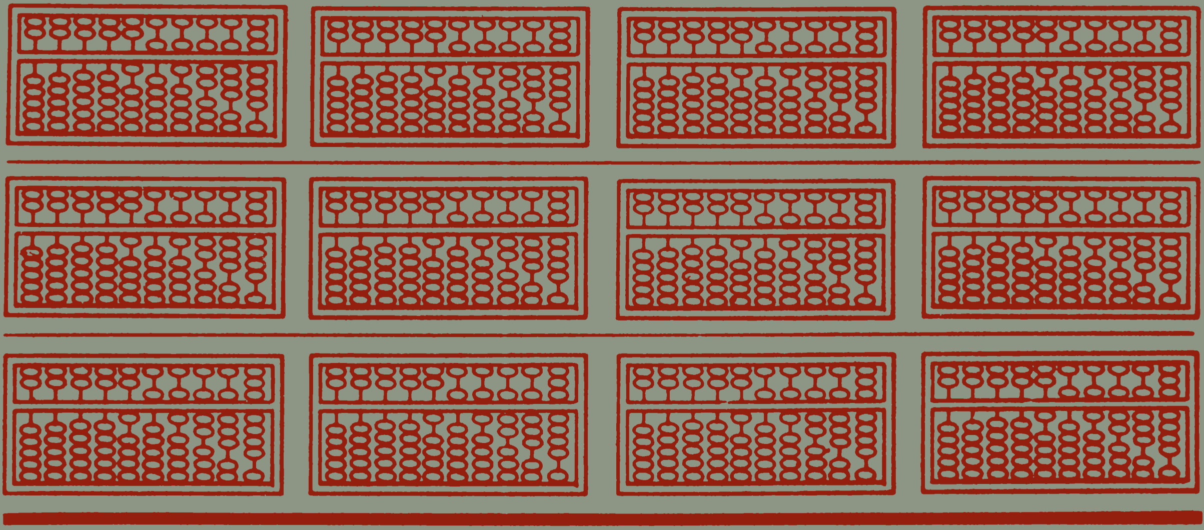
YOU CAN COUNT ON Abacus Software



SUPER Pascal

Compiler and Software Development System

By H. Schnepf

A Data Becker Product

Published by:

Abacus Software
P.O. Box 7211
Grand Rapids, MI 45910

Fourth Printing, November 1987
Printed in U.S.A.

Copyright © 1986	Data Becker GmbH Merowingerstraße 30 4000 Düsseldorf, West Germany
Copyright © 1986,1987	Abacus Software, Inc. 5370 52nd Street SE Grand Rapids MI 49508

ISBN 0-916439-41-0

FOREWORD

This is the handbook to your SUPER Pascal Development System for the Commodore 64 and 128. The minimum hardware required to run SUPER Pascal is a C-64 (or C-128 in C64 mode), a 1541, and a television or monitor. A second disk drive and printer will let you take full advantage of SUPER Pascal.

This handbook is designed to help you understand the workings of SUPER Pascal, assuming that you have had experience in the Pascal language. It is not a basic course in Pascal, and makes no claims to that effect. Before using SUPER Pascal, you should be familiar with the PASCAL language and how it works.

SUPER Pascal is a complete implementation of "Standard Pascal", based on the "Pascal User Manual and Report" (or "The Pascal Bible") by Kathleen Jensen and Niklaus Wirth. You'll find SUPER Pascal to be one of the most comprehensive Pascal systems ever offered for C-64 or similar machines. One of the problems with the C-64 is the slow transfer of data between computer and disk drive (250- 400 bytes per second); SUPER Pascal solves that problem, allowing you to transfer 1250 bytes per second from disk to computer and back. This means that disk access is increased threefold!

I should mention that when developing a comprehensive software package as large in scope as SUPER Pascal, errors may occur. We have done our best to make SUPER Pascal as bug-free as possible. Naturally if you do encounter problems, please let us know. Your suggestions are always welcome.

January 1985

H. Schnepf

TABLE OF CONTENTS

1.0 SUPER PASCAL - SYSTEM

1.1 SYSTEM OVERVIEW

Super Pascal works with one or two 1541 disk drives. If you are using two 1541 drives, the second 1541 drive should be designated as drive 9 (see your 1541 manual). SUPER Pascal refers to these as drive 0 and drive 1 (device numbers 8 and 9 respectively). Drive 0 acts as the master drive. SUPER Pascal searches drive 0 for the system programs, such as the Assembler, Compiler, Editor and Utility programs.

Keywords, commands, names and identifiers are entered in upper-case, i.e., ASCII codes 65-90 ($41-$5A). These codes are entered as unshifted keys in normal upper-case/graphics mode. In upper/lower-case mode, these characters are displayed as lower case. This feature is peculiar to Commodore machines, not Pascal.

The latter mode is the default when SUPER Pascal is initialized. Remember to type statements, identifiers and names in lower case when in this mode! If this confuses you, modes can be switched by pressing <C=+SHIFT>.

A few thoughts on the syntax for Pascal identifiers are in order here. These identifiers are names defined for constants, variables, filenames, procedures, functions, etc. Rules dictate that an identifier:

consists of no more than 8 significant characters

begin with a letter

must use letters and numbers for the remaining characters, as well as the character "_"(ASCII $5F), shown on the C-64 as a back-arrow.

Screen dumps (printer outputs of the screen) can be done with a suitable printer (Commodore, or another properly interfaced printer). Unusual printer set-ups can be "tuned" in by software changes.

A third item worth mentioning here is the input of direct commands, and response to the prompts. Generally, direct commands are issued using a single letter (e.g., "E" will call the Editor). Direct commands must be followed by pressing the <RETURN> key. Later descriptions of these commands do not mention this fact, so please keep this in mind. If the command requires further information, you will be prompted (NOTE: in some cases, a default value is available). If your input is illegal or invalid, the system will ignore it, and you'll have to re-enter the command correctly.

If the system is expecting a numeric value -- such as in the procedures `READ` and `READLN` -- and you input a non-numerical response, the system will respond `"IL.INPUT"`, and wait for the correct input. A <RETURN> without any other input is interpreted as a 0. Integer values can be input in decimal form or in hexadecimal form (e.g., 1024 = $0400) when preceded by a dollar-sign ("$"), .

Due to the limitations of the integer range (-MAXINT..+MAXINT [-32767..+32767]), any addresses from +32769 to +65535 should be specified using hexadecimal notation ($8000..$FFFF).

Another strength of SUPER Pascal is the system's high resistance to errors and bad input from the user. You'll be surprised at how well this program handles errors.

1.2 LOADING THE SYSTEM

Loading SUPER Pascal is extremely simple. After turning on the computer and disk drive(s), put the system disk into drive 0, and close the drive door. Then type `LOAD"*",8,1`<RETURN>. The autoboot (load/run procedure) takes over from there. If you happen to type `LOAD"*",8`<RETURN>, and leave out the 1, the system will respond with `READY`. In this case just type `RUN`<RETURN> to finish the loading process.

The autoboot displays the title screen, which asks you to press a key. Once you've pressed a key, SUPER Pascal will display:

```
LOADING ...

    *****  C-64  SUPER PASCAL - SYSTEM   5.3  *****
                   BY H. SCHNEPF
           (C)  COPYRIGHT 1985 DATA BECKER
              LICENSED BY ABACUS SOFTWARE
      *****************************************
```

When the load procedure is completed, Pascal is initialized, and the system displays the Main Menu:

```
        * C-64 PASCAL-SYSTEM 5.3. *
COMMANDS =  ...

A(SSEMBLER) H(ELP)       R(UNPRGM)
C(OMPILER)  J(UMP)       U(TILITY)
E(DITOR)    M(AP/DRIVE)  W(RITESRCE)
G(ETRAM)    P(UTRAM)


@
```

Note that the cursor is represented here by a "@"

The 64's BASIC operating system has now been temporarily replaced by the SUPER Pascal operating system; you can return to BASIC by shutting off the computer.

1.3 SHORT DESCRIPTION OF SUPER PASCAL

Let's look at the elements of the Main Menu individually -- this makes up the RUNPAC, a set of machine code routines and compiled Pascal routines, which allows us to create our own Pascal programs.

1.3.1 MAIN MENU

The Main Menu of SUPER Pascal has the following commands:

'A'	(=ASSEMBLER) Calls the 6510 assembler source-program.
'C'	(=COMPILER) Calls the Pascal compiler.
'E'	(=EDITOR) Calls the program to edit source files.
'G'	(=GETRAM) Load a file into memory from disk.
'H'	(=HELP) Prints Main Menu command list.
'J'	(=JUMP) Jump to machine code program found at specified address.
'M'	(=MAP/DRIVE) Displays disk directory.
'P'	(=PUTRAM) Saves specific memory area contents to disk.
'R'	(=RUN PROGRAM) Starts a Pascal program.
'U'	(=UTILITY) Calls utility program for working with files, etc.
'W'	(=WRITE SOURCE) Formatted output of source-file to disk or printer.

1.3.2 ASSEMBLER

The assembler is used to create 6510 machine code programs from an assembler source file. The machine code is stored as a file. The assembler source files must have the following form:

--> TEXT LINE   :ZZZZSLLLLLLLLSIIISOOOOOOOOOOOO...

ZZZZ	=	line number
S	=	space
LLLLLLLL	=	label field

maximum of 8 characters (same as for Pascal identifiers). Unused space in a label field is filled in with blank spaces.

III	=	instruction field

Operation codes (mnemonics) in 6510 assembler notation:

ASL	CLC	INC	PHA	SEI
BCC	CLD	INX	PHP	STA
BCS	CLI	INY	PLA	STX
BEQ	CLV	JMP	PLP	STY
BIT	CMP	JSR	ROL	TAX
BMI	CPX	LDA	ROR	TAY
BNE	CPY	LDX	RTI	TSX
BPL	DEC	LDY	RTS	TXA
BRK	DEX	LSR	SBC	TXS
BVC	DEY	NOP	SEC	TYA
BVS	EOR	ORA	SED	

Pseudo-operating-code notation:

.BA	=	begin assembly
.BY	=	insert byte
.CT	=	continue with source ...
.DL	=	define label
.DS	=	displacement
.EN	=	end assembly
.EQ	=	cond. assembly: equal 0
.NE	=	cond. assembly: not equal 0
.OC	=	object code clear
.OS	=	object code set
...	=	end of cond. assembly

OOOOOOOO..= Operand field

As operands are labels, decimal numbers, hex numbers and +/- combinations, the following types of addresses are permitted:

Operand	=absolute
Operand,X	=absolute indexed X
Operand,Y	=absolute indexed Y
(Operand,X)	=indirect indexed X
(Operand),Y	=indirect indexed Y
(Operand)	=indirect absolute
*Operand	=zeropage
*Operand,X	=zeropage indexed X
*Operand,Y	=zeropage indexed Y
#Operand	=immediate
#H,Operand	=immediate high-byte
#L,Operand	=immediate low-byte
A	=implicit Accumulator

A semicolon (;) at position 6, 15, or after 0 should precede any remarks or commentary.

1.3.3 COMPILER

The compiler compiles Pascal source programs from diskette. The user then has the option of putting the compiled program code (Pascal-Pcode) on disk, or keeping it in memory.

The compiler accepts and compiles the following reserved words found in the Pascal language:

WORDS:

AND	DO	FUNCTION	NIL	PROGRAM	TYPE
ARRAY	DOWNTO	GOTO	NOT	RECORD	UNTIL
BEGIN	ELSE	IF	OF	REPEAT	VAR
CASE	END	IN	OR	SET	WHILE
CONST	FILE	LABEL	PACKED*	THEN	WITH
DIV	FOR	MOD	PROCEDURE	TO	

* PACKED will not always compile!!

The following standard identifiers are also permitted:

Constants:	FALSE, MAXINT, TRUE
Types:	ALFA, BOOLEAN, CHAR, INTEGER, REAL, TEXT
Variables:	INPUT, OUTPUT

Procedures:

GET,	NEW,	PUT,
READ,	READLN,	RESET,
REWRITE,	WRITE,	WRITELN

Functions:

ABS,	ARCTAN,	CHR,	COS,
EOF,	EOLN,	EXP,	LN,
ODD,	ORD,	PRED,	ROUND,
SIN,	SQR,	SQRT,	SUCC,
TRUNC			

In addition, all the standard characters mentioned in The Pascal Users Manual and Report are accepted by the compiler:

Symbols:

.	,	=	<	[	(*
-	.	<>	<=	]	*)
*	..	>	(	-[	^
/	;	>=	)	]-	’
:=	:				

The following commands, etc., are additions to SUPER Pascal Development System:

Reserved Words:

AND	combines BYTE-values
ELSE	used as an alternative in a CASE statement
NOT	negates a BYTE-value
OR	ORr's BYTE-value
SHL	rapid integer multiplication by a factor of 2^n (n=0 to 16) without overflow checking

SHR	rapid integer division by a divisor of 2^n (n=0 to 16)
USERFUNC	declares an external function written in machine code
USERPROC	declares an external procedure written in machine code
XTRNFUNC	declares/defines an external Pascal function
XTRNPRGM	declares an external Pascal program
XTRNPROC	declares/defines an external Pascal procedure

Other Indicators:

Constants:

PI	the real number pi (3.141...)
STKPOI	the value of the variable stack pointer

Types:

BYTE	a single-byte, preceded by the character "#" (range: #0-#255 = #$00-#$FF)
STRING	a dynamic array of CHAR, defined as : RECORD LENGTH:BYTE; CHRS:[#0..L] OF CHAR END

Variables:

MEM	a pseudo-variable array that permits access to memory (similar to PEEK and POKE in BASIC)
RANDOM	a pseudo-variable that produces a random real number between 0 and 1;

Procedures:

ALLOCATE	sets pointer variables to an address accessible to the user.
CLOSE	close and clears the file in last buffer
CONTINUE	load and run a Pascal program from diskette
CLRTRAP	clear trap of runtime I/O errors
EXECUTE	load a Pascal program from disk, and run subprogram already in memory

HEX	converts integer or byte parameters to hexadecimal
INDVC	redirect input from device
KILL	delete an unprotected (unlocked) file from diskette
LOCK	close and protect a file on diskette
LOAD	loads external Pascal program, procedure or function into memory from diskette
MARK	records the current heap pointer
NAME	assign a filename to a file
OUTDVC	redirect output to device
RELEASE	set heap pointer to previously MARKed value
SEEK	set file position pointer for direct access
SETADR	define starting address for an external Pascal or machine language routine
SETDRV	define disk drive for file access
SETTRAP	activates trap for runtime-error (I/O errors)

Functions:

ANYKEY	returns TRUE if any keyboard input is present
EOF	returns TRUE if end of file or BREAK key pressed (:BOOLEAN)
FRAC	returns the fractional part of a real number (:REAL)
FREE	returns the amount of memory remaining on Pascal variable stack (:INTEGER)
GETKEY	returns the value of the next key in keyboard buffer; otherwise, waits for next key (:CHAR)
HBYT	returns the most significant byte value of an integer (:BYTE)
HXS	(=hexsum), returns the sum of two integer values without checking for overflow, used for calculating addresses (:INTEGER)
INT	returns the integer value of a real number or gives IL.QUANT.ERROR (:INTEGER)

`IOERROR`	returns the value for the I/O error `(:INTEGER)` as follows:

```
 0 = OK
 1 = DISK ERROR
 2 = NOT OPEN ERROR
 3 = NOT CLOSED ERROR
 4 = BUFFER OVERFLOW ERROR
 5 = DIRECTORY OVERFLOW ERROR
 6 = NOT FOUND ERROR
 7 = DISK OVERFLOW ERROR
 8 = DISK MISMATCH ERROR
 9 = ILLEGAL FILE-OPERATION ERROR
10 = AFTER EOF ACCESS ERROR
11 = IEEE-ERROR
```

`LOCALITY`	returns the current memory location of Pascal variables `(:INTEGER)`
`LOW`	converts an integer or a single number into high-byte,low-byte (when possible)
`LBYT`	returns the least significant byte value of an integer `(:BYTE)`
`LEN`	returns the length of a string `(:INTEGER)`
`ROUND`	returns the rounded value of any real number `(:REAL)`
`SIGN`	gives previous item an integer or a real value `(:INTEGER)`
`SIZE`	returns the number of bytes occupied by a Pascal variable `(:INTEGER)`
`TRUNC`	returns the integer portion a real value `(:REAL)`

Structural Commands:

These commands aid in structuring Pascal programs:

`FORWARD`	for forward definitions of PROCs and FUNCs according to "The Pascal Bible"
`SEGMENT`	to break a Pascal program into segments used for overlay techniques

`ASSEMBLE`	converts the text to follow from Pascal into assembly language.

Compiler Directives:

These commands change the defaults of the compiler.

`&ADR+`	activates output of addresses during compilation
`&ADR-`	deactivates output of addresses
`&CONTINUE`	instruct the compiler to continue compilation on the given source-file
`&INCLUDE`	instruct the compiler to include given sourcefile in the compilation presently being done
`&PCODE+`	activates the P-code output
`&PCODE-`	deactivate P-code display
`&TRUTH`	identify section of source file for conditional compilation

Error Messages:

The standard error messages identified by the compiler (according to the Pascal User Manual and Report) are as follows:

22:	'..' expected
23:	'.' expected
24:	',' or ')' expected
25:	BOOLEAN constant expected
60:	PROGRAM incomplete
182:	Parameter list of extern PRGM not allowed
183:	LOAD/SETADR only for externals
184:	Externals without address definition
185	Slice-ARRAY must be CHAR or BYTE type
186:	SLICE := SLICE not allowed
207:	BYTE-const too large
208:	Error in BYTE-const
209:	Error in HEX-const

210: Error in numeric const

400: FILE-element too long
401: STRINGS not allowed here
402: Too many identifiers
403: READLN/WRITELN only with TEXT
405: Too many segments
406: Nested segments not allowed
407: Separated segments not allowed
408: Compiling of segmented PRGMS to RAM not allowed
409: Too many parameters
410: Error in '&'-option
411: Too many nested sources

Runtime Errors

Runtime errors can also include I/O errors:

OUT OF RNG.	number out of range
NOT EXQ.	non-executable P-code
NUM. OV.	number overflow
BAD SUBS.	bad subscript
ILL. QUANT.	illegal quantity
STK. OV.	stack overflow
ZERO DIV.	division by 0
ILL. DVC.	illegal device number

Options: The following items may be changed when the compiler is started (contents in parentheses are defaults):

Start-of-program	($0800)
Starting address of heap	(end-of-program)
Max. address of variable stack	($9000)
Compiling mode	(disk), or RAM:
Memory location for comp.	($9000)
Test for end-of-memory	(yes) , or no:
File for post-mortem dump	(no) , or yes:
post-mortem filename	(P_M_DUMP)
Suppress program listing	(yes)
Suppress printer output	(yes)

1.3.4 EDITOR

The editor sets the source-program into a screen-oriented format. The line numbers displayed in edit mode are there for editing only -- they aren't part of the program itself. The following commands available for changing parameters in edit mode :

`'A:'`	`(=APPEND FILE)` Append specific file on diskette to file in memory.
`'C:'`	`(=CHANGE)` Change the character string following ':' to another string.
`'D'`	`(=DELETE)` Delete lines:

`D`	delete ALL lines
`D xxxx`	delete line xxxx
`D -xxxx`	delete up to and including line xxxx
`D xxxx-`	delete from line xxxx on
`D xxxx-yyyy`	delete from line xxxx to line yyyy

`'F:'`	`(=FIND)` Find and list the line containing the specified character string.
`'G:'`	`(=GET SOURCE FROM DISK)` Load a source file from diskette into the Editor.
`'H'`	`(=HELP)` Display Editor's command set.
`'L'`	`(=LIST)` List line(s); parameters are similar to 'D'.

'M'	(=MAP/DRIVE) Display disk directory; defines drive for 'A:', 'G:', 'P:', and 'U:' commands.
'N'	(=AUTO-NUMBERING) Automatically generate line numbers in increments of 5, with an option of changing the starting line number (Nxxxx).
'O'	(=OUTPUT DEVICE) Change output device for display to screen or printer.

0	-----	screen
04,0	-----	printer

'P:'	(=PUT SOURCE TO DISK) Save source file from editor to diskette (NOTE: If a file of the same name already exists on the diskette, the old file is overwritten).
'Q'	(=QUIT) Return to the Main Menu.
'R'	(=RENUMBER) Renumber lines in increments of 5, starting at line number 1000.
'S'	(=SHIFT LINE) Move line(s) to a different memory range (S xxxx - yyyy : zzzz ... move lines xxxx to yyyy to location after line zzzz).
'U:'	(=UPDATE FILE) Append source file in Editor to file on diskette.
'V'	(=VACANCY) List amount of memory left for text.

1.3.5 UTILITY

The Utility function has disk management commands, as well as some useful monitor commands; this section gives you working memory in $4000 - $C200, and this register can be used as standard RAM. Here are the commands:

'A'	(=ADVICE) Display any special information on a given file (data, version number, etc.).
'B'	(=BLOCKTABLE) Display a diskette blocktable (similar to block availability map).
'C'	(=COPY FILE) Copy file from one diskette to another.
'D'	(=DUPLICATE DISK) Duplicate an entire disk (only possible with two drives).
'E'	(=ENTER SECTOR) Store any sector (=512 bytes) of memory to diskette.
'F'	(=FETCH SECTOR) Load any sector of disk into memory.
'G'	(=GET FILE FROM DISK TO MEMORY) Load a file from diskette.
'H'	(=HELP) Display the Utility command list.
'I'	(=INSERT ADVICE) Input extra information (see ADVICE) to file on diskette.

'J'	(=JUMP) Jump to any program in memory.
'K'	(=KILL FILE) Scratch file from diskette.
'L'	(=LOCK FILE) Protect a file on diskette from killing of overwriting. Locked files appear in the directory in reverse video.
'M'	Display the disk directory. Also defines the drive for 'A', 'B', 'E', 'F', 'I', 'K', 'L', 'R', 'U', 'X' and 'Z' commands.
'N'	(=NEW MAP) Generate new directory (in disk-formatting and producing system disks).
'O'	(=ORGANIZE DISK) Reorganize disk contents; pack two disks' worth of material to one disk, giving more memory space (possible only with two drives).
'P'	(=PUT MEMORY AS FILE TO DISK) Store any memory range to diskette as a file.
'Q'	(=QUIT) Return to Main Menu.
'R'	(=RENAME FILE) Change the name of a file.
'S'	(=STORE BYTE INTO MEMORY) Place a value into any memory cell in the computer (similar to POKE).
'T'	(=TRANSFER MEMORY-PAGES) Transfer any one of 256 bytes to another area in memory.

'U'	(=UNLOCK FILE) Unlock file protection.
'V'	(=VIEW MEMORY) List any memory range in hexadecimal OR ASCII (memory dump).
'W'	(=WRITE DIRECTORY) Output all additional information in the disk directory.
'X'	(=XCLUDE BLOCK) Exclude a block on diskette from further use.
'Y'	(=FILE DUMP) List file on diskette in hex or ASCII.
'Z'	(=RELEASE BLOCK (SET ZERO)) Release used or kept block to diskette for later reference.

1.4 SYSGEN - SETTING UP YOUR SYSTEM

As already mentioned, SUPER Pascal supports the use of two floppy disk drives. However, the limitations of using only one drive are so minimal that you could easily get along with one drive (only a few of Utility Menu commands require two drives -- 'D' and 'O').

We'd now like to offer a few words of advice on the use of SUPER Pascal.

First, please keep in mind that copying the original disk for your own personal use is possible -- but the Compiler and Assembler on that backup won't run properly. All the other programs should run just fine, though.

The segmenting by the compiler (overlay-technique) requires the original diskette to be in drive 0. Similarly, the assembler looks for the source file in drive 0. Unfortunately, if you're using only one disk drive, the source code produced is saved on the original disk. It is best to use the system diskette only for compiling and assembling.

1.4.1 MAIN DISKETTES

Let's have a look at the procedures for formatting a Super Pascal disk:

Several basic disks can be created using SYSGEN, called using the 'R' command from the main menu. The program displays a header, and asks in which drive the new disk lies:

```
*   PASCAL-SYS.DISK. GENERATOR   *
************ vs 5.3 *************

'DRIVE(MAP) = 0'
```

The default drive is 0. Next, you'll be asked for the disk title -- supply a name for the disk. Next comes the message:

```
    INSERT DISK INTO DRIVE x
    ...PRESS:  "RETURN" IF DONE!
```

Just to make sure, the system will ask

```
SURE  TO REWRITE THE DISK ?  Y/N
```

since generating a diskette will destroy any old material previously on the disk.

If all is well, the program will format the diskette, put in a directory under the given name, and put a LOADDAT file onto the diskette.

WARNING!!!

A diskette formatted by SYSGEN is readable ONLY by Super Pascal -- you can't use this disk in BASIC, unless you format it normally. With a SYSGEN disk, it is vital that LOADDAT -- which contains the Pascal operating system -- be on the disk.

From the file UTILITY menu you can clear a Pascal directory using 'N', duplicate a disk with 'D', and reorganize data with 'O'.

If a read/write error occurs during formatting, you'll see the following error message:

```
FORMATTING OR FLOPPY ERROR!
...EXECUTION NOT SUCCESSFUL!

REPEAT WITH ANOTHER DISK ? N/Y
```

Try again; or, if you tell the system "n", it will go back to the Main Menu.

1.4.2 WORK DISKETTES

Now, using the file UTILITY program and the COPY command ('C'), you can make work disks of your choice, e.g.:

An Editor Disk would be make up of LOADDAT and C_EDITOR (_ represents the back arrow key). You could use such a disk for developing, editing and storing Pascal or assembly language source programs.

A Utility Disk would contain LOADDAT, C_UTILIT, C_PMDUMP and SYSGEN (more on this in Section 4.6). This is a good choice for some quick system work.

A Program Disk containing LOADDAT and the compiled Pascal programs and/or assembled machine-code programs of your choice. This would essentially be a user program disk, which would run on any C-64 without the help of the original diskette.

1.4.3 COMPILER DISKETTES

Once you've copied the different programs off of the original diskette (with the exceptions of the Compiler and Assembler, which are copy-protected), and put them into work diskettes to suit your own needs, you may want to delete those files from the original diskette (K command in Utility Menu). After doing so, you'll be left with LOADDAT, C_CPLR (Compiler) and C_ASMBLR (Assembler), as well as 25 blocks (= 100 kilobytes) available for assembling and compiling larger programs. When you are ready to compile you copy the source program from the work diskette (if you haven't a second disk drive) to the compiler diskette for writing and reading program code. We're following one of the oldest rules in computing here: Make backups whenever possible, and use the original only when necessary.

NOTE:

During compiling and/or assembling, the respective program will put a temporary file (or set of files) on diskette, which can be found by the source program. At least 3 blocks must be free on the diskette if you are running only one disk drive. The first temporary file (CODDAT) becomes the necessary program code after compiling/assembling; CODDAT is deleted after the compilation/assembly. The temporary files can be accessed ONLY if a break or error occurs during the compiling or assembly process.

We realize that, at first glance, the material given so far can look pretty intimidating to the beginner. Rest assured that, like BASIC, the more you work with this language, the more experienced you'll become in controlling its inner workings. Good luck with SUPER PASCAL!

2.0 MAIN MENU

The Main Menu is the outer-most command set of SUPER Pascal; it gives you access to the primary system programs, such as the Assembler, Compiler, Editor, etc., or you can use it to run your own programs. After user-written programs run, an "OK" message appears, and you are returned to the Main Menu. The cursor is displayed in the Main Menu as a '@' sign.

This menu also gives you the ability to load specific memory registers from disk or to save any memory range to disk.

The following is displayed when in the Main Menu:

```
          *  C=64 PASCAL-SYSTEM  5.3 *

COMMANDS = ...
A(SSEMBLER) H(ELP)       R(UNPRGM)
C(OMPILER)  J(UMP)       U(TILITY)
E(DITOR)    M(AP/DRIVE)  W(RITESRCE)
G(ETRAM)    P(UTRAM)
```

These are the direct commands mentioned earlier in this manual, which we will now cover in detail. Remember that all commands and responses to input must be followed by a <RETURN> (see 1.1).

2.1 MAIN MENU COMMANDS

2.1.1 A (= ASSEMBLER)

This command calls the onboard 6510 assembler, which allows you to convert 6510 assembly language into 6510 machine code. The assembler looks for an assembler source-program file on diskette, and will ask for input concerning this file:

```
FILE-TITLE = ?
DRIVE(MAP) = X
```

The default value of X is the number of the last disk drive used, so a <RETURN> here will usually suffice.

You could use an asterisk (*) instead of an actual file-name; this instructs the assembler to assemble the first textfile found. The assembler next offers a verification of filename and corresponding drive number:

```
CONFIRM "FILENAME,DRIVE_NR"? N/Y
```

Incorrect input of any kind will return you to the Main Menu. If all input is correct the assembler will load and run. This process begins with the loading of the file LOADDAT; both LOADDAT and the assembler program (C_ASMBLR) MUST be in drive 0. If the given name of the textfile isn't found, the assembler generates an error message, and returns you to the Main Menu. If the given file cannot be handled as a textfile, an error message will appear, and you return to the Main Menu.

The individual commands and operation of the assembler are handled in Chapter 5.

2.1.2 C (= COMPILER)

This command puts you in the compiler section, which allows you to create Pascal programs. One very important feature to this compiler is the fact that it accepts mixtures of 6510 assembly language and Pascal. When you press 'C' in the Main Menu, you will get prompts similar to those found in the assembler:

```
FILE-TITLE = ?
DRIVE(MAP) = X
```

The default value of X is the number of the last disk drive used, so a <RETURN> will usually suffice.

You could use an asterisk (*) instead of an actual filename; this instructs the compiler to assemble the first textfile found. The compiler next offers a verification of filename and corresponding drive number:

```
CONFIRM "FILENAME,DRIVE_NR"? N/Y
```

Incorrect input of any kind will return you to the Main Menu. If all input is correct the compiler will load and run. This process begins with the loading of the file LOADDAT; both LOADDAT and the compiler program (C_CPLR) MUST be in drive 0. If the given name of the textfile isn't found, the compiler generates an error message, and returns you to the Main Menu. If the given file cannot be handled as a textfile, an error message will appear, and you return to the Main Menu.

The program operation and individual commands of the compiler can be found in Chapter 4.

2.1.3 E (= EDITOR)

This command loads and runs LOADDAT, then the text-editor (the file C_EDITOR) from drive 0.

Assembler and Pascal source-programs can be modified using the editor, then saved to diskette in Pascal DOS. Chapter 3 contains the individual editor commands.

2.1.4 G (= GET FILE FROM DISK TO MEMORY)

This command loads any file into memory from diskette; this is especially useful for temporarily storing information, as well as specifically loading programs. The 'G' command will ask for input on the following parameters:

```
START-ADR. = ?
```

Input the starting address of the file to be loaded. As already mentioned in 1.1, the address can be input either in decimal or hexadecimal.

```
FILE-TITLE = ?
```

Input the name of the desired file.

```
    DRIVE(MAP) = X
```

Give the number of the drive containing the file. The default value for X will be the number of the last drive used, so you could just press <RETURN>, unless the file is in the "other" drive.

If all input is correct, the routine will load the file from diskette. The load routine is part of the system diskette program LOADDAT, which must be kept in drive 0. If this is not the case, or if the file is not found, a corresponding error message will be given, and program control will return to the Main Menu.

NOTE:

The 'G' command doesn't check to see if there is enough memory to hold the file being loaded, nor does it see if the memory address given matches the file's starting address. The file will be loaded at the stated starting address, and will end at the EOF (end-of-file) marker supplied on the file. The 'G' command can utilize the memory space from $0800 to $BBFF. This can be extended to include screen memory ($0400 - $07FF).

After loading, the end address (END ADDRESS + 1) is displayed; and program control returns to the Main Menu.

2.1.5 H (= HELP)

This command calls the complete command list, just to remind you what's available.

2.1.6 J (= JUMP)

This command lets you jump to any machine-language or Pascal routine in memory:

```
        START-ADR. = ?
```

Input starting address of the routine.

NOTE:

If you give the starting address of an incomplete, or non-debugged program, you may lose control of the system.

Memory from $0800 to $BBFF is at your disposal for programs. When working with a machine-language program, you could insert RTS, which will return you to the Main Menu, as long as locations $0028-$004F, $0340-$0379 and $BC00-$F2FF are unchanged. Another method would be to put in the m/l command JMP $C200, which also returns you to the Main.

2.1.7 M (= MAP/DRIVE)

The 'M' command displays the contents of a disk (the directory, or MAP) onscreen:

```
        DRIVE(MAP)= X
```

Response to this prompt will display the directory in the drive number given (default value of X is the drive number last used, so a <RETURN> will do in most cases).

The directory output is accomplished with the help of a routine in LOADDAT, so it is vital that LOADDAT be in drive 0 when the 'M' command is used.

A reminder: The directory in Pascal DOS is designed quite differently from that of "normal" Commodore DOS 2.6; in fact, SUPER Pascal cannot read a directory made under the standard operating system, nor can BASIC read a Pascal disk. With the exception of 22 blocks (with a standard block-size of 256 bytes each), the rest of the system disk is under Pascal DOS.

The directory will tell you the filenames and the amount of memory left on the diskette. Remember that a block in Pascal DOS is equal to 4k (4096 bytes), as opposed to the 256 bytes per block in DOS 2.6.

The directory of a system disk looks something like this:

```
   MAP OF DISK "PASCAL" :
LOADDAT  SYSGEN   C_EDITOR C_UTILITY
C_CPLR   C_ASMBLR C_PMDUMP
DISC 0 = 18 //
BLOCKS FREE !
```

Locked (protected) files appear with names in reverse video. For more information on locking and unlocking files, please see the chapter on utilities.

More detailed information concerning Pascal DOS and new disk commands can be found in Chapters 6 (Utility) and 7 (System- Specific Information).

2.1.8 P (= PUT MEMORY AS FILE TO DISK)

This command is the opposite of 'G' -- it saves any portion of memory to diskette as a data file. It will allow you to generate any specific information (data, program, etc.) on a file presently in memory, and put the information into the directory. The following parameters must be taken care of:

```
START-ADR. = ?
```

Input the address at which the information to be saved begins (as before, in either decimal or hexadecimal notation).

```
END-ADR.+1 = ?
```

Input the number immediately following the end address of the register (e.g., if the material stops at $0A00, input $0A01).

```
FILE-TITLE = ?
```

Type in the name as you wish to have it appear on the directory, bearing in mind these rules:

* Identifiers have up to eight characters.

* Identifiers must begin with an upper-case character.

* Remaining characters in an identifier must be upper-case characters, numbers and '_'.

```
DRIVE(MAP) = X
```

Give the drive number, or press <RETURN> for the default value.

After all parameters are in, on condition that no errors have occurred, the save process calls LOADDAT, and stores the file on diskette. As before, LOADDAT must be in system drive 0, or the routine will not work.

NOTE:

If there is a file of the same name already on the target disk, this older file will be scratched and replaced by the file being saved; in short, you'll lose the old file. There is an exception to this -- if the older file is locked (protected), you'll get the error message "ILL.FILE OPR. ERROR!".

If there isn't enough space on the disk, or if the disk has a write-protect tab, a respective error message will be displayed, and the 'P' command breaks off.

During a save, the memory configuration shifts: $0000-$CFFF is RAM; $D000-$DFFF is for I/O; and $E000-$FFFF contains the ROM (Kernal).

Barring errors, the program returns to the Main Menu.

2.1.9 R (= RUN PROGRAM)

The 'R' command gives the user the ability to call and run any compiled Pascal program on diskette. The command automatically loads the program into memory, and starts it, after filling in these parameters:

```
    FILE-TITLE = ?
```

Input the filename.

```
    DRIVE(MAP) = X
```

Give the corresponding drive number (or <RETURN> for default). After correct input, the program is loaded with the help of LOADDAT (read from drive 0); if LOADDAT cannot be found, an error message is displayed, and the 'R' command is ignored.

Here are two simple methods for calling programs:

a) After compiling a program, respond to the filename prompt with "*".

b) Call a program in 'R' mode using "*".

These cases assume that the system will immediately be able to find the program on disk.

There are times when runtime-errors will happen (i.e., problems during a program run); when this happens, the program returns you to the Main Menu, and gives you the error message and address of the error, thusly:

```
    ... ERROR IN $....
```

Here is a short list of runtime-errors:

`OUT OF RNG. ERROR!`	number out of legal range
`NOT EXQ. ERROR!`	program code cannot be executed

`NUM.OV. ERROR1`	numerical overflow beyond a predefined integer range
`B.SUBS. ERROR`	bad subscript (array index)
`IL.QUANT. ERROR!`	illegal quantity
`STK.OV. ERROR!`	overflow of stack (variables)
`ZERO-DIV. ERROR!`	division by zero
`IL.DVC. ERROR!`	illegal device number
`FLOPPY ERROR!`	error in data transfer via disk drive
`NOT OPEN ERROR!`	file not open
`NOT CLO. ERROR!`	`RESET/REWRITE` attempted on an open file
`BUF.OV. ERROR!`	attempt to use more than three file buffers
`DIR.OV. ERROR!`	not enough directory space
`NOT FND. ERROR!`	file not found
`DSC.OV. ERROR!`	not enough memory on diskette
`DSC.MISM. ERROR!`	illegal/ non-matching diskette
`IL. FILE OPR. ERROR!`	illegal file operation
`AFTER EOF ERROR!`	attempt to read file after EOF
`IEEE-ERROR!`	data transfer error in IEEE-bus

A successful program run will end with the message "`OK`" displayed.

2.1.10 U (= UTILITY)

This command loads and starts the utility section of the system diskette, first loading LOADDAT (in drive 0) and C_UTILITY.

The utility program permits a simple file-management system. However, you also get access to a set of monitor functions in this menu. In addition, 'U' mode lets you load and run programs without having to resort to LOADDAT, making the system disk unnecessary once the Utility Menu is loaded!

The idiosyncrasies of this menu are covered in Chapter 6.

2.1.11 W (=WRITE SOURCE)

'W' gives you a hardcopy (printout) of a source program. Essentially, this command will let you print out any text file, with line numbers to help you in debugging. These line numbers are NOT part of the program itself -- they are there as an aid to the user.

You have the option of either printing the program on a continuous-feed sheet (no pagination,etc.), or printing it out in a readable format, with page headers.
Once you choose 'W', you'll have to answer a few prompts:

```
    FILE-TITLE = ?
```

Input filename of the text to be printed.

```
    DRIVE(MAP) = X
```

Input drive number, or press <RETURN> for default.

After input, LOADDAT is loaded and run (did you remember to leave it in drive 0?). If the file isn't found, or if it isn't a textfile after all, the command will break off, and display an error message.

Assuming the WRITE routine hasn't hit any problems, a new set of parameters are displayed:

```
    PRT-DEVICE = 4,0
```

If necessary, you can change the primary (default 4) and secondary (default 0) addresses to suit your own printer.

```
    LINES/PAGE = 72
```

This is for page formatting -- the number 72 represents the total number of lines per page.

Once input is completed, the printing begins immediately; you may stop the printout at ant time using RUN/STOP, which will send you back to the Main Menu.

If you should have a different form of printer (different from a serial-port printer), you can change the primary address (4 = printer in serial port/ 5 = user port). Both device addresses reside in a subroutine at $CA03. Changing the device address can be done at $0373 (change to either 4 or 5).

2.2 EXIT TO BASIC

SUPER Pascal will return to BASIC when you press the RUN/STOP and RESTORE keys, which executes a RESET routine and does a BASIC cold-start. As long as $C200-$FFFF remains unchanged, you can get from BASIC back into SUPER Pascal by typing SYS 49664, which puts you in the Main Menu.

3.0 TEXT EDITOR

The editor is started from the Main Menu by pressing 'E'. If you make a diskette for editing, be sure to include LOADDAT -- again, LOADDAT is a necessity for booting this section -- in addition to the editor itself (C_EDITOR).

The following message is displayed in edit mode:

```
     *  C=64  SOURCE-EDITOR  5.3  *

COMMANDS = ...
A:(PPENDSRC)    L(IST)        Q(UIT)
C:(HANGE)       M(AP/DRIVE)  R(ENUMBER)
D(ELETE)        N(UMBERING)  S(HIFTLINE)
F:(IND)         O(UTPUTDVC)  U:(PDATESRC)
G:(ETSOURCE)    P:(UTSOURCE)  V(ACANCY)
H(ELP)
```

No cursor is displayed in edit mode. In this mode, you can edit Pascal and assembler programs as textfiles, and save them to disk for compiling/assembling later; this mode supplies 43000 bytes of memory available to the user. Note that auto-repeat is in effect for all keys.

Essentially, the editor lets you edit and augment programs, with line numbers supplied during editing. Each line can be 80 characters long - just as in BASIC - and you have full control of the normal screen editing keys (cursor up/dn/lft/rt;insert/delete). Revised lines are "installed" by pressing <RETURN> when you're through editing. If you type in a line number and <RETURN> only, and that number already exists, said line will be deleted.

Lines can be edited in any order, at any time; just move the cursor to the line in question, correct, and press <RETURN>.

There is one small limitation in editing: It is impossible to start a text line (i.e., immediately following a line number) with a number. If you do so, you'll get one of two messages:

```
ILLEG. LINE#!
EXECUTION NOT SUCCESSFUL!
```

If a line is typed in without line number, the first character will be read as a command, and again, you'll probably get

```
EXECUTION NOT SUCCESSFUL!
```

since the system will be confused by the number.

If command input is wrong, two common error messages are

```
ILLEG. SYNTAX!
EXECUTION NOT SUCCESSFUL!
```

These are the remaining error messages:

```
ILLEG. INPUT!
EXECUTION NOT SUCCESSFUL!
     (illegal device number)

ILLEG. TITLE!
EXECUTION NOT SUCCESSFUL!
     (illegal filename)

TITLE UNDEFINED!
EXECUTION NOT SUCCESSFUL!
     ('*' used for unspecified filename)

RAM OVERFLOW!
EXECUTION NOT SUCCESSFUL!
     (insufficient memory)

COMMAND IGNORED!
     (use of undefined command abbreviation)
```

Other errors encountered will be I/O errors, which will display messages, but will not dump you from the editor, or destroy your file.

3.1 EDITOR COMMANDS

Some of these commands have a colon (:) appended to them; the reason for this is a string or set of numbers are expected to follow. If mistakes are made in giving input, you'll be greeted with a syntax error. Remember, too, that all input must be concluded with <RETURN>.

3.1.1 A: (= APPEND FILE)

This command permits appending files on disk to files already in memory. Its syntax sounds like this:

```
A:FILENAME
```

with FILENAME representing the file to be appended (added).This means that the file is taken from the last disk drive used (which should be 0 immediately after the editor starts, but you can change that with the 'M' command). The editor will ignore any illegal input, and respond with an error message. When correct input has been supplied, the editor will get the file from disk, and append the two programs.

To avoid any conflicts, the second file (the one to be appended) should be shifted above the last address of the original file ('S').

If errors are encountered (file not found, file not a textfile, etc.), the procedure is stopped, but the original file will remain behind. On the chance that you run out of memory, the error message will read

```
RAM-OVERFLOW!
EXECUTION NOT SUCCESSFUL!
```

See 3.5 for help with memory trouble.

3.1.2 C: (= CHANGE)

This command makes it possible to replace any text string with a new string. Syntax:

```
C:STRING_OLD
```

refers to the old string. Alteration reads:

```
TO:STRING_NEW
```

A string can be defined as any character or set of characters found on the keyboard, and printed onscreen. The editor uses all material following the colons (:). Unused columns are filled in with blank spaces (NOTE: Do not end strings with a space yourself).

If the change involves replacing a short string with a longer one, see that the line doesn't have more than 80 characters, or this error message will turn up:

```
LINELENGTH EXCEEDED IN LINE:
... CURRENT TEXT LINE ...
```

You will have to go in and change this line "by hand"; 'C:' will not operate with overstepped lines. Errors will not cause you to lose your text, though (for additional help, see 3.5).

3.1.3 D (= DELETE)

This command deletes a line, or a number of lines, specified by the user.

```
D
```

alone will delete all text in the editor. You'll get a warning --

```
SURE TO DELETE THE COMPLETE SOURCE? Y/N
```

-- to avoid deleting something you may not want dumped. Respond 'Y' if you want to dispose of the text.

```
    Dxxxx
```

deletes line #xxxx; this is equivalent to typing just the line number with no text following.

```
    D-xxxx
```

deletes from beginning-of-file to line #xxxx.

```
    Dxxxx-
```

deletes from #xxxx to end-of-text.

```
    Dxxxx-yyyy
```

deletes from xxxx to yyyy. If yyyy is a number less than xxxx, then no text is scratched.

Input not following these patterns will be ignored, and treated as syntax errors, excepting input using additional spaces between parameters.

3.1.4 F: (= FIND)

The 'F:' command is handled much like the 'C:' command; it allows you to find any text string:

```
    F:STRING
```

The editor will then list all lines containing this string. The listing can be stopped and started by pressing the spacebar. The RUN/STOP key aborts the listing, and halts the 'F:' command.

3.1.5 G: (= GET SOURCE FROM DISK)

This command will load a textfile from diskette for editing. The command syntax is similar to 'A:':

```
G:FILENAME
```

FILENAME, of course, refers to the file to be loaded from the last drive used, or the drive stated by the 'M' command.

An asterisk (*) can also be used for FILENAME, provided that '*' has been predefined (see also 'P:'). If no such file has been defined, or if an error has been caused from 'A:' or 'U:' commands, you'll see

```
TITLE UNDEFINED!
EXECUTION NOT SUCCESSFUL!
```

onscreen; if this, or some other error message comes up, the command given by the user will be ignored.

Immediately after all proper input, the 'G:' command will load the file requested from diskette into memory. The system will arrange the file into lines numbered in fivefold steps, beginning at 1000 (i.e., 1000,1005,1010,1015,etc.). NOTE: The line numbers are there for your convenience only--they are not in fact part of the file itself.

Errors, such as file not found, no textfile, read error, etc., will stop the command, and send you back to the editor. Whatever text loaded into the system before the error will be available to you.

If there isn't enough memory to handle the file, this message appears:

```
RAM OVERFLOW!
EXECUTION NOT SUCCESSFUL!
```

However, you WILL be able to edit the text loaded up to the time of the overflow.

NOTE:

Any text in memory when the 'G:' command is called will be lost and overwritten by the new material. Be sure that this old material is saved before calling a new file. If you choose not to save it, the 'G:' command will ask:

```
SURE NOT SAVING THE SOURCE? Y/N
```

giving you the option of saving or not.

3.1.6 H (= HELP)

'H' prints the complete command set onscreen, to remind you of all sections of the program (MAIN/EDITOR/UTILITY).

3.1.7 L (= LIST)

This command allows you to list all or part of the textfile for review or debugging, using the "artificial line numbers". Here are the individual versions of LIST:

```
L
```

lists entire text from beginning to end.

```
Lxxxx
```

lists line number xxxx.

```
L-xxxx
```

lists text from beginning up to line xxxx.

```
        Lxxxx-
```

lists lines xxxx to the end of the file.

```
        Lxxxx-yyyy
```

lists from xxxx to yyyy. If yyyy is less than xxxx, then no lines will be listed.

The listing can be slowed with the CTRL key, or stopped and started by the spacebar. Press the RUN/STOP key to abort the listing altogether.

3.1.8 M (= MAP/DRIVE)

For details on the 'M' command, see Chapter 2.1.7 ('M' in Main Menu).

Keep in mind that disk drive 0 will be the "main drive", i.e., that the system will look there for LOADDAT and the respective system programs. The 'M' command will let you change drive numbers for 'A:', 'G:', 'P:', and 'U:'.

3.1.9 N (= AUTO-NUMBERING)

This command automatically generates line numbers in steps of 5, allowing you to add text. There are two methods of starting auto-number mode:

```
        N
```

which begins with a number 5 higher than the last number of text. If no previous text exists, then 'N' will start at line 1000.

```
        Nxxxx
```

begins at line xxxx (determined by the user) and goes in five-step increments from there.

Auto mode will switch off if:

you move to a different line for editing, and press <RETURN>.
you enter a <RETURN> after a line number.

During auto mode, no other editing commands can be accessed; in order to return to editing, use one of the above methods.

3.1.10 O (= OUTPUT DEVICE)

This lets you select the output device to be used. When the editor starts, the output device is obviously the screen, but using

```
Ox,y
```

will let you redefine this device number; x represents the primary address, i.e., the device number proper, and y the secondary address. If no number is given for y, the default value will be 0. Here are three ways to reset the output to the screen:

```
O0,0        (or)        O0            (or)          O
```

Input of an illegal device number (other than 0 or 4-7) or secondary address (other than 0-15) will result in

```
ILLEG. INPUT!
EXECUTION NOT SUCCESSFUL!
```

being displayed.

After redefining the output channel, the entire output -- which would normally appear onscreen -- will go to the specified device; this feature can be very useful for the 'F:', 'L' and 'M' commands. On the other hand, your best bet for a hardcopy of the text would be the 'W' command in the Main Menu, since that command gives you a neatly formatted printout of a file.

Output mode can be halted with the RUN/STOP key, or an error will change the readout back to the screen.

3.1.11 P (= PUT SOURCE TO DISK)

This takes a text file from the editor, and saves it to disk. This text file can later be compiled or assembled:

```
P:FILENAME
```

FILENAME is, of course, the name under which you want the file saved to disk. You may first want to check the directory or change drives ('M' command).

Rather than use a filename, you could use the identifier "*", in connection with the 'G:' command which also allows for predefined filenames. If such a filename hasn't been defined, or you have accidentally used 'A:' or 'U:', the system will display

```
TITLE UNDEFINED!
EXECUTION NOT SUCCESSFUL!
```

The following prompt is displayed to insure against any other bad input:

```
CONFIRM "FILENAME,DRIVE_NR"? N/Y
```

Confirmation ("Y") begins the save procedure.

Any errors occurring during the save sequence (e.g., bad syntax, illegal identifier) will display a corresponding error message, and bring the command to a halt.

The syntax rules for Pascal identifiers must be followed (as we've mentioned before at Chapter 2.1.8). Be sure to reread those rules, as it will make your file storage easier.

NOTE:

It is very important that you give a textfile a different name from the compiled "program" version, when saving to disk -- Pascal syntax suggests a PROGRAM header (PROGRAM PROGRAM NAME; ...), to avoid any overwriting problems.

Here are the important identifiers:

S_NAME	name of a Pascal source-file (source)
A_NAME	name of an assembler source-file (asmblr)
C_NAME	name for Pascal program code (code)
M_NAME	name for 6510 object code (M-prgm)

The remaining files (data) have no specific identifiers.

If you give the file to be saved a name identical to a file already on disk, the old file will be scratched, and replaced by the new. However, if the file on disk is locked (protected), the save process will abort, and this message will be displayed:

```
ILL.FILE OPR.
```

Corresponding error messages will come up for any I/O errors. Once the 'P:' section is done, the old text remains in the editor for your work.

3.1.12 Q (= QUIT)

This command leaves the editor and returns you to the Main Menu. If there is text in the editor when the 'Q' command is given, the system will ask whether you want to save the file or not:

```
SURE NOT SAVING THE SOURCE? Y/N
```

Choosing 'Y' (yes) erases the file and returns you to the Main Menu.

3.1.13 R (=RENUMBER)

The 'R' command comes in handy for renumbering programs (for, say, renumbering a program after editing). The numbering begins at 1000, and increases in 5-step increments.

If the need arises for more than four lines' worth of space for additional text, just insert a new line number, a (* comment *), <RETURN>, and run the 'R' command; this will give you 9 lines to work with.

3.1.14 S (=SHIFT LINE)

Often the user will find it necessary to move an entire set of program text to another place (e.g., when appending files); to accomplish this, we've included the 'S' command. The syntax must be typed in as follows:

```
Sxxxx-yyyy:zzzz
```

The command moves lines xxxx through yyyy to the place defined by zzzz. If the value for line yyyy is less then that of line xxxx, the command will be ignored. On the other hand, if zzzz is defined within the ranges xxxx to yyyy, the system will say

```
ILLEG. INPUT!
EXECUTION NOT SUCCESSFUL!
```

After moving text, it should be renumbered (see 'R' command). Now you can work with the newly-moved text.

3.1.15 U: (=UPDATE FILE)

To some extent, this is a companion to the 'A:' command -- it allows you to append text to files already on disk. The opening syntax sounds like this:

```
U:FILENAME
```

-- FILENAME representing the file on disk (you may first want to verify that filename with the 'M' command). The system will ask for verification:

```
CONFIRM "FILENAME,DRIVE_NR"? N/Y
```

A positive response ("Y") starts the save routine; any other character will cancel the command.

If the file is locked (protected), the 'U:' command displays

```
ILL.FILE OPR.
```

and halts the command; any I/O errors will also display messages and abort the command, although the text will remain in the editor.

NOTE:

The 'U:' command can, with repeated use, produce extremely long text files -- longer, in fact, than the 'G:' command will be able to handle. Keep this in mind, and watch file size carefully.

3.1.16 V (=VACANCY)

This command returns the amount of memory free in the editor at any time (the empty editor has 43000 bytes free). Any time that memory runs out will cause the following to be displayed. (see 3.5 for a solution).

```
RAM OVERFLOW!
EXECUTION NOT SUCCESSFUL
```

3.2 EDITING PASCAL PROGRAMS

This chapter will briefly cover writing Pascal programs in edit mode.

Source programs are input using the syntax described in the "Pascal User Manual and Report". The reserved words (keywords and word symbols) and identifiers use the ASCII characters from $41 to $5A (upper-case). These will be printed differently on the C-64, as we mentioned at the beginning of this manual.

In the default mode of SUPER Pascal (lower/upper-case mode), these characters appear as lower-case, rather than upper- case (to avoid confusion, you can go back to upper- case/graphics mode by pressing C=/SHIFT). Identifiers are distinguished by the "_" character (ASCII $5F, or "back-arrow").

The "{" and "}" characters, unavailable on the C-64, are replaced in SUPER Pascal by "(*" and *)". Any other characters, strings or CHAR-types are those used on the 64.

There are a few restrictions imposed by the 64 in developing Pascal programs, but these are so trivial, that they probably won't make that much difference in your programming:

> Textlines (including line number) cannot exceed 80 characters (solution -- divide text into smaller sections);
>
> First character of text (immediately following the line number) cannot be a number (solution -- start text with a space);
>
> No blank text lines (answer -- input as a blank comment (* *));

Aside from that, the 'N' command (see Chapter 3.1.9) helps in making room for plenty of program development, where there seems to be no room.

Thanks to the large amount of text memory (43k), you can edit and write Pascal programs that are downright huge. Large programs can be divided into smaller sections, stored on disk in this form, and edited piece-by-piece. A simple command at the end of each file tells the compiler that this is only part of a program. We have managed to develop and effectively compile a Pascal program of six separate sections of 40k each, and as far as we know, the ability to divide programs is limitless, and can be used at your own discretion (do keep in mind, though, that these sections must be absolutely correct, syntactically speaking, before compiling).

The compiler command for continuing with another program section should read:

```
    &CONTINUE(FILENAME, DRIVE_NR);
```
(or)
```
    &C(FILENAME,DRIVE_);
```

This tells the compiler to get FILENAME for compiling, once it's through with the present file, compile FILENAME from disk.

A simpler method:

```
    &CONTINUE(FILENAME);
```
(or)
```
    &C(FILENAME);
```

can be used if the additional source sections can be found on the same diskette as the first program section.

Next in the intermediate commands for the compiler is a second routine for larger programs, which basically includes one much-used routine at a certain spot in the program, rather than type in that routine time and again:

```
    &INCLUDE(FILENAME,DRIVE_NR);
```
(or)
```
    &I(FILENAME,DRIVE_NR);
```

This command interrupts the compiling of the present file, and pulls the specified file (FILENAME) from disk, compiles that, and continues with the old file. The file called by '&I' is now an integral part of the original program. Needless to say, the text called by '&I' should be debugged and ready to go before compiling, to avoid errors.

The short versions of this command are:

```
    &INCLUDE(FILENAME);
```
(and)
```
    &I(FILENAME);
```

which, as above, will work if the file to be INCLUDEd is in the same drive as the original file.

A program section can contain a number of '&C' and '&I' commands. A program can work with up to 4 nested '&I' commands.

The `INCLUDE` command allows similar `CONST`-, `VAR`-, `PROCEDURE`-, `FUNCTION`- or statement definitions to be used in different programs. Also, individual routines can be used again and again, e.g., you could use the demo program "Hilbert-Curve" in one of your high-resolution programs.

There's a third aid in designing Pascal source-code; the command for conditional compilation. That is, versions of a program which differ from one another in a few respects can be attended to as one program. The conditional option reads:

```
      &TRUTH(BOOLEAN_CONST);
(or)
      &T(BOOLEAN_CONST);
```

This command tells the compiler to translate this section of the program text only if the Boolean constant is TRUE. If the constant is FALSE, the compiler ignores the text and continues searching until the TRUE conditional command is found. The compiler will compile the TRUE version only.

It should be self-evident that control can be turned on with '`&T(TRUE)`' and off again with '`&T(FALSE)`'.

The remaining compiler commands ('&P' and '&A') don't deal so much with the source text as they do with information about the compiling process. We will cover these in detail in the chapter on the compiler (Chapter 4).

3.3 EDITING ASSEMBLER PROGRAMS

This chapter will cover only the text editing assembler programs; particulars of the assembler can be found in Chapter 5.

The assembler source follow tightly-assigned rules of syntax, in order for the text to be properly converted to 6510 machine language. At the same time, the source code must also be readable.

These two items are often the reasons for an assembler source-code to have specific columns drawn within a text line. They aren't normal procedure; those columns are for the user's convenience. Though they aren't necessary to compiling, four-digit line numbers (1000-9999) are also included for user-readability. These are the same numbers generated by 'R' (see 3.1); starting number 1000, steps of 5, up to 9999, 43k of memory.

On to those column divisions: The assembler has certain ranges for specific material within a line -- a label field, an instruction- or operator-field, and an operand- or address-field. The room left on a line can be used for comments if desired.

Here's a sample text line, with its individual features defined:

--> text line : `ZZZZ LLLLLLLL III OOOOOOOO...`

POSITION 1-4 (`ZZZZ`=line number)

Field for 4-digit line number.

POSITION 5 (space)

Blank space, separating line number from label field.

POSITION 6-13 (`LLLLLLLL`=label field)

This is where labels are placed for recognition by the assembler program. The labels are linked together into an array. All identifiers are allowed here as labels. The structural rules for labels are:

8 significant characters (no more can be used per field), whereby

the first character must be a letter, and

the remainder can be letters, numbers or "_".

Unused positions on a label field will be made up of spaces. If no label exists in a line of text, then the entire field will be blank (spaces).

POSITION 14 (space)

A space separating the label field from the instruction field.

POSITION 15-17 (`III`=instruction or operator field)

This field is where the 6510 mnemonic instructions proper are put. The abbreviations here are identical to Commodores 6510 definitions. Here is a list of these instructions:

ASL	BPL	CLV	EOR	LDX	PLA	SEC	TAY
BCC	BRK	CMP	INC	LDY	PLP	SED	TSX
BCS	BVC	CPX	INX	LSR	ROL	SEI	TXA
BEQ	BVS	CPY	INY	NOP	ROR	STA	TXS
BIT	CLC	DEC	JMP	ORA	RTI	STX	TYA
BMI	CLD	DEX	JSR	PHA	RTS	STY	
BNE	CLI	DEY	LDA	PLP	SBC	TAX	

Along with these instructions, there are several pseudo- commands, which behave like assembler commands:

.BA	.DL	.EQ	.OS
.BY	.DS	.NE	...
.CT	.EN	.OC	

These pseudo-operations begin with a '.' at position 15. Chapter 5 contains detailed descriptions of these commands, but we shall touch on these pseudo-operands here.

POSITION 18 (space)

A blank space, separating the instruction field from the operand field to follow.

POSITION 19 ff. (OOOOOOOO... = operand field)

Column 19 is the starting place of the operand field, the length of which varies. This field gives the operands for the operation, i.e., the parameters for the machine language commands. Here is where the symbolic or absolute addresses appear, containing an address to which the routine should jump, or a particular item on which the command should operate.

The length of the operand field depends on the type of addressing used in the command, the length of the label used, and the address range. The maximum length of an operand is limited to linelength (80 characters, including line number).

If you should wish to add any commentary (similar to REMS in BASIC) that the system will ignore, you can do so after the operand field by adding a space and a semicolon (;). You can also insert comments after placing a semicolon at position 6 or 15.

We have some other commands in the assembler which are quite different from the norm (.CT,.NE, etc. -- see earlier in this chapter); these affect the design on source-code and assembler processes.

The first of these is a string command:

```
    .CT FILENAME
```

This command (mnemonic for "ConTinue") instructs the assembler to translate the source file on diskette listed in the operand field as FILENAME. .CT allows you to work with assembler programs much larger than the editor can handle; you can edit them in smaller sections, then connect them into one unit with .CT.

Next we'll discuss the conditional commands

```
    .EQ OPERAND    and        .NE OPERAND
```

in detail. These commands, similar to the conditional compiler instructions (see 3.2), give the assembler the ability to choose between different versions of a source program, and choose the one conditionally proper for editing and saving. The mnemonic for .EQ means "If operand equals 0"; i.e., the command converts the program that follows ONLY if the given operand is equal to zero. In other words, the program would be acceptable if the constant label at the beginning of the program equals 0.

The pseudo-instruction .NE ("if operand is not equal to 0") works much the same way, except the operand would have to be anything but 0.

A conditional program section must be ended by the pseudocode

```
    ...
```

which sets the assembler back to normal.

The following pages contain a sample program showing all these features. It demonstrates carriage return output for the C-64, PET and ABC:

```
.
.
.
1100 C64       .DL 1       ;set label C64 to 1
1105 PET       .DL 0       ;set label PET to 0
1110 ABC       .DL 0       ;set label ABC to 0
.
.
1200 CR        .DL $0D     ;set label CR to 13
.
.
1300           .NE C64     ;cond. assemb. if C64<>0
.
.
1350 BSOUT     .DL $F1CA   ;label BSOUT = address
.
.
1390           ...         ;end of cond. assemb.-C64
1395 ;
1400           .NE PET     ;cond. assemb. if PET <>0
1405 ;                       ignored if PET = 0!
.
.
1450 BSOUT     .DL $FFD2   ;BSOUT = address
.
.
1490           ...         ;end cond. assem. PET
1495 ;
1500           .NE ABC     ;cond. assem. if ABC <>0
1505 ;                        ignored here if ABC=0!
.
.
1550 BSOUT     .DL $FF00   ;set label BSOUT
```

```
.
.
1590          ...          ;end cond. assemb. ABC
.
.
2000 OUTCR    ;             CR-output program,
2005          LDA #CR      ;load CR-code and perform
2010          JSR BSOUT    ;output routine,
2015 ;
2020          .NE ABC      ;cond. assemb. if ABC <>0
2025 ;              give LF code (linefeed) as well
2030          LDA #$0A    ;as CR (cancel by C64/PET)
2035          JSR BSOUT    ;ignored here if ABC
2040 ;                      is equal to 0
2040          ...          ;end ABC cond. assembly
.
.
.    ;program for other versions
.
3000          .CT DEMO_2  ;assembly command, which
3005                    ;converts source file DEMO_2
3010 ;                     if found on diskette.
3015 ;
.
.
```

3.4 MIXED PROGRAMS

A major strength of the SUPER Pascal compiler is its ability to handle mixed programs -- in other words, it isn't limited to Pascal; it can also deal with machine-language routines within a Pascal program. Assembler source-code is called into a Pascal routine with PROCEDURE or FUNCTION; it's a simple matter to treat assembly routines as "normal" Pascal functions or procedures. For more information on this matter, see Chapter 4.1.2.3. For the moment, however, we'll look at a few relevant aspects.

As we said above, machine language routines can be inserted in a Pascal program with PROCEDURE or FUNCTION. In addition to standard procedures of machine-code syntax (see 3.3 and 5), certain parameters must be fitted to the routine to make it a Pascal-compatible procedure or function. The assembler section can be cordoned off with BEGIN and END, while the compiler command

```
ASSEMBLE;
```

calls the program section.

This command should be developed somewhat to define the exact name and location of the file (if the disk has more than one assembler program):

```
ASSEMBLE(FILENAME,DRIVE_NR);
```

The system looks for the FILENAME on the drive selected. If the file is on the "working disk", all you need type in is

```
ASSEMBLE(FILENAME);
```

Pascal-level constants can be defined within the assembler routine. Furthermore, the contents of the Pascal variable stack can be read within such a m/l routine with STKPOI.

The machine-code routine is concluded with the instruction RTS, which sends us back to the Pascal routine currently called. The chapter concerning the assembler (5) suggests that the pseudo-command .EN should be used after RTS. NOTE: the next line of the source program should be in Pascal. The following routine will give you an example of what has been discussed: Essentially, this program is a Pascal routine containing a machine-language subroutine to change the 64's screen colors.

```
1500  PROCEDURE SCREENCOLOR (COLOR:BYTE);
1505  {}
1510  {Pascal routine with a call to change screen)
1515    {colors (parameters in single-byte form)}
1520  {}
1525
1530  ASSEMBLE;
1535 ;
1540 CPUPORT  .DL $01
1545 SCREE_RG .DL $D020
1550 ;
1555          SEI             ;broaden IRQ for
1560          LDA *CPUPORT    ;different memory
1565          ORA #3          ;configuration
1570          STA *CPUPORT    ;(I/O enable)
1575          LDY #0          ;set index register
1580          LDA (STKPOI),Y  ;get parameters
1585          STA SCREE_RG    ;set screen color
1590          LDA *CPUPORT    ;reset memory
1595          AND #$FC        ;config.I/O disable
1600          STA *CPUPORT    ;
1605          CLI             ;IRQ permit
1610          INC *STKPOI     ;set-raise Pascal
1615          BNE EXIT        ;variable stack and
1620          INC *STKPOI+1   ;thus parameters
1625 EXIT     RTS             ;return to Pascal
1630 ;
1635          .EN             ;end Assem routine
1640 {}
1645 {now back to the Pascal routine}
1650 {}

2000 SCREENCOLOR(#0);
2005 {call to switch on screen color BLACK}
```

3.5 INTERNAL ORGANIZATION OF THE EDITOR

We'll spend this chapter looking at the design of the editor, and give a few explanations on how to write textfiles in this mode. The chapter will conclude with memory management during edit mode.

With the exception of one critical section in machine language, the editor itself is a Pascal program (see the complete listing in the Appendix of this manual). When editing, the program organization and variables carry greater weight than individual textlines; here we can see a connection between necessary memory usage and the program's ability to find lines quickly. The best type of program in Pascal is a solidly-written one.

Program design involves being able to edit, delete, move and augment lines without any problems to the system. The contents of a given line is defined in terms of STRING-length, which is, of course, limited to line-length (80 chars.). Aside from that, the Pascal-heap (which contains the dynamic variables) can also be altered. However, removing text strings which are no longer needed can pose some problems with pointer manipulation, thus slowing the program down. In addition, leaving these text strings in can take up a good deal of memory.

There is, however, a way out of our memory problems: Rather than keep the text in memory, the editor will allow us to move memory around, and make adjustments for text storage on disk to be called up later; as this space fills up from loading (and as the strings become unnecessary), the garbage collection is speeded up, and the heap pointers don't get pushed about by the collection. Another good reason for using temporary storage is that it increases the amount of memory available for writing programs. The compiler commands `&CONTINUE`, `&INCLUDE`, `ASSEMBLE` and the assembler command `.CT` are used in this respect.

3.6 TEXTFILE DESIGN

We'll conclude Chapter 3 with some remarks about the structure of textfiles. They contain the information which will eventually become programs (sourcefiles) in SUPER Pascal.

SUPER Pascal textfiles are designated with the filetype declaration TEXT per standard Pascal (i.e., the `FILE OF CHAR` according to the "Pascal User Manual and Report"), with the filename and type followed by a carriage return (ASCII $0D). As mentioned before, source programs are saved without line numbers or other characteristics of textfiles. An end-of- file (EOF) character will not be produced, and is not an absolute requirement of the files. An `EOF` can be set in the information section of the directory.

If have a source program that's unfamiliar to you, and you'd like to have a look at it (for editing, curiosity, whatever), it's possible to write a file-conversion program, to turn the file from SUPER Pascal back to a textfile. We suggest that you have a good look at the "Pascal User Manual and Report" for proper Pascal syntax and programming methods.

4.0 PASCAL COMPILER

As mentioned previously, this manual is an instruction book for SUPER Pascal, nothing more -- it is not a Pascal tutor, nor is it intended to be. We suggest that you check the chapter at the end of this book ("For Further Reading") for a list of beginning Pascal texts.

The basic concepts and definitions of Pascal were essentially invented by Niklaus Wirth, a professor at the Technische Hochschule (technical college) of Zurich, Switzerland). Professor Wirth literally "wrote the book" on the subject -- the Pascal User Manual and Report, which is the acknowledged text on the language. The language name, Pascal, is in honor of the French mathematician Blaise Pascal(1623-1662), who built a working mechanical calculator in the 17th century.

4.1 SCOPE OF THE LANGUAGE

The SUPER Pascal compiler contains the complete set of commands and "figures of speech" used in standard Pascal, plus a few commands that we included to make C-64 programming easier. The chapters that follow give definitions of these commands.

4.1.1 STANDARD LANGUAGE ELEMENTS

All Pascal programs are written in a block-oriented manner, with program blocks divided into two sections -- an arrangement section and a command section. The arrangement section contains the definitions and declarations necessary to the problems being handled, while the command section contains the statements used for comparison, problem solving, etc. Subprograms (procedures and functions) contained within such a main program are written with the same block orientation. This top-down concept is an important reason for the structural elegance of Pascal programs. There's another, more functional reason for top-down programming: That structure dictates the sequence in which the different language elements should appear. Now, on to the basics: The standard symbols and identifiers are listed below followed by their definitions.

Every Pascal program begins with the program header, which gives filename and individual parameters of a file. A header looks like this:

```
PROGRAM PROGRAM NAME (FILEPARAMETERLIST);
```

PROGRAM is the reserved word for the program header.

PROGRAM NAME represents the program identifier by which the program is known; the SUPER Pascal compiler will save the program code to diskette under this name.

NOTE:
The source program MUST be saved under a different name than the name on the program header; otherwise, the sourcefile will be overwritten by program code with the same name.

FILEPARAMETERLIST refers to the other identifiers within a file (placed within parentheses, with each parameter separated by commas), which are used in the program. Two examples of this are INPUT and OUTPUT. These last two, needed for accessing external files, are relics from the early days of the mainframe computer -- used to call not only program code, but the devices involved in retrieving data. On a home computer such as the 64, this sort of thing is unnecessary for file retrieval, since we have immediate access to disk. So, INPUT and OUTPUT for us stands for the keyboard and screen respectively. You can use the parameter list -- but you don't absolutely HAVE to; the program header can be abbreviated:

```
PROGRAM PROGRAM NAME;
```

which makes life much simpler.

The next must be included in the course of the arrangement section -- the label declaration:

```
LABEL LABELLIST;
```

LABEL is the reserved word for label declaration.

LABELLIST stands for the label number(s) (separated by comma(s)). The label number should be used in connection with the GOTO command; the number can be any whole number between 0 and 32767 (maxint).

The section following the label declaration is the constant arrangement:

```
     CONST CONSTANTLIST;
```

CONST is the reserved word for constant arrangement.

CONSTANTLIST refers to a set of constant definitions (separated by semicolons (;)) required in the program -- these definitions consist of the definition, followed by an equal sign (=), followed by the value of the definition. It is permissible in SUPER Pascal to arrange defined values for simple calculations, or for comparisons between constants. Here are some possibilities, encompassed in one example:

```
CONST VALUE       = 3;            NUMBER = 0;
      ENDVALUE =VALUE * $FF +5;RETURN= CHR($0D);
      ASCII       = ORD(X) ;      LESS   = PRED(MAX);
```

The full scope of these comparisons are described in 4.1.2.1 under language extensions.

The constants

FALSE (ordinal value 0),

TRUE (ordinal value 1),

and
MAXINT (ordinal value 32767)

are predetermined values.

There are two more constants, which aren't used in constant declarations per se, but are handled within a program as if they are constants:

NIL (as pointer constant for zero)

and

`[]` as constant for a blank number.

The declarations end with the type declaration:

```
     TYPE TYPELIST;
```

TYPE is the reserved word for the type arrangement.

TYPELIST stands for the sequence of type definitions (each separated by semicolons (;)). The basic type definitions are set out thoroughly in the Pascal User Manual and Report. It is possible to set subtypes in conjunction with constants:

```
 TYPE INDEX      =  0..PRED(MAXVALUE);
    COMMAND    =  ARRAY [CHR(0)..PRED(' ')] OF CHAR;
    ARRAY      =  ARRAY [1..10*10] OF INTEGER;
    DAY        =  (SU,MO,TU,WE,TH,FR,SA);
    WORKDAYS   =  SUCC(SU)..PRED(SA);
```

User-defined scalar types (such as above) can go as high as 256 values.

Predetermined types:

```
BOOLEAN    = (FALSE, TRUE);

CHAR       = (CHR(0)..CHR($FF));

INTEGER    = -MAXINT..MAXINT;
```

and

REAL

In addition to these, the following list contains reserved words usable for structure commands AND variable types:

ARRAY

Arrays can be defined without limit as

```
    ARRAY [DIM1,DIM2...] OF ELEMENT;

    ARRAY [DIM1][DIM2]... OF ELEMENT;
```

or

```
    ARRAY [DIM1] OF ARRAY [DIM2] ... OF ELEMENT;
```

-- take your choice.

RECORD

Fixed or variable records can be defined; variable records can be set up within a CASE-list; the component variables (tagfield) will only be allowed with a specific array definition in a memory location; the component variable can be used with just a type name, so the array isn't an important part of the record.

SET

Quantities of all scalar types (`REAL`) are permitted; a `SET` can contain a maximum of 256; the range must not go beyond 0..255.

FILE

`FILE` is not an element of a structured type itself, i.e., it doesn't fit in as an element of an array, record, pointer or file. A `FILE` can be no more than 512 bytes (size of the file buffer).

^ (POINTER)

The definition of pointer types is used in connection with the design of list and branch structures (progression and recursion):

```
     BRANCH = MARK;
     JOINT  = RECORD
               ENTRY:ARRAY [0..7] OF CHAR;
               LBRANCH,
               RBRANCH :BRANCH
              END;
```

These structured types are predetermined according to the Pascal User Manual and Report:

ALFA = ARRAY [0..7] OF CHAR;

and

TEXT = FILE OF CHAR;.

Next section deals with variable declarations:

```
     VAR VARIABLELIST;
```

VAR is the reserved word for variable declaration.

VARIABLELIST states any set of variable groups, each group separated by a semicolon (;). A variable group consists of a series of variable identifiers (separated by commas (,)and ended by a colon (:)),for example:

```
 VAR  FLAG,SWITCH           :BOOLEAN;
      CH                    :CHAR;
      VALUE,NUMBER,SUMQ     :INTEGER;
      TITLE,FILENAME        :ALFA;
      ARRAY                 :ARRAY [0..9] OF INTEGER;
      HEAP                  :^INTEGER;
```

The variables

INPUT

and

OUTPUT

are predefined (under type TEXT).

In connection with these variable types, here are the memory requirements for each type:

BOOLEAN, CHAR and user-specific variables	- 1 byte
INTEGER- and pointer-variables	- 2 bytes
REAL-variables	- 6 bytes
SET-variables	- 32 bytes

The variable arrangement set ends with the declaration of procedure and function headers:

```
    PROCEDURE PROCEDURENAME (PARAMETERLIST);
```

and

```
    FUNCTION FUNCTIONNAME (PARAMETERLIST):TYPENAME;
```

PROCEDURE is the reserved word for procedure assignment.

FUNCTION assigns function (reserved word).

TYPENAME defines the function return value, i.e. the type of function called. SUPER Pascal allows all types except FILE for this value.

PARAMETERLIST defines the parameters for the given function/procedure. Syntax follows the "Pascal User Manual and Report". SUPER Pascal permits all sorts of parameters:

```
    parameter transfer by number,

        "         "    by name

    and

    procedure and function transfer by name.
```

Two examples:

```
FUNCTION FILEHANDLING(ELMNT:INTEGER;VAR SEQU:TEXT;
                       PROCEDURE ERROR):BOOLEAN;

PROCEDURE TEST(VAL1,VAL2:INTEGER;VAR MESG:ALFA;
               FUNCTION CHECK:BOOLEAN);
```

These type definitions must be determined and set up before this routine, either as computer default (predefined by system) or by the user.

The assignment set is done: Now we go on to the command set. This section contains the program activity proper; the entire command section is defined (or bordered, if you prefer) by the reserved words

BEGIN

and

END

which give the start and end of the program. These two words are set off as individual statements (nothing else on that line).

The individual statements from standard Pascal are given here, but you'll have to check a Pascal instruction book for proper syntax and use; any special differences between standard and SUPER Pascal will be laid out below:

:= (assignment)

Assignment operator goes to the left -- the expression (variable) goes to the right of the equal sign.

IF ... THEN ... ELSE ...

Checks conditions, and branches to whichever next statement applies -- much more practical than Commodore BASIC's IF/THEN construct.

CASE ... OF ... END

Sets up a multiple-choice of sorts, going to `END` if all else fails. SUPER Pascal lets you make an infinite number of `CASE`s. If none of the choices work out, the program will immediately go to the next statement.

WHILE ... DO ...

No difference between standard and SUPER Pascal.

REPEAT ... UNTIL ...

No difference between standard and SUPER Pascal.

FOR ... TO/DOWNTO ... DO

Loop number can be defined as any scalar variable (aside from `REAL` variables).

WITH ... DO ... (record access)

This statement simplifies access to the array variables of a record. This can save lots of memory in a source program by calling up a number of array lists during runtime.

GOTO ...

This statement makes the program jump to the label specified (similar to BASIC's GOTO). Pascal very seldom uses `GOTO`, mainly because earlier generations of the language never even HAD such a command; but there are a few exceptional situations that might be made simpler by using this command. For example, when errors crop up, it was possible to tell the compiler to take an "easy out" using GOTO, to cease program flow and return control to you. Most of the time, though, other Pascal commands (e.g., IF/THEN/ELSE) had to suffice.

SUPER Pascal has a fully implemented GOTO statement; that is, no limitations. You can use the statement to call up a procedure, function, or even another program without compunction. One warning: If GOing TO a loop or set of loops, be sure that the loop (set) is properly structured -- or the system might get caught in an endless loop, which tends not to return program control to you.

Standard procedures which are predefined in standard and SUPER Pascal consist of:

DISPOSE

This procedure is used in standard Pascal for freeing up parts of memory by changing the dynamic (pointer-) variables on the heap (memory heap for dynamic variables). The area can next be cleared out with NEW (see corresponding section).

The memory reserved for dynamic variables is quite different in a small computer (such as a C-64) as opposed to a mainframe. Mainframes have massive amounts of memory, and seldom need to do very much adjusting. SUPER Pascal, like most other Pascal versions for home equipment, has a DISPOSE command. There are, of course, other commands used to change the heap pointers (MARK and RELEASE).

GET

GET sets the read pointer (which is set for an opened file by RESET) for a file element.

Syntax: `GET(FILEVARIABLE);`

FILEVARIABLE stands for the identifier declared in the variable type FILE. GET allows you to set the access pointer for the next character to be read (see remarks under READ/READLN).

NEW

NEW allocates and reserves a section of memory on the heap using a pointer variable under Pascal control, i.e., the pointer containing the value for the heap pointer is set to the next free space on the heap. The setting of the heap pointer depends upon the size of the pointer variable.

Syntax: `NEW (POINTERVARIABLE);`

POINTERVARIABLE stands for the identifier declared under POINTER (see arrangement section). This variable is accessed with POINTERVARIABLE^.

NOTE:
Treat this command with care:

It can be adjusted to point to any memory location -- careless adjustment could be fatal to your variables, and even the program. Be sure you know where you're aiming it....

A pointer set to NIL obviously sets that value to 0, i.e., memory location $0000.

PACK

This procedure packs spread-out structured variables together (saves memory) in standard Pascal. SUPER Pascal already does this internally, so this command isn't available.

PUT

Sets write pointer in conjunction with REWRITE (analogous to GET).

Syntax: `PUT(FILEVARIABLE);`

FILEVARIABLE stands for the identifier declared in the arrangement section as FILE>.

READ

This procedure has a double purpose: First, it serves to allocate access to existing file elements under one target variable; second, it sets the read pointer to the next file element.

Syntax: `READ(FILEVARIABLE,TARGETVARIABLELIST);`

FILEVARIABLE stands for the identifier for accessing the existing file variables. When handling a file as INPUT, this variable we needn't be so explicitly described. The syntax for the simple version is

```
  READ(TARGETVARIABLELIST);
```

TARGETVARIABLELIST stands for a set of file elements (each separated by commas), for a set of variables -- these represent at least one identifier. Obviously, the target variables must be of the same type as the file elements. An exception to the rule applies to TEXT filetypes (thus, INPUT files as well). The target variables for these files can be declared as CHAR, INTEGER and REAL types. Target species INTEGER and REAL will automatically convert ASCII representations of strings in the file to the binary coding used within variables.

There is another point to consider about the above. When reading INTEGER and REAL values from a TEXT file, a numeric string must be ended with the proper syntax. This end character will not be read: Rather, the read-access pointer will point to this character after the read operation. This closing character must be recognized, or the next INTEGER or REAL read operation will not take place. When READ encounters a numeric variable, the access pointer will go to the start of the next number string. Leading spaces are ignored when reading a REAL or INTEGER variable.

One item to be observed is the line delimiter used in TEXT filetypes (the C/R, or carriage return: ASCII $0D, or 13 in decimal). When the file-access pointer comes upon this character, the pointer is set to the function return value of the function EOLN to TRUE. This return value is 0 on reading a numeric variable; the read pointer then goes to the start of the next line.

The variable INPUT has a peculiar relationship to the TEXT type itself. This INPUT has its own file buffer (not to be confused with the keyboard buffer, in which any key pressed is held temporarily, until the C-64 operating system gets around to working with it). Suppose we have an empty input buffer; if we call up file INPUT, the C-64 system routine GETLINE is called upon once this buffer is filled. GETLINE reads the keyboard and puts the given characters onscreen. Program control remains in this routine until the <RETURN> key is pressed (EOLN). The line of text is taken into the input buffer after <RETURN>. Now the input buffer will be provided with the first target variable from the READ call. The access pointer sets itself for however many characters are in the buffer, as is necessary for the provision of the target variables. A new READ command, or arrangement for another target variable immediately puts the next character into the input buffer. The procedure GET(INPUT) works in much the same way in resetting pointers by one character position.

The input buffer's involvement in this reading process continues until the access pointer reaches a carriage return, which is interpreted as a space (or as a 0). The next GETLINE and READ will cause the input buffer to refill, whereby the access pointer goes back to the beginning of the buffer.

One difference between READing a diskette file and the file INPUT is the response to illegal characters used for numerical variables: Such characters (or bad syntax) will give you:

```
IL.INPUT
```

Bad access to a diskette file will stop the program with a runtime error, while bad INPUT will let you re-enter the input without program stoppage.

READLN

READLN will require some re-reading of READ. Here we have an additional use for the access pointer. Once all target variables are fed in, READLN sets the pointer to the next carriage return; i.e., the next read access can be from a diskette file AND call INPUT and GETLINE. It's possible to call up READLN without a target variable list:

```
READLN(FILEVARIABLE);
```

or simpler still, for reading INPUT alone:

```
READLN;
```

This procedure will set any file access pointer to the next carriage return.

Because `READLN` will only run properly if a c/r is read, the command is best used with `TEXT` filetypes.

If a file access (regardless of being called with `GET`, `READ` or `READLN`) isn't preceded by a `RESET` for opening said file, the runtime-error

```
NOT OPEN ERROR!
```

will appear, and the program will cease.

This last doesn't apply to file INPUT, which doesn't open files per se; `RESET(INPUT)` has a special meaning, to be covered later.

RESET

`RESET` opens a file for reading purposes, i.e., the file access pointer will be set for the first element of this file.

Syntax: `RESET(FILEVARIABLE);`

FILEVARIABLE represents the identifier for a `FILE` variable declared in the assignment section of the program.

`RESET` utilized with `INPUT` deserves special mention: The standard procedure `RESET(INPUT)` resets the the read pointer (in the input buffer) back to the start-of-buffer, then allows the buffer to read any new file INPUT. This, unlike `READLN`, makes it possible to provide correct input, if the file was incorrect.

REWRITE

Analogous to `RESET`: `REWRITE` opens a file for writing, i.e.,the file access pointer is set to the start of the new sector of said file.

Syntax: `REWRITE(FILEVARIABLE);`

FILEVARIABLE identifies the FILE variable declared in the assignment section.

NOTE:

A file already on diskette (i.e., bearing the same name) will be deleted by REWRITE, except when the file is locked (protected), in which case the runtime-error message

```
      IL.FILE OPR. ERROR!
```

is displayed, and program run is stopped.

REWRITE(OUTPUT) isn't needed by our system, so it has been left out.

We should supply some background information on the file procedures RESET and REWRITE: When a diskette file is opened (regardless of purpose [read or write]), the file- access is limited to the file buffer. This file buffer is 1024 bytes in size (1k, if you prefer); RESET loads the first sector (512 bytes) of the opened file into the buffer.

Now, a relatively simple access mechanism in the file access pointer can pull type and variable declarations set up in the file elements. Once the access pointer goes beyond the first sector, the first sector moves to the "bottom" half of the buffer, and the new sector is loaded into the "upper" half. This adds up to fast and efficient file access. When the access pointer gets to the conclusion of the last file element (end-of-file, or EOF), the return value of the standard function EOF is set to TRUE. If, by some chance, the read access finishes reading the file without running into EOF, the runtime-message

```
      AFTER EOF ERROR!
```

is displayed, and the program stops.

The write access to a file is similar to reading a file. REWRITE sets the access pointer to the beginning of the extant file, reserving the file buffer. WRITEs and PUTs utilize this "half-and-half" buffer usage (see the previous paragraph).

SUPER Pascal has a special command that must be used to end the write process, and commit the remaining contents of the buffer to the diskette file. This command is `CLOSE`, in a slightly different form from standard Pascal, in that it closes the file opened with `RESET` or `REWRITE`, and clears the file buffer for access to another file.

Now, regarding our fast file access system from a few paragraphs back: SUPER Pascal has three such file buffers (each 1k) set up. This means that you can only access three files at one time, regardless of whether they are being accessed for reading or writing. This may not seem like much, but we've had no problems in terms of practical usage with SUPER Pascal. In fact, you'll find that three 1k buffers will be quite enough for handling the most complicated file operations (reading sourcefiles, generating codefiles, generating revised data, accessing sourcefiles temporarily for 2-pass assembly, accessing variable data for a post-mortem-dump, etc.).

If you happen to try opening a fourth file buffer with `RESET` or `REWRITE`, you'll get a

```
    BUF.OV. ERROR! (buffer overflow)
```

and a stopped program.

One interesting feature of the file buffer system is the ability, when a `RESET` and `GET` are called, to load any sector of a program-code file, and join it with any callable external procedure or function. SUPER Pascal gives you a number of direct commands to call up such routines ('M', 'G', 'P', 'W'); these commands are contained in LOADDAT.

One thing not clearly discussed in the standard Pascal literature should be mentioned here: Should a file be opened for reading with `RESET`, you can switch from read access to write access at any time (i.e., regardless of whether the pointer is at the beginning, middle, or end of the file). This is simply a matter of using `PUT` instead of `GET`, and `WRITE` instead of `READ`. So, it's an easy matter to add new data to any spot on the file ("UPDATE"). The new file length, and the `EOF` marker are adjusted accordingly with this switch. Once you switch from `READ` to `WRITE`, though, switching back to `READ` isn't possible; otherwise, the new data will be lost.

If a file, opened with REWRITE or RESET, has not been closed properly (CLOSE), the program will halt, and

```
NOT CLO. ERROR!
```

will be displayed. As we mentioned before, this doesn't apply to file INPUT. Attempts to read or write to an unopened file (with RESET or REWRITE) will cause the system to state:

```
NOT OPEN ERROR!
```

This also doesn't apply to INPUT or OUTPUT.

WRITE

Like READ, WRITE has a double function: To arrange a new file element; and to set the write pointer to the next position available for that file element.

Syntax: `WRITE(FILEVARIABLE,SOURCELIST);`

FILEVARIABLE refers to the identifier for accessing extant file-variables. When file OUTPUT is being handled, this variable is unnecessary. The "short" syntax for this occasion is:

```
WRITE(SOURCELIST);
```

SOURCELIST stands for at least one expression (more than one would have each separated by commas) representing output of file elements. It's obvious that the source expression(s) must be of the same type as the file elements. The file type TEXT (OUTPUT also fits in this class) is an exception to the rule. Such source expressions as CHAR, BOOLEAN, INTEGER and REAL can be listed in these files. These last three will be automatically converted in the file from the binary system (used internally) to ASCII. Integer or real output, as with boolean expressions, will be set out in a specific format. Boolean sets will be printed without leading or ending spaces.

OUTPUT is particularly helpful in making formatted screen or printed output, when used with the write procedure. This array format is separated from the source expression by a colon (:), thus:

```
WRITE(FILEVARIABLE,SOURCEEXPRESSION:ARRAYFORMAT);
```

The file variable given must be a `TEXT` type. The source expressions will now be put out in `ARRAYFORMAT`, right-justified. `ARRAYFORMAT` must be an `INTEGER` type. The maximum allowable value for an `ARRAYFORMAT` hinges on the usual 132 characters per printer-line; the output buffer is also limited to 132 characters. If this number is surpassed (or if the value is less than 0), the runtime error

```
    IL.QUANT. ERROR! (illegal quantity)
```

is displayed, and the procedure stops.
There is a further adjustment that can be made for the formatted output of real numbers:

```
WRITE(FILEVARIABLE,REALEXPRESSION:ARRAYFORMAT:PLACE
FORMAT);
```

As above, the file-variable must be `TEXT`. `ARRAYFORMAT` is also covered above. `PLACEFORMAT` -- which must be `INTEGER` -- the numbers are formatted in fixed-point notation (i.e., decimal points are neatly lined up). There is a limitation; the numbers are limited to 11 decimal places, and the last place rounded off. This can be adjusted to anywhere between 0 and 31 decimal places. A larger value for `PLACEFORMAT` will give you an

```
    IL.QUANT. ERROR!
```

In contrast to the array format, the place format will take negative values. This means that the real number is given in floating-point notation, but a negative sign is attached. The range exists between -1 and -11; any steps out of range bring up

```
    IL.QUANT. ERROR!
```

See the next section (`WRITELN`) for output options.

WRITELN

WRITELN basically goes under the same rules as WRITE. However, WRITELN also seeks out the target file(s) and resets the write pointer at each C/R, then after printing, looks for the next line.

Syntax: `WRITELN(FILEVARIABLE);`

"Short" syntax (for use with OUTPUT):

```
WRITELN;
```

The command produces a formatted sourcelist. WRITELN is used only with TEXT files.

If attempts to access a file are made without a previous REWRITE or RESET, you'll see

```
NOT OPEN ERROR!
```

and the program will halt.

NOTE:

It is always possible to write to a file opened with RESET i.e., a file opened for reading). Internal control can switch the pointer to the end of the file, so that new data can be written in. If the file in question is locked, the data stays untouched, and the runtime-error

```
IL.FILE OPR. ERROR!
```

appears.

The functions defined by standard Pascal fall under three categories:

Type-conversion functions
Conditional functions
Mathematical functions

The first of these, the type-conversion functions, serves to convert a quantity of one type into a quantity of another type. This allows different types to be compatible to one another. To this set belong:

CHR

This converts any scalar argument (REAL numbers) to CHAR.

Syntax:

```
CHR(EXPRESSION)     (function type:CHAR)
```

EXPRESSION stands for any scalar quantity. For example, typing in

```
CHR(65)    or    CHR($41)
```

gives the letter A and

```
CHR(13)    or    CHR($0D)
```

gives a carriage return (C/R). Naturally, you are limited to the size of the character set (0-255, or $00-$FF). Any value above or below yields

```
IL.QUANT. ERROR!
```

ORD

This function performs the opposite of CHR; from an integer to any scalar argument (REAL).

Syntax:

```
ORD(EXPRESSION)     (function type:INTEGER)
```

EXPRESSION stands for any quantity. The function call gives the respective ordinal number; numbering begins with 0. CHAR-size determines the consequent ASCII-code of the ordinal value.

The second group of functions are those which read all conditions in the system (I/O register, for example), and act on the truth of those conditions (TRUE or FALSE). These functions are therefore BOOLEAN in nature.

EOF

Reads the file access (`GET`, `READ` or `READLN`) for end-of-file; case is TRUE if reached, FALSE if not.

Syntax:

```
EOF(FILEVARIABLE);   (function type:BOOLEAN)
```

`EOF` can also check on file `INPUT`:

```
EOF(INPUT)
```

or in short form (without the argument)

```
EOF
```

Since the C-64 views keyboard input in the same way as file input, the `EOF` flag can be set to TRUE by pressing the RUN/STOP key.

EOLN

This function can detect whether the read pointer finds a carriage return in a `TEXT` file; TRUE if so, FALSE if not.

Syntax:

```
EOLN(FILEVARIABLE)   (function type:BOOLEAN)
```

This function is also useful for `INPUT`.

ODD

Gives information regarding the remainder of an integer divided by 2. If the remainder is 1, the function is TRUE; otherwise, FALSE.

Syntax:

```
    ODD(EXPRESSION)      (function type:BOOLEAN)
```

The EXPRESSION used must be an INTEGER.

The third group of functions embrace the mathematical ("computing") functions. Two common functions:

PRED

and

SUCC

which are used to determine the Predecessor and Successor to the argument(s).

Syntax:

```
PRED(EXPRESSION)      (function type:EXPRESSION-TYPE)
SUCC(EXPRESSION)      (function type:EXPRESSION-TYPE)
```

EXPRESSION must be defined as a REAL number. The return value of the function will consistently be the same type as the argument (EXPRESSION). PRED will be less than the defined value, while SUCC will be greater than that value. This function should not be used with undefined values.

The remaining functions are arithmetical, and particularly useful for scientific programs:

ABS (determines absolute value)
SQR (squares value)

Syntax:

```
ABS(EXPRESSION)      (function type:EXPRESSION-TYPE)
SQR(EXPRESSION)      (function type:EXPRESSION-TYPE)
```

Both these functions will work with INTEGERs or REAL numbers.

ARCTAN	(reverse of TAN-function)
COS	(COSINE-function)
EXP	(exponent)
LN	(logarithm)
SIN	(SINE)
SQRT	(square root)
TRUNC	(whole numbers (left of decimal pt.))
ROUND	(round off to next whole number)

These functions all have same syntax and type:

```
funct(EXPRESSION)   (function type:REAL)
```

EXPRESSION is the expression used for the function; this argument can be either INTEGER or REAL. The function value, returned, however, will consistently be REAL, so SUPER Pascal has more call for the functions TRUNC and ROUND than you would in standard Pascal. These functions give you integers. In order to maximize the use of TRUNC and ROUND (i.e., to avoid limiting these functions to -MAXINT to +MAXINT) these two functions belong to the REAL types. Converting these numbers to integers is possible with INT:

```
INT(TRUNC(EXP(1)))        -- gives the integer 2

INT(ROUND(EXP(1)))        -- gives the integer 3
```

If illegal arguments are given for the functions LN and SQRT (negative numbers, 0 for LN), the program will stop and display the error message

```
IL.QUANT. ERROR!
```

Those are the standard functions in Pascal. The next few paragraphs discuss the combining operations which can go within the expressions. The reserved words according to standard Pascal are:

AND and **OR**

for logical boolean comparisons. The result is always a BOOLEAN expression.

NOT

for logical negation. The result is likewise BOOLEAN.

IN

to test for quantity relationship. Result: BOOLEAN.

DIV

for whole-number division of integers. Result will be INTEGER.

MOD

for determining the remainder of integers. Result: INTEGER.

+ and **-**

as leading characters, and for the addition and subtraction of integer and/or real numbers. An integer results from an equation made up of integers; otherwise, the result is real.

*****

is used to multiply integer and/or real numbers. As above, an all-integer equation yields an integer result; otherwise the result is a real number.

/

for dividing integer and/or real numbers. The quotient will always be a real number.

Pascal recognizes a number of comparative operations (see below). These comparisons must be used in conjunction with like types of numbers, i.e., all integers, all real, etc. The result of such expressions is BOOLEAN.

=	Test for equality
<>	Test for inequality
<	Test for "less than"
<=	Less than or equal to test amount
>	Greater than test quantity
>=	Greater than or equal to test quantity

Please check your Pascal User Manual and Report for standard usage of these elements. Now, on to one command that has nothing to do with the assignment set, or the command section: Rather, it deals greatly with compiler control:

FORWARD

This directive allows you to define blocks within a program which the compiler will treat as procedures or functions. Thus, these procedures/functions which have been predefind can be called repeatedly: This is useful for such things as recursion routines (see the Appendix for the HILBERT curve sample program).

4.1.2 LANGUAGE EXTENSIONS

The language extensions in SUPER Pascal were required for two reasons:

First it's a difficult task to put a Pascal implementation into a computer the likes of the C-64; its memory capacity, the fact that it is an 8-bit machine (8 bits=1 byte), and its input/output functions require some changes from any mainframe version of Pascal.

Second, a complete language and programming system had to be set up within the 64 which would bypass the standard operating system, and cut down the time factor.

4.1.2.1 ADDITIONAL ASSIGNMENTS, PROCEDURES AND FUNCTIONS

As with the normal assignment section, the block-design sequence applies here. The assignment set begins with the

`PROGRAM HEADER ASSIGNMENT`

and the

`LABEL ASSIGNMENT.`

Once again, as mentioned previously, these parameters can only be contained in the list in the program header. There are really no extra commands for these lists.
On the other hand, the

`CONSTANT ASSIGNMENTS`

have a few extra surprises:

`PI` as the real constant 3.1415926536E+00

`STKPOI` -- the pointer for the Pascal variable stack.

`STKPOI` is the two-byte pointer for the lowermost memory cell for the stack (top-of-stack). This pointer can call for parameters in the first line, or call certain parameter values: These values are of the `BYTE` type, and are characterized by the symbol

#

Let's clarify this a bit -- this constant can handle a single-byte value, and not a 2-byte integer, e.g.:

```
    CYAN = #3;                          SPRITE_ = #$3C;
```

The other supplements in constant assignment allow the use of simple constant expressions. The following are allowed:

DIV, MOD, SHL, SHR, *, +, -	for integer constants and/or their corresponding expressions
PRED, SUCC, ORD, LOW	for all constants and/or corresponding expressions
CHR	for integer and byte constants and/or corresponding expressions
LBYT, HBYT	for integer constants and/or corresponding expressions

TYPE ASSIGNMENT

There are two additional types:

BYTE = #0..#255;.

Defines the numerical contents of a one-byte-sized memory location. The other predetermined type is

STRING

which allows you to predefine any sequence of characters of a length up to to 132 characters (maximum print line in Super Pascal). A blank line is permissible in the form ''. Characters for string constants are, of course, treated as CHAR constants.

Let's take a quick look at how to handle string lengths. The type STRING is handled by the pointer like this:

```
RECORD LENGTH:BYTE; CHARACTER[1...LENGTH] OF CHAR
END;
```

This means that every time a new string is read, placed or generated on the heap (the memory heap for dynamic variables), more memory will be provided. MARK and RELEASE are also commands that can be taken into consideration when managing memory. This doesn't apply, however, to programs already containing string constants; they are automatically provided for in the compiling process. Internally-defined record elements are not accessible to the user.

Another intriguing point is the compatibility between a STRING and an

```
     ARRAY[INDEX] OF CHAR.
```

This means that opposite assignments and comparisons are possible. It also means that if a STRING quantity is longer than the defined ARRAY, the string will be tailored accordingly; then again, if the string is shorter than the chosen array, spaces will be inserted after the string to bring it to the same size as the array. The heap changes with the combination of a CHAR array with a string; the compiler, however, will only watch string length to avoid overflow. One great advantage to STRING types is the possibility of immediately reading these with READ or READLN (and with INPUT) from files. Here's an example:

```
CONST     LINELNGTH = #80;     {constant decl.}
VAR       TITLE:ALFA;          {variable decl.}
          LINE:ARRAY[0..PRED(LINELNGTH)] OF CHAR;
          TEMP,
          LINE :STRING;

BEGIN                         {command section}
 READLN(LINE);                {read string input}
 LINE:=LINE;                  {provide an array}
 IFLINE[0] IN ['A'..'Z'] THEN
 TEMP:=LINE;                  {provide temp. string}
 TITLE:=TEMP              {provide an ALFA quantity}
END;
```

VARIABLE ASSIGNMENTS

MEM :ARRAY[$0000..$FFFF] OF BYTE

RANDOM :REAL

MEM can access the entire memory of the C-64. That is, it can perform this task if the elements of this array are defined as BYTE types. MEM will also allow you to rearrange any memory cell (vague equivalent of "POKE") and read these cell contents (similar to "PEEK"), e.g.:

MEM[$277]:=LOW('A');	writes an "A" to the first memory location in the keyboard buffer and
NUMBER:=MEM[$C6];	transfers the number of the key pressed to the byte-variable NUMBER.

The variable RANDOM produces a random REAL number, which lies in the range:

```
0<= RANDOM < 1 .
```

RANDOM is best used in programming that requires random numbers; be forewarned, however, that the sequence of random numbers given isn't all THAT random -- a seed number is determined at startup, and the set of numbers depends upon that seed for its sequence.

COMMAND SET

The only modification to the command section is the CASE statement, with an ELSE-branch. Syntax:

CASE ... OF ... ELSE ... END

This means that if none of the criteria for the CASE statement are met, the ELSE will be the next command executed. Here's a sample program:

```
CASE CHARACTER OF
  'A':ONE;
  'B':TWO;
  'C':BEGIN ONE;TWO END;
  'D':THREE
  ELSE BEGIN ONE;TWO;THREE END
END;
```

If none of the values contained in 'A' .. 'D' are encountered, CHARACTER will go to the ELSE sequence: 'BEGIN ONE;TWO;THREE END'.

In contrast, this case statement operates differently without the ELSE:

```
CASE CHARACTER OF
  'A':ONE;
  'B':TWO;
  'C':BEGIN ONE;TWO END;
  'D':THREE
END;
```

NOTE:
As in an IF/THEN statement, ELSE shouldn't have a semicolon preceding it. The compiler will generate an error message otherwise.

STANDARD PROCEDURES

There are a number of procedures in Super Pascal that are unavailable to Standard Pascal. They are:

ALLOCATE

Unlike NEW, a pointer variable can be assigned to a memory cell by the user.

Syntax: ALLOCATE(POINTERVARIABLE,EXPRESSION);

POINTERVARIABLE stands for the identifier declared as a POINTER type in the assignment section. Access to this variable occurs with POINTERVARIABLE^.

EXPRESSION stands for that expression determining the pointer address. This expression must be an INTEGER. You can, for example, define an internal 2-byte address pointer as ^INTEGER, and easily manage memory in Super Pascal. Here's a sample program, using ALLOCATE:

```
TYPE  LINE = ARRAY[0..39] OF BYTE;
      SCRN = ARRAY[0..24] OF LINE;
VAR   I      :INTEGER;
      TEMP   :LINE;
      SCRNRAM:^SCRN;
BEGIN
 ALLOCATE(SCRNRAM,$400);TEMP:=SCRNRAM^[24];
 FORI:=0 TO 23 DO SCRNRAM^[SUCC(I)]:=SCRNRAM^[I];
 SCRNRAM^[0]:=TEMP
END;
```

This program gives you a continual screen scroll from top to bottom under Pascal control. This uses the procedure ALLOCATE(SCRNRAM,$400) to put the screen-repeat memory into $400 (decimal 1024). Bear in mind that the color RAM should be moved as the screen has been shifted, for the best demonstration of the program.

NOTE:
This procedure doesn't give you free reign over program code or other variables. A complete knowledge of memory layout will be necessary.

CLOSE

See the section on standard language elements.

Syntax: CLOSE(FILEVARIABLE);

FILEVARIABLE is the FILE type defined in the assignment block. This procedure will put the buffer contents to the last file opened for writing, and close the file; the file buffer will then be cleared for the next access. CLOSEing an unOPENed file produces the runtime error:

```
NOT OPEN ERROR!
```

and a program break.

CLRTRAP

This command, used without other parameters, clears the runtime error trap for I/O (input/output) errors. This means that after calling this procedure, neither a text error message nor a program break will occur. The I/O error trap is switched on with `SETTRAP`.

CONTINUE

This procedure lets you load and start an entirely different Pascal program.

Syntax: `CONTINUE(FILENAME,DRIVE_NR);`

The new program must be in the drive number indicated (`DRIVE_NR`), and must be listed under the proper identifier (`FILENAME`); the procedure finishes the loading process. LOADDAT is necessary to this procedure, so it must be in drive 0. If, by some chance, LOADDAT isn't available, a respective error message and program break happens. The program is loaded into the memory range where it was compiled.

A return to the original program isn't a vital part of this procedure, which makes possible the use of

EXECUTE

This procedure is similar to `CONTINUE` in calling a new program; in this case, though, it acts as a subroutine for the running program.

Syntax: `EXECUTE(FILENAME,DRIVE_NR);`

This procedure concludes the program load so that this procedure will execute under the conditions given by `CONTINUE`. As above, `FILENAME` and `DRIVE_NR` must correspond, and LOADDAT must be located in drive 0. The loaded program will be placed in the memory range at which it was compiled, and will use the variable stack range assigned by the compiler. Needless to say, the memory of the program first in memory must not run into any conflict with the registers of the currently loaded program. You'll have to program VERY carefully in terms of memory management and variable assignment.

HEX

This procedure converts integers and byte-numbers into hexadecimal numbers.

Syntax: `WRITE(FILEVARIABLE,...HEX(EXPRESSION)...);`
or
`WRITELN(FILEVARIABLE,...HEX(EXPRESSION)...);`

EXPRESSION stands for any `INTEGER` or `BYTE` expression. The expression can be input either in decimal or in hex (the latter with a dollar-sign preceding the number,e.g., $0A3F).

INDVC

Switches the active input device.

Syntax: `INDVC(EXPRESSION1,EXPRESSION2);`

`EXPRESSION1` refers to the desired primary address (device number), while `EXPRESSION2` gives the secondary address within the device. Both must be `INTEGER` types, with the primary address set within limits (0 - 255). Any number beyond or below this range will present

```
IL.QUANT. ERROR!
```

as a runtime error, and the program will stop.

When Super Pascal is initialized, the primary and secondary addresses are 0, which follows the `INPUT "GETLINE"` (from the keyboard). `EXPRESSION1` changes that device number in `INDVC` until a new procedure call changes it to another device, or if switched "manually". Runtime errors will reset the input device number to 0.

NOTE:
The primary address 2 will not operate the user port: It is NOT available as an `INDVC`. Although the possibility exists to adapt Super Pascal for this port, the system "as-is" will only work with serial devices.

KILL

KILL will delete unlocked (non-protected) files from the diskette and directory.

Syntax: `KILL(FILEVARIABLE);`

FILEVARIABLE is the label for the file to be scratched. If this is attempted with a locked file, the runtime error

```
IL.FILE OPR. ERROR!
```

appears, and the program stops. Locked files can only be dealt with in the Utility segment of the program. If the file isn't found in the running disk drive, again, an error message and a program end will occur.

LOCK

This procedure can be used in the same manner as CLOSE, i.e., for closing previously opened files. However, LOCK has one extra feature -- it protects files from overwriting and deleting.

Syntax: `LOCK(FILEVARIABLE);`

FILEVARIABLE is the FILE declared in the assignment section. A file need be locked only once (no need to do so repeatedly, unless you need access to the file, and have to unlock it). Attempts to scratch a LOCKed file will result in the program stopping, and

```
IL. FILE OPR. ERROR!
```

LOAD

LOAD puts an external Pascal routine into memory from diskette.

Syntax: `LOAD(FILENAME,DRIVE_NR);`

All that this command does is load the program, as opposed to CONTINUE and EXECUTE. The program must be loaded using the proper FILENAME and disk drive (DRIVE_NR). The load procedure itself requires the aid of LOADDAT (which must be in drive 0).

This procedure loads the program code into the memory location at which the code was compiled.

Program routines called with CONTINUE and EXECUTE

a) can handle independent Pascal programs; and
b) can be called for at any time;

while LOAD

a) will load independent programs, AND simple external procedures and functions (XTRNPRGM, XTRNPROC & XTRNFUNC); and
b)only offers one chance to use the command.

NOTE:
LOAD offers you no control over whether there is sufficient memory for the routine being loaded; you'll have to be very precise in knowing how much memory is involved, and how it is distributed.

Other examples are quoted in Chapter 4.1.2.2.

MARK

Together with RELEASE, MARK serves to control management of the heap (memory heap for dynamic variables).

Syntax: MARK(POINTERVARIABLE);

POINTERVARIABLE stands for the identifier for an ^INTEGER pointer variable, which becomes the active heap-pointer when the procedure is called. This is the pointer to the topmost portion of the variable stack and the ever-growing heap. Even when the heap is cleared (see NEW), any input strings cause the heap to begin growing yet again. If a situation occurs where the heap is unnecessary for storing strings or dynamic variables, RELEASE sets the pointer back to the POINTERVARIABLE. The next memory cell is available to you. See RELEASE for a short example.

NAME

This procedure allows you to give a program a different name from that stated by the current identifier.

Syntax: `NAME(FILEVARIABLE,EXPRESSION);`

FILEVARIABLE stands for the identifier which was declared within the assignment section as a `FILE` variable. As long as no changes have been made to this variable using `NAME`, the file variable will go under its "normal" identifier, i.e., by that filename on diskette. After providing the **EXPRESSION**, which must be `ALFA` or `STRING`, the file-variable will be changed to that name. Here's a short example:

```
VAR SOURCE:TEXT;       {formal declaration of file}
    TITLE:ALFA;           {variable        SOURCE,}
.
.
RESET(SOURCE);READ(SOURCE);CLOSE(SOURCE);
           {access to a file with the name SOURCE}

.
.
NAME(SOURCE,'OTHER');  {provision of actual name }
.                     {OTHER for the formal var.  }
.                     {SOURCE,                    }
REWRITE(SOURCE);      [access to file with current}
WRITELN(SOURCE,'1.LINE');  {name OTHER instead of}
CLOSE(SOURCE);   {the formal identifier SOURCE,   }
.
.
NAME(SOURCE,TITLE)    {provision of name contained}
.                     {in title as current file-  }
.                     {name, etc.                 }
```

OUTDVC

OUTDVC switches current output device.

Syntax: `OUTDVC(EXPRESSION1,EXPRESSION2)`

EXPRESSION1 stands for the primary address of the desired device, while EXPRESSION2 gives the appropriate secondary address. Both must be INTEGERs; the primary address must be within the boundaries of 0 to 255. Any differing address yields

```
IL. QUANT. ERROR!
```

and the program stops.

Thus, this procedure defines the output device to be used in conjunction with OUTPUT, WRITE and/or WRITELN. Primary and secondary default addresses (i.e., when Super Pascal is started up) are 0 (screen). An OUTDVC call (say, to send output to the printer) will remain at that address until the procedure is called again. Runtime errors automatically switch the output device back to 0.

NOTE:
As mentioned with INDVC, the user port is inaccessible.

RELEASE

This procedure represents the counterpart to the abovementioned procedure MARK; with it, the heap memory can be released from any earlier definition by MARK.

Syntax: `RELEASE(POINTERVARIABLE);`

POINTERVARIABLE is the ^INTEGER identifier for a pointer variable, which is contained in the "interim" heap pointer. This value will dictate where the heap pointer will be set. Below is an example of both MARK and RELEASE:

```
VAR  HEAP1,HEAP2,HEAP3:^INTEGER;   {decl. of three}
INFO;ALFA;                        {pointer variables of}
LINE:STRING;                      {^INTEGER type      }
.
MARK(HEAP1);                      {heap pointer starts }
.                                 {at HEAP1            }
MARK(HEAP2);                      {current heap pointer}
.                                 {at HEAP2            }
READLN(LINE);                     {read a string placed}
INFO:=LINE;                       {on the heap  and    }
.                                 {INFO provided       }
RELEASE(HEAP2);                   {reset heap pointer  }
.                                 {to value HEAP2      }
MARK(HEAP3);                      {freeze up current   }
.                                 {pointer at HEAP3    }
TRE_TREE;                        {call routine w/ dy- }
PRINT_TREE;                      {dynamic variable use}
RELEASE(HEAP3);                   {reset to value be-  }
.                                 {fore procedure call }
RELEASE(HEAP1);                   {release entire heap }
          etc.
```

SEEK

Used like RESET and REWRITE in opening files, the difference being that RESET and REWRITE use the access pointer "as-is", while SEEK lets you set that pointer.

Syntax: `SEEK(FILEVARIABLE,EXPRESSION);`

FILEVARIABLE stands for the identifier set up in the assignment section (FILE type). EXPRESSION sets the position of the file-access pointer. In cases where numbers might be an element in such a file (e.g., TEXT), EXPRESSION must be a REAL number. The access pointer will always take on the whole-number portion of that REAL expression. Negative numbers lead to the message

```
IL.QUANT. ERROR!
```

and a break, while numbers that overshoot the end-of-file give

```
    AFTER EOF ERROR!
```

and a subsequent program end.

The distinction between read/write operations in SEEK mode depends on the operation which follows: GET, READ or READLN puts you in read mode, while PUT, WRITE or WRITELN lets you write to the file. The access pointer will move to the next spot after each access.

After write access, any data after the write-position will be lost. The read/write operation is concluded with CLOSE or LOCK.

SETADR

This procedure contacts a running program to find and load an existing routine. Unlike LOAD, this is for resident or quasi-resident routines (esp. assembler routines). Syntax:

```
SETADR(PROCEDURE_FUNCTIONS_NAME,EXPRESSION);
```

PROCEDURE_FUNCTIONS_NAME represents the identifier of the externally called procedure/function. These "EXTERNALS" shall be called by these names which should have been defined in the procedure/function assignments. The procedure SETADR establishes the connection between name and actual address during runtime; this address is an integer stated in EXPRESSION. This can be handy when a routine is needed time and again (see 4.1.2.2 for an example).

SETDRV

This procedure sets the number of the current disk drive. The file-opening procedures RESET, REWRITE and SEEK come after SETDRV.

Syntax: `SETDRV(EXPRESSION);`

EXPRESSION must be an integer, and must state drive number (0 or 1, nothing else).

Initializing the program creates a default value of drive 0. And already-open file needs no further file definition; drive number will hold until the file is closed.

SETTRAP

SETTRAP (no parameters) switches the I/O error trap back on. Having the I/O error trap on will produce error messages and program breaks if (when) the time comes. Switching the trap off gives no messages, but programs will cease.

FUNCTION ASSIGNMENTS

There are three groups of predefined functions in Super Pascal:

Type conversion functions
Conditional functions
Mathematical functions

The first group consists of:

INT

This function converts real numbers into integers.

Syntax: INT(EXPRESSION) (function type: INTEGER)

EXPRESSION refers to a REAL expression. The conversion naturally works only if the quantity remains within the limits of -MAXINT...+MAXINT. Otherwise

```
IL.QUANT. ERROR!
```

comes up as a runtime error.

HBYT
LBYT

Both these functions deal with conversion of integers into BYTE quantities, simultaneously isolating high-bytes (HBYT) and low-bytes (LBYT).

Syntax: `HBYT(EXPRESSION)` (function type:`BYTE`)
`LBYT(EXPRESSION)` (function type:`BYTE`)

EXPRESSION is any integer. The function delivers the most significant byte (`HBYT`) of this integer, and the least significant byte (`LBYT`) of same. `BYTE` is the result in both cases. This can be convenient for m/l programming, since both "half-bytes" add up to one integer.

LOW

This function converts any scalar argument type (`REAL`) into a `BYTE` quantity.

Syntax: `LOW(EXPRESSION)` (function type:`BYTE`)

EXPRESSION stands for any scalar type. Using this function with integers will limit you to conversions up to 255 ($FF).

The second group of functions don't just operate as predefined functions which give Boolean information (yes/no cases), rather control internal system conditions.

ANYKEY

This function is chiefly used in programs involving input from the user, or just pausing until the user hits a key to go on. No parameters are needed:

Syntax: `ANYKEY` (function type:`BOOLEAN`)

It can be used, for example, for programming a wait loop, or perhaps you can have the system do something else while it's waiting for a keypress:

```
WHILE NOT ANYKEY DO;
```

It's just as simple to set up a conditional branch:

```
IF ANYKEY THEN ... (ELSE ...);
```

GETKEY

This function is comparable to BASIC's GET statement; it awaits input from the keyboard. No other parameters are necessary.

Syntax: `GETKEY` (function type:`CHAR`)

This allows you to read characters from the keyboard; every character will be pulled from the keyboard buffer (i.e., with `GETKEY`, every character will go to the buffer first). Here's an example of using `GETKEY` to control a program:

```
    CASE GETKEY OF ... (ELSE ...) END;
```

IOERROR

This has already been mentioned in connection with `CLRTRAP` and `SETTRAP`; it checks for I/O errors -- and if it finds one, looks to see which error it is. This function, too, can be called without argument.

Syntax: `IOERROR` (function type:`INTEGER`)

The only sensible time to use `IOERROR` is when the error trap has been switched off with `CLRTRAP`; otherwise, the system automatically reacts to any I/O errors. If, however, the trap is off, `IOERROR` will call up the number of such an error (NOTE: The program won't halt in this state). Here are the error numbers (all `INTEGER`, by the way):

FLOPPY ERROR	(1)
NOT OPEN ERROR	(2)
NOT CLO. ERROR	(3)
BUF.OV. ERROR	(4)
DIR.OV. ERROR	(5)
NOT FND. ERROR	(6)
DSC.OV. ERROR	(7)
DSC.MISM. ERROR	(8)
IL.FILE OPR. ERROR	(9)
AFTER EOF ERROR	(10)
IEEE-ERROR	(11)

No I/O errors gives the function a 0.

Runtime errors that aren't I/O-based - as already mentioned - always stop the program; these same errors aren't affected by the error trap's status:

```
OUT OF RNG. ERROR
NOT EXQ. ERROR
NUM.OV. ERROR
B.SUBS. ERROR
IL.QUANT. ERROR
STK.OV. ERROR
ZERO-DIV. ERROR
IL.DVC. ERROR
```

See Chapter 2.1.9 (RUN PROGRAM) for the definitions of these messages.

There are other functions grouped with these three conditionals.

FREE

Reads the amount of available memory between heap and stack at any time; no argument is needed.

Syntax: `FREE` (function type:`INTEGER`)

The value returned to you is expressed in 256-byte increments (pages), i.e., 1 block=256 bytes, 4 blocks=1K, etc. It's possible to end up with a

```
STK.OV ERROR
```

depending on the memory available.

LEN

LEN is an integer which supplies the length of a string, i.e., the number of characters in a string.

Syntax: `LEN(EXPRESSION)` (function type:`INTEGER`)

EXPRESSION refers to the string expression. This function is quite useful for determining the length of an unknown string within a file, and determining what to do about the length of same. The maximum allowable length of a string is the available size of the I/O buffer, while the maximum length of a printed line is 132 characters.

NOTE:
Attempts to overshoot these maximum lengths will lead to a system error.

SIZE

The size of a Pascal-variable can be found within a program with this function.

Syntax: `SIZE(TYPENAME)` (function type:`INTEGER`)

TYPENAME stands for the identifier stated in the type assignment of the program. Therefore, it is NOT the name of the variable itself, but of the type of variable. The return value will be given in bytes (rather than blocks); the value is given as an integer, in connection with `ALLOCATE`, i.e., the memory adjustments for pointer variables can be used.

The third group of functions contains the following three:

HXS

(HeX-Sum) This can be used for adding two integers without worry of overstepping the integer range.

Syntax: `HXS(EXPRESSION1,EXPRESSION2)` (func.type:`INTEGER`)

`EXPRESSION1` and 2 stand for the two integers to be added. For example,

```
     HXS($7F00,$1A80)  =  $9980
```

and

```
     HXS($A000,-$3800) = $6680.
```

SIGN

The SIGN-function gives a preceding character with a numerical expression:

Syntax: `SIGN(EXPRESSION)` (function type:INTEGER)

EXPRESSION can be either INTEGER or REAL. The function's result will give this EXPRESSION as an integer (the positive number ... +1; negative, ... -1). A functional argument of 0 gives a 0 result; similar to this example:

```
    EXPRESSION = SIGN(EXPRESSION) * ABS(EXPRESSION)
```

FRAC

The mathematical function FRAC delivers the opposite of the already-mentioned TRUNC -- it gives you the fractional section of a real number.

Syntax: `FRAC(EXPRESSION)` (function type:REAL)

EXPRESSION stands for any REAL expression; FRAC will separate the decimal numbers, and these numbers will be the result sent back to you. The leading character works the same here as in identifying functions.

These are the additional functions that you'll find in Super Pascal. The final section consists of the mathematical operations used within expressions. In addition to the normal operators:

SHL and **SHR**

Defined as reserved words. SHL (SHift Left) moves the bit pattern of an integer to the left, while SHR (SHift Right) moves an integer quantity to the right. The number of bits shifted is controlled by two operands:

```
Syntax:   EXPRESSION1 SHL EXPRESSION2 (type:INTEGER)
            EXPRESSION1 SHR EXPRESSION2 (type:INTEGER)
```

EXPRESSION1 and 2 can be any integers; the result will also be an integer.

Besides bit manipulation, these operations can also be used for quick multiplication with a factor of `2^EXPRESSION (SHL)` or fast division with `2^EXPRESSION (SHR)`. Examples:

```
    4747 SHL 2 = 4747 * (2^2) = 4747 * 4

    1111 SHR 4 = 1111 / (2^4) = 1111 / 16
```

AND OR NOT

These comparatives are `BYTE` types in SUPER Pascal; used for comparing bit patterns and checking memory contents, the result will consequently be a `BYTE` quantity. For example:

`#3 AND#12(#$03 AND #$0C)` gives the byte `#3 (=#$03)`,
`#161 OR #25 (#$A1 OR #$19)` gives `#185 (=#$B9)`
and
`NOT #200 (NOT #$C8)` gives the byte `#55 (=#$37)`.

4.1.2.2 ADDITIONAL PROGRAM STRUCTURES, EXTERNALS, SEGMENTS

This chapter will cover the techniques of program division and structure in Pascal, along with the connection and declaration of `EXTERNALS` and machine language routines. We'll try to include some common examples as we go along.

At the top of the list is the segmenting of Pascal programs. This division -- better known as overlay-technique -- involves breaking a larger program into several cooperative program blocks; this is called into play with the command

SEGMENT

Like `FORWARD`, this command is neither a reserved assignment command, command symbol nor execution command; rather, it's a control command for the compiler. The `SEGMENT` command is always in the same spot (syntactically speaking) as `FORWARD`, i.e., immediately after the procedure and/or function header. SEGMENT tells the compiler to treat the entire block of this procedure or function as a portion to be followed by other sections that will be compiled in the same memory range. The compiler notes the starting address of this block, and compiles those which follow at the same starting address. In a way, the segments are compiled as parallel program sections. The amount of memory reserved is dependent upon the longest segment being compiled.

There are a few ground rules for defining segmented blocks:

a) They must be arranged one immediately after another,

b) They must be defined as the same program level, regardless of label,

c) Interlocked routines should be avoided and

d) The whole number 8 should not be overstepped.

These limitations are not that bad, considering that you can sidestep some of them. For example, within a segment-assigned block, any deep function/procedure can be nested -- just as long as the remaining segments use the same procedures/functions. You should keep in mind that when working with segmented programs, the segments cannot be placed in the proper sequence by the computer itself; the computer will compile according to the sequence found on the diskette.

The 8-segment limit is really no problem, since there will be very few occasions when you'll write a program as large as that. One good example of a segmented program is the compiler itself; it's made up of the following segments:

INITIALIZATION	in which the predefined identifiers, functions and procedures are declared,
MAIN SECTION	which takes up compiling a block,
ASSEMBLER SECTION	which assembles the built-in assembly routines, and the
CONCLUDING SECTION	statistical evaluation.

These 4 segments, if put together normally, would take up a substantial amount more memory ($0800 - $C200); as compiled here, they only take up $0800 - $9000! Needless to say, segmenting programs is quite a practical move with the C-64.

NOTE:
Any reloading of segments requires that those segments all be in the same disk drive. Once the program is started, the disk drives can be switched around within the program. An additional file buffer will not be necessary for reloading segments.

The next two commands for developing larger programs shouldn't be unfamiliar to you, since we've mentioned them earlier:

CONTINUE EXECUTE

These procedures will let you load and run separate compiled (and complete) programs. There are differences between the two:

CONTINUE allows chaining of different programs, i.e., the new program can utilize variables and such defined in the previous program, or use its own definitions. No memory collisions can occur with continue.

EXECUTE allows separate Pascal programs to be called as subroutines to the main program. Memory must be set aside for both programs, so a solid knowledge of memory layout and management would be wise before using this technique.

"EXTERNALS"

We designed this category to allow for generating external programs and/or program routines. You can see the disadvantages of EXECUTE (see above); the command discussed here lets you define program routines as procedures or functions. The compiler recognizes these external reserved words:

XTRNPROC	(eXTeRNal PROCedure)
XTRNFUNC	(eXTRNal FUNCtion)

The compiler registers these as declared procedure/function identifiers and their respective parameter lists. The proper block for this procedure/function is canceled, since it is, of course, assigned externally. Now, in order for the program to find the `EXTERNAL`, a `LOAD` (for implicit address assignment) or `SETADR` (for explicit assignment) must be included (see 4.1.2.1). The assigned routines for `XTRNPROC` and `XTRNFUNC` are nested with the main program's variable stack. You can define all parameters in SUPER Pascal as predefined variable types; same goes for function values.

External procedures/functions will be compiled as such, i.e., contained in the program header NOT by the word `PROGRAM`, but rather by their `XTRN` identifiers and parameter lists. The rest is compiled like a normal Pascal program block.

To round out the set, we come to declaration of entire external programs, which are handled like the above externals.

`XTRNPRGM` (eXTeRNal PRoGraM)

No further parameters are needed; the main program calls the external program using the identifier defined in the main routine. Here again, we must be concerned about the starting addresses of external and main program, whether loading implicitly or explicitly (`LOAD` and `SETADR`, respectively). External programs are compiled "normally".

When using externals, it's important to remember that the main program will load the externals into the given memory cells; there is always a possibility of memory collision, if you haven't planned your memory layout carefully. Calling externals with `CONTINUE` or `EXECUTE` avoids these problems.

The last point we'll cover in this sector will be the "`USER`" routines.

These represent an extra step beyond the external routines. Unlike the externals, user routines are external machine-language routines, though assigned like standard procedures and functions. There are two types:

USERPROC
USERFUNC

You would then give these routines identifiers matching those given within the main program's assignment section. For more details on handling these routines, see Chapter 4.1.2.3 on the "internal" m/l routines.

During runtime, the procedure SETADER must be used to assign the jump address. The machine language routine must be consistent with the start-of-program, because, for example, when loading a code-file into the file buffer with RESET, the buffer address will equal the jump address. To help you out a bit, here are some items concerning memory information, and a sample program.

Regarding program design----

> The main program (example) should run and work in $2000 - $9FFF.
> Three independent subprograms will be generated in $0800 - $1FFF (SUB1, SUB2 and SUB3).
> Registers $A000 - $A7FF have been assigned to a procedure (X_PROC), while $A800 - $AFFF have been assigned a function (X_FUNC).
> An exit program (BADEXIT) has been designed for $0800 - $8FFF.
> A machine-language routine (TEST1 and TEST2) will be defined in $F100 - $F27F and $F280 - $F3FF, in connection with the file USERCODE.

```
PROGRAM EXAMPLE;
 CONST               {address declaration for SUPER}
  BUFFER1 = $F100;   {Pascal system file buffers;  }
  BUFFER2 = BUFFER1 + $400;  {an opened file will}
  BUFFER3 = BUFFER2 + $400;  {go to the first}
 .                           {free buffer   }
 .
 TYPE
  RECORD
      RANGE:ARRAY[0..99] OF ALFA;
      SET  :SET OF CHAR;
     END;

 .
 VAR
  TABLE :ARRAY[1..3] OF RECORD;
  FLOW  :INTEGER;
```

```
.
PROCEDURE REGULATE;          {Decl. of a normal }
 BEGIN ... END;              {Pascal procedure;  }
XTRNPROC X_PROC           {declaring an external }
 (A,B:INTEGER;MSG:STRING); {procedure w/parameter}
                              {transfer; }
XTRNFUNC X_FUNC        {declaring an external}
                         { function }
 (CH:CHAR;VAR TITLE:ALFA):BOOLEAN
.
FUNCTION READALFA            {decl. of a normal }
 (VAR READFILE:TEXT):ALFA;   {Pascal function;   }
 VAR
  INPUT:STRING;
 BEGIN
  .
  READLN(READFILE,INPUT);READALFA:=INPUT;
  .
 END;
.
USERPROC TEST1               {declares an )
 (VAR TAB:RECORD);           {assembler-procedure;}
USERFUNC TEST2:BOOLEAN;      {declares an }
.                            {assembler-function;}
.
PROCEDURE INIT;SEGMENT,      {declares a procedure}
 BEGIN                       {segment;           }
  .
 LOAD(X_PROC,0);      {load X_PROC from drive 0;}
 LOAD(X_FUNC,0);      {load X_FUNC from drive 0}
 SETADR(TEST1,BUFFER1);        {address transfer  }
 SETADR(TEST2,BUFFER1+$180); {address transfer; }
 RESET(USERCODE);              {load program-code }
 .                             {into file buffer1}
 FOR FLOW:=1 TO 3 DO
  TEST1(TABLE[FLOW]);     {multiple call of TEST1;}
 IF NOT TEST2 THEN       {call of Boolean function}
  BEGIN                      {TEST2            }
   CLOSE(USERCODE);
   CONTINUE(BADEXIT,0)       {prg. jump to BADEXIT}
   END;
  .
  END;
PROCEDURE PART1(JOB:KENNER);{declaring second  }
  SEGMENT;                  {segment block;    }
  BEGIN
   .
   CASE JOB OF        {call for one of the three }
    LOAD :EXECUTE(SUB1,1);   {Pascal subprograms -}
    SAVE:EXECUTE(SUB2,1);    {SUB1, SUB2 or SUB3  }
    REGISTER :EXECUTE(SUB3,1)
    ELSE ...
   END;
   .
  END;
FUNCTION PART2:BOOLEAN;      {declaring the third }
  SEGMENT;                   {segment block ;     }
  .
  BEGIN
   REGULATE;
   .
   PART1(MENU);             {call another segment;}
   .
   PART2:=          {provision for function val.;}
    READALFA(INPUT)='END'
  END;
PROCEDURE EXIT;SEGMENT;   {declaration of fourth }
  BEGIN ... END;           {segment block         }
BEGIN                       {main program          }
  INIT;                 {call for INIT. segment; }
  IF PART2 THEN EXIT        {call for OK-output;  }
  ELSE CONTINUE(BADEXIT,0) {call for error-output;}
END.
```

4.1.2.3 ASSEMBLER ROUTINE DESIGN

Inserting assembler routines in a Pascal program is a subject already touched upon in Chapter 3.4; see that section for a sample program. Here, however, we'll look at the "mechanism" used for parameters and function return values. More detailed information on 6510 machine language will be found in Chapter 5.2, but for LEARNING machine code, we suggest you read books dealing directly with the subject (see Appendix).

Here are the commands accepted by the compiler for integrating 6510 code and Pascal (pseudo-instructions: For the complete set, see Chapter 5.3):

.BA

This pseudo-instruction will tell the assembler the starting address of the program to be assembled (also, the address is vital to the Pascal program itself). This is the routine which embeds the routine into the Pascal program.

.OC

This pseudo-command suppresses the machine-language output, once the generation of the addresses (for the address label) is complete (note: this command is not provided in Pascal itself). The machine code will be produced within the Pascal program sequence.

.CT

This pseudocode will chain assembler sources (not possible in Pascal proper).

Keep the following in mind regarding parameter and function values: The place will be reserved on the Pascal variable stack for functions defined in the assignment section, and for the function return value, i.e., the top- of-stack will be adjusted accordingly. This will happen regardless of whether it is a regular Pascal function, a machine-language function, or an external "USER" function. (NOTE:Please see Chapter 4.1.1 for variable size, and use of the function SIZE). Machine programs have a different access mechanism to the stack -- indirect-indexed addressing.

The relative address (calculated from top-of-stack) is put into the Y register of the CPU; and the instruction

```
    LDA(STKPOI),Y
```

lets any byte be put on the Pascal stack. If parameter bytes go over 256, the most significant byte will be incremented by the zeropage pointer `STKPOI`. `STKPOI` (address $2E) is recognized by Pascal as a predefined quantity.

When a function return value should be put onto the stack, it must appear in the proper place on the stack `(STA(STKPOI),Y)`, i.e., above all eventual given parameters.

The stack pointer will again be corrected at the end of the machine-language routine, i.e., set to the value preceding the call of the m/l routine.

This point should be remembered when integrating m/l and Pascal; constants can be set up for the Pascal section within the m/l section.

The sample here may clear up some of the mystery of parameter and function return values:

An assembler routine assigned with

```
FUNCTION DEMO(MSG:STRING;CHARACTR:CHAR;VAR
WORD:ALFA):INTEGER
```

and called with

```
IF 36 = DEMO('HELLO',CX,TITLE) THEN ...
```

whereby CX should be a `CHAR`-variable, and `TITLE` and `ALFA`- variable. Below is an illustration of stack management (TOS=top-of-stack):

```
                         high address
 1.  TOS before entering |_____..._____|
   comparative expression|_____..._____|
 2.  Deposit a value of  |_____$00_____|
     36..................|_____$24_____|..........
 3.Arranging a place for |_____..._____|         |
   the funct return value|_____..._____|.........|
 4.  Deposit the string  |____ADR H____|      |  |
     address 'HELLO'.....|____ADR L____|....  |  |
 5.  Deposit CX..........|_____..._____|...|  |  |
 6.  Deposit address for |____ADR H____|   |  |  |
     variable TITLE......|____ADR L____|...|  |  |
 7.  TOS enters DEMO     |      |      |   |  |  |
     (= STKPOI).................|          |  |  |
 8.  Parameter range which                 |  |  |
     can be accessed with                  |  |  |
     (STKPOI),Y............................|  |  |
 9.  Deposit function                         |  |
     return value.............................|  |
10.  STKPOI corrected when leaving            |  |
     DEMO.....................................|  |
11.  Comparative operation                       |
     taken up....................................|
```

4.1.2.4 COMPILER COMMANDS

We mentioned before that you can embed different compiler directives within a Pascal program. These commands are all preceded with an ampersand (the '&' character). You can use the "long form", or an abbreviated versions of the commands -- here are both versions (the short versions are printed here in parentheses):

&ADR+	(&A+)
&ADR-	(&A-)
&CONTINUE	(&C)
&INCLUDE	(&I)
&PCODE+	(&P+)
&PCODE-	(&P-)
&TRUTH	(&T)

&CONTINUE and &INCLUDE, used for inserting and appending program sources, have already been discussed in Chapter 3.2. &TRUTH, used in conditional compiling, has also been explained. The remaining commands (&ADR and &PCODE) serve to control address declaration and PCODE output.

&ADR+ will switch on address output, giving the memory address for every line: This is useful for debugging runtime-errors. This output can be switched off with &ADR-.

PCODE output is switched on using &PCODE+, and off with &PCODE-. For every PCODE instruction given, the compiler generates a mnemonic command abbreviation, with the memory location and necessary parameters (in bytes). The PCODE abbreviations are as follows:

```
ADDI = ADD IMM. WORD
CALI = CALL INDIRECT
CALL = CALL ABSOLUTE
CALS = CALL SEGMENT
CPIB = COMPARE IMMEDIATE BYTE
CPIN = COMPARE IMM. n BYTES
CPIW = COMPARE IMMEDIATE WORD
EQUN = COMPARE n BYTES (=)
GEQN = COMPARE n BYTES (>=)
GETN = GET n BYTES (>)
GOTO = GO TO
GRTI = COMPARE IMM. WORD (>)
GRTN = COMPARE n BYTES (>)
INCT = INCREMENT STACK
JCDO = COND.-JUMP DOWN
JCUP = COND.-JUMP UP
JMPC = COND.-JUMP ABSOLUTE
```

```
JUMP = JUMP ABSOLUTE
LEQN = COMPARE n BYTES (<=)
LESN = COMPARE n BYTES (<)
LITB = LOAD IMMEDIATE BYTE
LITW = LOAD IMMEDIATE WORD
LODA = LOAD ADDRESS
LODB = LOAD BYTE
LODS = LOAD STRING
LODW = LOAD WORD
LODX = LOAD INDEXED
LSSI = COMPARE IMM. WORD (<)
MULI = MULTIPLY IMM. WORD
NEQN = COMPARE n BYTES (<>)
NEWN = NEW n BYTES
NOP  = NO OPERATION
OPRC = OPERATION CODE
PFIX = PREFIX OPR. CODE
PUTN = PUT n BYTES
RTRN = RETURN ABSOLUTE
RTNS = RETURN SEGMENT
STOB = STORE BYTE
STOS = STORE STRING
STOW = STORE WORD
STOX = STORE INDEXED
SUBI = SUBTRACT IMM. WORD
TBYT = CHECK BOUNDS
WRTA = WRITE ARRAY
```

The &ADR and &PCODE commands can be started with a general command at the start of the compiling process, then left on for the entirety of the procedure.

4.2 OPTIONS

SUPER Pascal offers a number of options for the compilation process itself. You do, of course, have the "option" of not choosing any options -- before compiling, the system will ask you

```
      DEFAULT OPTIONS ? N/Y
```

and if you wish to compile "as-is", press "N". If, however, you choose "Y", the options will run off in sequence, beginning with

```
START-OF-PROGRAM
```

which allows you to change the staring address to your liking. You have $0800 to $C1FF to work with, and, under very special circumstances, the file buffer range ($F100 to $FEFF) at your disposal as well. With free choice of starting address, it's possible for you to easily develop a larger program packet from smaller units (with the help of the memory map). The default value --

```
      START OF PRGM = $0800
```

-- can be retained, or changed (decimal OR hex value).

```
VARIABLE MEMORY
```

The compiler prompts with

```
      START OF HEAP = EOPGM
```

to tell you the starting point of the heap (storage for dynamic variables), from bottom of heap to the top of the stack (used for static variables). The default is EOPGM (end-of-program), i.e., the heap will be placed immediately after the end of the program being compiled. You, however, can reorganize the heap to your preference. After defining the start-of-heap,

```
      TOP OF STACK = $9000
```

denotes the default for the end of the stack. Be sure the input is correct or

```
     ILLEG. DECLARATION!
```
or even
```
     START OF HEAP EXCEEDS TOP OF STACK!
```

can occur. All in all you have from $0800 to $C1FF for program code and variable storage, and, in special circumstances ONLY, $F100 to $FEFF (file-buffer space). If all input so far has been proper, we go on to

```
COMPILATION MODE
```

The compiler prompts you with

```
     P.-CODE TO DISK ? N/Y
```

You have your choice of either compiling to diskette or compiling in RAM.

> Diskette Compilation:
> The default mode writes the p-code generated to disk as a temporary file (CODDAT); the fix-up information used to complete the compiling phase is placed in the so-called FIXUP-FILE. The fix-up procedure is necessary to eventually install the correct addresses into the program code once the single-pass compiler is done. Analogous to this is the management of assembler program sections, which are assembled with a two-pass process -- this is the reason for the second choice ---
>
> RAM Compilation:
> The compiler generates p-code directly into RAM memory. The fix-up process and the two-pass procedure will be handled in memory as well. The advantage to RAM mode lies in the higher working speed, since no write operation is required of the disk drive at the time; however, one way or another, you'll still end up saving the Pascal source to diskette.

In order to generate Pascal programs in RAM that you'll want to run later, the compiler will claim some memory for itself, and will let you determine the memory at which the compiled program will be located. The system will ask

```
     STORING ADRS. = $9000
```

The default is $9000 (the compiler itself takes up $0800 - $8FFF); memory available to you is $9000 - $C1FF.

Owing to parallel addressing, compiling segmented programs in RAM is impossible; if attempted, the compiler will give an error message.

VARIABLE CONTROL

The compiler prompts with

```
     TESTS OF BOUNDS ? N/Y
```

which gives you the choice of controlling the low-range defined variables. The default identifies the variable-defined boundaries, and is extraordinarily important for array-indices. The control is accountable for `IL. QUANT. ERROR` messages (runtime). The control mechanism will be set into the program as additional p-code.

NOTE:
Choosing variable control (bound test) should be for security of program control, on condition that the program has been thoroughly tested first. For example, false array indices (outside a defined array) tend to cause extremely nasty and hard-to-localize errors. Be very sure that the program is as completely debugged as possible (and, of course, that enough memory is available).

POST-MORTEM-DUMP

A particular problem in compiled programs is the diagnosis, analysis and cure of runtime errors; the problem is often a serious one in Pascal. SUPER Pascal has the ability to make a "post-mortem-dump", i.e., after running into a runtime error, the program section is dumped with corresponding section, function, procedure, and gravity of the error; also, the variables are listed with defined names and contents at that moment. Normally, the post-mortem-dump is suppressed, but this can be changed with the prompt

```
     SUPPRESS PMDUMP ? N/Y
```

Unless stated otherwise by you, a file will be dumped as

```
DUMP-TITLE = P_M_DUMP
```

The printout will consist of the source-code on the one side, and the coded program on the other. NOTE: You'll be better off debugging the source-code, and just re-compiling the source.

A/P OPTION

By default, the compiler ignores the integral commands &ADR+/ &ADR- and &PCODE+ and &PCODE-:

```
IGNORE A/P-OPT. ? N/Y
```

Change this option ONLY if you're utilizing these commands.

OUTPUT FORM/HARDCOPY

The last option gives control over the output form during the compilation process. Default value for output is "suppressed":

```
SUPPRESS OUTPUT ? N/Y
```

'N' will give you a line-by-line listing of the source text onscreen. If output is suppressed, the compiler generates an asterisk (*) for each line, and lists only the names of procedures and functions being compiled.

```
SUPPR. HARDCOPY ? N/Y
```

clarifies whether the compiler will run output normally (onscreen) or send the output to a printer. If the latter is desired,

```
OUTPUT DEVICE 4,0
```

will be the default for the primary and secondary device numbers. Incorrect input will produce

```
ILLEG. INPUT!
```

4.3 THE COMPILATION PROCESS

Pascal sourcecode (as well as procedures and functions written in 6510 assembler notation) will be converted by the SUPER Pascal compiler into a viable pascal program. The compiler is accessed from the Main Menu using the C-command. This subprogram awaits a source program (textfile) from diskette. Once in the C-command menu, the system asks for the filename to be compiled, and the disk drive in which said file can be found:

```
FILE-TITLE = ?
DRIVE(MAP)  = x
```

Rather than give a filename, you can use an asterisk (*), which tells the system to compile the last program contained within the editor. The system will ask for confirmation:

```
CONFIRM "FILENAME,DRIVE_NR?" N/Y
```

Improper input will return the system to the Main Menu.

If all input is acceptable, the compiler loads into the computer from the system diskette; remember to have the disk with LOADDAT and C_CPLR in drive 0. If these programs aren't in drive 0, or the textfile isn't in the stated disk drive, the system will generate appropriate error messages, and return to the Main Menu. If the file turns out NOT to be a textfile, a corresponding error message will be displayed, and the compiler will abort to the Main Menu.

After the compiler has initialized, and the source program has been opened by the compiler, the following will appear:

```
READY TO COMPILE: PROGRAM "NAME,DRIVE_NR"!
```

`NAME` represents the identifier for `PROGRAM` in the program header; `DRIVE_NR` stands for the drive in which the source file exists.

If a source other than a program (e.g., an external function or external procedure) is to be compiled, the above messages will use the appropriate word (XTRNPROC/XTRNFUNC) instead of the PROGRAM symbol. In conclusion, the program will ask

```
DEFAULT OPTIONS ? N/Y
```

to confirm whether to use internally defined parameters or not ('Y' if so). If the response is 'N', the relevant prompts will run by you (see Chapter 4.2).

Externals have no default values, so you'll have to go through the options menu to provide parameters (again, see Chapter 4.2). NOTE: Externals have no variable range of their own available.

Now the compiler will take the source program, and produce a viable program code. Any syntax errors will be pointed out by the compiler (see 4.4).

Assuming no errors have cropped up, the program codes are linked and saved; after this, the compiler returns program control to the Main Menu. If, however, a compiling error arises in the text, or if the RUN/STOP key is pressed, the compiler will immediately load and run the editor, to let you edit the program. From there, you'll have to return to the Main Menu to recompile the program.

4.4 ERROR MESSAGES

This chapter deals with the handling and classification of syntax errors which might arise in the program text. For those Pascal novices, you'll run into many such errors in your first few attempts at programming; don't let this get you down -- expert programmers slip up a lot, too. If, after having problems, you consider switching to a language other than Pascal, remember that Pascal has error control seldom seen in other languages. The compiler drops out at the slightest discrepancy.

The compiler will display the error number, the offending line, and mark the error itself with an up-arrow (^) (Note: This display will either occur on the screen or the printer, dependent on what you have defined as an output device). Screen output will await your acknowledgement of the error (press

<SPACE> to continue). The compiler will then look for the next convenient place to go, and continue compiling from that point on.

If the syntax problem is a meaningless write error (e.g., ',' instead of ';'), the compiler gives you a WARNING rather than an ERROR.

The compilation process can be stopped at any time with the RUN/STOP key; this will automatically load and run the Editor section, and the source code being worked on at the time. This also happens at the end of the entire compilation, if any errors have cropped up. Once edited, the program can be re-saved using "*" to represent the most recent filename used.

The total number of errors and warnings is displayed at the end of compilation (see Chapter 1.3.3 for a complete list of error messages, and the error lists used at the end of this manual).

4.5 END OF COMPILATION

How the compilation ends depends upon the manner of compilation. Errors in the sourcecode call the editor program, and reload the program (see Chapter 3 [Editor] and 4.3 [Compilation Process]).

However, if all goes well, the compiler prompts for a statistical summary:

```
STATISTICAL SUMMARY? N/Y
```

'Y' (yes) puts out a list of data concerning the program -- see next page:

```
     STATISTICAL SUMMARY OF "NAME____":
     ==================================
     NO ERRORS! // xx WARNINGS!
     MAXIMUM OF STATIC LEVELS = x
     MAXIMUM OF VALID IDENTIFIERS
      INCL. PREDFND. IDENT'S = xx
     AT THE SCOPE OF "NAME____"
     MAX. OF VALID PARAMETERS = xx
     DECLARATIONS IN DETAIL ...
          DIV. REFERENCES = xx
                CONSTANTS = xx
                VARIABLES = xx
        FIELD-IDENTIFIERS = xx
               PROCEDURES = xx
                FUNCTIONS = xx
       PARAMETERS-BY-NAME = xx
                    TOTAL = xx

PRGM-PCODE AT: $xxxx ... $xxxx (= $xxxx)
HEAP/STACK AT: $xxxx ... $xxxx (= $xxxx)

LINKING AND SAVING "NAME____" .....
    ---> PRESS "RETURN"
```

Immediately following the last output line, the compiler begins fixing up the p-code, diskette compilation, and connecting segmented program code. To see how things came out, right after the compiler returns to the main menu, hit R and "*" to run the compiled program.

4.6 LOCALIZING RUNTIME ERRORS

Runtime errors are those errors which aren't found during compilation; in fact, the only time that you WILL find them is when the program is up and running. The program will stop and give you a runtime message; this doesn't give you specific information as to why the error occurred.

SUPER Pascal helps you avoid runtime errors. Clearing the I/O trap will skip over I/O errors. Most of it has to come from YOU, though; the best way to avoid errors is to do as much "fine work" in the testing stages as is possible.

One especially important factor in debugging is the ability to find the problem areas, i.e., the place at which the runtime error occurred, and what state the data is in at this point. The "classical" solution is to surround the suspected areas with `WRITE` statements

a) to convey up to what point the program runs properly
and
b) to output "suspicious" variables.

This, however, is time-consuming work. SUPER Pascal, which gives an error message and the memory location involved, takes at least some of the mystery out of finding the problem. Attempts at verifying errors by recompiling the source using "`&ADR+`" is inexcusable.

Another aid is the post-mortem-dump. A program interruption gives all available information, and allows you to find those especially tenacious runtime errors. The PM-dump is in the Options menu (see Chapter 4.2).

If this option is chosen, a special marker will be put into the program; when a runtime error is encountered, the error display will automatically load and run the post-mortem output control.

All available data up to and including the error will be tabulated and listed. The variables will be listed by their identifiers AND present contents (when possible). LOADDAT (for loading), C_PMDUMP (output program proper) and the respective program file must all be available. The PM-dump file will be in its specified drive, but LOADDAT and C_PMDUMP MUST be in drive 0.

The PM output program will ask whether the output will be onscreen or to the printer. This determines format for array and record variables.

Another trick in SUPER Pascal for finding runtime errors lies in the Editor and Utility programs. Frankly, these don't help all that much -- they can be in connection with the following:

> The program call from the MAIN menu sets the address pointer to the system address $0363 (ADR_PRPO).
>
> The return from a program to the MAIN menu -- from program end or runtime error -- makes an indirect spring using the address pointer at $0361 (ADR_EXPO).
>
> Calling a program will set in the `MAIN` a so-called `WARMFLAG` ($0360), setting that flag to 0.
>
> If the program is has a starting address matching up with the pointers `ADR_PRPO` and `ADR_EXPO`, every program break will jump immediately back into the program. It goes to the start-of-program, be it first time or re-entry (This is controlled by `WARMFLAG`).

5.0 THE 6510 ASSEMBLER

The 6510 assembler runs completely in harmony with SUPER Pascal. This assembler is, however, a separate program, and must not be confused with the compiler-integrated assembler segment. Essentially, the assembler takes 6510 assembler source code and helps you turn these source codes into functional 6510 machine code.

The assembler itself is a Pascal program, but that makes no difference: It will still turn out acceptable 6510 code, and you will only occasionally notice that the SUPER Pascal assembler is slower than a standard machine code assembler/monitor.

The great strength to this assembler is its ability to assemble huge source texts; a splendid example of this is the m/l runtime packet in SUPER Pascal which has 200K of assembler squeezed into 8K of program code (divided into 7 individual files).

Another advantage to this assembler is the fact that when machine language is being generated directly to disk, there is no possibility of memory collision occurring.

5.1 SOURCETEXT DESIGN

This material has already been touched upon in Chapter 3.3. Bearing that in mind, we'll only recap the most important items here, just to avoid repeating ourselves too much.

The assembler converts a textfile into 6510 machine language (if the source text is in proper syntactical form). Each line is set up in a columnar arrangement, with each column reserved for a specific purpose:

```
Text line      : ZZZZ LLLLLLLL III OOOOOOOO...
```

POSITION 1-4 (`ZZZZ` = line number)

This field contains the line number.

POSITION 5 (space)

This column separates the line number from the next item (label field) with a space.

POSITION 6-13 (LLLLLLLL = label field)

This field contains the label by which specific areas within an assembler program are recognized. Labels are written in the same manner as Pascal identifiers:

8 significant characters
First character must be a letter
Remaining characters can be letters, numbers and/or '_' (ASCII $5F, the back arrow on the C-64)

POSITION 14 (space)

Separates the label field from the instruction field.

POSITION 15-17 (III = instruction (operator) field)

This field contains the 6510 mnemonic instruction (see 5.2), and will also accept pseudocommands (see 5.4).

POSITION 18 (space)

This space separates the instruction field from the operand.

POSITION 19 ff. (OOOOOOOO... = operand field)

The operand field, in which the operand corresponding to the operation (see above) is contained; the first line gives the address type (see 5.3 for an explanation of addressing).

Commentary can be supplied after the operand field; begin the comment line with a semicolon (;).

5.2 COMMAND SET

The 6510 assembler built into SUPER Pascal accepts standard 6510 (or 6502, if you prefer) mnemonics, as well as pseudo-instructions (preceded by a period '.'). Here are all the 6510 operation codes:

ADC	Add memory to accumulator with carry
AND	"and" memory with accumulator
ASL	shift one bit left (memory or accumulator)
BCC	branch on carry clear
BCS	branch on carry set
BEQ	branch on result zero
BIT	test bits in memory with accumulator
BMI	branch on result minus
BNE	branch on result not zero
BPL	branch on result plus
BRK	force break
BVC	branch on overflow clear
BVS	branch on overflow set
CLC	clear carry flag
CLD	clear decimal mode
CLI	clear interrupt disable bit
CLV	clear overflow flag
CMP	compare memory and accumulator
CPX	compare memory and x-register
CPY	compare memory and y-register
DEC	decrement memory by one
DEX	decrement x-register by one
DEY	decrement y-register by one
EOR	"exclusive-or" memory with accumulator
INC	increment memory by one
INX	increment x-register by one
INY	increment y-register by one
JMP	jump to new location
JSR	jump to subroutine (retain return address)
LDA	load accumulator with memory
LDX	load x-register with memory
LDY	load y-register with memory
LSR	logical shift right (memory or accumulator)

NOP	no operation
ORA	"or" memory with accumulator
PHA	push accumulator on stack
PHP	push processor status on stack
PLA	pull accumulator from stack
PLP	pull processor status from stack
ROL	rotate one bit left (memory or accumulator)
ROR	rotate one bit right (memory or accumulator)
RTI	return from interrupt
RTS	return to subroutine (back to main prg.)
SBC	subtract memory from accumulator w/ carry
SEC	set carry flag
SED	set decimal mode
SEI	set interrupt disable status
STA	store accumulator in memory
STX	store index x in memory
STY	store index y in memory
TAX	transfer accumulator to index x
TAY	transfer accumulator to index y
TSX	transfer stack pointer to index x
TXA	transfer index x to accumulator
TXS	transfer index x to stack pointer
TYA	transfer index y to accumulator

5.3 TYPES OF ADDRESSES

The opcodes quoted in the last chapter are actually quite versatile -- they can be addressed in different ways. The different types of addresses and their symbols are listed below. These types can be defined in the LABEL EXPRESSION, with the respective operand and type stated there. The expressions can

be made of symbolic labels
be in either decimal or hexadecimal form
be in CHAR form (ASCII)
present arguments for functions in H and L (high-byte, low-byte form).

The elements named can be combined with + and - for addition or subtraction. Examples of these label expressions:

```
OUTPUT          $D           0                 L,BUFFER
I_O_PORT        $001A        13                H,MEMORY
TIMER1          $FFFE        1024              H,10000
'X'             . .          ADDRESS+1         LBL-2
L,BUFFER+41     EXIT+$10     $400+65+$F0000-MEM_ADR
```

The address types for the 6510 CPU:

IMP (implied)	Syntax: no operand section
ACC (accumulator)	Syntax: A
IMM (immediate)	Syntax: #LABEL EXPRESSION
ABS (absolute)	Syntax: LABEL EXPRESSIONS
ABX (absolute,X)	Syntax: LABEL EXPRESSION,X
ABY (absolute,Y)	Syntax: LABEL EXPRESSION,Y
ZPG (zero page)	Syntax: #LABEL EXPRESSION
ZPX (zero page,X)	Syntax: #LABEL EXPRESSION,X
ZPY (zero page,Y)	Syntax: #LABEL EXPRESSION,Y
IXX (indexed,X)	Syntax: (LABEL EXPRESSION,X)
IXY ((indexed),Y)	Syntax: (LABEL EXPRESSION),Y
IND (indirect)	Syntax: (LABEL EXPRESSION)
REL (relative)	Syntax: LABEL EXPRESSION

These types coincide with the opcodes in the following table.

OPCODE	IMP	ACC	IMM	ABS	ABX	ABY	ZPG	ZPX	ZPY	IXX	IXY	IND	REL
ADC			*	*	*	*	*	*		*	*		
AND			*	*	*	*	*	*		*	*		
ASL		*		*	*		*	*					
BCC													*
BCS													*
BEQ													*
BIT				*			*						
BMI													*
BNE													*
BPL													*
BRK	*												
BVC													*
BVS													*
CLC	*												
CLD	*												
CLI	*												
CLV	*												
CMP			*	*	*	*	*	*		*	*		
CPX			*	*			*						
CPY			*	*			*						
DEC				*	*		*	*					
DEX	*												
DEY	*												
EOR			*	*	*	*	*	*		*	*		
INC				*	*		*	*					
INX	*												
INY	*												
JMP				*								*	
JSR				*									
LDA			*	*	*	*	*	*		*	*		

OPCODE	ADDRESS TYPES												
	IMP	ACC	IMM	ABS	ABX	ABY	ZPG	ZPX	ZPY	IXX	IXY	IND	REL
LDX			*	*		*	*		*				
LDY			*	*	*		*	*					
LSR		*		*	*		*	*					
NOP	*												
ORA			*	*	*	*	*	*		*	*		
PHA	*												
PHP	*												
PLA	*												
PLP	*												
ROL		*		*	*		*	*					
ROR		*		*	*		*	*					
RTI	*												
RTS	*												
SBC			*	*	*	*	*	*		*	*		
SEC	*												
SED	*												
SEI	*												
STA				*	*	*	*	*		*	*		
STX				*			*		*				
STY				*			*	*					
TAX	*												
TAY	*												
TSX	*												
TXA	*												
TXS	*												
TYA	*												

5.4 PSEUDO OPERATION CODES

The pseudo operation codes accepted by the assembler are for controlling the assembler and generating code. All pseudo opcodes are preceded by a period (.):

CONTROL pseudo opcodes:

.BA (Begin Assembly)
Syntax: `.BA` ADDRESSEXPRESSION

This command defines the starting address for the machine code. ADDRESSEXPRESSION stands for an absolute address in decimal or hexadecimal form, or for an expression already defined as a label.

NOTE:
Expressions should NOT contain spaces; any material after spaces will be viewed as commentary by the assembler (and consequently ignored).

.CT (ConTinue with ...)
Syntax: `.CT` FILENAME

This command appends separate source programs. FILENAME stands for the file desired in the current disk drive.

.DL (Define Label)
Syntax: `.DL` ADDRESSEXPRESSION

This command will determine the comparison between an already-used label name, and the address given at ADDRESSEXPRESSION.

.EN (ENd of assembly)
Syntax: `.EN`

This signals the conclusion of assembly. `.EN` can be defined as a label (e.g., `END .EN`).

.EQ (if EQual to 0)
.NE (if uNEqual to 0)
. . . (end of condition)

SYNTAX:
```
.EQ ADDRESSEXPRESSION
.NE ADDRESSEXPRESSION
... ADDRESSEXPRESSION
```

These instructions will handle conditional assembly; the details are handled in 3.3.

.OC (Objectcode Clear)
.OS (Objectcode Set)

Syntax:
```
.OC
.OS
```

These codes switch the machine code generator off (.OC) and on (.OS); the default value of the generator (i.e., at power-up) is ON. This option allows insertion of already- assembled external program code.

PROGRAMMING pseudo opcodes:

.BY (BYte (table)

Syntax:
```
.BY BYTELIST
```

This can insert any sequence of bytes of running machine code. The number of bytes are governed by the size of the bytelist (min. 1 BYTE - max. to end-of-line). If more bytes need be generated, just keep calling up .BY in the lines to follow. A BYTELIST is any set of bytes; they can be in decimal or hex; or, individual characters and strings, e.g.:

```
.BY 0 128 255
.BY 0 $80 $FF
.BY 'A' 'C' 'E' 'U'
.BY 'SUPER PASCAL'
.BY 0 'A' $80 'DATA' 255
```

A commentary character (;) or a c/r ends the bytelist in any line.

.DS (DiSplacement)
Syntax: .DS ADDRESSEXPRESSION

This command can create large memory ranges in machine language. The assembler generates code from $00 through the amount stated in ADDRESSEXPRESSION; the assembly continues with the next available memory address.

.SA (Set Address)
Syntax: .SA LABELEXPRESSION

From this command, the assembler generates a 2-byte address (low-byte/high-byte) and puts it into the code. LABELEXPRESSION is an expression made up of any labels and/or absolute addresses (hex or decimal).

5.5 RUNNING THE ASSEMBLER / OPTIONS

The assembler is loaded from the MAIN menu using the 'A'command; it will load an assembler sourcecode from diskette. The system prompts with:

```
FILE-TITLE = ?
DRIVE(MAP) = x
```

The default value for x is the number of the disk drive last used; by rights, then, you need only press <RETURN>.

If the file to be assembled was edited most recently, you can simply respond to the FILENAME prompt by pressing * and <RETURN>. The system asks for verification:

```
CONFIRM "FILENAME,DRIVE_NR"? N/Y
```

Any incorrect input will abort the assembler, and return you to the MAIN menu.

When all materials have been properly entered, the assembler will load from the system diskette (which requires LOADDAT and C_ASMBLR in drive

0). If the textfile is not found, or if the file isn't a textfile, the system will display the proper error message and return you to the MAIN menu.

Once the assembler has initialized, and the sourcecode file has been opened, the assembler displays

```
*    C=64 6510 ASSEMBLER  5.3 *
```

and prompts with

```
LISTING ? Y/N
```

so it knows whether or not to run a program listing (not designed like the source text, but rather a listing of memory locations and machine code in hex notation). If commentary running over 80 characters per line exists, the right portion of the commentary will be cut off. The system questions further:

```
HARDCOPY ? Y/N
```

-- giving you the option of seeing the listing on screen or on paper. If you choose the latter, the output device numbers will be requested:

```
OUTPUT-DVC = 4,0
```

Once this is confirmed, PASS 1 of the assembly process will commence.

The assembler might find some syntax or formula errors: An error message and the offending line will be displayed. For example,

```
2005 1_BUFFER LDA #1
```

will generate

```
ILLEG. CHARACTER IN LABEL ERROR IN ..
2005 1_BUFFER LDA #!
```

If you're reading this onscreen rather than on a printout, the assembler will wait for you to press the <RETURN> key before continuing -- to give you a chance to write the problem down.

Here are the possible error messages:

```
ILLEG. CHARACTER IN LABEL ERROR IN ..
ILLEG. MNEMONIC ERROR IN ..
ILLEG. PSEUDO ERROR IN ..
ILLEG. OPERAND ERROR IN ..
ILLEG. BYTE-DEFINITION ERROR IN ..
LABEL NOT FOUND ERROR IN ..
DUPLICATE LABEL ERROR IN ..
ILLEG. ADDR. MODE ERROR IN ..
ILLEG. INDEX ERROR IN ..
ILLEG. ADDRESS ERROR IN ..
LONG BRANCH ERROR IN ..
.EN MISSING ERROR IN ..
```

If the first pass goes without a hitch, the assembler announces the good news:

```
PASS 1 OK
```

-- and starts PASS 2, which assembles the file, and stores it on diskette as a temporary file (CODDAT). Errors are displayed just as in PASS 1.

If all has gone well, the assembler lets you know --

```
PASS 2 OK.
---> 0 ERRORS _--
```

-- and asks for the name of the object code file:

```
TITLE OF OBJECT-FILE =
```

You give the identifier that you wish this m/l program to have.

Next, you'll be asked about the fate of the label list:

```
LABEL-FILE TO DISC ? Y/N
```

Choosing 'Y' will make the system ask for a filename:

```
TITLE OF LABEL-FILE =

LABEL-FILE TO PRTR ? Y/N
```

This gives you the option of printing out the label list file. If 'Y' is chosen, the system will ask for the printer address:

```
OUTPUT-DVC = 4,0
```

If you so desire, the label file can be sent to the screen and/or the printer. One more time, you'll be asked about the label list:

```
LABEL-LISTING ? Y/N
```

This time, if you say 'Y', the system will put this listing onscreen. The list is arranged in alphabetical order of labels, together with their address definitions. The list can be stopped and resumed by pressing <SPACE>. The RUN/STOP key aborts the output, and returns you to the MAIN menu. Choosing 'N' for the label list prompt will also send you back to the MAIN section.

If errors are found during PASS 2, the system scratches (deletes) the temporary file CODDAT; the label list is still accessible, however. Leaving the assembler automatically loads and starts the Editor, which loads the bad sourcecode, so that you can immediately go in and debug it. From there, you must go back to the MAIN before calling the Assembler.

Pressing the RUN/STOP key while in the Assembler will display a

```
BREAK ...
```

and load the Editor and sourcefile.

Here is a short program demonstrating the design of an assembly program, the program listing output, and the label list output. The program should switch the C-64's screen on and off in intervals of one second.

```
1000 DEMO      .BA $0800;DEMO PROGRAM SWITCH SCREEN
1005 ;
1010 CPUPORT   .DL 1   ; DETERMINE MEM. CONFIGURATION
1015 VICREG17  .DL $D0000+17   ; SW. SCREEN BIT 4 OFF
1020 MARKER    .DL $FF00  ;MARKER CELL FOR SCR. MODE
1025 ;
1030 START     LDY #0          ;RESET COUNTER
1035           STY MARKER      ;INITIALIZE MARKER
1040 LOOP0     JSR SWITCH      ;SWITCH SCR. MODE
1045           DEC MARKER      ;MARK BIT 0
1050 LOOP1     JSR DUMMY    ;11-COUNT CPU TIME DELAY
1055           DEX             ;LOWBYTE COUNT
1060           BNE LOOP1       ;256 TIMES
1065           DEY             ;HIGHBYTE COUNT
1070           BNE LOOP1       ;256 TIMES
1075           JMP LOOP0  ;SWITCH -- BREAK CONDITION
1080 ;                          ;IS HERE
1085 SWITCH    LDA *CPUPORT
1090           ORA #1          ;I/O BANK ON
1095           STA *CPUPORT
1100           LDA VICREG17
1105           EOR #$10        ;INVERT BIT 4
1110           STA VICREG17
1115           LDA *CPUPORT
1120           AND #$FC        ;RAM-BANK ON
1125           STA *CPUPORT
1130 DUMMY     RTS
1135 ;
1140 END       .EN

------------------------------------------------

1000 $0800          DEMO  .BA $0800
                     ;DEMO PROGRAM SWITCH SCREEN
1005 $0800          ;
1010 $0800          CPUPORT  .DL 1
                     ;DETERMINE MEM. CONFIGURATION
1015 $0800          VICREG17 .DL $D0000+17
                     ; SW. SCREEN BIT 4 OFF
1020 $0800          MARKER   .DL $FF00
                     ;MARKER CELL FOR SCR. MODE
```

```
1025 $0800              ;
1030 $0800 A0 00        START    LDY #0 ;RESET COUNTER
1035 $0802 8C 00 FF              STY MARKER
                         ;INITIALIZE MARKER
1040 $0805 20 17 08     LOOP0    JSR SWITCH
                         ;SWITCH SCR. MODE
1045 $0808 CE 00 FF              DEC MARKER;MARK BIT 0
1050 $080B 20 2B 08     LOOP1    JSR DUMMY
                         ;11-COUNT CPU TIME DELAY
1055 $080E CA                    DEX ;LOWBYTE COUNT
1060 $080F D0 FA                 BNE LOOP1 ;256 TIMES
1065 $0811 88                    DEY ;HIGHBYTE COUNT

1075 $0814 4C 05 08              JMP LOOP0
                         ;SWITCH -- BREAK CONDITION
1080 $0817              ;         ;IS HERE
1085 $0817 A5 01        SWITCH   LDA *CPUPORT
1090 $0819 09 01                 ORA #1 ;I/O BANK ON
1095 $081B 85 01                 STA *CPUPORT
1100 $081D AD 11 D0              LDA VICREG17
1105 $0820 49 10                 EOR #$10;INVERT BIT 4
1110 $0822 8D 11 D0              STA VICREG17
1115 $0825 A5 01                 LDA *CPUPORT
1120 $0827 29 FC                 AND #$FC ;RAM-BANK ON
1125 $0829 85 01                 STA *CPUPORT
1130 $082B 60           DUMMY    RTS
1135 $082C              ;
1140 $082C              END      .EN

PASS 2 OK.

> 0 ERRORS <

                       LABELLIST

CPUPORT  $0001  DEMO   $0000  DUMMY  $082B
END      $082C  LOOP0  $0805  LOOP1  $080B
MARKER   $FF00  START  $0800  SWITCH $0817
VICREG17 $D011
```

6.0 UTILITY MENU

The Utility program is an extremely useful software packge. You know that a utility is universally defined as a program that helps you program; our packet gives you simple disk management and help in running SUPER Pascal. Pressing 'U' brings you to the Utility menu; it is important that two programs, LOADDAT and C_UTILIT, be in disk drive 0.

One advantage of the Utility program is that the loading of programs can be handled in this section itself. Once this menu is started, the system diskette is no longer needed. Here's what you'll see on initialization:

```
            *  C=64  FILE-UTILITY  5.3  *

COMMANDS = ...

A(DVICE)          J(UMP)               S(TOREMEM)
B(LOCKTABLE)      K(ILLTITLE)          T(RNSFRMEM)
C(OPY)            L(OCKFILE)           U(NLOCKFILE)
D(UPLICATE)       M(AP/DRIVE)          V(IEWMEM)
E(NTERSECT)       N(EWDISC)            W(RITEDIR)
F(ETCHSECT)       O(RGANIZE)           X(CLUDEBLC)
G(ETRAM)          Q(UIT)               Z(EROBLOCK)
I(NSERT ADV)      R(ENAME)
```

The cursor always turns into a dollar-sign ($) when you're in Utility. As mentioned previously, typing the first letter and <RETURN> gets you the individual menu selections; all other input requires pressing the <RETURN> key at the conclusion of the input line. Numbers can be entered in decimal or hexadecimal form (preceded by $, of course). False string input will be answered with

```
ILLEG. INPUT!
EXECUTION NOT SUCCESSFUL!
```

Improper numeric input will yield

```
INVALID INPUT
EXECUTION NOT SUCCESSFUL!
```

Disk access defaults to system drive 0; the 'M' command can redefine drive numbers.

The Utility program is fairly insensitive to errors -- any problems will bring up appropriate error messages, and hand control back to the Utility menu itself.

You have the option of sending Utility output to the screen or a printer. The default printer address is 4,0 but you can change it at any time with

```
@X,Y
```

with X representing the primary address, and Y the secondary address.

6.1 UTILITY COMMANDS

6.1.1 A (= ADVICE)

This command lets you review the user-specific information in any file (assuming that information has been added -- see 'i'). First prompt is:

```
FILE-TITLE =
```

to which you respond with the filename which has the information you want to view. The system searches the disk drive and displays

```
ADVICE TO "FILENAME,DRIVE_NR":
current information ...
```

If no information exists,

```
... NO ADVICE INSERTED!
```

appears. If the file itself doesn't exist, you'll see

```
     TITLE NOT FOUND!
     EXECUTION NOT SUCCESSFUL!
```

6.1.2 B (= BLOCKTABLE)

This command displays the block table map (or block availability map, as it's known in BASIC) of a diskette. The table is in Pascal-DOS, which means that the diskette is divided into 40 blocks of 4K each, with each block subdivided into 8 512-byte sectors.

'B' displays individual blocks with symbols explaining status. Here are the symbols, and their definitions:

`F (FREE)`
 The block displayed is ready to be used. The internal content is 0.

`I (INVALID)`
 This block shouldn't be changed; it contains the disk directory and information (see 6.1.1 and 6.1.9). Internal value is 255 ($FF).

`U (USED)`
 This block is filled; internal value is >= 80 and <96.

`X (eXCLUDE)`
 This block has been reserved from the DOS (see the 'X' command); the block can be freed up with the 'Z' command. Internal value is 256 ($FE).

The block table of the system disk looks something like this:

```
BLOCK-TABLE OF DISC "PASCAL ,0":
... ('XCLUDE,FREE,INVALID,USED)

 0:  I U U U U    U U U U U
10:  U U U U U    F F F F F
20:  F F F F F    F F F F F
30:  F F F F F    F F F F F
```

6.1.3 C (= COPY FILE)

Here you can copy Pascal-DOS files, regardless of type. The system prompts for the following parameters:

```
SOURCE - DRIVE = ?
DESTINAT-DRIVE = ?
FILE-TITLE = ?
```

Any bad input will repeat the prompts. Once all input is sent, the system will copy the file. If the system has two disk drives, the program will perform data transfer in a block-wise manner, while the single-drive system will load the file, ask you to change to the destination disk, and press <RETURN>, which will save the file to the new diskette.

Other information (ADVICE) is copied as well as the file. The procedure ends with "READY" displayed. If there is insufficient space on the destination disk, you'll get either

```
DISC OVERFLOW!
EXECUTION NOT SUCCESSFUL!
```

or

```
MAP OVERFLOW!
EXECUTION NOT SUCCESSFUL!
```

If the destination disk has a filename identical to the file you're copying, you'll get

```
FILENAME EXISTS ON DESTINATION-DISC!
SURE TO REWRITE FILE ? Y/N
```

to which if you respond 'Y', the old file will be overwritten by the new.

6.1.4 D (= DUPLICATE DISC)

In cases where large amounts of information must be copied (or, for that matter, all 40 blocks of a diskette), the 'D'command is at your disposal; it can be used ONLY with a two-drive system:

```
SOURCE - DRIVE = ?
DESTINAT-DRIVE = ?
```

requires your response (0/1). Since the destination diskette may be overwritten, you'll get this prompt to confirm:

```
DISC MAY BE USED;  SURE TO REWRITE ? Y/N
```

Once the parameters have been given, the system performs blockwise copying --

```
COPYING; PLEASE WAIT!
BLOCK IN PROGRESS ... x
```

If you try this command with only one disk drive, the system will protest:

```
NO DUPLICATING WITH SINGLE-FLOPPY!
EXECUTION NOT SUCCESSFUL!
```

Should something go wrong to stop the copying process (e.g., drive switched off, no diskette in drive, unformatted diskette, etc.), a corresponding error message appears, and execution ceases.

NOTE:
Duplication can only be done on diskettes formatted with SYSGEN!!

6.1.5 E (= ENTER SECTOR)

This command allows any 512 byte memory range to be saved to any sector of the diskette. The following parameters are requested:

```
RAM-ADR = ?
SECTOR# = ?
```

Input errors will make these prompts repeat.

Disk sectors are lined up in a logical sequence, with eight sectors to a block (sectors 0-7 in block 0, sectors 8-15 in block 1, etc.), up to 319 sectors. Double-drive systems offer sectors from 0-639.

In cases where the block is marked "I" or "U" (see 6.1.2), the system will ask for confirmation:

```
CONDITION OF CORRESPONDING BLOCK: x
SURE TO SAVE INTO THIS SECTOR ? Y/N
```

NOTE:
There is a possibility of overwriting old data, or even destroying the disk directory (sector 0); be careful.

6.1.6 F (= FETCH SECTOR)

The 'F' command is the reverse of 'E'; it will transfer any sector from diskette into memory. Prompts:

```
SECTOR# = ?
RAM-ADR = ?
```

Illegal input will be ignored. Once loaded, the sector can be displayed with the 'V' command.

NOTE:
The 'F' command doesn't check for sufficient memory space when loading. You have the entire memory from $4000 to $C200 available for this command (and, in exceptional cases, $0400-$07FF (screen memory)).

6.1.7 G (= GET FILE FROM DISC TO RAM)

This command loads any file from diskette to the computer, which can be useful for temporarily storing information as well as loading programs. This dialogue occurs:

```
    START-ADR = ?
```

-- give the address of where you want the program in memory (either in decimal or hexadecimal).

```
    FILE-TITLE = ?
```

-- you supply the filename.

```
    DRIVE(MAP) = x
```

-- give the drive number where the file can be currently found (default is the last-used drive).

```
    END-ADR+1 = $xxxx
```

--assuming the rest of the input was valid, give the ending address in memory.

NOTE:
This command doesn't test for available memory, or whether any collisions may occur (see the NOTE at 'F' for available memory).

6.1.8 H (= HELP)

This command displays the complete command list for the Utility menu.

6.1.9 I (= INSERT ADVICE)

Advice (extra information) is put in using this command (and read with 'A'). Mostly, this advice can consist of version number, memory range, starting address, etc.).

```
FILE-TITLE = ?
```

asks for the filename to which you want to add comments.

```
CONFIRM "FILENAME,DRIVE_NR"? N/Y
```

asks for verification. If the title isn't on the diskette, the machine responds with

```
TITLE NOT FOUND!
EXECUTION NOT SUCCESSFUL!
```

The prompt for the information will read:

```
WRITE THE ADVICE (MAX. 63 CHAR.)
AND TERMINATE WITH 'RETURN' !
```

The comments will be stored in sectors 1-5 (block 0) of the diskette.

6.1.10 J (= JUMP)

This allows a jump to any machine language or Pascal program in memory.

```
PRGM-ADR. = ?
```

-- you give the jump address.

NOTE:
There is no control over memory overlapping.

You have $4000 to $C200 at your disposal for a jump. When through with the routine, it would be wise to have

```
     JMP $0800
```

for the last command (this returns you to the Utility menu); do NOT return to $0028-$004F, $0340-$0379 or $0800-4000.

The 6510 command

```
     JMP $C200
```

will return you to the MAIN menu.

6.1.11 K (= KILL TITLE)

The 'K' command allows you to delete diskette files no longer needed.

```
     FILE-TITLE = ?
```

asks for the filename you wish scratched.

```
     CONFIRM "FILENAME,DRIVE_NR ? N/Y
```

asks for verification; 'Y' will delete the file (the disk drive can be redefined with the 'M' command).

If the file isn't in the drive, you'll get

```
     TITLE NOT FOUND!
     EXECUTION NOT SUCCESSFUL!
```

and a stopped command.

If the file is locked, the Utility will recheck --

```
FILE IS LOCKED!
SURE TO KILL THE FILE ? N/Y
```

Pressing 'Y' will kill the locked file.

At the conclusion of the process, the revised directory will be displayed onscreen.

6.1.12 L (= LOCK FILE)

Files can be protected from overwriting and deletion by this command. The Utility asks:

```
FILE-TITLE = ?
```

If file isn't existent, the system says

```
TITLE NOT FOUND!
EXECUTION NOT SUCCESSFUL!
```

The appropriate file is locked (and is shown in the directory in reverse video).

6.1.13 M (= MAP/DRIVE)

The 'M' command serves to display the directory (or MAP) of a diskette onscreen.

```
DRIVE(MAP) = x
```

x defines the drive desired; the default is the last utilized disk drive, so a <RETURN> alone will often suffice.

Note that the 'M' command reads Pascal-DOS disks ONLY! Since the DOS has been rewritten, the system cannot read disks formatted in the "normal" way. The system disk, with the exception of 22 blocks of normal size (256 bytes), the entire Pascal disk is under Pascal-DOS.

The directory shows filenames and the number of blocks still available on the disk (remember, Pascal blocks equal 4K each).

The map of the boot diskette looks like this:

```
MAP OF DISC "PASCAL  ":
LOADDAT    SYSGEN      C_EDITOR   C_UTILITY
C_CPLR  C_ASMBLR  C_PMDUMP
DISC 0 = 18 //
BLOCKS FREE !
```

Further information about chosen files can be had with the 'W' command. See Chapter 7 for more information about Pascal-DOS.

6.1.14 N (= NEW DISC)

This command clears the directory of a diskette already formatted using SYSGEN. First, state which drive has the disk to be "newed out":

```
DRIVE(MAP) = x
```

Default value is the last disk drive accessed. Incorrect input is treated as mentioned earlier.

For security reasons, the Utility asks the user for confirmation:

```
DISC MAY BE USED; SURE TO REWRITE ? Y/N
```

If you wish to go on, respond with 'Y'; the system will ask:

```
DISC-TITLE = ?
```

-- you give the name you want given to the diskette.

```
N OF DISCS = ?
```

Answering '1' will new one disk (40 blocks); '2' (assuming you have two drives) will new BOTH disks for use as one (totaling 80 blocks).

The new directory will be listed on the screen

6.1.15 O (= ORGANIZE DISC)

This command works in connection with 'N' -- where in 'N', two disk drives are used to create one directory, this command can return us to "single disk" status. Also, the disk is "organized" -- a closer packing of files.

```
DRIVE(MAP) = x
```

x = the drive number which contains the system diskette. Concluding with

```
NEW SIZE =
```

reorganizes the disk. If your input above is equal to 2, be sure that the second disk is in drive 1 (the second drive). An input of 1 separates the material in drive 0 from the files in drive 1. This procedure must conclude with the 'N' command if the second diskette has files on it, the Utility says:

```
DISC >= 1 NOT FREE!
SURE TO RESIZE DISC ? Y/N
```

6.1.16 P (= PUT RAM AS FILE TO DISC)

Store any memory contents to diskette as a datafile (see 'G' to retrieve). Parameters are as follows:

```
     START-ADR.  = ?
     END-ADR.+1  = ?
     FILE-TITLE  = ?
     DRIVE(MAP)  = x
```

Addresses can be in decimal or hex; filenames must be given per syntax for Pascal identifiers:

8 significant characters
1st char. must be a letter
remaining chars. can be letters, numbers and '_'

Default for x is the last drive number accessed.

NOTE:
Any file already on the destination diskette with the same name as the file being saved will be overwritten, unless the original file is locked: Then

```
     IL.FILE OPR. ERROR!
```

will appear.

You have the following memory available to you for this procedure:

```
$0000-$CFFF(RAM); $D000-$DFFF(I/O); $E000-
$FFFF(KERNAL)
```

6.1.17 Q (= QUIT)

Exits Utility menu and goes to MAIN.

6.1.18 R (= RENAME FILE)

'R' lets you rename any file in the directory. The system will ask:

```
     FILE-TITLE  = ?
```

You give the filename to be changed (NOTE: The disk with this file must be in the directory, or a 'TITLE NOT FOUND!' error will appear).

```
REPLACEMENT=
```

is the prompt for the new filename. The directory will then be changed, and the revised map shown onscreen.

If the new filename already exists, the Utility states

```
TITLE EXISTS ON THIS DISC!
EXECUTION UNSUCCESSFUL!
```

and the procedure is left undone.

It is also possible to change the diskette name itself with 'R'.

6.1.19 S (= STORE BYTE INTO MEMORY)

Byte information can be immediately changed in the 64, and stored in memory. The system will ask:

```
MEM-ADR =
CONTENTS=
```

Give the memory address and the contents of that address (both in decimal or $hexadecimal); a CONTENT of over 255 ($FF) will be ignored by the Utility.

NOTE:
No testing for the legality of the content in the memory location.

6.1.20 T (= TRANSFER MEMORY-BLOCK)

Here you can transfer memory contents (for test purposes) in increments of 1 memory page (256 bytes); however, the starting address can be virtually any number (see below), just as long as you don't go past page borders (low-order byte = $00).

```
ADR OF SOURCE - PAGE = ?
ADR OF DESTINAT-PAGE = ?
```

The input can be either in hex or decimal.

NOTE:
The target range isn't tested for what sort of manipulation it can perform. Use this command only if you're well-versed in memory management. You have available memory of $4000- $CFFF.

6.1.21 U (= UNLOCK FILE)

Opposite of 'L' -- unlock secured files.

```
FILE-TITLE = ?
```

requests the filename, which must be in the disk drive (drive can first be redefined with 'M', as necessary). If the file isn't onhand, the system responds with

```
TITLE NOT FOUND!
EXECUTION NOT SUCCESSFUL!
```

and starts over again. Assuming that all is well, the file is unlocked, and the revised directory is displayed.

6.1.22 V (= VIEW MEMORY)

The 'V' command lists any memory range to the screen or a printer; contents will be printed out in hexadecimal and -- when possible -- in ASCII form (hex-dump). The Utility asks for

```
START-ADR. = ?
END-ADR.+1 = ?
```

which can be given in either hex or decimal. If the start and end addresses are identical, the Utility will show the one line on screen. If a fair amount of memory is requested, the prompt

```
HARDCOPY TO PRINTER ? Y/N
```

will appear. Any changes to the printer addresses can be made according to the introduction to this chapter.

A memory dump of, say, $C200-$C22F would look like this:

```
<<MEMORY DUMP>>

$C200:20 21 CA 12 C2 00 00 BC   !......
$C208:F1 C1 00 00 00 C2 FD FF  ........
$C210:00 00 4D 19 C2 FF B1 11  ..M.....
$C218:C6 80 1C 2A 20 43 3D 36  ...* C=6
$C220:34 20 20 50 41 53 43 41  4  PASCA
$C228:4C 2D 53 59 53 54 45 4D  L-SYSTEM
```

Memory contents in the range $20-$7f are ASCII characters. This listing can be stopped and restarted with the <SPACE> bar; pressing RUN/STOP, however, aborts the program.

6.1.23 W (= WRITE DIRECTORY)

This command sends an entire disk directory to the printer in extended form (with extra information):

```
CONDITION:
```
Whether the file is locked or unlocked.

```
STARTBLOCK
```
The first block in which information is stored on diskette.

```
LENGTH
```
File length -- in an X,Y format (X= number of 256-byte pages, and Y= remainder not counted in X).

```
ADVICE
```
File information. If non is available, '---' is printed.

After calling the 'W' command, you'll be asked for the drive number:

```
DRIVE(MAP) = x
```

As before, the default for x is the last drive used.

Finally, the output mode will be asked for:

```
HARDCOPY TO PRINTER ? Y/N
```

Responding with "Y" will start printer output. Printer specification should be done with '@X,Y', as previously mentioned in the introduction to Chapter 6.

You have a choice of seeing the extended directory onscreen or on paper ; you have control over the first by pressing the <SPACE> bar to stop and resume output. Pressing RUN/STOP breaks off either screen or printer output.

Here is a sample directory -- one of a disk just formatted with SYSGEN:

```
<< DIRECTORY OF DISC "PASCAL  ,0" >>

FILE-TITLE "LOADDAT"
CONDITION:LOCKED     STARTBLOCK: 1
   LENGTH:  63.255   ADVICE: ---

(list of  files)

TOTAL:  1 DISC // 1 TITLES //
        5 BLOCKS  (35 FREE)//
```

6.1.24 X (= 'XCLUDE BLOCK)

This command allows you to set aside blocks of memory from regular use by the DOS. Such a block registers in the block availability map with a value of 254 ($FE), and is marked on the block table with an 'X'. Answer the prompt

```
EXCLUDING-BLOCK =
```

with an appropriate number; attempts to exclude the directory block (block 0) will be turned away with

```
INVALID INPUT!
EXECUTION NOT SUCCESSFUL!
```

The new BAM will be displayed on the screen.

Excluded blocks can be accessed with the 'E' command.

6.1.25 Y (= LIST FILE)

This generates a hex-dump from any Pascal-DOS-accessible file (similar to 'V', which dumps a certain memory range).

```
FILE-TITLE = ?
```

Give the name of the file to be listed -- must be in the directory (and, consequently, in the disk drive).

```
HARDCOPY TO PRINTER ? Y/N
```

'Y' sends the file to the printer, rather than to the screen.

NOTE:
File dumps will begin with $0000, regardless of memory address at which the file is located.

Onscreen dumps can be stopped and resumed with the <SPACE> bar -- RUN/STOP aborts any dump format.

Here's a sample dump:

```
<< FILE-DUMP of "LOADDAT ,0" >>

$0000:20 21 CA 12 F7 81 00 BA   !......
$0008:F1 C1 00 00 00 F7 FE FF  ........
$0010:00 00 4D 34 F8 4C 09 CA  ..M4.L..
$0018:4C 06 CA 5E 02 79 41 26  L..^.YA&
$0020:F1 08 5F FA F7 08 FC 08  .._.....
$0028:40 29 79 5E 02 79 6B 79  @YY^.YkY
```

6.1.26 Z (= RELEASE BLOCK TO ZERO)

This releases an excluded block for regular use by the Pascal-DOS, and give the block a value of 0 (marked in the block table with an 'F').

```
RELEASING-BLOCK (TO ZERO) =
```

Input any number except 0 (directory block).

If this block is already occupied with memory, the system will confirm:

```
BLOCK IS USED!  SURE TO RELEASE ? Y/N
```

REMEMBER: If you say 'Y' after this prompt, the data that was in this block is lost forever.

The procedure concludes with a display of the revised BAM.

7.0 SYSTEM-SPECIFIC INFORMATION

This chapter should give you enough detailed information about SUPER Pascal's design to let you develop, adapt and change it to suit your own needs. You can reach this information with your own file access.

7.1 SYSTEM SIZE AND DEFINITION

Variable Design

BOOLEAN VARIABLES are one byte in size, and are one of two values:

```
FALSE   =   0000 0000
TRUE    =   0000 0001
```

CHAR and BYTE VARIABLES represent user-specified scalar variables, and run in a range from

```
          0    ($00)    =   0000 0000
to
        255    ($FF)    =   1111 1111
```

CHAR VARIABLES stand for the ASCII codes of the characters in the C64's system.

INTEGER VARIABLES are two-byte, binary-coded numbers, where the msb (most significant bit) contains the integer information (0 for positive, 1 for negative). They have the following range:

```
-32767 (-MAXINT) ($8001) = 1000 000  0000 0001
              -1 ($FFFF) = 1111 1111 1111 1111
               0 ($0000) = 0000 0000 0000 0000
               1 ($0001) = 0000 0000 0000 0001
+32767 (+MAXINT) ($7FFF) = 0111 1111 1111 1111
```

REAL VARIABLES total 6 bytes in binary-coded exponential form. The most significant byte represents the binary exponent:

```
(2^) -127 ($01)      =      0000 0001
(2^)    0 ($80)      =      1000 0000
(2^)  127 ($FF)      =      1111 1111
```

The remaining five bytes represent the normal Mantissa, i.e., the msb is always 1, so that its representation is assured. The Mantissa function is integral (1 = positive, 0 = negative). For example:

-23.5 would be (in binary) -10111.1 = -1.01111 * 10 ^ 100
with the following 6 bytes: $84 $BC $00 $00 $00 $00

Zero is not available here. The value 0 would be arranged with an exponent of 0 ($00) thus:

```
$00 $80 $00 $00 $00 $00
```

ADDRESS quantities are represented in two bytes.

SET VARIABLES can contain up to 256 elements (256 bits = 32 bytes). The lesser byte represents the elements 0-7, while the greater byte stands for the elements 248-255.

ARRAY VARIABLES are represented sequentially, from lowest to highest address.

RECORD VARIABLES are analogous to array variables.

Here are the variables in sequence from top-of-stack to bottom-of-stack:

```
                              HIGH           ADDRESS
                             |___________________________|
A,B:INTEGER;                 |________A(HIGH)____________|
                             |_________A(LOW)____________|
                             |________B(HIGH)____________|
                             |_________B(LOW)____________|
C :REAL;                     |________C( EXP)____________|
                             |______C(MAN.HIGH)__________|
                             |______C(MAN.__1_)__________|
                             |______C(MAN.__3_)__________|
                             |______C(MAN.__4_)__________|
                             |______C(MAN.__5_)__________|
                             |______C(MAN._LOW)__________|
D,E:ARRAY[1..3] OF CHAR;     |_________D[3]______________|
                             |_________D[2]______________|
                             |_________D[1]______________|
                             |_________E[3]______________|
                             |_________E[2]______________|
                             |_________E[1]______________|
F   :RECORD                  |_____K(248..255)___________|
G   :BOOLEAN;                |_____K(240..247)___________|
H   :BYTE;                   |_____K(  ...    )__________|
 CASE I:INTEGER;             |_K(8..15) / J(HIGH)________|
     1:(J:STRING);           |_K(0..7)  / J(LOW) ________|
     2:(K:SET OF 0..255)     |______I(HIGH)______________|
    END;                     |______I( LOW)______________|
                             |_________H_________________|
                             |_________G_________________|
                             | LOWEST ADDRESS            |
```

PROCEDURE/FUNCTION descriptions take up 6 (7) bytes, and are set on the stack with every procedure/function call. These 6 (7) bytes represent:

Dynamic link (2 bytes)
Return address(2 bytes)
Static link (2 bytes)
(segment nr. (1 byte))

System-defined runtime errors are as follows:

```
 0 = OK                       (I/O error 0)
 1 = NA
 2 = IL.INPUT
 3 = NA
 4 = OUT OF RNG.
 5 = NOT EXQ.
 6 = NUM.OV.
 7 = B.SUBS.
 8 = IL.QUANT.
 9 = STK.OV.
10 = ZERO-DIV
11 = IL.DVC.
12 = FLOPPY-                 (I/O ERROR  1)
13 = NOT OPEN                (I/O ERROR  2)
14 = NOT CLO.                (I/O ERROR  3)
15 = BUF.OV                  (I/O ERROR  4)
16 = DIR.OV.                 (I/O ERROR  5)
17 = NOT FND.                (I/O ERROR  6)
18 = DSC.OV.                 (I/O ERROR  7)
19 = DSC.MISM.               (I/O ERROR  8)
20 = IL.FILE OPR.            (I/O ERROR  9)
21 = AFTER EOF               (I/O ERROR 10)
22 = IEE -                   (I/O ERROR 11)
```

7.2 MEMORY LAYOUT AND ADDRESSES

SUPER Pascal uses the following addresses in the 64:

\$0028..\$0029	Start-of-stack pointer
\$002A..\$002B	unused reserve pointer
\$002C..\$002D	base-pointer
\$002E..\$002F	top-of-stack pointer (STKPOI)
\$0030..\$0031	pointer for current heap
\$0032..\$0039	diverse pointers (usable in assembler routines)
\$003A..\$004F	Fetch routine for "P-machine"
\$0050..\$0066	sundry zero-page cells
\$0067..\$006F	assorted C-64 system registers
\$0100..\$0184	INPUT buffer
\$0185..\$01F9	6510 machine stack
\$01FA..\$01FF	RANDOM variable

\$0340..\$0348	Descriptor for 1st file buffer
\$0349..\$0351	Descriptor for 2nd file buffer
\$0352..\$035A	Descriptor for 3rd file buffer
\$035B	Error-trap flag
\$035C	I/O ERROR number
\$035D	Working disk drive
\$035E	"EXECUTE" flag
\$035F	temporary disk drive
\$0360	Warm flag
\$0361..\$0362	MAIN menu pointer
\$0363..\$0364	Start-of-program pointer
\$0365..\$036C	Filename for source transfer
\$036D	Transfer drive for PUT/GET sector
\$036E..\$036F	Sector number for PUT/GET sector
\$0370	INPUT device
\$0371..\$0372	INPUT secondary address
\$0373	OUTPUT device
\$0374..\$0375	OUTPUT secondary address
\$0376..\$0379	assorted uses
\$037A..\$03FF	OUTPUT buffer

`$0800.. ...`	Start of programming memory
`$BBFF`	End of regular free stack
`$BC00..$C1FF`	MAIN menu variable stack
`$C200..$C7FF`	MAIN menu
`$C800..$F0FF`	SUPER Pascal runtime packet
`$CA03`	JUMP for external printer routine
`$CA06`	JUMP on GET sector
	(variable transfer on Pascal stack:
	Drive number (high)
	Drive number (low)
	Sector number(high)
	Sector number(low)
	RAM address (high)
	RAM address (low))
`$CA09`	JUMP on PUT sector
	(variable transfer like GET sector)
`$CA0C`	JUMP on runtime error
	(error number put on Pascal stack)
`$CA0F`	JUMP to MAIN menu
	(JMP $C200)
`$CA12`	Indirect JUMP on program end
	(regular:MAIN menu (JMP($0361)))
`$CA15`	Indirect JUMP to program
	(regular at $0800 (JMP ($0363)))
`$F300..$F6FF`	1st file buffer
`$F700..$FAFF`	2nd file buffer
`$FB00..$FEFF`	3rd file buffer
`$FFFA..$FFFF`	Machine vectors

7.3 DISKETTE ORGANIZATION

Diskettes are laid out in Pascal-DOS, i.e., 320 sectors (0..319), with each sector totaling 512 bytes. Data is transferred in this DOS by the GET-sector and PUT-sector routines ($CA06 and $CA09 respectively). Each sector in Pascal-DOS is double the size of a normal DOS sector (256 bytes); the changed DOS cuts the number of available disk blocks from 683 to 640, with the remaining 43 blocks unused by SUPER Pascal. The blocks are arranged as follows:

```
Track   1 - 17  /   Sector 20
Track  18       /   Sector 0, 1, 9, 10, 18
Track  19 - 24  /   Sector 18
Track  25 - 30  /      -
Track  31 - 34  /   Sector 16
Track  35       /   Sector 6 - 16
```

The blocks T1/S20 and T2/S20 contain the loader software for changing the DOS in SUPER Pascal. T18/S0 and T18/S1 hold the directory and BAM in regular DOS, while T18/S9, S10, S18, as well as T17/S20, T16/S20, T15/S20 and T14/S20 store the SUPER Pascal boot software.

The 320 sectors of a Pascal diskette aren't read individually; rather, in clusters of eight (blocks). Such a block comprises 8 X 512 bytes = 4096 bytes or 4k. This block-wise arrangement of sectors gives you a total of 40 blocks per diskette, which increases to 80 blocks when two drives are used in concert. The first block of every diskette (#0) is reserved for internal use (contains #255). Block 0 of sector 0 is set aside for the Pascal DOS directory; this directory is arranged schematically. Sectors 1..5 of block 0 are used for storing advice (additional information). The remaining sectors (6 and 7) are free.

GET-sector and PUT-sector (mentioned previously) allow access to all 320 sectors. With the help of these routines, you can reserve blocks for your own file- and diskette management, or data handling; you can also handle program control of the directory. Just use these routines as USER functions:

```
USERFUNC GETSECTOR(DRIVE,SECTOR,RAMPOINTER:INTEGER)
:BOOLEAN;
```

and

```
USERFUNC PUTSECTOR(DRIVE,SECTOR,RAMPOINTER:INTEGER)
:BOOLEAN;
```

and use SETADR to get the desired address. Calling the function transfers the disk drive number, sector and RAM pointer to the specified memory range. If you've declared the memory range as a variable, you'll have to give the function as parameters of the variable address (LOCALITY). The return value of the function is FALSE for bad execution, and TRUE if everything runs correctly.

The directory is accessed in SUPER Pascal in a similar manner; the directory is loaded into an appropriate variable range. This declaration has the following design:

```
 (START (top end) OF DIRECTORY)

EQUALIZE   :BYTE;
WORKBLOCK  :BYTE;
BLOCKTABLE :ARRAY[0..79] OF BYTE;
LASTBYTE   :ARRAY[0..37] OF BYTE;
STARTBLOCK :ARRAY[0..37] OF BYTE;
FIXFLAG    :SET OF 0..37;
WORKNAME   :ALFA;
TITLETABLE :ARRAY[0..37] OF ALFA;
DISCNAME   :ALFA;
DISCNUMBER :BYTE;
DISCSIZE   :BYTE;
 (END (bottom end) OF DIRECTORY)
```

These variable declarations take up exactly 512 bytes (the sector with logical number 0):

```
Address  0        ($000)         Diskette size (0 or 1)
  "       1       ($001)         Diskette number(0 - 1)
  "       2..9    ($002..$009)   Diskette name  (ALFA)
  "      10..313  ($00A..$139)   up to 38 filenames
                                 (ALFA)
  "     313..321  ($13A..$141)   temp. work name (ALFA)
  "     321..353  ($142..$161)   32 * 8 bits, first 38
                                 with  LOCK flag
  "     354..391  ($162..$187)   38 * 1 byte in
                                 position as EOF in
                                 last "1541" block
  "     392..429  ($188..$1AD)   38 * startblock in
                                 filename order
  "     430..509  ($1AE..$1FD)   80 * 1 byte for
                                 blocktable
  "     510       ($1FE)         temporary work block
  "     511       ($1FF)         fillbyte
```

NOTE:
Try out program control via directory with a scratch disk FIRST! Rebuilding a directory from scratch is rough work -- make sure that your variable declarations work out properly.

8.0 PROGRAM EXAMPLES AND GRAPHIC EXTENSIONS

8.1 THE EDITOR PROGRAM

The complete Editor program is listed here as a demonstration program (the Super Pascal Editor itself). You may have ideas on changing the program to suit your own needs. AUTO LINE MODE offers machine-code-like programming in Pascal.

```
                {PASCAL - TEXT - EDITOR}

PROGRAM EDITOR;

LABEL 99;

CONST BUFFER   =$F300;    KEY_CNT   =$C6;
      KEY_BUF  =$0277;    MAXLW_NR  =1;
      CRT_DVC  =0;        BCSP      =CHR($9D);
      CRSRUP   =CHR($91);CRTN       =CHR($D);
      SCRNLENG =80;       LWTEMP    =$035F;
      WARMFLG  =$360;     ADR_EXPO  =$0361;
      ADR_PRPO =$363;     ADR_COMM  =$0365;
      MAIN_JMP =$CA12;

     HEAD      ='* C=64 SOURCE-EDITOR  5.3 *';
     ILL_LINE  ='ILLEG. LINE#';
     NOTXT_FL  ='NO TEXT-FILE';
     SURE_NSS  ='SURE NOT SAVING THE SOURCE';
     EX_N_SUC  ='EXECUTION NOT SUCCESSFUL!';
     ILL_SYN   ='ILLEG. SYNTAX';
     RAM_OVER  ='RAM OVERFLOW';
     TITLE_ND  ='TITLE UNDEFINED';
     ILL_TITLE ='ILLEG.TITLE';
     ILL_INPUT ='ILLEG. INPUT';
     TO_       ='TO:';
     L_LEN_EX  ='LINELENGTH EXCEEDED IN LINE:';
     SURE_D_S  ='SURE TO DELETE ALL THE  SOURCE';
     HELP      ='HELP FOR:';
     BYTE_FREE ='0 BYTES FREE!';
     ONLY_ENT  =' PLEASE ONLY ENTER:';
     DRV_MAP   ='DRIVE(MAP)';
     CONFIRM   ='CONFIRM "';
     COM_IGN   ='COMMAND IGNORED!';

TYPE REF       = ^ITEM;
     ITEM         = RECORD
                       NR:INTEGER;
                       NX:REF;
                       ST:STRING;
                     END;
     BUFFSIZE  = ARRAY [0..511] OF BYTE
     INARRY    = ARRAY [0..PRED(SCRNLENG)] OF CHAR;
VAR  SOURCE                               :TEXT;
     LOADDAT                         :FILE OF BUFFSIZE;
     LINE,TRNSLINE,TPMLINE,FIRST           :REF;
     FROM,TIL,HNTR,NUM,AUTO_NUM,DRIVE     :INTEGER;
     SPARE                               :INARRY;
     CH                                   :CHAR;
     TITLE,SEEKSTR                        :STRING;
     NOT_DEF,SAVED,AUTO_FLAG              :BOOLEAN;
     BEGINHEAP,LFDHEAP,ADRPOI             :^INTEGER;
     COMMON                               :^ALFA;
     NUMBER,LETTER                    :SET OF '0'..'9';

XTRNFUNC MAP_EXT:BOOLEAN;

FUNCTION COMPARE(SUSTR,TESTR,STRING;
 STRTPOS:BYTE):BYTE;
ASSEMBLE;
;
;***************************

;*  SEARCH -- ROUTINE        *

;***************************
```

```
;

POI        .DL STKPOI+4
HBAS       .DL STKPOI+6
TEMP       .DL STKPOI+8
;
START      LDY #4
LOOP       LDA (STKPOI),Y
           STA POI-1,Y
           DEY
           BNE LOOP
           LDA (HBAS),Y
           STA *TEMP
           SEC
           LDA (POI),Y
           SBC *TEMP
           BCC EXIT
           SBC (STKPOI),Y
           BCC EXIT
           STA *TEMP+1
           CLC
           LDA (STKPOI),Y
           TAX
           ADC *POI
           STA *POI
           BCC LOOP1
           INC *POI+1
LOOP1      LDY *TEMP
           INX
LOOP2      LDA (POI),Y
           CMP (HBAS),Y
           BNE INCTEST
           DEY
           BNE LOOP2
           BEQ EXIT
INCTEST    INC *POI
           BNE INCTEST1
           INC *POI+1
```

```
INCTEST1  DEC *TEMP+1
          BPL LOOP1
          LDX #0
EXIT      TXA
          LDY #5
          STA (STKPOI),Y
          TYA
          CLC
          ADC *STKPOI
          STA *STKPOI
          BCC EXIT1
          INC *STKPOI+1
EXIT1     RTS
;
          .EN

PROCEDURE JUMPMAIN;ASSEMBLE;
          JMP MAIN_JMP
          .EN

PROCEDURE STOP(MESSAGE:STRING);
     BEGIN
      WRITE(MESSAGE,'!',' ');
      WRITELN(EX_N_SUC);AUTO_FLAG:=FALSE;
      GOTO 99
     END;

PROCEDURE SYN_STOP;
     BEGIN STOP(ILL_SYN) END;

PROCEDURE OV_STOP;
     BEGIN STOP(RAM_OVER) END;

PROCEDURE TEST_SURE(MSG:STRING);
     BEGIN
      READLN;WRITE(MSG,'? Y/N',BCSP);
      READ(CH);WRITELN;
      IF CH<>'Y' THEN BEGIN WRITELN(COM_IGN);
        GOTO 99 END
     END;
```

```
PROCEDURE TEST_FOR_SAVE;
     BEGIN
      IF NOT SAVED THEN TEST_SURE(SURE_NSS)
     END;

PROCEDURE WAIT_BRK;

PROCEDURE BREAK;
     BEGIN
      IF EOF THEN BEGIN READLN;OUTDVC(CRT_DVC,0);
       GOTO 99 END
     END;

  BEGIN
     BREAK;
     IF ANYKEY THEN
      IF GETKEY=' ' THEN
       REPEAT
          WHILE NOT ANYKEY DO BREAK
       UNTIL GETKEY=' '
  END;

PROCEDURE IGN_SPACE;
 BEGIN WHILE CH=' ' DO READ(CH) END;

PROCEDURE GETCH;
 BEGIN READ(CH);IGN_SPACE END;

PROCEDURE TEST_SYNTAX;
 BEGIN
  IF EOLN THEN SYN_STOP;GETCH;
  IF (CH<>':') OR EOLN THEN SYN_STOP
 END;

PROCEDURE SET_LAST;
 BEGIN LINE^.NR:=MAXINT
  LINE^.NX:=NIL;MARK(LFDHEAP) END;
```

```
PROCEDURE CLEAR;
 BEGIN
  RELEASE(BEGINHEAP);NEW(LINE);SET_LAST;
   FIRST:=LINE;
  SAVED :=TRUE
 END;

PROCEDURE GET_NUM(VAR LN_NR:INTEGER);
 BEGIN
  IF NOT (CH IN NUMBER) THEN SYN_STOP;
  LN_NR:=0;
  WHILE CH IN NUMBER DO
   BEGIN
    IF LN_NR >3275 THEN STOP(ILL_LINE);
    LN_NR:=10*LN_NR - 48 + ORD(CH);
    IF NOT EOLN AND (INPUT^ IN NUMBER) THEN
     READ(CH)
    ELSE CH:=' '
   END
  END;

PROCEDURE GET_SECND(TESTCH:CHAR);
 BEGIN
  GETCH;
  IF CH<>TESTCH THEN SYN_STOP;
  IF NOT EOLN THEN BEGIN GETCH; GET_NUM(TIL) END
 END;

PROCEDURE FROM_TIL;
 BEGIN
  FROM :=0;TIL:=PRED(MAXINT);
  IF NOT EOLN THEN
   BEGIN
    GETCH;
    IF CH='-' THEN
     BEGIN
      IF EOLN THEN SYN_STOP;
      GETCH;GET_NUM(TIL)
     END
    ELSE
     BEGIN
      GET_NUM(FROM);
      IF NOT EOLN THEN GET_SECND('-') ELSE
       TIL:=FROM
     END
    END
   END;

PROCEDURE GET_TITLE(FOR_GET:BOOLEAN);
 BEGIN
  TEST_SYNTAX;
   IF INPUT^='*' THEN
    BEGIN
     IF NOT_DEF THEN STOP(TITLE_ND);
     IF FOR_GET THEN TEST_FOR_SAVE
    END
   ELSE
    BEGIN
     IF NOT(INPUT^ IN LETTER) THEN
     STOP(ILL_TITLE);
    READ(TITLE);
    IF FOR_GET THEN TEST_FOR_SAVE;
    NOT_DEF:=FALSE;COMMON^:=TITLE
   END
  END;

PROCEDURE RENUMBER;
 BEGIN
  NUM:=1000; LINE:=FIRST;
  WHILE LINE^.NX<>NIL DO
   BEGIN LINE^.NR:=NUM;NUM:=NUM+5;
    LINE:=LINE^.NX END
 END;

PROCEDURE PREPARE;
 BEGIN
  SETDRV(DRIVE);NAME(SOURCE,COMMON^);
  MEM[LWTEMP]:=LOW(DRIVE)
 END;
```

```
PROCEDURE SAV_SRCE(FOR_PUT:BOOLEAN);
 BEGIN
  GET_TITLE(FALSE);
  READLN;
  WRITE(CONFIRM,COMMON^,',',DRIVE,'"? N/Y',BCSP);
  READ(CH);WRITELN;
  IF CH<>'Y' THEN BEGIN WRITELN(COM_IGN);
   GOTO 99 END;
  PREPARE;
  IF FOR_PUT THEN REWRITE(SOURCE)
  ELSE
   BEGIN
    RESET(SOURCE);
    WHILE NOT EOF(SOURCE) DO READLN(SOURCE)
   END;
  LINE:=FIRST;
  WHILE LINE^.NX<>NIL DO
   BEGIN WRITELN(SOURCE,LINE^.ST);
    LINE:=LINE^.NX END;
   CLOSE(SOURCE);SAVED:=TRUE
  END;

PROCEDURE LOAD_SRCE;
 VAR CNT:INTEGER;
 BEGIN
  PREPARE;
  RESET(SOURCE);CNT:=0;
  WHILE(SOURCE^<>CRTN) AND (CNT<=80) AND NOT
   EOF(SOURCE) DO
    BEGIN
     CNT:=SUCC(CNT);
     IF (CNT>80) OR (SOURCE^<' ') THEN
      BEGIN CLOSE(SOURCE);STOP(NOTXT_FL) END;
     GET(SOURCE)
    END;
 IF SOURCE^<>CRTN THEN
  BEGIN CLOSE(SOURCE);STOP(NOTXT_FL)  END;
 CLOSE(SOURCE);RESET(SOURCE);
 WHILE NOT EOF(SOURCE) DO
```

```
  BEGIN
   LINE^.NR:=NUM;READLN(SOURCE,LINE^.ST);
    NUM:=NUM +5;
   IF FREE<=3 THEN
    BEGIN SET_LAST;CLOSE(SOURCE);OV_STOP END;
   NEW(TMPLINE);LINE^.NX:=TMPLINE;LINE:=TMPLINE
  END;
 SET_LAST; CLOSE (SOURCE)
END;

PROCEDURE SEEK(LN_NR:INTEGER);
 BEGIN
  LINE:=FIRST;WHILE LINE^.NR<LN_NR
   DO LINE:=LINE^.NX
 END;

PROCEDURE CHANGE;

 VAR
  OLD_LINE;NEW_LINE:INARRY;
  POSITION:BYTE;
  SEEKLEN,OLDLEN,NEWLEN,FSTLEN,CMPLEN,DELTA,
   FLOT:INTEGER;
  SPEC:RECORD CASE INTEGER OF
        0:(HEAP:^INTEGER);
        1:(LENG:^BYTE);
        2:(ADRS:INTEGER)
       END;
 BEGIN
 TEST_SYNTAX;READLN(SEEKSTR);SEEKLEN:=LEN(SEEKSTR);
 MARK(SPEC.HEAP);
 WRITE(TO_);READ(TITLE);WRITELN;RESET(INPUT);
 IF EOLN THEN FSTLEN:=0 ELSE FSTLEN:=LEN(TITLE);
 DELTA:=SEEKLEN-FSTLEN;SPARE:=TITLE;
 RELEASE(SPEC.HEAP);LINE:=FIRST;POSITION:=#0
 WHILE (LINE^.NX<>NIL) AND NOT EOF DO
  WITH LINE^ DO
   BEGIN
    CMPLEN:=ORD(COMPARE(SEEKSTR,ST,POSITION));
    IF CMPLEN<>0 THEN
```

```
     BEGIN
      OLDLEN:=LEN(ST);
      IF (OLDLEN-DELTA)>(SCRNLENG-4) THEN
       BEGIN WRITELN(L_LEN_EX);WRITELN(NR,ST);
        GOTO 99 END;
      OLD_LINE:=ST;NEW_LINE:=OLD_LINE;
     FOR FLOT:= 0 TO PRED(FSTLEN) DO
      NEW_LINE[PRED(CMPLEN+FLOT)]:=SPARE[FLOT];
     FOR FLOT:= PRED(CMPLEN+SEEKLEN) TO
       PRED(OLDLEN) DO
       NEW_LINE[FLOT-DELTA]:=OLD_LINE[FLOT];
     ST:=NEW_LINE;SPEC.LENG^:=LOW(OLDLEN-DELTA);
     SPEC.ADRS:=HXS(SPEC.ADRS,SUCC(OLDLEN-DELTA));
     RELEASE(SPEC.HEAP);
      POSITIN:=PRED(LOW(CMPLEN+FSTLEN);
     IF FREE<=3 THEN OV_STOP
    END
   ELSE BEGIN LINE:=NX;POSITION:=#0 END
  END
 END;

PROCEDURE COMMANDS;

BEGIN
 WRITELN('COMMANDS = ...');
 WRITELN('A:(PPENDSRC) L(IST)       Q(UIT)');
 WRITELN('C:(HANGE)     M(AP/DRIVE)  R(ENUMBER)');
 WRITELN('D(ELETE)      N(UMBERING)  S(HIFTLINE)');
 WRITELN('F:(IND)       O(UTPUTDVC)  U:(PDATESRC)');
 WRITELN('G:(ETSOURCE) P:(UTSOURCE) V(ACANCY)');
 WRITELN('H(ELP)');WRITELN
END;
```

```
          {MAIN PROGRAM}

BEGIN

 IF MEM[WARMFLG]=#1 THEN

  BEGIN WRITELN;RELEASE(LFDHEAP) END

 ELSE

 BEGIN
  NUMBER:=['0'..'9'];LETTER:=['A'..'Z'];
  ALLOCATE(COMMON,ADR_COMM);
  SETADR(MAP_EXT,BUFFER1);
  AUTO_FLAG:=FALSE;NEW(TRNSLINE);
  MARK(BEGINHEAP);MARK(LFDHEAP);CLEAR;
  ALLOCATE(ADRPOI,ADR_PRPO);
  FROM:=ADRPOI^;
  ALLOCATE(ADRPOI,ADR_EXPO);
  TDRPOI^:=FROM;
  NOT_DEF:=MEM[WARMFLG]<>#2;
  IF NOT NOT_DEF THEN
   BEGIN
    DRIVE:=ORD(MEM[LWTEMP]);NUM:=1000;
    LOAD_SRCE
   END
  ELSE DRIVE:=0;
  WRITELN;
  WRITELN(HEAD:34);
  WRITELN;COMMANDS;
  MEM[WARMFLG]:=#1;SAVED:=TRUE
 END;
```

```
REPEAT

 IF AUTO_FLAG THEN
  BEGIN
   WRITE(AUTO_NUM,' ':NUM);
   READLN(CH);WRITELN;
   MEM[KEY_BUF]:=LOW(CRSRUP);
    MEM[SUCC(KEY_BUF)]:=LOW(CRTN);
   MEM[KEY_CNT]:=#2;
   AUTO_NUM:=AUTO_NUM+5;
   IF AUTO_NUM>=32750 THEN STOP(ILL_LINE)
  END;

 READ(CH);IGN_SPACE;WRITELN;

 IF CH IN NUMBER THEN    {LINE NUMBER INPUT}
  BEGIN
   GET_NUM(NUM);IF NUM<>AUTO_NUM-5 THEN
     AUTO_FLAG:=FALSE;
   SEEK(NUM);
  IF LINE^.NR=NUM THEN
   IF EOLN THEN
    IF NOT AUTO_FLAG THEN LINE^:=LINE^.NX^
    ELSE AUTO_FLAG:=FALSE
   ELSE READ(LINE^.ST)
  ELSE
   IF NOT EOLN THEN
    BEGIN
     NEW(TMPLINE);
     TMPLINE^:=LINE^;LINE^.NR:=NUM;
     LINE^.NX:=TMPLINE;READ(LINE^.ST)
    END
   ELSE AUTO_FLAG:=FALSE;
  IF AUTO_FLAG THEN
   BEGIN
    SPARE:=LINE^.ST;NUM:=0;
    WHILE SPARE[NUM]=' ' DO NUM :=SUCC(NUM)
   END;
  SAVED:=FALSE;
  IF FREE<=3 THEN OV_STOP;
  MARK(LFDHEAP)
 END
 ELSE                         {COMMAND INPUT}
  BEGIN
   AUTO_FLAG:=FALSE;CASE CH OF

     'A':BEGIN {APPEND}
          RENUMBER;GET_TITLE(FALSE);
            NOT_DEF:=TRUE;LOAD_SRCE
         END;

     'C':BEGIN {CHANGE}
        CHANGE;SAVED:=FALSE
       END;

     'D':BEGIN {DELETE}
          FROM_TIL;
           IF (FROM=0) AND (TIL=PRED(MAXINT)) THEN
            BEGIN
             TEST_SURE(SURE_D_S);
             CLEAR
            END
           ELSE
            IF FROM<=TIL THEN
             BEGIN
              SAVED:=FALSE;
              SEEK(FROM);TMPLINE:=LINE;
              SEEK(SUCC(TIL));TMPLINE#:=LINE^
             END
          END;

    'F':BEGIN {FIND}
         TEST_SYNTAX;
         READ(SEEKSTR);LINE:=FIRST;
         WHILE LINE^.NX<>NIL DO
          BEGIN
           IF COMPARE(SEEKSTR,LINE^.ST,#0)<>#0 THEN
            WRITELN(LINE#.NR,LINE#.ST);
       LINE:=LINE^.NX;
       WAIT_BRK
      END
    END;

 'G':BEGIN {GET}
      GET_TITLE(TRUE);
      CLEAR;NUM:=1000;
      LOAD_SRCE
     END;

 'H':BEGIN {HELP}
      WRITELN(HELP,HEAD);
      WRITELN;COMMANDS
     END;

 'L':BEGIN {LIST}
      FROM_TIL;SEEK(FROM);
      WHILE LINE^.NR<=TIL DO
       BEGIN
        WRITELN(LINE^.NR,LINE^.ST);
        LINE:=LINE^.NX;
        WAIT_BRK
       END
      END;

 'M':BEGIN {MAP}
     READLN;
     WRITE(DRV_MAP,' = ',DRIVE,BCSP);
     IF DRIVE>9 THEN WRITE(BCSP);
     READ(FROM);WRITELN;
     IF (FROM<0) OR (FROM>MAXLW_NR) THEN
       STOP(ILL_INPUT);
     DRIVE:=FROM;
     SETDRV(0);RESET(LOADDAT);
     GET(LOADDAT);GET(LOADDAT);GET(LOADDAT);
     CLOSE(LOADDAT);
     SETDRV(DRIVE);
     IF NOT MAP_EXT THEN
      BEGIN DRIVE:=0;SETDRV(0) END
```

```
   END;

'N':BEGIN {AUTO-NUMBERING}
    IF EOLN THEN
     BEGIN
      AUTO_NUM:=1000;LINE:=FIRST;
      WHILE LINE^.NX<>NIL DO
       BEGIN AUTO_NUM:=LINE^.NR+5;
         LINE:=LINE^.NX END
     END
    ELSE
     BEGIN
      GETCH;GET_NUM(AUTO_NUM);
      IF NOT EOLN THEN SYN_STOP
     END;
    AUTO_FLAG:=TRUE;NUM:=0
   END;

'O':IF EOLN THEN OUTDVC(CRT_DVC,0)
        {SET OUTPUT DVC}
     ELSE
      BEGIN
       GETCH;GET_NUM(FROM);TIL:=0;
       IF NOT EOLN THEN GET_SECND(',');
       IF NOT ((FROM IN [0,4..7]) AND
         (TIL<=15)) THEN
        STOP(ILL_INPUT);
       OUTDVC(FROM,TIL)
      END;

'P':SAV_SRCE(TRUE); {PUT}

'Q':BEGIN {QUIT}
     TEST_FOR_SAVE;
     OUTDVC(CRT_DVC,0);
     JUMPMAIN
    END;

'R':RENUMBER; {RENUMBER}
```

```
     'S':BEGIN {SHIFTLINE}
         FROM_TIL;TEST_SYNTAX;
         GETCH;GET_NUM(HNTR);
         IF(HNTR>=FROM) AND (HNTR<=TIL) THEN
          STOP(ILL_INPUT);
         SEEK(SUCC(HNTR));
         TRNSLINE^:=LINE^;TMPLINE:=LINE;
         SEEK(FROM);
         TMPLINE^:=LINE^;TMPLINE:=LINE;
         SEEK(SUCC(TIL));
         TMPLINE^:=LINE^;LINE^:=TRNSLINE^;
         RENUMBER;SAVED:=FALSE;
        END;

     'U':BEGIN {UPDATE}
         NOT_DEF:=TRUE;SAV_SRCE(FALSE)
        END;

     'V':WRITELN(FREE*25-77+(FREE*6)
         DIV 10,BYTE_FREE)
          ELSE
           BEGIN
            WRITELN(COM_IGN,ONLY_ENT);
            COMMANDS
           END
          END
          END;
     99: READLN

       UNTIL FALSE

END.
```

8.2 "RPN" PROGRAM

Here is the complete program listing for RPN, which you'll find on your SUPER Pascal diskette in both compiled form and sourcecode. RPN simulates the functions of an RPN pocket calculator; some runtime errors will occur whatever shape the program is in, since some transcendental math functions can cause such errors. The modular structure of this program allows for easy modification.

```
      (*****************************************)
      (*                                       *)
      (*               R P N                   *)
      (*              -------                  *)
      (*    THIS PROGRAM SIMULATES THE         *)
      (*    FUNCTIONS OF A CALCULATOR          *)
      (*    WHICH USES REVERSE POLISH NOTATION *)
      (*    (RPN) (NOTE: ALL INPUT MUST        *)
      (*    CONCLUDE WITH <RETURN>             *)
      (*****************************************)

PROGRAM RPN;

CONST MAXBEF    =79;      CUP          =CHR($91);
      WARMFLG  =$360;     ADR_PRPO     =$363;
      ADR_EXPO =$361;

VAR  REG                      :(X,Y,Z,T);
     STACK                    :ARRAY[X..T] OF REAL;
     LSTX,S1,KEYIN,ZW,QU      :REAL;
     FLOT,CON,PLACE,FIELD     :INTEGER;
     CX                       :CHAR;
     LSTRI                    :STRING;
     BEFARR                   :ARRAY[0..MAXBEF]OF CHAR;
     ADRPOI,HEAP              :^INTEGER;
```

```
PROCEDURE EXIT;

 ASSEMBLE;
       JMP $C200
        .EN

PROCEDURE ENTER;
 BEGIN
  FOR REG:=Z DOWNTO X DO
   STACK[SUCC(REG)]:=STACK[REG]
 END;

PROCEDURE CALC(RESULT:REAL;SINGLE:BOOLEAN);
 BEGIN
  LSTX:=STACK[X];STACK[X]:=RESULT;
   IF NOT SINGLE THEN
    FOR REG:=Y TO Z DO
     STACK[REG]:=STACK[SUCC(REG)]
 END;

PROCEDURE PRTSTK;
 BEGIN
  WRITELN(CUP,CUP,CUP,CUP,CUP,CUP);
  WRITELN('T = ':10,STACK[T]:FIELD:PLACE);
  WRITELN('Z = ':10,STACK[Z]:FIELD:PLACE);
  WRITELN('Y = ':10,STACK[Y]:FIELD:PLACE);
  WRITELN('X = ':10,STACK[X]:FIELD:PLACE);
  WRITELN;WRITELN(' ':39,CUP)
 END;

PROCEDURE COMANDS;
 BEGIN
  WRITELN;WRITELN;
  WRITELN('<<COMMANDS FOR "RPN" >>':32);
  WRITELN('=====================':29);
  WRITELN('A=ABSOLUTE   B=ROUND      C=COSINE');
  WRITELN('D=ROLL DOWN  E=EXP        F=FRAC');
  WRITELN('G=GETMEM     H=CLEAR X    I=INTEGER');
  WRITELN('K=RECIPROCAL L=LN         M=MEM');
  WRITELN('N=ENTER      O=OUTP.FORM. P=PI');
  WRITELN('Q=SQUARE      R=SQROOT      S=SINE');
  WRITELN('T=TANGENT     U=ROLL UP     V=SIGN');
  WRITELN('W=CH.SIGN     X=LAST X      Y=X CH Y');
  WRITELN('Z=RAND.NUM.   @=ARCTAN');WRITELN;
  WRITELN('RELATIONS: <,>,=');
  WRITELN('OPERATORS:+,-,*,/');
  WRITELN;WRITELN;WRITELN;WRITELN;WRITELN;
  WRITELN;WRITELN;WRITELN;WRITELN;WRITELN;
  WRITELN(CUP,CUP,CUP,CUP,CUP)
 END;

PROCEDURE JOB;
 BEGIN
  CASE CX OF
   'A':CALC(ABS(STACK[X]),TRUE);
   '@':CALC(ARCTAN(STACK[X]),TRUE);
   'B':CALC(ROUND(STACK[X]),TRUE);
   'C':CALC(COS(STACK[X]),TRUE);
   'D':BEGIN
        KEYIN:=STACK[X];
        FOR REG:=X TO Z DO
         STACK[REG]:=STACK[SUCC(REG)];
        STACK[T]:=KEYIN
       END;
   'E':CALC(EXP(STACK[X]),TRUE);
   'F':CALC(FRAC(STACK[X]),TRUE);
   'G':BEGIN ENTER;STACK[X]:=S1 END;
   'H':STACK[X]:=0.0;
   'I':CALC(TRUNC(STACK[X]),TRUE);
   'K':CALC(1/STACK[X],TRUE);
   'L':CALC(LN(STACK[X]),TRUE);
   'M':S1:=STACK[X];
   'N':ENTER;
   'O':BEGIN
        FLOT:=INT(STACK[X]);
        IF FLOT < 12 THEN
         IF FLOT > 0 THEN
          BEGIN
           WRITELN(CUP,CUP,CUP,CUP,CUP,' ':39);
           WRITELN(' ':39);WRITELN(' ':39);
          WRITELN(' ':39);WRITELN;
          PLACE:=-FLOT;
          FIELD:=ABS(INT(ROUND(100*FRAC(FLOT))))
         END
        END;
  'P':BEGIN ENTER;STACK[X]:=PI END;
  'Q':CALC(SQR(STACK[X]),TRUE);
  'R':CALC(SQRT(STACK[X],TRUE);
  'S':CALC(SIN(STACK[X],TRUE);
  'U':BEGIN
      KEYIN:=STACK[T];ENTER;
      STACK[X]:=KEYIN
     END;
  'V':CALC(SIGN(STACK[X],TRUE);
  'W':STACK[X]:=-STACK[X];
  'X':BEGIN ENTER;STACK[X]:=LSTX END;
  'Y':BEGIN
       KEYIN:=STACK[X];STACK[X]:=STACK[Y];
       STACK[Y]:=KEYIN
      END;
  'Z':BEGIN ENTER;STACK[X]:=RANDOM END;
  '<':CALC(ORD(STACK[Y]<STACK[X]),FALSE);
  '=':CALC(ORD(STACK[X]=STACK[Y]),FALSE);
  '>':CALC(ORD(STACK[X]=STACK[Y]),FALSE);
  '+':CALC(STACK[X]+STACK[Y],FALSE);
  '-':CALC(STACK[Y]-STACK[X],FALSE);
  '*':CALC(STACK[X]*STACK[Y],FALSE);
  '/':CALC(STACK[Y]/STACK[X],FALSE);
 END
END;
```

```
          (* ****************************** *)
          (* *** MAIN OF RPN         **** *)
          (* ****************************** *)

BEGIN

 IF MEM[WARMFLAG]=#0 THEN
  BEGIN
   MARK(HEAP);COMANDS;LSTRI:='SXC/';
   BEFARR:=LSTRI;MEM[WARMFLAG]:=#1;
   ALLOCATE(ADRPOI,ADR_PRPO);
   FLOT:=ADRPOI^;ALLOCATE(ADRPOI,ADR_EXPO);
   ADRPOI^:=FLOT;
  END
 ELSE
  BEGIN
   WRITELN('PRESS "SPACE" !':32,CUP);
   WHILE GETKEY<>' 'DO;
   WRITELN(' ':39,CUP);
   WRITELN(CUP,' ':39,CUP,CUP);
   WRITELN('':39,CUP)
  END;
 FOR REG := X TO T DO STACK[REG]:=0;
 S1:=0;FIELD:=0;PLACE:=-11;PRTSTK;
  WHILE NOT EOF DO
   BEGIN
    READ(CX);RESET(INPUT);
    WHILE(INPUT^=' ') AND NOT EOLN DO READ(CX);
    IF INPUT^ IN ['0'..'9'] THEN
     BEGIN
      READLN(KEYIN);ENTER;STACK[X]:=KEYIN
     END;
    ELSE
     BEGIN
      READ(CX);JOB;READLN
     END;
    PRTSTK;RELEASE(HEAP)
   END;
  EXIT
END.
```

8.3 THE GRAPHICS PACKET

You won't need the 64's graphic capabilities in normal use of SUPER Pascal. However, S_GRAPH ill let you perform high-resolution tasks in your own program routines. The routine is treated as a Pascal routine during compiling -- to install this routine into your own programs, just use the compiler command

&INCLUDE(S_*GRAPH)

S_*GRAPH is written in machine code for the sake of speed, but is clearly written to allow you to make your own changes. The HILBERT-CURVES program in Chapter 8.2.1 uses the routine, and shows a few changes that can be performed.

```
          (****************************)
          (*                          *)
          (* GRAPHICS PACKET FOR C64  *)
          (*                          *)
          (****************************)

PROCEDURE GRAPHIC

(COM:GRAPHICCOMMAND;VAL1,VAL2,VAL3,VAL4:INTEGER);

ASSEMBLE;

;
CPUPORT   .DL $0001      ;DEFINE MEMORY
CONFIGURATION
VIDCTR    .DL $D000      ;VIDEO CONTROLLER
BITMAP    .DL $2000      ;GRAPHIC SCREEN
COLRAM    .DL $0800      ;COLOR RAM
;
TMPMOD    .DL $FF01      ;DEFINE TEMPORARY
TMPPOI    .DL $FF02      ;MEMORY LOCATIONS
PLFLG     .DL $FF03
;
```

```
TMP        .DL STKPOI+4    ;DEFINE ZEROPAGE CELLS
XKOR       .DL STKPOI+6    ;IN SUPER PASCAL
YKOR       .DL STKPOI+8    ;VIA STACK POINTER
COLOR      .DL STKPOI+9
XKOR1      .DL STKPOI+34
ZW         .DL STKPOI+36
ZA         .DL STKPOI+37
MSK        .DL STKPOI+38
DIF0       .DL STKPOI+39
DIF1       .DL STKPOI+40
DIF2       .DL STKPOI+41
DIF3       .DL STKPOI+42
DIF4       .DL STKPOI+43
DIF5       .DL STKPOI+44
YKOR1      .DL STKPOI+45
;
START      LDA #1          ;I/O ON AND
           ORA *CPUPORT    ;PASCAL RAM OUT
           STA *CPUPORT
           LDY #8
           LDA (STKPOI),Y ;CALL GRAPHIC COMMAND FROM
           ASL A           ;STACK, AND USE AS POINTER
           TAX             ;IN JUMP TABLE
           LDA SPRGTAB,X
           STA *TMP
           LDA SPRGTAB+1,X
           STA *TMP+1
           JMP (TMP)       ;JUMP INDIRECTLY TO CALLED
                              ROUTINE
           ;
SPRGRTAB   .SA GRAPHIN     ;GRAPHIC SCREEN ON
           .SA GRAPHOUT    ;GRAPHIC SCREEN OFF
           .SA GCLEAR      ;CLEAR GRAPHIC SCREEN
           .SA COLCLEAR    ;CLEAR COLOR SCREEN
           .SA DOT_ON      ;SET DOT
           .SA DOT_OFF     ;CLEAR DOT
           .SA LINESET     ;DRAW LINE
           .SA LINECLR     ;CLEAR LINE
           .SA REVERS      ;REVERSE GRAPHIC SCREEN
           ;
```

```
GRAPHIN    LDA VIDCTR+17  ;SWITCH ON GRAPHIC SCREEN
           STA TMPMOD
           LDA VIDCTR+24
           STA TMPPOI
           LDA #$3B
           STA VIDCTR+17  ;BITMAP MODE
           LDA #$28
           STA VIDCTR+24  ;BITMAP AFTER $2000
           JMP EXIT
           ;
GRAPHOUT   LDA TMPMOD     ;GRAPHIC SCREEN OFF
           STA VIDCTR+17
           LDA TMPPOI
           STA VIDCTR+24
           ;
EXIT       LDA #$FC       ;PROGRAM EXIT
           AND *CPUPORT   ;PASCAL RAM SWITCHED ON
           STA *CPUPORT   ;AND I/O REGISTER OFF
           CLC            ;PASCAL STACK SET BACK A
           LDA #9         ;TOTAL OF 9 BYTES
           ADC *STKPOI    ;(1 BYTE + 4 INTEGER)
           STA *STKPOI
           BCC EXIT0
           INC *STKPOI+1
EXIT0      RTS            ;BACK TO PASCAL
           ;
           ;
GCLEAR     LDA #H,BITMAP  ;CLEAR GRAPHIC SCREEN
           STA *TMP+1
           LDY #L,BITMAP
           STY *TMP
           LDX #$20       ;# OF PAGES
           TYA
GCLEAR1    STA (TMP),Y
           INY
           BNE GCLEAR1
           INC *TMP+1
           DEX
           BNE GCLEAR1
           JMP EXIT
```

```
          ;
COLCLEAR  DEY              ;CLEAR COLOR SCREEN
          DEY
          LDA (STKPOI),Y ;LOWBYTE OF VAL1 FROM.
          ASL A            ;STACK AS SCREEN COLOR
          ASL A
          ASL A
          ASL A
          STA *COLOR
          DEY
          DEY
          LDA(STKPOI),Y   ;LOWBYTE OF VAL 2 FROM
          AND #$0F         ;STACK AS BORDER COLOR,
          ORA *COLOR       ;AND STORED WITH
          LDX #L,COLRAM    ;SCREENCOLOR
          STX *TMP
          LDX #H,COLRAM
          STX *TMP+1
          LDY #0
          LDX #3
          ;
COLCL0    STA (TMP),Y      ;SCREEN INFORMATION STORED
          DEY              ;IN COLOR RAM
          BNE COLCL0
          DEX
          BMI COLCL0
          INC *TMP+1
          BNE COLCL0
          LDY #$E8
          BNE COLCL0
COLCL1    STA (TMP),Y
          JMP EXIT
          ;
DOT_OFF   JSR SET0         ;UNSET DOT
          JMP EXIT
          ;
DOT_ON    JSR SET1         ;SET DOT
          JMP EXIT
          ;
```

```
SET0       LDX #$80        ;ROUTINE FOR SETTING OR
           .BY $2C         ;UNSETTING
SET1       LDX #0          ;DOT-POINTS
           STX PLFLG
           JSR TESTCOR     ;GET & TEST COORDINATES
           JSR HPOSN       ;CALC MEMORY POSITION
           JMP PLOT        ;DOT SET/CLEAR
           ;
PLOT       LDY #0          ;DRAW/CLEAR DOT IN
           LDA *MSK        ;POSITION CALCULATED
           BIT PLFLG
           BPL PLOT0
           EOR #$FF
           AND (TMP),Y
           .BY $2C
PLOT0      ORA (TMP),Y
           STA (TMP),Y
           RTS
           ;
TEST_X_Y   JSR TETCOR
           RTS
           ;
TESTCOR    DEY
           LDA (STKPOI),Y ;HIBYTE OF VAL1 OR VAL3
           STA *XKOR+1     ;(=X) CALLED FROM STACK
           DEY
           CMP #1          ;>1?
           BCC TEST1
           BNE IGNOR       ;IGNORE AND EXIT
           LDA #$3F
           CMP (STKPOI),Y ;>=320?
           BCC IGNOR       ;IGNORE AND EXIT
TEST1      LDA (STKPOI),Y ;LOWBYTE OF VALUES 1 OR 3
           STA *XKOR       ;CALLED AND STORED
           DEY
           LDA (STKPOI),Y ;HIGHBYTE OF VAL2 OR VAL4
           BNE IGNOR       ;(=Y) CALLED FR STACK
           DEY             ;<>0?:IGNORE AND EXIT
           LDA (STKPOI),Y ;LOWBYTE OF VALUES 2 AND 4
           CMP #200        ;CALLED FROM STACK; >=200?
```

```
          BCS IGNOR      ;IGNORE AND EXIT
          STA *YKOR      ;STORE Y-COORDINATES
          RTS
          ;
IGNOR     PLA            ;COMMAND EXECUTION
          PLA            ;FOR IGNORING ILLEGAL
          PLA            ;NUMBERS; SUBROUTINE-LEVEL
          PLA            ;CORRECTION
          JMP EXIT
          ;
HPOSN     AND #7         ;CALC MEMORY ADDRESSES
          STA *TMP
          LDA *XKOR+1
          STA *TMP+1
          LDA *YKOR
          LSR A
          LSR A
          LSR A
          TAX
          LDA *XKOR
          AND #$F8
          ;
          CLC
          ADC *TMP
          BCC HPOSN0
          INC *TMP+1
HPOSN0    CLC
          ADC LOWTAB,X
          STA *TMP
          LDA *TMP+1
          ADC HIGHTAB,X
          ADC #H,*BTIMAP
          STA *TMP+1
          LDA *XKOR
          AND #7
          TAX
          LDA BITTAB,X
          STA *MSK
          RTS
          ;
```

```
LINECLR    JSR SET0        ;CLEAR LINE
           JMP LINE0
LINESET    JSR SET1        ;DRAW LINE
LINE0      LDA *YKOR       ;FIRST COORDINATES
           STA *YKOR1
           LDA *XKOR
           STA *XKOR1
           LDA *XKOR+1
           STA *XKOR1+1
           LDY #4          ;SECOND SET OF COORDINATES
           JSR TESTCOR     ;CALLED, TESTED AND
           LDA *XKOR       ;UTILIZED
           LDX *XKOR+1
           LDY *YKOR
           PHA         ;DEPENDENT & INDEPENDENT
           LDA *XKOR1+1    ;COORDINATES DETERMINED
           LSR A           ;INDEPENDENTS INCREMENTED
           LDA *XKOR1
           ROR A           ;ENDPOINTS CLEARED/SET
           LSR A
           LSR A
           ;

           ;                    ;
           STA *ZW
           PLA
           PHA
           SEC
           SBC XKOR1
           PHA
           TXA
           SBC XKOR1+1
           STA *DIF3
           BCS LINE3
           PLA
           EOR #$FF
           ADC #1
           PHA
           LDA #0
           SBC *DIF3
```

```
LINE3       STA *DIF1
            STA *DIF5
            PLA
            STA *DIF0
            STA *DIF4
            PLA
            STA *XKOR1
            STX *XKOR1+1
            TYA
            CLC
            SBC *YKOR1
            BCC LINE4
            EOR #$FF
            ADC #$FE
LINE4       STA *DIF2
            STY *YKOR1
            ROR *DIF3
            SEC
            SBC *DIF0
            TAX
            LDA #$FF
            SBC *DIF1
            STA *ZA
            LDY *ZW
            BCS LINE5
LINE1       ASL A
            JSR R_L
            SEC
LINE5       LDA *DIF4
            ADC *DIF2
            STA *DIF4
            LDA *DIF5
            SBC #0
LINE2       STA *DIF5
            STY *ZW
            JSR PLOT
            INX
            BNE LINE6
            JMP EXIT
```

```
LINE6       LDA *DIF3
            BCS LINE1
            JSR U_O
            CLC
            LDA *DIF4
            ADC *DIF0
            STA *DIF4
            LDA *DIF5
            ADC *DIF1
            BVC LINE2
SETCELL     INC *TMP         ;SET UP 8 X 8 MATRIX
            BNE SETCELL0     ;(BOTTOM)
            INC *TMP+1
SETCELL0    LDA *TMP
            AND #7
            BNE SETCELL2
            INC *TMP+1
            LDA #$38
SETCELL1    CLC
            ADC *TMP
            STA *TMP
            BCC SETCELL2
            INC *TMP+1
SETCELL2    RTS
            ;
U_O         BMI SETCELL        ;LEAVE 8 X 8 FIELD
            ;
TOPCELL     LDA *TMP           ;SET UP TOP OF 8 X 8
            BNE TOPCELL0       ;MATRIX
            DEC *TMP+1
TOPCELL0    DEC *TMP
            AND #7
            BNE SETCELL2
            DEC *TMP+1
            DEC *TMP+1
            LDA #$C8
            BNE SETCELL1
            ;
```

```
RGHCELL    LSR *MSK       ;SET UP RIGHT SIDE OF
           BCC RGHCELL2   ;8 X 8 FIELD
           ROR *MSK
           INY
           LDA #8
RGHCELL1   CLC
           ADC *TMP
           STA *TMP
           BCC RGHCELL2
           INC *TMP+1
RGHCELL2   RTS
           ;
R_L        BPL RGHCELL     ;LEAVE 8 X 8 MATRIX
           ;
LINKS      ASK *MSK       ;DESIGN LEFT SIDE OF 8 X 8
           BCC RGHCELL2   ;FIELD
           ROL *MSK
           DEC *TMP+1
           LDA #$F8
           BNE RGHCELL1
           ;
REVERS     LDY #0         ;INVERSE VIDEO SCREEN
           LDA #H,BITMAP
           STY *TMP
           STA *TMP+1
           LDX #$20
REVERS1    LDA (TMP),Y
           EOR #$FF
           STA (TMP),Y
           INY
           BNE REVERS1
           INC *TMP+1
           DEX
           BNE REVERS1
           RTS
           ;
```

```
LOWTAB     .BY $00 $40 $80 $C0  ;LOWBYTE
           .BY $00 $40 $80 $C0  ;MULTIPLICATION
           .BY $00 $40 $80 $C0  ;TABLE
           .BY $00 $40 $80 $C0
           .BY $00 $40 $80 $C0
           .BY $00 $40 $80 $C0 $00
           ;
HIGHTAB    .BY $00 $01 $02 $03  ;HIGHBYTE
           .BY $05 $06 $07 $08  ;MULTIPLICATION
           .BY $0A $0B $0C $0D  ;TABLE
           .BY $0F $10 $11 $12
           .BY $14 $15 $16 $17
           .BY $19 $1A $1B $1C $1E
           ;
BITTAB     .BY 128 64 32 16 8 4 2 1 ;BIT TABLE FOR
                                    ;MASK BITS
           .EN

PROCEDURE GRAPHIN (*GRAPHIC SCREEN ON*);
 BEGIN GRAPHIC(GRIN,0,0,0,0) END;

PROCEDURE GRAPHOUT (*GRAPHIC SCREEN OFF*);
 BEGIN GRAPHIC(GROT,0,0,0,0) END;

PROCEDURE GRAPHCLR (*GRAPHIC SCREEN CLEAR*);
 BEGIN GRAPHIC(GCLR,0,0,0,0) END;

PROCEDURE COLCLR (*SCOLR,BCOLR:INTEGER*);
 BEGIN GRAPHIC(CCLR,SCLOR,BCLOR,0,0) END;

PROCEDURE DOT (X,Y:INTEGER);
 BEGIN GRAPHIC(ON,X,Y,0,0) END;

PROCEDURE UNDOT (X,Y:INTEGER);
 BEGIN GRAPHIC(OFF,X,Y,0,0) END;

PROCEDURE LINE(A1,B1,A2,B2:INTEGER);
 BEGIN GRAPHIC(LINS,A1,B1,A2,B2) END;

PROCEDURE CLINE(A1,B1,A2,B2:INTEGER);
```

```
 BEGIN GRAPHIC(LINC,A1,B1,A2,B2) END;

PROCEDURE REVERS;
 BEGIN GRAPHIC(REV,0,0,0,0) END;

(*  END OF GRAPHIC ROUTINE  *)
```

8.3.1 HILBERT CURVES

The program listed below gives you a practical demonstration of the material covered previously concerning graphics. The program draws meandering lines, and can also do recursion. We suggest that you read Algorithms and Data Structures by Niklaus Wirth. The program is stored on your system disk under the name HILBERT. If you compile this program be sure to change the defaults as listed in the program.

```
PROGRAM HILBERT;
     (*--------------------------------*)
     (* START-OF-PROGRAM: $0C00       *)
     (* HEAP            : EOPRGM       *)
     (* TOP-OF-STACK    : $2000       *)
     (*--------------------------------*)

CONST
     HX0=320; HY0=192;
     CLRHOM=CHR($93);
     BACKSPC=CHR($9D);

TYPE
GRAPHICCOMMAND=(GRIN,GROT,GCLR,CCLR,ON,OFF,LINS,
LINC,REV);

VAR  DEPTH,X0,Y0,HX,XX1,XX2,I,YY1,YY2:INTEGER;
     CHARIN                           :CHAR;

          (*----------------------------*)
          (*                            *)
              &INCLUDE(S_GRAPH);
          (*                            *)
          (* &INCLUDE CAN INSERT ANY    *)
          (* USER PROGRAM               *)
```

```
PROCEDURE DRAW;
 BEGIN
   LINE(XX1,YY1,XX2,YY2);
   XX1:=XX2;YY1:=YY2 END;
PROCEDURE B(I:INTEGER);FORWARD;
PROCEDURE C(I:INTEGER);FORWARD;
PROCEDURE D(I:INTEGER);FORWARD;

PROCEDURE A(I:INTEGER);
BEGIN
 IF I> 0 THEN
  BEGIN
   D(I-1);XX2:=XX1-HX;DRAW;
   A(I-1);YY2:=YY1-HY;DRAW;
   A(I-1);XX2:=XX1+HX;DRAW;
   B(I-1)
  END
 END;

PROCEDURE B;
BEGIN
 IF I> 0 THEN
  BEGIN
   C(I-1);YY2:=YY1+HY;DRAW;
   B(I-1);XX2:=XX1+HX;DRAW;
   B(I-1);YY2:=YY1-HY;DRAW;
   A(I-1)
  END
 END;

PROCEDURE C;
BEGIN
 IF I> 0 THEN
  BEGIN
   B(I-1);XX2:=XX1+HX;DRAW;
   C(I-1);YY2:=YY1+HY;DRAW;
   C(I-1);XX2:=XX1-HX;DRAW;
   D(I-1)
  END
 END;
```

```
PROCEDURE D;
BEGIN
 IF I> 0 THEN
  BEGIN
   A(I-1);XX2:=YY2-HY;DRAW;
   D(I-1);XX2:=XX1-HX;DRAW;
   A(I-1);YY2:=YY1+HY;DRAW;
   C(I-1)
  END
 END;

BEGIN                 (*MAIN OF HILBERT*)
 WRITELN(CLRHOM);WRITELN;
 WRITELN('HILBERT - CURVES':26);
 WRITELN;
 WRITELN('THIS PROGRAM DRAWS HILBERT CURVES');
 WRITELN('WITH THE HIGH-RES-GRAPHICS OF THE C-64');
 WRITELN('         SEE:  NIKLAUS WIRTH,         ');
 WRITELN('    ALGORITHMS AND DATA STRUCTURES    ');
 WRITELN('            TEUBNER PUB.,            ');
 WRITELN;
 WRITELN('DEPTHS OF RECURSION CAN BE INPUT');
 WRITELN('BETWEEN THE NUMBERS OF 1 AND 6');
 WRITELN;
 WRITELN('RUN/STOP EXITS HIGH-RES MODE;');
 WRITELN(' "E" EXITS THE PROGRAM ALTOGETHER.');
 WRITELN;
 WRITELN;
 REPEAT
  WRITE('CHOICE (1-6,E) =?',BACKSPC);
  REPEAT
   CHARIN:=GETKEY
  UNTIL CHARIN IN ['1'..'6' ];
  WRITELN(CHARIN);
  IF CHARIN <>'E' THEN
   BEGIN
    DEPTH:=ORD(CHARIN)-ORD('0');
    GRAPHIN;GRAPHCLR;COLCLR(0,5);
    HX:=HX0;X0:=HX DIV 2;
    HY:=HY0;Y0:=HY DIV 2;
    I:=0;
    REPEAT;
    I:=I+1;
    HX:=HX DIV 2;HY:=HY DIV 2;
    X0:=X0 + HX DIV 2;
    Y0:=Y0 + HY DIV 2;
    XX1:=X0;YY1:=Y0;
    XX2:=XX1;YY2:=YY1;
    A(I)
   UNTIL I = DEPTH;
   REPEAT UNTIL EOF; (*WAIT FOR BREAK *)
  GRAPHOUT
  END
 UNTIL CHARIN='E'
END.
```

8.4 C64 TO PASCAL DOS

The program C64TOPAS on the main disk converts files from C64 format to SUPER Pascal DOS format. The program is started by running it from the Main Menu in the following manner:

```
R
FILE-TITLE = C64TOPAS
DRIVE(MAP) = 0
```

```
* FILE-TRANSFER C64-DOS TO PASCAL-DOS *
*************** VS  5.3 ***************

TITLE OF SOURCE-FILE (C64-FILE) = ...
?
```

Enter the C-64 file name.

The program will then ask if the file is stored in program or sequential format.

```
PROGRAM OR SEQUENTIAL (P/S)?
```

Next enter the new name for the SUPER Pascal file.

```
TITLE OF PASCAL-FILE =?
```

Insert the C-64 formatted disk into drive 0.

```
INSERT DISC WITH SOURCE-FILE (C64-FILE)
INTO DRIVE 0!   PRESS"RETURN" IF DONE

INSERT THE DESTINAT'-DISC (PASCAL-DISC)
INTO DRIVE 0!   PRESS "RETURN" IF DONE.
```

The file will be converted to SUPER Pascal DOS format

9.0 APPENDIX

9.1 ERROR LIST

This is the complete SUPER Pascal list of compiler errors, per the Pascal User Manual and Report.

```
 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:  'A"' EXPECTED
12:  'U"' 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
22:  '..'EXPECTED
23:  '.' EXPECTED
24:  ',' OR ')' EXPECTED
25:  BOOLEAN CONSTANT EXPECTED
:
50:  ERROR IN CONSTANT
51:  ':=' EXPECTED
52:  'THEN' EXPECTED
53:  'UNTIL' EXPECTED
54:  'DO' EXPECTED
```

```
55:  'TO' OR 'DOWNTO' EXPECTED
56:  'IF' EXPECTED
57:  'FILE' EXPECTED
58:  ERROR IN FACTOR
59:  ERROR IN VARIABLE
60:  PROGRAM INCOMPLETE
:
101:  IDENTIFIER DECLARED TWICE
102:  LOW BOUND EXCEEDS HIGHBOUND
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
110:  TAGFIELD TYPE MUST BE SCALAR OR SUBRANGE
111:  INCOMPATIBLE WITH TAGFIELD TYPE
113:  INDEX TYPE MUST BE SCALAR OR SUBRANGE
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
121:  FILE VALUE PARAMETER NOT ALLOWED
122:  FORWARD DECLARATION 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:  TESTS ON EQUALITY ALLOWED ONLY
133:  FILE COMPARISON NOT ALLOWED
```

134:	ILLEGAL TYPE OF OPERAND(S)
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
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 OR FUNC NOT ALLOWED
165:	MULTIDEFINED LABEL
166:	MULTIDECLARED LABEL
167:	UNDECLARED LABEL
168:	UNDEFINED LABEL
169:	ERROR IN BASE SET
170:	VALUE PARAMETER EXPECTED
171:	STANDARD FILE WAS REDECLARED
177:	ASSIGNMENT TO FUNCTION IDENTIFIER NOT ALLOWED HERE

178:	MULTIDEFINED RECORD VARIANT
179:	X-OPT OF ACTUAL PROC OF FUNC DOES NOT MATCH FORMAL DECLARATION
182:	PARAMETER LIST OF EXTERN PRGM NOT ALLOWED
183:	LOAD/SETADR ONLY FOR EXTERNALS
184:	EXTERNAL WITHOUT ADDRESS-DEFINITION
185:	SLICE-ARRAY MUST BE OF TYPE CHAR OR BYTE
186:	ASSIGNMENT OF SLICE TO SLICE NOT ALLOWED
:	
201:	ERROR IN REAL CONSTANT: DIGIT EXPECTED
202:	STRING CONSTANT MUST NOT EXCEED SOURCE LINE
203:	INTEGER CONSTANT EXCEEDS RANGE
206:	INTEGER PART OF REAL CONSTANT EXCEEDS RANGE
207:	BYTE-CONST TOO LARGE
208:	ERROR IN BYTE-CONST
209:	ERROR IN HEX-CONST
210:	ERROR IN NUMERIC-CONST
:	
250:	TOO MANY NESTED SCOPES OF IDENTIFIERS
251:	TOO MANY NESTED PROCEDURES AND/OR FUNCTIONS
252:	TOO MANY FORWARD REFERENCES OF PROC ENTRIES
257:	TOO MANY EXTERNALS
258:	TOO MANY LOCAL FILES
259:	EXPRESSION TOO COMPLICATED
:	
398:	IMPLEMENTATION RESTRICTION
399:	VARIABLE DIMENSION ARRAYS NOT IMPLEMENTED
400:	FILE-ELEMENT TOO LONG
401:	STRING NOT ALLOWED HERE
402:	TOO MANY IDENTIFIERS
403:	READLN/WRITELN ONLY WITH TEXT
404:	PROGRAM INCOMPLETE
405:	TOO MANY SEGMENTS
406:	NESTED SEGMENTS NOT ALLOWED
407:	SEPARATED SEGMENTS NOT ALLOWED
408:	COMPILING OF SEGMENTED PRGMS TO RAM NOT ALLOWED
409:	TOO MANY PARAMETERS
410:	ERROR IN '&' OPTIONS
411:	TOO MANY NESTED SOURCES

9.2 FOR FURTHER READING

ON PASCAL:

Alpert/Stephen: PASCAL, A structured strong Language
BYTE 3/78 BYTE Publications

Barron, D.W.: PASCAL, The Language and its Implementation
John Wiley & Sons, New York

Bowles: USCD PASCAL
BYTE 5/78

Jensen/Wirth: PASCAL User Manual and Report
Springer Verlag, New York

Zaks, R: Introduction to PASCAL including USCD PASCAL
Sybex, Berkeley CA

ON THE C-64 AND MACHINE LANGUAGE:

Angerhausen/Becker/English/Gerits:
Anatomy of the Commodore 64
Abacus Software, Grand Rapids MI

Englisch, L.: The Advanced Machine Language Book for the Commodore-64
Abacus Software, Grand Rapids MI

English/Szczepanowski: The Anatomy of the 1541 Disk Drive
Abacus Software, Grand Rapids MI

Commodore 64 Programmer's Reference Guide

9.3 INDEX

Auto-Run Super Pascal Programs

To make an auto-run Super Pascal program disk:

Load Super Pascal into your computer.

From the main menu run the SYSGEN program by:

```
r          [RETURN] for r(unprgm)
SYSGEN     [RETURN]
```

This creates a Super Pascal disk. When this is finished remove the newly created Super Pascal Disk and insert the Master Super Pascal disk.

From the main menu goto the Utility menu by pressing:

```
u          [RETURN] for u(tility)
```

Copy the program you wish to automatically start, using the c(opy) command to the new Super Pascal Disk as follows:

```
c          [RETURN] for c(opy)
source - drive: 0
destinat-drive: 0
file-title = program name
```

When the copy is finished rename "`program name`" to "`startup`" usi
the r(ename) command as follows:

```
r          [RETURN] for r(ename)
file-title = program name
replacement = startup
```

Restart the C-64 system and with the Super Pascal disk you created in the disk drive simply type:

```
LOAD "*",8,1
```

The Super Pascal System will be loaded and your program will automatically start.

Super Pascal Addendum

This addendum consists of clarifications and corrections to the Super Pascal 64 manual. Page numbers refer to those in the Super Pascal 64 manual.

A. (text follows program code at bottom of p. 86)

The type compatibility between STRING and CHAR array also means that the procedures `WRITE` and `WRITELN` can output quantities of type CHAR in addition to quantities of type STRING. For example, `WRITELN(TITLE:10);` is absolutely correct in Super Pascal if `TITLE` is defined as type ALFA.

B. (page 87, following the description of `RANDOM` and preceding `COMMAND SET`)

Following the variable declaration comes the

Procedure declaration

and

Function declaration

Except for the structuring and compiler instructions to be discussed later, we will not say anything more about these two here.

The next part of a program block is the

Statement section

with its sequence of statements. Two extensions of Super Pascal should be mentioned in the area of the statement section. The first is regarding the

```
:=  (assignment) statement
```

To allow for easy access to variables of type FILE and ARRAY OF CHAR or ARRAY OF BYTE, the following access mechanisms are provided:

File access

Instructions with the following syntax:

```
FILEVARIABLE(INDEX):=ELEMENT;
```

or

```
DESTVAR:=FILEVARIABLE(INDEX);
```

can be used to access a precisely defined element of a file (random access) for both reading and writing, depending on the assignment.

FILEVARIABLE stands for the identifier which was declared as a variable of type FILE in the declaration section.

ELEMENT stands for a expression of the type of the elements of the file in question.

DESTVAR stands for the identifier of a variable of the type of the elements of the file in question.

INDEX stands for the number of the desired file element. The elements are placed in the file sequentially and the first element has the number 0. The INDEX expression must be of type REAL so that large files can be accessed. The integer portion of the index expression will always be chosen. Negative values or values which are too large lead to runtime errors:

IL.FILE OPR. ERROR or AFTER EOF ERROR!

If the element type of the file is a structured type, individual sub-variables can also be accessed:

```
FILEVARIABLE(5000).CITY:='NEW YORK';
```

If the file element contains a field definition of type ALFA. Something like this is also allowed:

```
IF FILEVARIABLE(5000).CITY[0]='N' THEN ...
```

NOTE:
This method of file access implicitly includes opening and closing the file, which takes a noticeable amount of time on the C64 because of the slow transfer of data to and from the disk drive. Care must also be taken to ensure that three file buffers of the Super Pascal system are available for file access. The file being addressed must be accessible in the working disk drive (see the procedure SETDRV).

Array access

In addition to the assignment of entire arrays or individual array elements, sections of arrays (called slices) can be accessed in Super Pascal. This is especially useful when working with CHAR arrays and string quantities.

The syntax is as follows:

```
ARRAYVAR[>INDEX]:=EXPRESSION;
```

and

```
DESTVAR:=ARRAYVAR[>INDEX];
```

In the first case, the quantity indicated by EXPRESSION is placed in the array designated by ARRAYVAR at position INDEX. The lowest array element has the number 0. INDEX must be of type INTEGER, while the array variables must be of type ARRAY OF CHAR or ARRAY OF BYTE.

NOTE:
During these assignment, the quantity EXPRESSION is placed over the specified array range in its entirety, regardless of whether it fits this range or not. Under certain circumstances, neighboring variables may be overwritten! This assignment technique should be used only for known relationships.

For example:

```
    TITLE[>4]:=1234;
```

places the binary coding of the integer value 1234 in positions 4 and 5 of the array `TITLE`.

```
    TITLE[>4]:=TITLE;
```

leads to a "dangerous" range overflow because it places the entire variable `TITLE` in the variable area at position 4 and beyond.

In the opposite assignment:

```
    DESTVAR:=ARRAYVAR[>INDEX];
```

the destination variable will be filled in its entire length with the array elements of `ARRAYVAR` at position `INDEX` (inclusive). Missing values will be taken from the variable storage adjacent to `ARRAYVAR`.

Although the constructs presented here do not conform to the Pascal concept, they do provide an easy way to process elements of differing types and sizes, especially for system programming, when applied conscientiously. If a particular problem is to be solved using good Pascal style, there are other ways of accomplishing the same things.

C. (This text is the conclusion of `CLOSE`, p. 89, bottom)

NOTE:
The `CLOSE` procedure must be used for a file opened for writing or the information last written to the file will be lost. The information will be written to the file buffer, but not actually stored in the given file. The buffer is not written to disk until it is full or the file is closed.

D. (Re-definition of LOAD,p. 93)

LOAD
LOAD loads an external Pascal program routine into memory.

```
Syntax:
LOAD(PROCEDURE_FUNCTION_NAME,FILENAME,DRIVE_NR);
```

In contrast to CONTINUE and EXECUTE, the LOAD procedure allows only an external program routine to be loaded. The external routine declared under an arbitrary identifier (PROCEDURE_ FUNCTION_NAME) will be loaded into memory during the program run. It must be available under the given identifier (FILENAME) in the given drive (DRIVE_NR). The procedure or function identifier (PROCEDURE_FUNCTION_NAME) must not be the same as the disk entry (FILENAME). The loading procedure itself is performed by a utility routine in the file LOADDAT. LOADDAT must be present in drive 0 or the program run will stop with an error message.

Calling the loaded function is no longer part of the procedure; it takes place as with a normal procedure or function via the identifier declared with the reserved word symbols XTRNPRGM, XTRNPROC, and XTRNFUNC.

E. (text added to OUTDVC, p. 95 under **NOTE:**)

NOTE:
The inadequate input/output interface built into the C64 under the primary address 2 (RS-232) is not available via OUTDVC. If you are interested, you can make an adaptation with Super Pascal. OUTDVC addresses only the devices connected to the serial input/output bus.

F. (add to SEEK, p. 96-97)

The SEEK procedure can only be used on files which are available in the drive defined as the current working drive. If the file is not found, the program will stop with an appropriate error message. The working drive can be defined with the procedure SETDRV, discussed later.

This procedure positions the access pointer to the file element whose ordinal number is determined by the value representing EXPRESSION. The first element of a file, the element to which the access pointer is set by RESET or REWRITE, has the ordinal number 0. The difference between read and write access results from the operation following the SEEK procedure. GET, READ, and READLN cause read accesses, while PUT, WRITE, and WRITELN write to the file. After each access, the access pointer is advanced one element.

After a write access, any data behind the write position will be erased. Only writing can continue in the file. Termination of the read/write operations is done with CLOSE or LOCK. It is not possible to write to a file which has been LOCKed. If an attempt is made, the message

```
   IL.FILE OPR. ERROR!
```

will occur and the program will be terminated.

G. (add to Chapter 4.6, p. 125, end of page)

If, at the beginning of a program, its entry address is taken from the pointer ADR_PRPO and placed in the pointer ADR_EXPO, then every program end will lead back to the called program. All you must do is check at the beginning of the program whether it is being called for the first time and must be initialized or whether this is a re-entry. This can be determined from the WARMFLAG; if it is set at the beginning of the program, it can be used to recognize a re-entry and bypass the initialization routine. All variables will remain intact.

The only problem is the actual jump to the MAIN menu (QUIT). This is possible via a small assembly language routine which executes a 65XX JUMP to the MAIN menu. More details can be gathered from the listing of the editor program in Chapter 8.

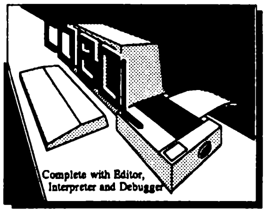
Make Your Choice

C-64 or C-128

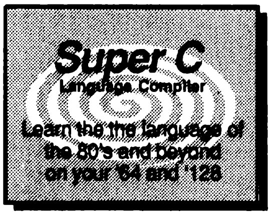
BASIC Compiler
omplete BASIC compiler ıd development package. peed up your programs 5x ı 35x. Compile to machine ɔde, compact p-code or ɔth. '128 version: 40 or 80 ol. monitor output and AST-mode operation. '128 ersion includes extensive 0-page programer's guide. . great package that no oftware library should be ɪithout. C-64 $39.95 C-128 $59.95

BASIC Compiler
Speeds up your BASIC programs by 3 to 20 times. For C-64 and C-128

COBOL
Now you can learn COBOL, the most widely used commercial programming language, on your 128 or 64. COBOL Compiler package comes complete with syntax-checking editor, interpreter and symbolic debugging aids. New '128 version works with 40/80 column monitors and is quicker than the '64 version. C-64 $39.95 C-128 $59.95

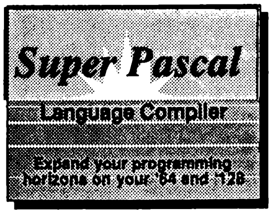


Super C Compiler
ɔr school or software ɘvelopment. Learn the C nguage on the '64 or '128. ompiles into *fast* machine ɔde. Added '128 features: P/M-like operating system;)K RAM disk. Combine M/L C using CALL; 51K avail- ɔle for object code; Fast ɘading; Two standard I/O ɔrarys plus math & graphic ɔraries. C-64 $59.95 C-128 $59.95


Super C
Language Compiler
Learn the the language of the 80's and beyond on your '64 and '128

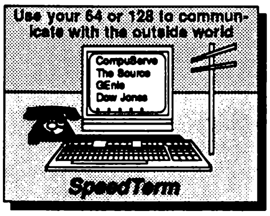
New C-128 Version!

Super Pascal Compiler
Complete system for developing applications in Pascal. Extensive editor. Standard J & W compiler. Graphics library. Added '128 features: RAM disk; 100K source/one drive or 250K/two; 80/40 column. If you want to learn Pascal or develop software using the best tool available, Super Pascal is your first choice. C-64 $59.95 New! C-128 $59.95

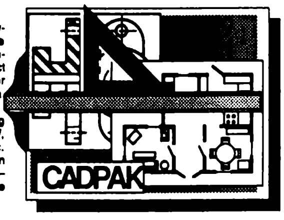


C-64 sion!

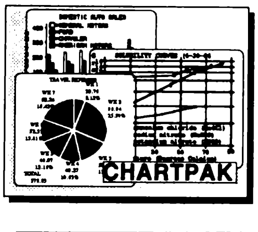
Speedterm
ɘt your 64 or 128 communɘate with the outside world. ɔbtain information from ɘrious computer networks. lexible, command driven ɘrminal software package. upports most modems. modem and Punter transfer rotocol. VT52 terminal emuɪtion with cursor keys, large 5K capture buffer & user ɘfinable function keys. ɘw! C-64 $39.95 C-128 $39.95



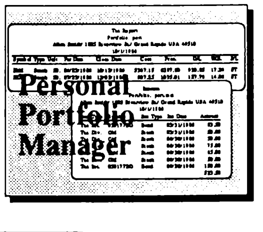
Cadpak
Easy-to-use interactive drawing package for accurate graphic designs. Dimensioning features to create exact scaled output to all major dot-matrix printers. Input via keyboard or lightpen. Two graphic screens for COPYing from one to the other. DRAW, BOX, ARC, ELLIPSE, etc. available. Define your own library of symbols/objects–store up to 104 separate objects. C-64 $39.95 C-128 $59.95



Chartpak
ɾreate professional quality harts *fast*— without programming. Enter, edit, save nd recall data. Interactively uild pie, bar, line or scatter raph. Set scaling, labeling nd positioning. Draw charts different formats. Statistical ɔutines for average, standard eviation, least squares and ɔrecasting. Use data from preadsheets. Output to most rinters. C-64 $39.95 C-128 $39.95



PPM
Comprehensive portfolio management system for the 64 and 128. Manage stocks, bonds, mutual funds, T-bills; record taxable or non-taxable dividends & interest income; reconcile each brokerage account cash balance with the YTD transaction file; on-line quotes through Dow Jones or Warner. Produces any type of report needed to analyze a portfolio or security. C-64 $39.95 C-128 $59.95



ɔall **now** for the name of the dealer nearest you. ɔr order directly form Abacus using your MC, Visa r Amex card. Add $4.00 per order for shipping. ɔoreign orders add $10.00 per item. Call (616) 41-5510 or write for your **free** catalog. 30-day money back software guarantee. Dealers inquires welcome–over 1500 dealers nationwide.

Abacus

P.O. Box 7219 Dept M2 Grand Rapids, MI 49510
Phone 616/241-5510 • Telex 709-101 • Fax 616/241-5021

Abacus You Can Count On

Register this software and be eligible to win additional software free in our monthly drawing.

Return this card to register your purchase and to receive free technical support for this product. You may also order a backup copy of this program.

Monthly drawing winner will be notified by mail. Good Luck!

FOLD HERE→ AND TAPE

900
Product ID

REGISTRATION CARD

Registration # 050509 Program:______________________
Name____
Address____
____
City____________________State____________Zip__________

Purchase Information:
Dealer____
Address____
City____________________State____________Zip__________

Return this registration card to obtain a backup copy of the above program for a handling charge of $10.00. A check, money order, or credit card number must accompany this request. Purchase orders are <u>not acceptable.</u>

BACKUP COPY?

☐ No, do not send a backup copy, but register my purchase.

☐ Yes, send a backup copy. $10.00 payment is enclosed.

Credit card#_________________________________
Expiration Date____/____/____

Abacus Software
You Can Count On
5370 52nd Street SE
Grand Rapids, MI 49508

Abacus

Software You Can Count On