclosed between parenthesis and separated by semicolons.

The only value a pointer may be initialized to is Nil.

Be careful of the following:

```
const
    e = 2.718281828;
    e: real = 2.718281828;
```

Although both statements are perfectly legal, they are not equivalent. The first assignment creates a real CONSTANT whose value may not be modified and which can be assigned in a constant declaration. The second assignment creates a real VARIABLE and generates six bytes of code for the real value.

The following examples should serve to show how to correctly initialize any typed constant.

```
type
    person_rec = record
                     name: string[20];
                     code: longint;
                 end;
```

```
(* Simple Constants *)

const
       five  =    5;   (* Byte *)
       bell  = #253;   (* Character *)
       note  =  'C';   (* Character *)
       half  =  0.5;   (* Real *)
       page6 = $0600;  (* Integer (Hex is acceptable!) *)

(* Typed Constants *)

const
         yes: string[3] = 'Yes';
       title: string[20] = 'CLSN Pascal';

         fib: array [1..8] of integer = (0, 1, 1, 2, 3, 5, 8, 13);
    identity: array [1..3] of array [1..3] of real =
                  ((1.0, 0.0, 0.0),
                   (0.0, 1.0, 0.0),
                   (0.0, 0.0, 1.0));

      person: person_rec = (name: 'Fred'; code: 28162);

       alpha: set of char = ['A'..'Z','a'..'z'];

      vowels: array [1..5] of char = ('A','E','I','O','U');
              (* or equivalent *)
      vowels: array [1..5] of char = 'AEIOU';

        root: pointer = nil;
```

Expressions

The precedence of the operators in CLSN Pascal is as follows. Operators with equivalent precedence are evaluated from left to right.

Operators	Precedence
@ (NOT	First
* / DIV MOD AND SHL SHR	Second
+ - XOR OR	Third
< > = <> <= >= IN	Fourth

Operators

The following table shows the legal operand types for the arithmetic operators and the resulting type of their conjunction. For operations between an INTEGER and a REAL, the INTEGER is first converted to a REAL before the operation takes place.

Operator	Operation	Operand Type	Result Type
+	Addition	Integer	Integer
		Real	Real
	Set Union	Set	Set
	Concatenation	String	String
-	Negation	Integer	Integer
		Real	Real
	Subtraction	Integer	Integer
		Real	Real
	Set Difference	Set	Set
*	Multiplication	Integer	Integer
		Real	Real
	Set Intersection	Set	Set
/	Division	Integer	Real
		Real	Real
DIV	Integer Division	Integer	Integer
MOD	Remainder	Integer	Integer
SHL	Multiplication	Integer	Integer
SHR	Multiplication	Integer	Integer
NOT	Negation	Boolean	Boolean
		Integer	Integer
AND	And	Boolean	Boolean
		Integer	Integer
OR	Or	Boolean	Boolean
		Integer	Integer
XOR	Exclusive Or	Boolean	Boolean
		Integer	Integer
=	Equality	Simple	Boolean
		Real	Boolean
		String	Boolean
		Set	Boolean
		Pointer	Boolean

<>	Inequality	Simple	Boolean
		Real	Boolean
		String	Boolean
		Set	Boolean
		Pointer	Boolean
<	Less than	Simple	Boolean
		Real	Boolean
		String	Boolean
>	Greater than	Simple	Boolean
		Real	Boolean
		String	Boolean
<=	Less or Equal	Simple	Boolean
		Real	Boolean
		String	Boolean
>=	Greater or Equal	Simple	Boolean
		Real	Boolean
		String	Boolean
>=	Superset	Set	Boolean
<=	Subset	Set	Boolean
IN	Membership	A simple type and a set of the same type	Boolean

@ Operator

The @ operator may precede a variable identifier in order to retrieve its address in memory. For a global variable the address returned is the absolute location of the variable. For a local variable the value returned is the offset of the variable on the stack.

Standard Pascal allows the @ to be used in the same way the ^ operator is used. CLSN Pascal sees the two operators in completely different ways, and one may never be used for the other.

The value returned by the @ operator is a longint with the highest byte representing the bank (0 = Main; 1..4 = Banks) and the low word representing the absolute memory location.

Value Typecasts

Value typecasts facilitate the changing of an expression of one type to another type. Any simple type identifier can be used to alter a type in addition to the two provided by CLSN Pascal, CHR and ORD. Some examples of value typecasting follow:

```
chr(65)      (*  'A' *)
ord('0')     (*  48  *)
boolean(1)   (* TRUE *)
```

Evaluation of Expressions

Expressions in CLSN Pascal are always completely evaluated; there is no short circuit boolean evaluation. For example, the following statement, although syntactically correct, will generate a run-time error when X is zero.

```
if (x <> 0) and ((y / x) > 10.0) then
```

The reason is that if even if X is zero the second term of the expression is still evaluated and, since division by zero is an error, the program will halt. The above statement should be coded as:

```
if (x <> 0) then
  if ((y / x) > 10.0) then
```

This removes the danger of division by zero. The same caution should be applied to pointers. Do not try to dereference a NIL pointer.

Statements

The FOR, GOTO, CASE, IF, REPEAT, and WHILE statements are all supported by CLSN Pascal. The WITH statement is not available. In order to allow machine language subroutines to be accessed, the INLINE statement is present.

FOR Statement

The FOR statement works the same as in Standard Pascal. Although it is technically an error to modify the control variable, the compiler will not signal an error if this is done. Doing so will not affect the FOR loop; it will still iterate as many times as was set by the initial statement.

GOTO Statement

The GOTO statement can only reference a declared label. GOTO can not be used to jump between procedures, and it should not be used to jump within another block (i.e. do not GOTO a label within a FOR loop).

CASE Statement

The CASE statement may only be used with simple types. If the CASE statement does not find any matches among the case constants, nothing is executed, unless an ELSE statement follows the last statement in which case its block is executed.

The CASE statement is the fastest form of comparison in CLSN Pascal.

Example

```
case ch of
  'E': edit_text;
  'C': clear_text;
  'L': load_text;
  'S': save_text;
  'Q': quit_flg:=true
else
  writeln('Invalid Key');
end;
```

WITH Statement

The WITH statement is not supported by this version of CLSN Pascal. This means that any record references must be spelled out completely.

INLINE Statement

With the INLINE statement, procedures and functions can incorporate machine language routines to help quicken the speed of a program. The format of the INLINE statement is:

```
INLINE(b1/b2/b3/...)
```

Where b1, b2 and b3 represent the machine language code. These values can be either byte constants or identifiers. An identifier will generate two bytes of of code representing its absolute address. If an identifier is preceded by a "<" or a ">" then only one byte of code is generated. Using a "<" will generate the low byte of the address while using ">" will generate the high byte of the address.

In the sample file UPSTR.PAS, the INLINE statement is used to convert a string to uppercase at machine language speed.

Procedures and Functions

CLSN Pascal has no real limit (other than memory and stack considerations) on the number of parameters or the type of the parameters that may be passed to a procedure or function. The only type that cannot be passed as a parameter is a procedure or a function. File types must be variable parameters.

Functions can only return a simple, real, string, or pointer type. An ARRAY, for example, cannot be returned.

Procedures and functions can be declared FORWARD so it may be called by a procedure that physically comes before it. The complete procedural heading is required, followed by the FORWARD clause. The procedure is later resolved by giving the procedure heading (without the parameter list) followed by the block. See the file SALPI.PAS for an example of a forward declaration.

File Input / Output

CLSN Pascal does not support a file buffer variable that can be accessed using the ^ operator. Also, the standard procedures GET and PUT are not available. The only facility for storing and retrieving records are the READ and WRITE procedures.

The procedures BLOCKREAD and BLOCKWRITE can be used to quickly transfer a block of data.

Any device available can be used as a file. For example the printer could be used as an output file, or the keyboard could be used as an input file by using 'K:' or 'P:' with the ASSIGN statement.

```
function key: char;
var
  ch: char;

begin
   assign(f,'K:'); reset(f);    (* Get input from keyboard *)

   read(f, ch); key := ch;      (* Get the key, return it *)

   close(f);                    (* Close the file *)
end;
```

NOTE: The function READKEY accomplishes all of this.

If an error occurs during an I/O routine a runtime error normally occurs, and the program halts. A program can perform its own error checking by using the [I] switch and investigating the function IORESULT. The following example determines if a file exists.

```
function exist(s: string): boolean;
var
    f: file of char;

begin
    assign(f, s);

    [I-];                        (* Turn error checking off *)

    reset(f);                    (* Try to open file *)

    [I+];                        (* Turn checking back on *)

    exist := (ioresult < 128); (* Less than 128 is success *)

    close(f);
end;
```

NOTE: The [I] switch is a statement and can only be used within a statement block.

Including Files

The INCLUDE statement can only appear outside a procedure or function block. An INCLUDE file may not INCLUDE another file. The filename must follow standard DOS convention and must include the extender; the extender ".PAS" is not assumed with the INCLUDE statement. An example of the INCLUDE statement follows:

```
const
   cx = 160;
   cy = 80;

include 'D:GRAPH.PAS';     (* The .PAS is required *)

var
  x, y: integer;
```

INCLUDE files can be used to create libraries of commonly used procedures and functions. For example, a series of procedures creating and maintaining windows could be stored in a file called 'D:WINDOW.PAS'. Then, if a new program needed windows, this file only needs to be INCLUDEd to access the routines.

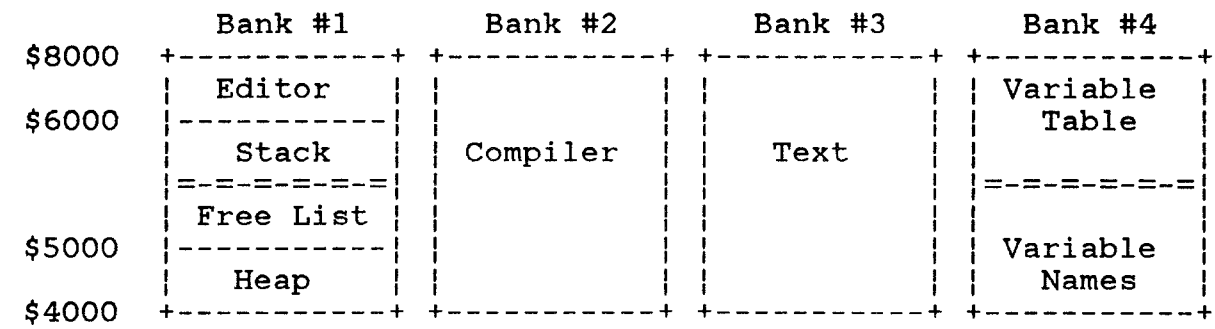
Also, when the source becomes too large to fit all at once in the editor, it can be broken up into smaller segments which a main file could then INCLUDE together. By doing this, CLSN Pascal can compile source code much longer than the 16k editor might imply.

Technical Information about CLSN Pascal

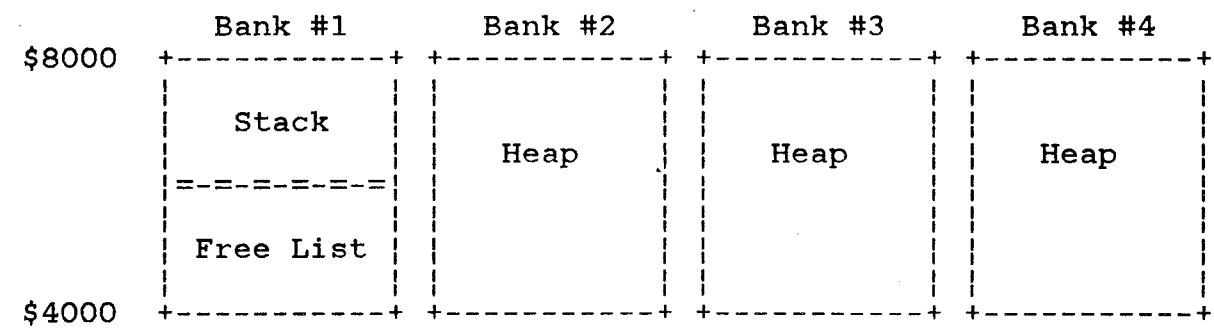
The Atari 130XE can access 128k of memory by breaking up half of it into four small 16k banks that can be accessed only one at a time. CLSN Pascal makes use of this memory scheme in order to allow an editor, a compiler, the source code, and the compiled code to all exist simultaneously in memory.

Main memory is always allocated in the same manner but the configuration of the banks will be in one of two states depending on whether or not the compiler and editor are in memory. Memory maps of both situations, along with main memory, follow.

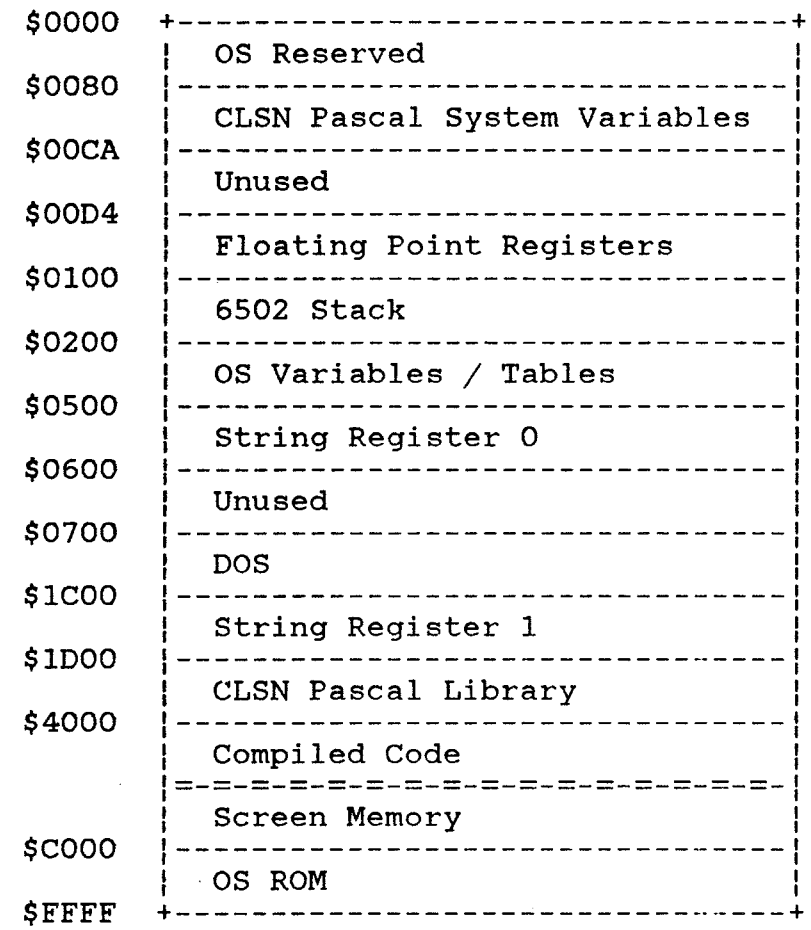
Bank Configuration #1: Editor and Compiler in Memory



Bank Configuration #2: A Compiled Program run from DOS



Main Memory



Variable Storage

Global variables, (variables not defined within a procedure or function) are stored in main memory intermixed with the compiled code. Local variables are allocated upon the stack during entry to the procedure or function. The stack size is initially 16384 bytes but is also dependent on the the number of dynamic variables allocated. See Heap Management for more information.

Integers

INTEGERs are stored differently depending on their range.

Range	Stored as	Size (in bytes)
0..255	Byte	1
-128..127	Shortint	1
0..65535	Word	2
-32768..32767	Integer	2
Otherwise	Longint	3

Real

A REAL value is stored in binary coded decimal and occupies six bytes. The first byte contains the sign of the number, the sign of the exponent, and the exponent (base 100) itself. The following five bytes are the mantissa.

Char

A CHAR value occupies one byte.

Boolean

A BOOLEAN value occupies one byte with 0 representing FALSE, and 1 representing TRUE.

Enumerated

Enumerated types occupy one byte if there are less than 256 elements and occupy two bytes otherwise.

String

STRINGs occupy as much memory as set in their declaration plus one byte for the length. The zeroeth element of the string is the length.

Set

SETs always occupy 32 bytes and do not depend on the size of the set type.

Records

The fields of the record are stored contiguously in memory. The size of a record is simply the sum of the size of the fields.

Arrays

ARRAYs are stored contiguously in memory with the rightmost dimension increasing first through memory.

Files

Files occupy one byte and are simply an index to an IOCB block, so its value ranges from 0 to 7.

Parameters and Results

Parameters to a procedure or function can be of any type, as opposed to function results which can only return a limited range of types.

Parameters

If the parameter is a value parameter, a copy of the value is placed on the stack. This is true for all values including ARRAYs, STRINGs, RECORDs, and SETs.

When a variable parameter is passed, only the address of the variable is placed on the stack.

Functions

Functions can only return a STRING, a REAL, a POINTER, or a simple type value. The following table shows where a return value is placed.

Return Type	Location	Name
Simple	$D4..$D6	IR0 (Integer Register 0)
Pointer	$D4..$D6	IR0 (Integer Register 0)
Real	$D4..$DA	FR0 (Floating Point Register 0)
String	$500..$5FF	SR0 (String Register 0)

Heap Management

An intricate part of Pascal is its ability to allocate variables dynamically. The heap routines are the heart of this system.

The heap management scheme employed in CLSN Pascal is divided into two parts, the free list and the heap. The heap, displayed in the diagram above, is the memory that is allocated for a program's use. The size of the heap when compiler and editor are in memory is about 4000 bytes. When the program is running on its own, the heap is about 48k.

The free list is a table of pointers into the heap stating which memory is in use and which memory is free. Located in bank 1 at $4000, the free list grows upward in memory toward the stack.

```
type
  free_record = record
                  bank: byte; (* Bank block is in         *)
                              (*    $00: End of Free List *)
                              (*   +$80: Block is in use  *)
                  mptr: word; (* Location of block        *)
                  size: word; (* Size of block            *)
                end;

var
  free_list: array [0..3275] of free_record absolute $14000;
```

When a block of memory is requested by the procedures GETMEM or NEW, the free list is searched for a free block large enough to fulfill the demand. If a block is found, the record is marked "in use" and stored back into the free list. In most cases, the size of the block will not exactly match the size of the request. When this

happens a new record is added to the free list pointing to the unused portion of the block.

When a block of memory is released by the procedures FREEMEM or DISPOSE, the heap routines unmark the specified block and search for any memory in the heap contiguous to this block. If any are found, the two blocks combine and become one large block.

An error will occur if there are no blocks large enough to fulfill the request, or if the free list collides with the stack.

The function MEMAVAIL, which tells how much free memory remains in the heap, returns the sum of the size of the free blocks. MAXAVAIL returns the size of the largest free block in the heap.

Run Time Errors

If a runtime error occurs, the program will display the error code and halt. If the compiler and editor are in memory, then, once a key is pressed, CLSN Pascal will recompile the code and locate the error in the source text. The following table lists the possible runtime errors generated by CLSN Pascal.

Error	Meaning
1	Too many open files A maximum of five files may be open simultaneously. This refers to any file (i.e. "K:", "S:"), as opposed to DOS which normally can only have three DISK files open at once.
2	Mathematical Overflow A mathematical operation resulted in a number too large or small, LN was passed a zero or negative number, or SQRT was passed a negative number.
3	Division by Zero The second argument of /, DIV or MOD was evaluated as zero.
4	Invalid Number A READ or VAL statement unsuccessfully converted a string into a number.
5	String Overflow A string function tried to create a string longer than 255 characters.
6	Stack Overflow The runtime stack collided with the free list. In other words, the stack can maintain no more local variables, or sustain any more procedure calls.
7	Out of Heap The request for dynamic memory cannot be satisfied. There are no free blocks of memory large enough for the demand.
8	Nil Pointer A pointer whose value was NIL was dereferenced, or a NIL value was passed to the procedures FREEMEM or DISPOSE.
9	Free List Overflow The free list collided with the stack; there are too many dynamic variables allocated.
128..255	I/O Error Codes See the DOS manual for complete explanation

About CLSN Software

CLSN Software is a very small company. The costs and considerations of creating a project as large as CLSN Pascal have prevented this manual from being any larger.

We hope, however, that the ease of use of this compiler will rejuvenate Atari programmers so articles on Pascal will become commonplace in Atari magazines.

Once again, thank you for purchasing this product.

CLSN Software
10 Arlington Place
Kearny, NJ 07032

Special thanks to Mom, Dad, Matt, and Pat - The special people in my life with three letter names, except for Matt who always likes to be difficult.

References

Chris Hawksley, "PASCAL: A Programming Guide to Computers and Programming," Cambridge University Press, 1986.

Seymour C. Hirsch, "PASCAL Programming," Prentice-Hall, Inc. 1987.

Elliot B. Koffman, "PASCAL," Addison-Wesley, 1989.

John Konvalina and Stanley Wileman, "Programming with PASCAL," McGraw-Hill, 1987.

Lisanti, Mann, and Zlotnick, "Algorithms, Programming, Pascal," Wadsworth, 1987.

Walter J. Savitch, "PASCAL: An Introduction to the Art and Science of Programming," Benjamin/Cummings Publishing Co., 1987.

The following pages contain an alphabetical listing of the library routines available in CLSN Pascal. Each entry contains a brief description of the procedure, function, or variable, its Pascal declaration, any special remarks about the procedure, an example of its use, and a reference to other related library routines.

ABS Function

Description

Returns the absolute value of the argument.

Declaration

```
function abs(x: real): real;
function abs(x: longint): longint;
```

Example

```
var
  x: integer;

begin
  x:=abs(-5);
end.
```

See also

hi,lo,ord

APPEND Procedure

Description

Opens an existing file and prepares it so data may be appended to the end of the file.

Declaration

```
procedure append(var f: file);
```

Remarks

The file to be appended must already exist on the disk otherwise a FILE NOT FOUND error will result.

Example

```
var
   f: text;
  ch: char;

begin
  assign(f,'D:OUTPUT'); rewrite(f);

  for ch:='A' to 'L' do   (* Create a file *)
    write(f,ch);
```

```
    close(f);

    append(f);                (* Append to it *)

    for ch:='M' to 'Z' do     (* More data *)
      write(f,ch);

    close(f);
  end.
```

See also
  assign,reset,rewrite

ARCTAN Function

Description
  Returns the arctangent of the argument.

Declaration
```
  function arctan(x: real): real;
```

Remarks
  The resulting angle will be in radians unless a DEGREES statement was executed.

Example
```
  begin
    writeln('PI=',4.0*arctan(1.0));
  end;
```

See also
  sin,cos,degrees,radians

ASSIGN Procedure

Description
  Associates a file variable with a specified external file.

Declaration
```
  procedure assign(var f: file; s: string);
```

Remarks
  Any standard device may be considered a file. For example, the keyboard 'K:' could be used an an input file, and the printer, 'P:' could be used as and output file.

  The file name must include the device prefix.

  File name validity is not checked until a reset, rewrite, or append statement is executed.

```
Example
  var
    f: text;

  begin
    assign(f,'D:OUTPUT'); rewrite(f);

    writeln(f,'This is a text file.');

    close(f);
  end.

See also
  append,reset,rewrite

--------------------
BLOCKREAD Procedure
--------------------
Description
  Reads a block of data from an open file into a buffer.

Declaration
  procedure blockread(var f: file; var buffer; size: longint);

Remarks
  The file may be of any type and the size specified is always in
bytes.

  The number of bytes actually transferred can be found by the
predefined global variable BLOCKSIZE.

  If the operation is successful IORESULT will equal 1.

Example
  See COPY.PAS

See also
  blocksize,blockwrite

---------------------
BLOCKWRITE Procedure
---------------------
Description
  Writes a block of data from a buffer to an open file.

Declaration
  procedure blockwrite(var f: file; var buffer; size: longint);

Remarks
  The file may be of any type and the size specified is always in
bytes.

  If the operation is successful IORESULT will be equal to 1.
```

```
Example
  See COPY.PAS

See also
  blockread,blocksize

--------------
CHOUT Variable
--------------
Description
  Provides a method for adding an offset to character output.

Declaration
  var
    chout: byte;

Remarks
  The value in CHOUT, which defaults to zero, is added to each
character before it is output.  The main purpose of CHOUT is to make
inverted output simpler, although it can be used to encrypt or
unencrypt output.

Example
  begin
    writeln('Normal');
    chout:=$80; writeln('Inverse'); chout:=$00;
    writeln('Normal');
  end.

------------
CHR Function
------------
Description
  Returns the character associated with the specified ordinal value.

Declaration
  function chr(x): char;

  (x may be of any ordinal type)

See also
  ord

---------------
CLOSE Procedure
---------------
Description
  Closes an open file.

Declaration
  procedure close(var f: file);
```

Remarks
  All remaining data is sent to the file, and the file is then closed.
  Always close a file when finished with it.

CLRSCR Procedure

Description
  Clears the screen.

Declaration
  procedure clrscr;

Remarks
  The screen is cleared and the cursor is sent home, but the current screen colors are not reset.

See also
  gotoxy

COPY Function

Description
  Returns a substring of a string.

Declaration
  function copy(s: string; p,n: longint): string;

Remarks
  The series of characters in the string s, starting at p and extending through p+n is returned as the result.

  It is an error to specify p as larger than the length of the string.

  If p+n is larger than the length of the string, then the section of the string from p to the end is returned.

Example
```
  var
    s: string;

  begin
    s:='CLSN Pascal';
    writeln(copy(s,6,6)); (* Pascal *)
  end.
```

See also
  delete,insert,pos

COS Function

Description
  Returns the cosine of the specified angle.

Declaration
  function cos(x: real): real;

Remarks
  The angle must be in radians, unless a DEGREES statement was executed in which case the angle must be in degrees.

DEC Procedure

Description
  Decrements an ordinal variable by 1.

Declaration
  procedure dec(x);

  (x may be variable of any ordinal type)

Remarks
  DEC(x) is mathematically equivalent to x:=x-1. However, the compiler generates different code for the two statements with DEC being more optimal.

Example
```
  var
    x: integer;

  begin
    x:=10;
    dec(x);          (* 9 *)
  end.
```

See also
  inc,pred,succ

DEGREES Procedure

Description
  Sets trigonometric angle measurement to degrees.

Declaration
  procedure degrees;

Remarks
  After executing this statement, the function SIN and COS will expect an angle measured in degrees. ARCTAN will return an angle in degrees.

See also
  radians

DELETE Function

Description
  Returns a given string with specified characters removed.

Declaration
  function delete(s: string; p,n: longint): string;

Remarks
  The characters in the string s, from p to p+n are deleted and the resulting string is returned.

  It is an error to pass p a value larger than the length of s.

  If p+n is larger than the length of s, only the characters up to p are returned.

Example
```
  var
    s: string;

  begin
    s:='D:FILE.TXT';
    s:=delete(s,1,2); (* FILE.TXT *)
  end.
```

See also
  copy,insert,pos

DISPOSE Procedure

Description
  Releases to the heap a dynamic variable that was allocated using NEW.

Declaration
  procedure dispose(var p: pointer);

Remarks
  The memory allocated by the dynamic variable is returned to the heap.

  An error occurs if p is NIL.

  It is an error to reference p^ after it has been disposed.

```
Example
  var
    temp: ^real;
     x,y: real;

  begin
    x:=1; y:=10;

    new(temp);                    (* Get a temporary variable *)

    temp^:=x; x:=y; y:=temp^;     (* Swap *)

    dispose(temp);                (* Release *)
  end.
```

DPEEK Function

```
Description
  Returns the word value in a specified memory location.

Declaration
  function dpeek(memloc: longint): word;

Remarks
  The byte at the memory location is returned.  See POKE for memory
organization.  It is suggested that DPEEK be avoided in favor of
pointers and absolute variable declarations.

Example
  var
    scmem: word;

  begin
    scmem:=dpeek(88); (* Get screen memory *)
  end.

  (or even better)

  var
    scmem: word absolute 88;

  begin                  (* No assignment, value is already there *)
  end.

See also
   dpoke,peek,poke
```

DPOKE Procedure

```
Description
  Stores a word in a specified memory location.
```

Declaration
  procedure poke(memloc,value: longint);

Remarks
  The low word of value is stored in the memory location memloc. See POKE for memory organization. It is suggested that DPOKE be avoided and in its place use pointers or absolute variable declarations.

See also
  poke,peek,dpeek

DRAWTO Procedure

Description
  Used in conjunction with plot to create a line between two points on the graphics screen.

Declaration
  procedure drawto(x,y: longint);

Remarks
  A line in the color set by the procedure SETCOLOR is drawn from the current cursor position to the specified coordinates. The current cursor position is set from the most recent plot or drawto. If no PLOT or DRAWTO was executed since entering the graphics mode, the line will originate from the upper left corner of the screen.

  An error occurs if x or y is out of range the screen.

See also
  graphics,plot,setcolor

EOF Function

Description
  States whether or not the end of file has been reached.

Declaration
  function eof(var f: file): boolean;

Remarks
  Returns true when the last component of the file is reached.
  Returns false otherwise.

```
Example
  var
    f: file of char;
   ch: char;
    s: string;

  begin
    write('File to view: '); readln(s);

    assign(f,s); reset(f);

    while not eof(f) do
      begin
        read(f,ch); write(ch);
      end;

    close(f);
  end.

See also
  eoln

-------------
EOLN Function
-------------
Description
  States whether or not the end of a line in a text file has been
reached.

Declaration
  function eoln(var f: text): boolean;

Remarks
  Returns true if the file pointer is at the end of the line.
  Returns false otherwise.

  Eoln may only be used on text files.

See also
  eof

---------------
ERASE Procedure
---------------
Description
  Deletes an external file.

Declaration
  procedure erase(f: string);

Remarks
  The specified file is removed from the disk.

  The file with the name f must not be open.
```

IORESULT will be equal to 1 if the operation is successful.

Example

```
begin
  erase('D:GARBAGE.DAT'); (* Erases file *)
end.
```

EXIT Procedure

Description

Leaves the current procedure.

Declaration

```
procedure exit;
```

Example

```
var
  x: integer;

begin
  while true do
    begin
      if keypressed then
        exit;

      writeln(x); inc(x);
    end;
end.
```

See also

halt

EXP Function

Description

Returns e raised to the power of the argument.

Declaration

```
function exp(x: real): real;
```

Example

```
begin
  writeln('e=',exp(1.0));
end.
```

see also

ln

FILLCHAR Procedure

Description
  Fills a block of memory with a value.

Declaration
```
  procedure fillchar(var p; size,num: longint);
```
  (p may be a variable of any type)

Remarks
  Fills a block of memory starting at p with the value specified by num.

Example
```
  type
    maze_type=array [1..24] of array [1..40] of char;

  procedure clear_maze(var maze: maze_type);
  begin
    fillchar(maze,sizeof(maze),0); (* Set it all to zero *)
  end;
```

See also
  move

FREEMEM Procedure

Description
  Releases a dynamic variable to the heap that was allocated using GETMEM.

Declaration
```
  procedure freemem(var p: pointer; size: longint);
```

Remarks
  The memory allocated by the dynamic variable is returned to the heap.

  The size must be the same as when allocated using GETMEM.

  An error occurs if p is NIL.

  It is an error to reference p^ after it has been disposed.

See also
  getmem

GETDOT Function

Description
  Returns the color of a point on the graphics screen.

Declaration
  function getdot(x,y: longint): byte;

Remarks
  The color at the specified coordinate is returned. This function must be preceded by the use of a GRAPHICS statement. An error occurs if the specified coordinate is out of range of the screen.

See also
  plot

GETMEM Procedure

Description
  Creates a dynamic variable on the heap.

Declaration
  procedure getmem(var p: pointer; size: longint);

Remarks
  GETMEM returns a pointer to a block of memory of the specified size.

  A runtime error occurs if the heap routines cannot allocate the memory requested.

See also
  freemem

GOTOXY Procedure

Description
  Position the cursor at the specified screen coordinates.

Declaration
  procedure gotoxy(x,y: longint);

Remarks
  The upper left corner of the screen represents (0,0).

  The maximum row and column depends on the current screen configuration.

  The cursor will not move until the next output to the screen takes place.

Example

```
procedure show_time;
var
  jlow: byte absolute 20;
   jhi: byte absolute 19;

begin
  gotoxy(2,0);                          (* Top of screen *)
  writeln(jlow+256*jhi,' jiffies');(* Show time passed *)
end;
```

See also

clrscr

GRAPHICS Procedure

Description

Initializes the Atari's graphics capabilities.

Declaration

```
procedure graphics(m: longint);
```

Remarks

The screen is cleared and a text or bit mapped screen will be generated according to the parameter m. If a bit mapped graphics screen is chosen, the procedures PLOT and DRAWTO may be used to create images. The following table summarizes the different modes available on the 130XE.

Graphics

Mode	Type	Columns	Rows	Colors
0	Text	40	24	2
1	Text	20	24	5
2	Text	20	12	5
3	Graphics	40	24	4
4	Graphics	80	48	2
5	Graphics	80	48	4
6	Graphics	160	96	2
7	Graphics	160	96	4
8	Graphics	320	192	1/2
9	Graphics	80	192	16
10	Graphics	80	192	8
11	Graphics	80	192	16
12	Text	40	24	5
13	Text	40	12	5
14	Graphics	160	192	2
15	Graphics	160	192	4

The following values may be added to the parameter m:

+16: Inhibit the text window

+32: Prevent clearing of the screen

```
Example
  var
    x: integer;

  begin
    graphics(7);

    for x:=0 to 159 do
      begin
        setcolor(x * 4 div 160);
        plot(x,0); drawto(x,95);
      end;
  end.

See also
  plot,drawto,setcolor,grx

--------
GRX File
--------
Description
  Provides a channel for output to the graphics screen.

Declaration
  var
    grx: text;

Remarks
  This device was provided so text may be printed on the screen while
in graphics mode 1 or 2.  A GRAPHICS statement must be executed before
any reference to this file.

Example
  begin
    graphics(2);
    writeln(grx,'LARGE TEXT');
    writeln('SMALL TEXT');
  end.

See also
  graphics

--------------
HALT Procedure
--------------
Description
  Stops program execution immediately.

Declaration
  procedure halt;

Remarks
  The program ends immediately.  Open files are not closed.
```

```
Example
  var
    s: string;

  begin
    write('File: '); readln(s);

    if (s='') then
      begin
        writeln('ERROR: Invalid File Name');
        halt;
      end;
     .
     .
     .

  end.

See also
  exit

-----------
HI Function
-----------
Description
  Reveals the hi order byte of the argument.

Declaration
  function hi(x: longint): byte;

Example
  begin
    writeln(hi($1234));  (* $12 *)
  end.

See also
  lo

-------------
INC Procedure
-------------
Description
  Increments an ordinal variable by 1.

Declaration
  procedure inc(x);

  (x may be variable of any ordinal type)

Remarks
  INC(x) is mathematically equivalent to x:=x+1.  However, the
compiler generates different code for the two statements with INC
being more optimal.
```

```
Example
  var
    x: integer;

  begin
    x:=10;
    inc(x);        (* 11 *)
  end.

See also
  dec,succ,pred

---------------
INSERT Function
---------------
Description
  Returns a string with characters inserted at a specified point.

Declaration
  function insert(sub,s: string; p: longint): string;

Remarks
  The string sub is inserted into the string s at point p.  All
characters from p to the end of the string are moved to make room for
the new string.

  It is an error to assign p a value larger than the length of s.

  A runtime error occurs if the resulting string is longer than 255
characters.

Example
  var
    s: string;

  begin
    s:='ABEF';

    s:=insert(s,'CD',3); (* 'ABCDEF' *)
  end.

See also
  copy,delete,pos

------------
INT Function
------------
Description
  Returns the integer portion of a real value.

Declaration
  function int(x: real): real;
```

Remarks
  Rounding is always downward.

Example
```
  begin
    writeln(int(pi));    (* 3.0000000 *)
  end.
```

See also
  round,trunc

IORESULT Function

Description
  Returns the status of the last I/O operation.

Declaration
```
  function ioresult: byte;
```

Remarks
  A value less than 128 is considered a successful operation.

  For errors greater than 127, see the DOS manual for a more complete explanation.

  Program termination by an I/O error can be avoided by using the [I] switch.

Example
```
  function exist(s: string): boolean;
  var
    f: file of char;

  begin
    assign(f,s);

    [I-]; reset(f); [I+];  (* Try to open file *)

    exist:=(ioresult<$80); (* Inspect ioresult *)

    close(f);
  end;
```

KEYPRESSED Function

Description
  Tells whether or not a key has been pressed.

Declaration
```
  function keypressed: boolean;
```

```
Example
  procedure wait;
  var
    ch: char;

  begin
    while not keypressed do
      writeln('Waiting...');

    ch:=readkey;
  end;

See also
  readkey

---------------
LENGTH Function
---------------
Description
  Returns the length of a given string.

Declaration
  function length(s: string): byte;

Remarks
  A null string has a length of zero.

Example
  var
    s: string;
    i: integer;

  begin
    write('Enter a word: '); readln(s);

    for i:=length(s) downto 1 do
      write(s[i]);    (* Reverse word *)

    writeln;
  end.

See also
  copy,delete,insert,pos

-----------
LN Function
-----------
Description
  Returns the natural logarithm of the argument.

Declaration
  function ln(x: real): real;
```

```
Remarks
  A runtime error occurs if x is less than or equal to zero.

Example
  begin
    writeln(ln(exp(pi))); (* pi *)
  end.

See also
  exp

-----------
LO Function
-----------
Description
  Reveals the low order byte of the argument.

Declaration
  function lo(x: longint): byte;

Example
  begin
    writeln(lo($1234));  (* $34 *)
  end.

See also
  hi

-----------------
MAXAVAIL Function
-----------------
Description
  Returns the largest contiguous block of memory available in the
heap.

Declaration
  function maxavail: longint;

Remarks
  Since a runtime error will occur if a call is made to GETMEM or NEW
with a request for more memory than available, this function can be
used to prevent an unexpected program termination.

See also
  getmem,new,memavail

-----------------
MEMAVAIL Function
-----------------
Description
  Returns the total amount of memory left in the heap.

Declaration
  function memavail: longint;
```

Remarks
  The size of all free blocks of memory are summed and returned.

See also
  maxavail

MEMPTR Function

Description
  Creates a generic pointer given a bank and an address.

Declaration
  function memptr(bank, addr: longint): pointer;

See also
  getmem,freemem

MOVE Procedure

Description
  Moves a block of data from one area of memory to another.

Declaration
  procedure move(var source,dest; size: longint);

  (source and dest are variables of any type)

Remarks
  Move can move a block of memory in and between banks and main memory but it will not handle overlaps.

Example
```
  type
    chtype=array [0..127] of array [0..7] byte absolute $E400;

  var
    chset: chtype absolute $E000;
    chnew: chtype;

  begin
    move(chset,chnew,sizeof(chtype));  (* Copy the character set *)
    (* or even better *)
    chnew := chset;
  end.
```

NEW Procedure

Description
  Creates a dynamic variable on the heap.

```
Declaration
  procedure new(var p: pointer);

Remarks
  NEW will create a dynamic variable on the heap.

  A runtime error occurs if the heap routines cannot reserve enough
memory for the variable.

See also
  dispose

-----------------
NOSOUND Procedure
-----------------
Description
  Turns off all sound.

Declaration
  procedure nosound;

Example
  begin
    while not keypressed do        (* Wait for a key *)
      sound(0,random(256),10,10); (* Random noise *)

    nosound;                       (* Silence *)
  end.

See also
  sound

------------
ODD Function
------------
Description
  Determines if the argument is odd.

Declaration
  function odd(x: longint): boolean

Remarks
  Returns true if x is odd, otherwise it returns false.

------------
ORD Function
------------
Description
  Returns the ordinal number of an ordinal type expression.

Declaration
  function ord(x): longint;

  (x is an ordinal type)
```

Example

```
var
  ch: char;

begin
  for ch:='A' to 'Z' do
    writeln(ch,' has an ordinal value of ',ord(ch));
end.
```

See also
  chr

PADDLE Array

Description
  Returns the position of one of the eight paddle controllers.

Declaration

```
var
  paddle: array [0..7] of byte absolute 624;
```

Remarks
  The position of the specified controller is returned. It can range from 0 to 228.

See also
  ptrig,stick,strig

PALETTE Array

Description
  Sets the color of the specified color register.

Declaration

```
var
  palette: array [0..4] of byte absolute 708;
```

Remarks
  Palette is a global variable and values are assigned to it as such. The values assigned represent the color and can range from 0 to 255; the index represents the color register. The following table reveals the values of different colors.

Value	Color	Value	Color
0	Gray	128	Blue
16	Gold	144	Light Blue
32	Orange	160	Turquoise
48	Red-Orange	176	Green-Blue
64	Pink	192	Green
80	Purple	208	Yellow-Green
96	Purple-Blue	224	Orange-Green
112	Blue	240	Light Orange

A value of 0 to 15 (dark to light) representing the hue may be added to the value of the color.

The color registers for graphics modes varies. The following table summarizes exactly what each color register controls.

Color Register	Graphics Mode	Controls
0	0	Nothing
	1,2	Character Color
	3 to 7	Color for SETCOLOR(1)
	8	Nothing
1	0	Character Luminance
	1,2	Character Color
	3,5,7	Color for SETCOLOR(2)
	4,6	Nothing
	8	Graphics Luminance
2	0	Background Color
	1,2	Character Color
	3,5,7	Color for SETCOLOR(3)
	4,6	Nothing
	8	Background Color
3	0	Background Color
	1,2	Character Color
	3 to 7	Nothing
	8	Background Color
4	0	Graphics Border
	1 to 7	Graphics Border/Background SETCOLOR(0)
	8	Graphics Border

Note: Colors may not exactly match the above chart depending on the television and its current adjustments.

See also
setcolor

PEEK Function

Description
  Returns the byte value in a specified memory location.

Declaration
  function peek(memloc: longint): byte;

Remarks
  The byte at the memory location is returned. See POKE for memory organization. It is suggested that PEEK be avoided in favor of pointers and absolute variable declarations.

See also
  poke,dpeek,dpoke

PLOT Procedure

Description
  Positions the cursor at a specified coordinate on the graphics screen and lights a point in the color set by the procedure setcolor.

Declaration
  procedure plot(x,y: longint);

Remarks
  Used in conjunction with DRAWTO, images can be created on the graphics screen. This command must be preceded by a call to the procedure GRAPHICS.

  An error occurs if x or y is out of the range of the screen.

See also
  graphics,drawto,setcolor,getdot

POKE Procedure

Description
  Stores a single byte in a specified memory location.

Declaration
  procedure poke(memloc,value: longint);

Remarks
  The low byte of value is stored in the memory location memloc. Memory is mapped as follows:

Location	Area
$00000-$0FFFF	Main Memory
$14000-$18000	Bank #1
$24000-$28000	Bank #2
$34000-$38000	Bank #3
$44000-$48000	Bank #4

It is suggested that poke be avoided. Instead use pointers or absolute variable declarations.

Example

```
procedure cursor_off;
begin
  poke(752,1);
end;

(or even better)

procedure cursor_off;
var
  crsinh: byte absolute 752;

begin
  crsinh:=1;
end;
```

See also

peek,dpoke,dpeek

POS Function

Description

Returns the position of a substring within a string.

Declaration

```
function pos(sub,s: string): byte;
```

Remarks

The position of the first occurrence of sub within s is returned.

If sub is not found within s, a zero is returned.

Example

```
var
  s: string;

begin
  s:='AB CD EF';

  while pos(' ',s)<>0 do
    delete(s,pos(' ',s), 1); (* ABCDEF *)
end.
```

See also
  copy,insert,delete

PRED Function

Description
  Returns the preceding ordinal value of the argument.

Declaration
  function pred(x);

  (x is an ordinal type and the function returns a value of the same type)

See also
  succ,inc,dec

PTRIG Array

Description
  Reveals the state of a given paddle button.

Declaration
  var
    ptrig: array [0..7] of byte absolute 636;

Remarks
  Returns 0 if the specified paddle button is pressed, otherwise it returns 1.

See also
  paddle,stick,strig

RADIANS Procedure

Description
  Sets trigonometric angle measurement to radians.

Declaration
  procedure radians;

Remarks
  After executing this statement, the function SIN and COS will expect an angle measured in radians. The ARCTAN function will return an angle in radians.

See also
  degrees

RANDOM Function

Description
  Returns a random number in a given range.

Declaration
```
  function random(max: longint): longint;
```

Remarks
  This function returns a random number between zero and max-1.

Example
```
  begin
    graphics(8); setcolor(1);

    while not keypressed do
      plot(random(320),random(192)); (* Plot points *)
  end.
```

READ Procedure (Text Files)

Description
  Inputs values for variables from either the keyboard or a file.

Declaration
```
  procedure read([var f: text] var v1[,v2,v3...]);
```

  (Where v1,v2,v3... represent integer, real, character, or
   string variables)

Remarks
  Input is obtained from the keyboard unless a file has been specified.

  When a variable of type char is requested, READ obtains the next character in the file and stores its value in the variable.

  When a variable of any string type is requested, READ takes all the characters up to, but not including, the end of line marker (#155) and stores them into the string variable.

  When a variable of an integer type is requested, READ first skips leading blanks. A series of digits led by an optional sign is expected, and if not supplied will generate a runtime error. Hexadecimal notation (i.e $0500) is also acceptable.

  When a variable of type real is requested, READ first passes over leading blanks. A series of digits separated by an option decimal point, led by an optional sign, and followed by an option exponent is expected and if not supplied will generate a runtime error.

See also
  readln,readkey,write

READ Procedure (Typed Files)

Description
  Inputs records for variables from a typed file.

Declaration
  procedure read(var f: file; var v1[,v2,v3...]);

  (Where f is a file of any type except text, and v1,v2,v3... are all variables of the type of the file)

Remarks
  READ takes successive components from the file and stores them into the variables requested.

See also
  write

READKEY Function

Description
  Waits for a key to be pressed and returns its value.

Declaration
  function readkey: char;

Example
```
  var
    ch: char;

  begin
    repeat
      ch:=readkey;     (* Get a key *)
      write(ch);       (* Echo it *)
    until (ch=#27);
  end.
```

READLN Procedure

Description
  Performs a READ on a text file, then moves past the end of line marker to the next line of the file.

Declaration
  procedure readln([var f: text] var v1[,v2,v3...]);

  (Where v1,v2,v3... represent integer, real, character, or string variables)

Remarks
  READLN performs exactly the same function as READ on a text file, except that after retrieving the requested values, READLN moves the past the end of line marker to the next line of the file.

Example
```
  (* Displays a text file *)

  var
    s: string;
    f: text;

  begin
    assign(f,'D:TEXT.DOC'); reset(f);

    while not eof(f) do
      begin
        readln(f,s);   (* Get a line *)
        writeln(s);    (* Display it *)
      end;

    close(f);
  end.
```

RENAME Procedure

Description
  Gives a new name to a disk file.

Declaration
  procedure rename(old,new: string);

Remarks
  The file with the name old is given the name new.
  Old must have the drive identifier (i.e. D1:), but new must not have the drive identifier.

  An error occurs if the file does not exist.

See also
  erase

RESET Procedure

Description
  Moves the file pointer to the beginning of the file.

Declaration
```
procedure reset(var f: file);
```

Remarks
The file must have been previously associated with a filename by use of the ASSIGN procedure.

If RESET is used on an already open file, the file is closed then reopened.

An error will occur if the file does not exist.

See also
assign,rewrite

RESETDIR Procedure

Description
Prepares the file directory to be read.

Declaration
```
procedure resetdir(var f: text);
```

Remarks
In order to read the disk directory, ASSIGN should be called with a text file variable and the file mask. Successive READLNs can then be used to access the directory information.

Example
```
procedure print_directory(mask: string);
var
  f: text;

begin
  assign(f,mask); resetdir(f); (* Prepare the directory *)

  while not eof(f) do
    begin
      readln(f,s); writeln(s); (* Get file listing *)
    end;

  close(f);                    (* Close the file! *)
end;
```

REWRITE Procedure

Description
Deletes the contents of a file and moves the file pointer to the beginning of the file.

Declaration
  procedure rewrite(var f: file);

Remarks
  The file must have been previously associated with a filename by use of the ASSIGN procedure.

  If the file exists, the contents are deleted, otherwise a new file is created.

See also
  assign,reset

ROUND Function

Description
  Rounds a real value into an integer value.

Declaration
  function round(x: real): longint;

Remarks
  Round converts the given real number to the nearest integer and returns the value.

  If the value is out of the range of a longint, an error will occur.

See also
  int,trunc

SETCOLOR Procedure

Description
  Sets the current drawing color.

Declaration
  procedure setcolor(c: longint);

Remarks
  This command is different from the BASIC setcolor command in that it does not set a color for a color register, but rather chooses the color register for successive PLOT and DRAWTO statements. To change the color of a color register see PALETTE.

See also
  graphics,plot,drawto,palette

SIN Function

Description
  Returns the sine of the specified angle.

Declaration
```
  function sin(x: real): real;
```

Remarks
  The angle must be in radians, unless a DEGREES statement was executed in which case the angle must be in degrees.

See also
  arctan,cos

SIZEOF Function

Description
  Returns the size in bytes of the specified variable or type identifier.

Declaration
```
  function sizeof(x): longint;
```

  (Where x is a variable of any type or a type identifier)

Remarks
  It is suggested that SIZEOF be used with function such as MOVE and FILLCHAR.

Example
```
  var
    maze: array [1..10] of array [1..10] of byte;

  begin
    fillchar(maze,sizeof(maze),0); (* Clear the maze *)
  end;
```

SOUND Procedure

Description
  Turns on or off one of the four sound voices, and sets the pitch, distortion and volume.

Declaration
```
  procedure sound(n,p,d,v: longint);
```

Remarks
  N is the sound channel and allowed values are 0 to 3.
  P is the pitch and can range from 0 (high) to 255 (low).
  D is the distortion and can be set from 0 to 15 (10 is pure)
  V is the volume and has a maximum value of 15.

The following table show the values that will produce musical tones.

Note	Pitch	Note	Pitch	Note	Pitch	Note	Pitch
C	29	D#	50	F#	85	A	144
B	31	D	53	F	91	G#	153
A#	33	C#	57	E	96	G	162
A	35	C	60	D#	102	F#	173
G#	37	B	64	D	108	F	182
G	40	A#	68	C#	114	E	193
F#	42	A	72	C	121	D#	204
F	45	G#	76	B	128	D	217
E	47	G	81	A#	136	C#	230
						C	243

Example

```
  begin
    while not keypressed do
      sound(0,random(256),10,15); (* Random noise *)

    nosound;
  end.
```

See also

nosound;

SQR Function

Description

Returns the argument squared.

Declaration

```
function sqr(x: longint): longint;
function sqr(x: real): real;
```

SQRT Function

Description

Returns the square root of the argument

Declaration

```
function sqrt(x: real): real;
```

Remarks

A runtime error occurs if x is less than zero.

STR Function

Description

Converts a number to a string representation.

Declaration
```
  function str(x: real; d: longint): string;
```

Remarks
  This function converts the number to a string representation.
  D is the number of digits after the decimal point to show. If d is less than zero then the number is converted to exponential form.

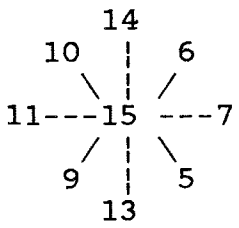
See also
  val

STICK Array

Description
  Reveals the current direction of a given joystick.

Declaration
```
  var
    stick: array [0..3] of byte absolute 632;
```

Remarks
  The direction of the specified joystick is returned as a value which can be interpreted as follows:



Example
```
  begin
    while (strig[0]=0) do     (* Wait for the trigger *)
      writeln(stick[0]);      (* Show direction *)
  end.
```

See also
  paddle,ptrig,strig

STRIG Array

Description
  Reveals the status of the specified trigger button.

Declaration
```
  var
    strig: array [0..3] of byte absolute 644;
```

Remarks
  The status of the specified trigger button is returned. A 1 means the button is not pressed; a 0 means it is pressed.

See also
  stick,paddle,ptrig

SUCC Function

Description
  Returns the next successive value of the argument.

Declaration
  function succ(x);

  (Where x is of any ordinal type.
   The return type is the same type as the argument)

See also
  pred,inc,dec

TRUNC Function

Description
  Truncates a real value into an integer value.

Declaration
  function trunc(x: real): longint;

Remarks
  Trunc converts the given real number to the nearest integer below the value given and returns this value.

  If the value is out of the range of a longint, an error will occur.

See also
  int,round

UPCASE Function

Description
  Returns the uppercase representation of a lower case letter.

Declaration
  function upcase(ch: char): char;

Remarks
  If the letter given is lowercase, the uppercase analog is returned, otherwise the character given is returned.

VAL Procedure

Description
  Converts a numeric string to a numeric value.

Declaration
```
  procedure val(s: string; var v: real; var p: longint);
```

Remarks
  Leading blanks in the string are skipped and an attempt to convert the string into a number is made. If successful, the value is stored in v and p is set to 0. If unsuccessful, v is undefined and the position of the error is stored in p.

See also
  str

WHEREX Variable

Description
  Contains the current column of the cursor.

Declaration
```
  var
    wherex: word absolute $55;
```

Remarks
  This variable can be used to modify the position of the cursor, but the cursor will not move until the next WRITE statement.

WHEREY Variable

Description
  Contains the current row of the cursor.

Declaration
```
  var
    wherey: byte absolute $54;
```

Remarks
  This variable can be used to modify the position of the cursor, but the cursor will not move until the next WRITE statement.

WRITE Procedure (Text Files)

Description
  Outputs formatted values to the screen or a file.

Declaration
  procedure write([var f: file of text;] v1[,v2,v3...]);

  (Where v1,v2,v3... are values of integer, character, string, real or boolean type)

Remarks
  If no file is specified the screen is assumed.

  Output begins at the current cursor position and all values specified are output one after the other. Output parameters have the syntax:

  value[:spacing[:decimal]]

  Where spacing is an integer value representing a padding of blanks that the value is to be output on, right justified. Decimal is the number of digits following the decimal point that are to be displayed and is valid only for real values. If the value is real, and decimal is not specified, the result is output in exponential form.

  If the value is of type BOOLEAN, the words TRUE or FALSE are output.

  If the value does no fit on the padding, the entire value is output regardless.

  Write should never be called with a function that calls a WRITE or a READ statement.

See also
  writeln,read,readln

WRITE Procedure (Typed Files)

Description
  Outputs a variable to a file.

Declaration
  procedure write(var f: file; v1[,v2,v3...]);

  (Where f is a file of any type except text, and v1,v2,v3... are a;; variables of the type of the file)

Remarks
  WRITE outputs all of the variables specified to successive file components.

See also
  read

WRITELN Procedure

Description

Outputs formatted values to the screen or a file.

Declaration

```
procedure writeln[([var f: file of text;] v1[,v2,v3...])];
```

(Where v1,v2,v3... are values of integer, character, string, real or boolean type)

Remarks

This procedure works exactly like WRITE on a text file, except that after all of the specified values are output, an end of line character (#155) is sent to the file.

WRITELN without any parameters sends an end of line marker alone to the file.

See also

write,read,readln

Version 1.1 Information

Corrections

Page six, as listed in the manual, is not free. String register #1 uses page six, not $1C00 as listed in the manual.

On page 50, a small fragment of code was listed with the MOVE procedure. The type declaration is in error and should be:

```
type chtype=array [0..127] of array [0..7] of byte;
```

Using CLSN Pascal with SpartaDOS

In order for CLSN Pascal to work properly with SpartaDOS, SpartaDOS must be configured to USE OSRAM.

Addendum

Three new procedures have been added to the CLSN Pascal library. These are described below.

UPDATE Procedure

- Description -
Opens a file for random access.

- Declaration -
procedure update(var f: file);

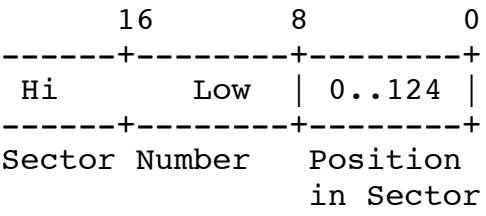
- Remarks -
A file should be opened using UPDATE when the NOTE and POINT procedures are to be used upon that file.

NOTE Function

- Description -
Returns the physical position of the file pointer.

- Declaration -
function note(var f: file): longint;

- Remarks -
The longint returned contains the sector and byte position in the following manner:



POINT Procedure

- Description -
Moves the file pointer to a given position.

- Declaration -
procedure point(var f: file; p: longint);

- Remarks -
The position the file pointer is sent to, p, must have the sector and byte position organized as described above.

An error occurs if the new file position is not part of the given file.