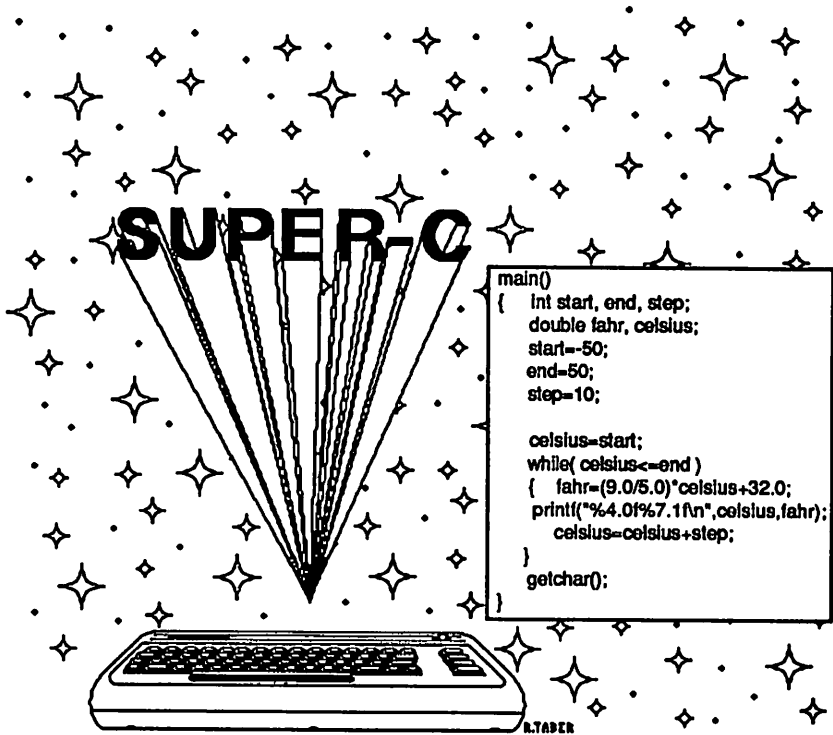


SUPER-C

for the

C-128 or C-64

by F.J. Hauck and T. Eirich



A Data Becker product from

Abacus  Software

233277

Copyright Notice

Abacus Software makes this package available for use on a single computer only. It is unlawful to copy any portion of this software package onto any medium for any purpose other than backup. It is unlawful to give away or resell copies of this package. Any unauthorized distribution of this product deprives the authors of their deserved royalties. For use on single-site multiple computers, please contact Abacus Software to make arrangements.

Warranty

Abacus Software makes no warnings, expressed or implied, as to the fitness of this software package for any particular purpose. In no event will Abacus Software be liable for consequential damages. Abacus Software will replace any copy of this software which is unreadable, if returned within 30 days of purchase. Thereafter, there will be a nominal charge for replacement.

Second Printing, December 1986

Copyright © 1986

Data Becker GmbH
Merowingerstr. 30
4000 Dusseldorf, West Germany
Abacus Software, Inc.
P.O. BOX 7219
Grand Rapids, MI. 49510

ISBN 0-916439-59-3

Preface

The programming language C has been in existence since about 1972. It was developed by Dennis Ritchie and Brian Kernighan for the operating system UNIX. More than 90% of UNIX was written in C. With the spread of UNIX, C enjoyed greater popularity as well. Super C now makes it possible to program in C on the C-64 or C-128.

Like Pascal, C is a structured programming language. Programs are written without jumps (goto's). You can program small problems in functions (comparable to subroutines in BASIC), and then call this with certain data. Once programmed, only the operation of these functions need be known. This concept results not only in programs which are easier to read but also to easier to maintain. By building libraries of functions, new programs can incorporate them to quickly create solutions to a wide assortment of problems.

In C there are data types comparable to those in Pascal. Very important in C is the pointer, which plays a large role. While it is more of an appendage to Pascal, C supports the pointer with pointer arithmetic. This moves C in the direction of assembly languages. Like assembly languages, certain problems can be solved easily using pointer arithmetic, without assuming anything about the computer being used. This means that programs will run on any C machine without significant changes.

Super C is a system of programs. Super C contains a command processor with various resident commands, such as to display the contents of the disk. In addition, there are commands which can be loaded (transient commands). These can also be written in C themselves.

An editor makes it easy to enter C programs. Programs are translated into machine language by the compiler. This system includes the entire range of the C language except for bit fields.

The linker links together separately compiled programs and the standard libraries. Graphics and mathematical functions are also available as library functions.

The Super C version for the C-128 also has a RAM disk. This can be used like a normal disk drive. The RAM disk has a tremendous speed advantage over normal disks. The contents of the RAM disk can also be saved on diskette as a package so that they are loaded when Super C is started. Super C can work with a total of up to eight disk drives.

Franz J. Hauck
Thomas Eirich

Table of Contents

Part I. Tutorial	1
1. Introduction	3
2. The Super C command processor	5
2.1 Starting Super C	5
2.2 CCP resident commands	6
2.3 CCP transient commands	8
3. The RAM disk in the C-128	13
4. C editor	15
4.1 The command "new"	15
4.2 Inserting and deleting lines	16
4.3 Saving text	19
4.4 Loading text	20
4.5 Block commands	21
4.5.1 Deleting block	
4.5.2 Move blocks	22
4.5.3 Copy block	23
4.6 Search and replace	24
4.6.1 Searching	24
4.6.2 Replacing	25
4.7 Back to the CCP	26
5. Your first C program	27
5.1 Editing	28
5.2 Compiling	30
5.3 Linking	33
5.4 Executing	36
6. Introduction to C	37
6.1 Overview	37
6.1.1 The first program	37
6.1.2 Objects	39
6.1.3 Loops	41
6.1.4 Symbolic constants	44
6.1.5 Arrays	45
6.1.6 Character arrays	47

6.2 Expressions and declarations	48
6.2.1 Names	48
6.2.2 Types	48
6.2.3 Constants	49
6.2.4 Memory classes	51
6.2.5 Arithmetic operators	52
6.2.6 Comparisons, logical operators	53
6.2.7 Type conversions	54
6.2.8 Increment and decrement	54
6.2.9 Bit operations	55
6.2.10 Assignments	56
6.2.11 Conditional evaluation	57
6.2.12 Precedence and order of operators	58
6.2.13 Additional operators	59
6.2.14 Program text	60
6.3 Control structures	60
6.3.1 Block	61
6.3.2 if instruction	61
6.3.3 switch instruction	63
6.3.4 while instruction	64
6.3.5 for instruction	65
6.3.6 do instruction	66
6.3.7 break instruction	66
6.3.8 continue instruction	67
6.3.9 goto instruction and labels	68
6.4 Program structures	69
6.4.1 Functions	69
6.4.2 Arguments	72
6.4.3 Global definitions	73
6.4.4 Declarations	74
6.4.5 Local definitions	76
6.4.6 Initializations	76
6.4.7 Macros	79
6.4.8 Chaining files	81
6.5 Pointers, addresses, and arrays	81
6.5.1 Pointers	82
6.5.2 Address arithmetic	83
6.5.3 Pointers and arrays as arguments	85
6.5.4 Complex declarations	86
6.5.5 Pointer arrays	87
6.5.6 Pointers and multi-dimension arrays	88

6.6 Structures and variants (struct/union)	89
6.6.1 Declarations of structures	90
6.6.2 Access to components	91
6.6.3 Functions and structures	92
6.6.4 Recursive structures	93
6.6.5 Variants	94
6.6.6 Type definitions	95
6.7 Programming environment	96
6.7.1 Files	96
6.7.2 EOF	97
6.7.3 STDIO	98
6.7.4 Additional functions	98
6.7.5 Error handling	99
6.7.6 Interruption	99

II. System Guide 103

1. The command processor	103
1.1 Start, NMI, and RESET	103
1.2 Device identifiers and filenames	105
1.3 Extensions	106
1.4 Passing arguments	107
1.5 Character sets	108
1.6 Monitors	109
1.7 Resident commands	109
1.8 Transient commands	111
1.8.1 device	111
1.8.2 copy	112
1.8.3 f	114
1.8.4 lram (V3 only)	115
1.8.5 sram (V3 only)	115
1.8.6 sysgen (V3 only)	115
1.8.7 c system	117
1.8.8 c1	117
1.8.9 type	118
2. RAM disk	119
2.1 Deviations from Commodore DOS	119
2.2 Track/sector division	120
2.3 The RAM disk commands	121

3. The C editor	123
3.1 Character sets and text display	123
3.2 Control keys	125
3.3 Parameter inputs	128
3.3.1 Key input	128
3.3.2 Input a number	128
3.3.3 Input a string	129
3.3.4 Block input	129
3.3.5 Destination input	130
3.4 Commands	131
3.5 Error messages	139
4. C compiler	140
4.1 Start without arguments	140
4.2 Start with arguments	142
4.3 Compiler error messages	143
4.4 Finding errors	
5. Linker	150
5.1 Start without arguments	150
5.2 Start with arguments	152
5.3 Error messages	153
6. C programs	155
6.1 Start	155
6.2 Operating modes	156
6.3 Run-time errors	159
6.4 Memory layout	162
7. The Library functions	163
7.1 Standard C libraries	164
7.1.1 <code>erron()</code> , <code>erroff()</code> , <code>nmion()</code> , <code>qerror()</code> , <code>error()</code> , <code>exit()</code>	165
7.1.2 <code>open()</code> , <code>close()</code>	166
7.1.3 <code>putc()</code> , <code>fputc()</code> , <code>getc()</code> , <code>fgetc()</code>	168
7.1.4 <code>getchar()</code> , <code>putchar()</code>	168
7.1.5 <code>gets()</code> , <code>fgets()</code> , <code>puts()</code> , <code>fputs()</code>	169
7.1.6 <code>fgetf()</code> , <code>fputf()</code>	169
7.1.7 <code>fopen()</code> , <code>fclose()</code>	170

7.1.8	strlen(), strcmp(), strncmp()	171
7.1.9	strcat(), strncat(), strcpy(), strncpy()	172
7.1.10	strchr(), strrchr()	173
7.1.11	cursor(), exec()	173
7.1.12	cmove(), move()	174
7.1.13	alloc(), free()	175
7.1.14	settime(), gettime()	176
7.1.15	keys()	177
7.1.16	call()	177
7.1.17	fast(), slow()	178
7.1.18	window()	178
7.1.19	vdcin(), vdcout()	178
7.1.20	io\in(), io\out()	179
7.1.21	is80()	179
7.1.22	Formatted output	180
7.1.23	Formatted input	184
7.1.23.1	Reading strings	187
7.1.23.2	Error messages	187
7.1.23.3	sscanf and fscanf	188
7.2	The graphics library	189
7.2.1	graphic(), graphon(), graphoff(), isgraph()	190
7.2.2	backgr(), clrmap(), colors(), setcol()	191
7.2.3	dot(), dotin(), bdot()	192
7.2.4	line(), bline(), mline(), oline()	195
7.2.5	setplot(), plot()	197
7.2.6	shape(), fill()	198
7.2.7	pushobj(), plotobj(), fplotobj(), bplotobj()	199
7.2.8	mask, bmap, mapv()	201
7.2.9	Layout of the graphics memory	202
7.2.10	Demo program	203
7.2.11	Storing the graphics	203
7.3	Math library	204
7.3.1	sin(), cos()	204
7.3.2	tan(), atan()	205
7.3.3	abs(), sgn(), rnd()	205
7.3.4	sqr(), sqrt()	206
7.3.5	log(), exp()	206
7.4	ctype.h	207

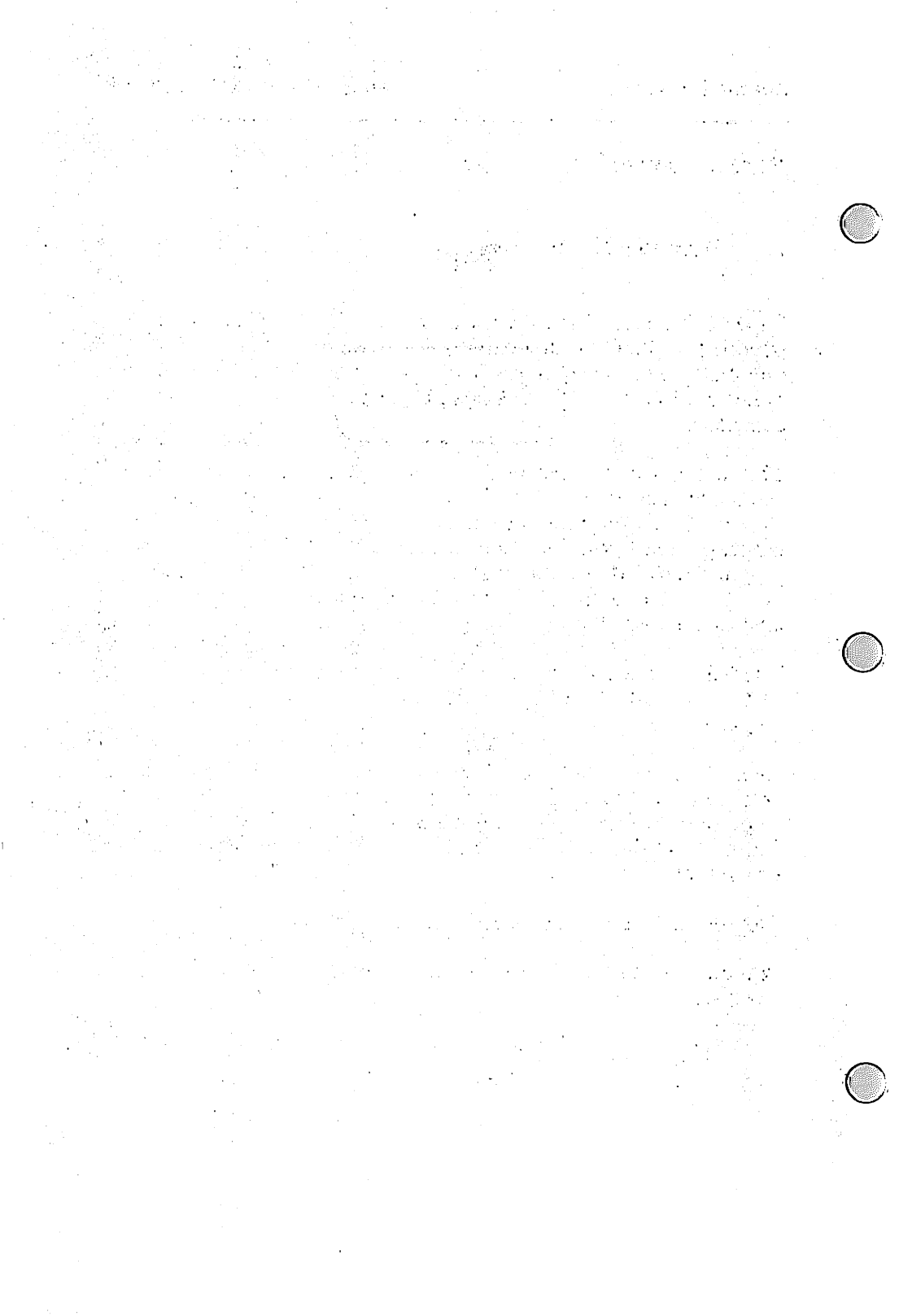
8. C language description	209
8.1 Introduction	209
8.2 Text conventions	209
8.2.1 Comments	209
8.2.2 Names	209
8.2.3 Keywords	210
8.2.4 Constants	210
8.2.4.1 Integer constants	210
8.2.4.2 Char constants	211
8.2.4.3 Floating-point constants	211
8.2.5 Strings	212
8.2.6 Example	212
8.3 Object names	213
8.3.1 Storage classes	213
8.3.2 Types	214
8.3.3 Hardware-specific type data	215
8.4 Objects and L-values	216
8.5 Conversion of a type	216
8.5.1 Integer values between each other	216
8.5.2 Floating-point values between each other	216
8.5.3 Floating-point and integer values	217
8.5.4 Addresses and integer values	217
8.5.5 The standard conversions	217
8.6 Syntax notation	218
8.7 Expressions	218
8.7.1 Simple expressions	219
8.7.2 Unary operators	221
8.7.3 Multiplication, division	222
8.7.4 Addition, subtraction	223
8.7.5 Shift operations	224
8.7.6 Comparisons	224
8.7.7 Equivalence comparisons	225
8.7.8 Bit operations	226
8.7.9 Logical operations	226
8.7.10 Condition evaluation	227
8.7.11 Assignments	227
8.7.12 Lists	228
8.8 Declarations	229
8.8.1 Storage classes	230
8.8.2 Types	230
8.8.3 Data definitions	231

8.8.4	Type declarations	231
8.8.5	Functions	232
8.8.6	Declarators	232
8.8.7	Function declarator	234
8.8.8	Parameter declaration	234
8.8.9	Structures and unions	235
8.8.10	Enumerated type	236
8.8.11	Initializations	236
8.8.12	Abstract declarators	238
8.9	Statements	238
8.9.1	Blocks	239
8.9.2	while statement	239
8.9.3	do statement	240
8.9.4	for statement	240
8.9.5	if statement	241
8.9.6	switch statement	241
8.9.7	break statement	242
8.9.8	continue statement	242
8.9.9	return statement	242
8.9.10	Labels	242
8.9.11	goto statement	243
8.9.12	Empty statement	243
8.10	Scope	243
8.10.1	Scope of a name	243
8.10.2	Scope on an object	245
8.11	Preprocessor	245
8.11.1	Macros	245
8.11.2	Chaining files	247
8.11.3	Conditional compilation	247
8.11.4	Line numbering	248
8.12	Implicit declarations	248
8.13	Operations on different data types	249
8.13.1	Structures and unions	249
8.13.2	Functions	250
8.13.3	Arrays, pointers	250
8.13.4	Conversion of pointer values	251
8.14	Constant expressions	251
8.15	Portability	252
8.16	Differences from standard compilers	252
8.17	Differences from the C-Compiler 64	253

Part III Appendix	254
1.0 Keyboard V2	254
1.2 Keyboard V3 LC character set	255
1.3 Keyboard V3 CBM character set	256
2.1 Character codes	257
2.2 CTRL-codes V2	258
2.3 CTRL-codes V3	259
2.4 ESC-codes V3	261
3.0 Function overview	262
4.0 Listing of the header files	270
5.0 Listing "text.c"	277
6.0 Listing "sample .c"	281
INDEX	282

Part I

Tutorial



Part I. Tutorial Section

1. Introduction

Super C Version 2 requires a C-64 or a C-128, since it can be operated in the C-64 mode. Version 3 requires a C-128. At least one disk drive is required for either version. The drive may be either a 1541 or a 1571. A maximum of 8 disk drives may be connected.

This manual is divided into two parts: a tutorial section and a reference section. The tutorial section will familiarize you with SUPER C. If you aren't familiar with the programming in the C language, the tutorial section also contains an introduction to C programming. If you find that this introduction to C programming is too advanced for you, consult the bibliography for a more extensive introduction.

The reference section describes the specific features and capabilities of the SUPER C package and contains a description of the C language.

Even if you're already familiar with the C language, you should begin with the tutorial section. The tutorial section describes the important features and commands of SUPER C. After you're familiar with these commands, you can concentrate on programming in C.

The following conventions are used in this manual.

A request to press a key is noted by the key enclosed in square brackets.

[RETURN]	means press the key marked RETURN
[A]	means press the key marked A

A request to press several keys in succession is noted by the keys enclosed in square brackets and separated by commas.

[h], [e], [l], [l], [o], [RETURN]

means type the word hello followed by the RETURN key.

The above may also be shortened to:

hello [RETURN]

A request to press two keys simultaneously is noted by the keys enclosed in square brackets and separated by a plus (+) sign.

[RUN/STOP] + [RESTORE]

means press the [RUN/STOP] key and the [RESTORE] key simultaneously.

In reality, we don't mean simultaneously. Instead press the first key and hold it down, then press the second.

A request to press an uppercase letter is noted either of two ways:

[SHIFT] + [a] or [A]

A request to press a cursor key is noted as follows:

[C-UP]	or	↑	Cursor up
[C-DOWN]	or	↓	Cursor down
[C-LEFT]	or	←	Cursor left
[C-RIGHT]	or	⇒	Cursor right

2. The Super C Command processor

2.1 Starting Super C

Turn on your computer and disk drive.

Insert the diskette included with your SUPER C package into your drive and close the drive door. We will call this disk the **system diskette**. The system diskette is write and copy-protected. You cannot store any more programs on the system diskette.

For SUPER C Version 3 (C-128)

Before starting SUPER C, check the setting of the 40/80 column button. If it is depressed, SUPER C will start up in 80 column mode, Otherwise it will start in 40 column mode. There are two ways to start SUPER C Version 3.

- Press the RESET button.
- Type: `boot [RETURN]`

In either case the SUPER C system is loaded and started.

For SUPER C Version 2 (C-64)

Enter the following:

```
load "c-system",8,1 [RETURN]
```

The SUPER C system is loaded and started.

If you are using Version 2 on a C-128 you can press the RESET button or type `boot [RETURN]`. SUPER C will switch to '64 mode automatically.

In either Version 2 or Version 3 the title screen is displayed and the SUPER C command processor displays its prompt:

a :

The blinking cursor will appear after the prompt (a :). When you see this prompt, you know that SUPER C is ready to process a command. We call the command processor CCP. This stands for C Command Processor.

2.2 CCP resident commands

All commands are terminated with the [RETURN] key. Enter:

a:dir [RETURN]

The directory of the system disk (which is hopefully still in the drive) is displayed. Below is a partial listing of a directory:

0	"c-system #3.00a "	he 2a
1	"ce"	prg
28	"c4"	prg
29	"e8"	prg
99	"cc"	prg
28	"cl"	prg
43	"c-system"	prg
5	"copy"	prg
30	"device"	prg

Now enter:

a:err [RETURN]

The CCP reads the error message from the disk drive and displays it. It looks like this:

a:err
00, ok,00,00
a:

A third command sends disk commands to the disk drive. Type:

```
a:com v [RETURN]
```

The disk command `v` for validate is sent to disk drive `a`.

If the system diskette is still in the drive the following appears on the screen:

```
a:com v
26, write protect on,18,00
```

The error message is displayed because the validated directory cannot be written to the write-protected diskette. Here, we are just demonstrating the `com` command.

All commands must immediately follow the `a:` prompt. The command `com` gets the "v" as the argument. Arguments for a command are separated from the command by at least one space. Spaces within the argument are not allowed.

Using the `com` command any valid disk command may be sent to the disk drive. For example, to rename a file from `test` to `prog`, you would send the following:

```
a:com r:prog=test
```

If a filename has an embedded space, the CCP interprets this as a separator. Only the text preceding the space is considered the argument. To overcome this limitation, you can replace all embedded spaces with a shifted space:

```
[SHIFT]+[SPACE]
```

A shifted space appears as a small dot on the screen.

The valid disk commands are described in the 1541 or 1571 disk drive manuals. Some of these disk commands are described briefly in Part II Chapter 2.

2.3 CCP transient commands

In addition to the preceding *resident* commands, SUPER C supports *transient* commands.

A resident command is always in the computer's memory, ready to immediately perform a task.

Transient commands are stored on diskette and are loaded into memory before they are started. SUPER C has several transient commands on the system diskette. Later, you'll learn how to add your own transient commands.

One SUPER C transient command is `device`. Type the following:

```
a:device [RETURN]
```

The program `device` is loaded from the system disk and responds with:

```
DEVICE CHANGE PROGRAM V2.0
device a to b.
```

Press [RETURN] twice. The message:

```
DRIVE/DOS version
1541/70/71 V2.6
device is changed.
device a to b.
```

This transient command changed the device number of your disk drive. Device numbers range from 0 to 15, and for disk drives from 8 to 15. In Super C the numbers are represented by letters:

Device letter:	a	b	c	d	e	f	g	h

Device number:	8	9	10	11	12	13	14	15

Since you accepted the default device identifiers in the `device` transient command, a and b, the device address 8 is changed to 9. This can be changed to normal as follows:


```
[b], [RETURN], [a], [RETURN]
```

Now change drive a to drive b as follows::

```
[a], [RETURN], [b], [RETURN]
```

When another input appears, press

```
[RUN/STOP]
```

You'll find yourself back in the CCP. Read the directory in the normal manner. The following is displayed:

```
a:dir [RETURN]
file not found
a:
```

The error "file not found" message is printed whenever a transient command is not found or if a disk drive is not addressable. The CCP tried to read the directory of device a. Since we reassigned drive a to device address b, the error message is displayed. To be able to read the directory of drive b, change the prompt from a: to b: as follows:

```
a:b: [RETURN]
b:dir [RETURN]
...
```

The directory of the system disk is now displayed. Furthermore, the prompt is now b:. Change the prompt back to a:.

```
b:a: [RETURN]
a:b:dir [RETURN]
...
b:
```

You can prefix a command with a device identifier. This implies that the command is to be performed using the prefixed device. Therefore `b:dir [RETURN]` displays the directory of drive b even if the current drive is drive a.

You can prefix the `com` and `err` commands with a device identifier. For transient commands, the prefixed device identifier specifies the drive from which the command is loaded.

Try loading device form a:

```
b:a:device [RETURN]
file not found
a:b:device [RETURN]
```

```
DEVICE CHANGE PROGRAM...
```

Load device from b and change the address of your drive back to a again.

If you are using two disk drives and both are turned on, the above procedure does not work correctly if one drive is device 8 and the other is device 9. Instead, the device transient command will set both drives to device 9. In this case you won't need the device command.

Usually the device address of the two disk drives are set to 8 on power up. In this case, turn off one drive, change the device from a to b using the device command, and turn on the second drive.

Remember that the device command changes the device address in software only. If the drives are turned off or the RESET button is pressed, the device number reverts to normal.

Here's another transient command. Type:

```
a:copy [RETURN]
```

This loads the `copy` transient command from the system disk. After a short time the prompt appears again, on Super C Version 2 it is now light red. This signifies that a transient command is loaded and ready to run (On Version 3, the transient command `lram std.p`, which loads the contents of the RAM disk, is automatically executed after booting. The prompt is red on Version 3 because `lram` is present after booting). You can use the `copy` command to copy files from one drive to another. The `copy` command

remains resident until another transient command is loaded.
Insert a blank disk into drive a and enter the format command:

```
a:com n:programs,cc [RETURN]
```

If you are using more than one drive, pay careful attention to the device identifier. Before formatting a disk make sure that the correct drive is specified.

You should now create a work diskette for your C programs. The work diskette will require various link files. Create your work diskette as described below.

For one drive systems:

Insert the system disk in drive a and enter:

```
a:copy a:stdio.h to a:* [RETURN]
```

The copy program then loads the file `stdio.h` from drive a into memory. Once the file is loaded, the message

```
quit to save!
```

appears. Remove the system diskette and insert the formatted work diskette. Now press a key and the file `stdio.h` is written to the diskette.

```
quit to save! ok
```

When the prompt appears, the file has been copied.

Arguments for the `copy` command are separated by spaces. No spaces may be within an argument. A space in a filename must be replaced by a shifted space.

The first argument is the filename of the file to be copied. It is prefixed by a device identifier.

The second argument is the word `to`. It indicates the direction of the copy.

The third argument is the filename of the copied file. It is prefixed by a device identifier. An asterisk (*) may be substituted for filename and signifies that the filename is the same as the first argument.

For two disk drives systems:

Insert the system disk in a, the formatted work disk in b.
Enter:

```
a:copy a:stdio.h to b:* [RETURN]
```

The file is copied directly from drive a to drive b.

Now copy the following files onto the formatted work diskette using the procedures described above (one drive or two drive systems):

```
stdio.h           (which we already copied)
graphic.h
math.h
ctype.h
libc.l
libcs.l
libgraph.l
libmath.l
```

This procedure is necessary, many of these files will be used by the C compiler when you compile your programs.

Super C also works with dual disk drive systems. Here is an example using the copy program.

```
a:copy a0:stdio.h to a1:* [RETURN]
```

For directories use this form:

```
a:dir 1
a:dir 0
```

3. Version 3—RAM disk (C-128)

If you are using SUPER C Version 3, the CCP contains a RAM disk. The RAM disk uses device identifier `h`.

What is a RAM disk? A RAM disk is a simulated disk drive. Instead of storing data or programs on magnetic media a RAM disk stores the data or programs in memory. Keep in mind that the contents of a RAM disk are lost when the computer is turned off, but they can first be copied onto a normal diskette.

To display the directory of the RAM disk type:

```
a:h:dir [RETURN]
```

You'll notice that several files are already stored on the RAM disk. These are the files that you copied in the preceding section. These files are automatically copied to the RAM disk when SUPER C Version 3 is started.

Try using RAM disk with the commands `com` and `err`:

```
h:com r:stdio.x=stdio.h [RETURN]
```

This command changes the name of `stdio.h` to `stdio.x`.

```
h:com n:test [RETURN]
```

This command erases the contents of the RAM disk and gives it the name `test`. The RAM disk can be accessed just like a normal disk drive, by specifying the device identifier `h`:

Insert the system disk into drive `a` and enter the following command:

```
h:a:lram a:stdio.p [RETURN]
```

Another transient command, `lram`, copies the specified file to the RAM disk. The file `stdio.p` contains the list of files specified in the preceding section. `lram` loads them into the RAM disk.

You can also save the contents of the RAM disk using the command `sram`. This command groups the individual files on the RAM disk together and saves them in one file.

```
h:a:sram [RETURN]
```

If you have a one drive system, you must load the `lram` and `sram` commands without arguments from the system diskette. Then you can replace the diskette in drive a. Enter this command:

```
a:sram a:ram.p [RETURN]
```

The RAM disk is saved to the diskette with the name `ram.p`.

If you have a two drive system, you can enter the following:

```
h:a:sram b:ram.p [RETURN]
```

The system disk must be in drive a and the other disk in drive b. If the system diskette and the other diskette are reversed enter:

```
h:b:sram a:ram.p
```

Note: On Super C Version 3, the command `lram std.p`, is automatically executed during startup. This is why the prompt is red to begin with in Super C Version 3, `lram` is present as a resident command after booting.

4. C editor

The editor is one of the three system components. The editor is called `ce` and is loaded like a transient command:

```
a:ce [RETURN]
```

In Version 3, the editor can be used in either 40 or 80 column mode.

4.1 The new command

For practice let's enter a short document. You begin a new document with the editor command `new` (new document).

First press the `[F5]` key. This is the command key. The following message appears on the top screen line:

```
enter command
```

This means that the editor is waiting for you to enter an editor command.

Press the `[n]` key. The editor responds by displaying the command name on the first line of the screen:

```
new: length of line
```

The editor is now waiting for you to specify the maximum length of each document line. You can enter a value from 40 to 80. This line length cannot be changed later. When entering numbers, the editor accepts only the digits 0 to 9, the `[DEL]` and `[RETURN]` keys. `[DEL]` deletes the preceding character and `[RETURN]` ends the input. Enter the number 63 and then delete 63 by pressing `[DEL]` twice and enter 80. Then end the input by pressing `[RETURN]`. The following is displayed:

```
new: length of line 80
file:
```

The cursor is positioned after the word `file:` in the second screen line. The editor is waiting for you to enter the filename of the document. Enter:

```
new: length of line 80
file: testfile
```

[DEL] deletes the character preceding the cursor. The other control keys are inoperative. Suppose you want to enter `testfile` instead of `textfile`. You must [DEL] all the characters up to and including `x` and then re-type the remainder of the filename.

To terminate the input, press [RETURN].

```
new: length of line 80
file: testfile
---*---*---*---*---*---*---*---*---*---*
```

You have now opened a document with line length of 80 and a filename "testfile".

4.2 Inserting and deleting lines

The cursor is now positioned in the *text field*. This is the part of the screen below the top three status lines.

If you try press an alphabetic or numerical key, the message "last line" is displayed on the status line. The cursor advances but none of the characters are displayed.

```
last line
file testfile
---*---*---*---*---*---*---*---*---*---*
```

When a new document is opened, it contains only one line, a last line. You can't enter text in this line, nor can you move the cursor beyond this line. In both cases the editor responds "last line". To enter text into this document, you must insert additional lines before the last line.

Insert a few blank lines by pressing the [F7] key. The [F7] key is used for inserting lines. Press [F7] a total of six times.

Your text field is now 6 lines by 80 columns. You can position the cursor anywhere within this text field. Pressing a key enters the text at that location.

The [RETURN] key positions the cursor to the start of the next line. [SHIFT] + [RETURN] places the cursor at the end of the previous line.

Use the cursor keys and the ([SHIFT]) [RETURN] key to move around in the text field. Version 3 also allows the upper cursor keys to be used. On 40 column monitors you'll notice that the screen scrolls to the left when you move the cursor beyond column 20. If you then move the cursor left, the screen scrolls to the right. Since these screens can display only 40 character per line, the characters outside the screen are brought into view by scrolling.

If you are using Version 3 on the 80-column monitor the screen does not scroll.

Move the cursor to line one, column one. You can also use the control key [CLR] ([SHIFT] + [HOME]) to do this.

Now enter the following text. When typing, you can use the [DEL] and [INS] keys, which have the same function as in BASIC.

```
This is the first line of my document.  
Here is the second  
three  
4
```

Position your cursor somewhere in the second line and press the [F7] key (to insert blank lines). All characters from the cursor line to the end of the document are moved down one line and the cursor is positioned within in a blank line. Here you can insert text. If one line isn't enough, press the [F7] several times.

```

c-editor 1.0
file: testfile
----*-----*-----*-----*-----*-----*-----*
This is the first line of my document

Here is the second
three
4

```

To delete lines, use the key [F8] ([SHIFT]+[F7]). The line in which the cursor is positioned disappears and all subsequent lines are moved up. Press [F8] again and the second line of the document disappears. It is deleted and all subsequent lines are moved up.

```

c-editor 1.0
file: testfile
----*-----*-----*-----*-----*-----*-----*
This is the first line of my document.
three
4

```

The editor allows you to assign each line its own color. You set a color by pressing the color control keys ([CBM]+[1] to [CBM]+[8] or [CTRL]+[1] and [CTRL]+[8]). The line then takes on the color according to the color control key. This can be very helpful for beginners by allowing them to more easily see the structure of the program.

Place the cursor in line one and then press [CTRL]+[6]. The line becomes green. Move the cursor to line two and then press the keys [CBM]+[4]. The line becomes dark grey. The color of the last line cannot be changed and is always red. On 80 column monochrome monitors these commands will not work, use one color on monochrome monitors.

When inserting lines ([F7]) the color of the line moved down is used. If you press [F7] in line 2, the new blank line has the color dark grey.

If you want to do something like insert seven red lines between line

one and line two, you don't have to insert seven lines and then color these seven lines red, but just insert one line first. Set the color of this line to red and then insert the remaining six lines. These will have the color red as well.

You can now experiment a bit with colors and inserting and deleting lines before you go on to the next section. Version 3 owners who are using a monochrome monitor for 80 columns will not be able to use the color features. Since the color feature allows beginners to see the structure of the program very easily, we have used color in the examples on the system disk. On 80 column monochrome screens if a sample program from the system disk is missing some lines, they may be displayed in a color not visible on your monitor. Simply change the color to one that can be displayed on your monitor and save the new example program to your work disk.

4.3 Saving text

Now save the document. Insert a diskette into the drive. Press the command key [F5] followed by [s]. The command name `save` appears on the first line as the editor saves the document. After the document is saved, the cursor reappears in the text field.

Repeat the procedure ([F5], [s]). The following question appears behind the command name:

```
save: replace y/n?
```

In this case the file already exists on the diskette. The editor is asking you if the file should be replaced. If so press [y] for yes. If not, press [n] for no and you are returned to the text field.

4.4 Loading text

Now load a document.. Insert the system diskette into the drive. We'll load a sample program.

Press the command key [F5], followed by the [1] key for the command load. The command name load is displayed on the first line and the cursor is positioned after the "file:" prompt on the second line.

Enter the filename to be loaded:

```
load
file xxx text.c [RETURN]
```

The new document is loaded and replaces the previous document in memory.

After loading, the first page of the document is displayed and the cursor is positioned to line one, column one.

Move the cursor using the cursor keys through the document and examine the document, but don't change it. Try the right and left scrolling on 40 column displays. Since this document has enough lines, you can also test the up and down scrolling in which you move the cursor beyond the first or last line of the screen.

Use [RETURN] to get to the start of the next line or [SHIFT]+[RETURN] in order to jump to the end of the previous line.

You can also use two additional control keys in order to move the cursor through the document quickly. The [F1] key pages forward through the document. This always displays the next 22 lines. The [F2] key ([SHIFT]+[F1]) pages backwards through the document. Here 22 lines preceding the current cursor position are displayed on the screen.

4.5 Block commands

Block commands operate on multiple lines at a time. A block is a group of lines that belong together. When delimiting a block of text, the message marking out range always appears in the first line. This message indicates the block-input mode. During the block input, only a few of the control keys are active (cursor left, cursor right, cursor up, cursor down, [RETURN], [STOP]).

4.5.1 Delete block

To practice let's delete lines 64 to 95 of the sample document. We'll use the block command `erase`. You could also delete the lines individually with the control key [F8].

Move the cursor to line 64. You can determine the line number at which the cursor is positioned from status line. The first number indicates the column, the second the line. Press [F5] followed by [e]. The editor displays the command name `erase:` and the block-input mode.

```
erase: marking out range
```

In the block-input mode the lines which mark the block are displayed in reverse video. Use the keys [C-UP] and [C-DOWN] to vary the size of the block.

Enlarge the text block with the [C-DOWN] key until the reverse line field reaches to line 99. But since we want to erase only to line 95, decrease the size of the block with [C-UP] until the reverse block extends to line 95. After you have delimited the block press the [RETURN] key. A confirmation question now appears behind `erase:`

```
erase:are you sure y/n?
```

The [n] key terminates the command and returns you to the text input without erasing the block. The [y] key erases the block. Press the [y] key to delete the block.

You can exit the block input with the [STOP] key in case you don't want to delete it after all, or you marked it out wrong.

4.5.2 Move block

With the command move you can move text blocks to a different location in the document. In the sample document we will move lines 101 through 113 to line 58.

Move the cursor to line 101. Press [F5] followed by [m] for move block. As you can gather from the status line (marking out range), you are in the block-input mode. You must now mark the range which you want to move. This works the same as described in Section 4.5.1 Delete block.

After you have delimited the block, press the [RETURN] key. The the message "fixing target" appears after move: on the first line of the screen. Use the cursor keys to position the cursor to the destination line which is displayed in reverse. In a block which is already displayed in reverse, the destination line is displayed in normal video.

You now have to set the destination line where the block is to be inserted. You can use the following control keys for this:

[C-UP] and [C-DOWN] move the destination line up and down.

The [F1] and [F2] keys move the destination line 22 lines down or up, respectively.

The [g] key calls the command "goto". You enter a line number to which the destination line jumps.

[RETURN] ends the line input. The destination line may not lie within the previously marked block. The editor will otherwise respond "no target line" and will not end the destination input.

Move the destination line to line 58 with the control keys and end the input with [RETURN].

The marked block is moved and the document is displayed starting at the inserted block. If you move the cursor up one line you will see the line before the destination line.

You can also use the `goto` command by pressing the [F5] key followed by [g]. The command name `goto` appears in the first line. Enter a number of up to four places (9999) at the prompt. The cursor moves to the specified line, but not beyond the last line.

4.5.3 Copy block

Copying (transferring) a block is very similar to moving a block. In contrast to moving, a copy of the marked block is inserted before the destination line. This allows you to duplicate the block at another location within the document.

The `transfer` command is used to copy blocks. It works identically to the `move` command, with one addition. When setting the destination, the [HOME] key is enabled.

[HOME] causes the text area to be switched into the *extra text area*. The editor has two separate text areas, the file text area, where you have worked so far, and the extra text area. You can distinguish between the two text areas by means of the second screen line. For the file text area, "file:" and the filename is displayed, while "extra text" is displayed in the extra text area.

While you are editing the actual document in the file text area, text should be stored only temporarily in the extra text area. You can edit the extra text just like the file text, except that some commands are not allowed in the extra text area. Using them causes the editor to display the message "illegal text."

These commands include `load` and `save`. Change the text area once and try to use the `load` command ([HOME] [F5] [1]). The editor displays the "illegal text" error.

```
illegal text
extra text
---*---*---*---*---*---*---*---*---*---*
```

Back to the transfer command. You can transfer text from one area to the other by changing the text area with the [HOME] key while setting the destination.

For practice, transfer a block of text to the extra text area. Press [F5] followed by [t] for transfer. Then mark a block of text. While setting the destination, press the [HOME] key to enter the extra text area. Since the extra text area contains only one line at this time, the last line you cannot move the destination line within this area.

As soon as you have set the destination by pressing [RETURN], the text block is copied to the extra text area.

With the extra text area you can insert text blocks into other files, for instance. If you switch back to the normal text area by pressing [HOME], you can load a different text file and insert the extra text in the file text area with the transfer command.

You will learn another very useful application of the extra text area in Chapter 5 of the tutorial section.

4.6 Search and replace

The editor allows you to search for strings. It also allows you to replace strings.

4.6.1 Searching

The search string is specified with the hunt command. Press [F5] followed by [h]. The command name hunt: is displayed and prompts you to enter a search string.

If you make a mistake while typing the search string you can delete the character with the [DEL] key. When you have entered the complete search string press [RETURN]. For this example enter:

```
hunt: violet [RETURN]
```


Now press the [F3] key to begin the search. The cursor will move through the text and stop at the first character of the matching string.

Press [CLR] to jump to the start of the document and search with [F3]. The cursor will move to line 14 since it found a match for "violet". Press [F3] again. Since there are no more occurrences of "violet" the last line is displayed.

4.6.2 Replacing

To replace a string, you must specify both a search string and a replacement string. Press [F5] followed by [r] for replace.

The prompt `hunt :` is displayed. Enter the search string followed by [RETURN]. Next the prompt `rplc :` is displayed. Enter the replacement string followed by [RETURN].

Enter the following:

```
hunt: violet [RETURN]
```

```
rplc: purple [RETURN]
```

Here we want the string "violet" to be replaced by the string "purple".

Replace can be performed in one of two ways:

- 1) automatic replace
- 2) ask-before-replace

In automatic replace, all occurrences of the search string are replaced by the replacement string.

In the ask-before-replace, the editor prompts you before making any replacements.

To automatically replace all occurrences of the search string press the [F6] key.

To ask-before-replace press the [F4] key. In this case, the editor searches for the first occurrence of "violet" and displays it. You are prompted.

```
replace?y/n
```

Press the [y] key to replace the string. Press [n] to skip to the next occurrence. You can stop this function by pressing the [STOP] key at any time. The cursor then appears at the location in the document that the editor had reached while searching.

If during the replacement the maximum line length is exceeded by a replace, the editor terminates the operation without replacing the string at that location. The cursor is located at the first character of the occurrence and the error message "overflow in line" appears in the status line. You must then decide for yourself how to change the text.

For practice you can try to create an overflow-in-line error. You must choose a replacement string which is longer than the search string in order to do this.

4.7 Back to the CCP

You can terminate all procedures (searching, all inputs and questions, loading, printing, etc.), with the exception of saving, with the [STOP] key. Moreover, the functions listed above are only the most useful; you can find a complete description of all the functions in the reference section in Part II Chapter 3.

To exit the editor, press [F5] followed by [x]. The prompt are you sure y/n? asks you to confirm this. Press [n] to return to editing.

Press [y] to exit to the CCP. Make sure that you've saved your document before exiting the editor.

5. Your first C program

In this chapter we'll show you how to create a C program using the Super C Package. We'll show you how to edit a program, compile and link the program and finally run it.

When you use the Super C Package you will create three different files: the *source* file, the *link* file, and the *program* file. Using the C-editor, you create and modify the source file. The C-compiler converts this source file into an intermediate link file. Finally the C-linker combines this link file with any others required link files to produce an executable program file.

To differentiate among these three types of files, we recommend that you use filename extensions. In this manual, the following conventions are used:

<u>file type</u>	<u>extension</u>	<u>example</u>
source file	.s	test.s
link file	.o	test.o
program file	-	test

The following describes two different methods to develop a program using Super C.

In the first example, we assume that you are using Super C Version 3 and one disk drive. This version automatically gives you a second disk drive—a RAM disk.

In the second example, we assume that you are using Super C Version 2 and only one disk drive.

In either case, if you are using more than one real disk drive, you can change the examples to use the added drives by specifying the required device identifier.

Start the C system as described in Chapter 1. Once you are in the CCP, load the editor by typing:

```
a:ce
```

The Super C editor will now be loaded and started.

5.1 Editing

The C-editor is used to create and modify the program text. Chapter 4 described how to use the editor.

Since you are just learning how to use Super C, we won't have you edit a source file here. We have already prepared a source file for you named `sample.c` contained on the system disk. Load this source file into the editor by typing:

```
[F5] [1] sample.c [RETURN]
```

Let's imagine that you had entered the text for this source file with the C-editor and now want to compile it. It's not possible to change the diskette during compilation. Therefore all source files to be compiled must be on disks which can be in the drives at the same time. This means that if you have only one drive, all source files must be on one diskette. For Version 3, you can divide the source files between the RAM disk (h:) and the real drive. If you have more than one drive you can divide the files between several diskettes.

Source files may not be saved on the system disk. Therefore you must load the C compiler before compilation. To avoid changing diskettes when moving between the editor, compiler, and linker, you must reserve one disk drive for the system disk.

In the first example, the system diskette is in drive a: and in version V3 the source files are saved on the RAM disk h:.

For Version V3 (C-128)

Look at the text of the example program. At the start of the text there are some lines which look like this:

```
#include "stdio.h"
```

Change the text so that device identifier h: is in front of `stdio.h`:

```
#include "h:stdio.h"
```

The `#include` directive tells the compiler to insert the specified file into the text beginning at that location. Recall that the file `stdio.h` was copied to the RAM disk after booting up. The device identifier `h:` tells the compiler that this file is to be read from the RAM disk.

Save the sample program to the RAM disk. To do this, change the file name to include the device identifier to `h:`. Press the keys `[F5] [f]` to enter the file command. Then enter the new filename:

```
h:sample.c [RETURN].
```

Finally save the new source file with the save command:

```
[F5] [s]
```

For Version V2 (C-64)

The same method applies for version V2. You do not have to change the filename in the statement since the `include` file is loaded from drive `a:`.

Insert the previously created work diskette into drive `a:` and save the example program with the save command:

```
[F5] [s]
```

Make sure that all source files necessary for compilation are on the work diskette or the RAM diskette. You can use the editor command to display the diskette's directory. Remember to switch to the extra text area with `[HOME]` before loading the directory so that you do not overwrite the source file.

Then enter `[F5] [d] [a] [:] [RETURN]`. The `d` specifies directory and `a:` specifies drive `a:`. The directory is then loaded into the extra text area. The contents of the RAM disk is displayed by using `h:` instead of `a:`.

In Version V2 the files `stdio.h` and `sample.c` should be on the work diskette, or on the RAM disk for Version V3.

Normally both source files must be present. The file `stdio.h` was copied to the work diskette in Chapter 1.

Exit the editor with [F5] [x].

5.2 Compiling

The C-compiler is called `cc` and is loaded by typing [cc] [RETURN]. The system diskette must be in the disk drive. The compiler displays the compiler header message and asks you to enter the filename of the source file. For Version V2, replace the system diskette with the work diskette. Then enter `sample.c` as the source program name.

When entering file name into the compiler, only the control keys [DEL] and [CLR] ([SHIFT]+[HOME]) as well as [RETURN] are active. [DEL] clears the previous character and [CLR] deletes the entire input field. [RETURN] ends the input.

```
source file name: sample.c [RETURN]
```

Enter `h:sample.c` for Version 3.

Now you are asked to enter the link file name. The output of the compilation is written to this link file. If the source file input ends with `.c`, the link file defaults to the file name with the extension `.o`. We recommend that you use the same link file name, so the various link files do not build up on the diskette.

Clear the input field with [CLR] ([SHIFT]+[HOME]) and enter a name (such as `o.o`):

```
link file name: a:o.o [RETURN]
```

For Version 3 enter `h:` instead of `a:` because the link file is stored on the RAM disk.

Next enter the name of the error file. In this file all of the compiler error messages are stored. The default name is `error.e`, which is preceded with the device identifier of the source text.

```
error file name: a:error.e [RETURN]
```

In Version 3 the device identifier is set to h:.

Now the compiler has all the information which it needs and starts to compile the program.

In grey type the compiler prints the source files which it is processing, h:sample.c and h:stdio.h. This means that it is currently reading the corresponding source file. The yellow message `getchar` indicates that the compiler is compiling the function `getchar`. After this follows a grey `#` character. This shows that the compiler is finished reading the source file `stdio.h` and is continuing to read in `sample.c` at the line after the one in which `stdio.h` was called. Then the compiler outputs `main` in yellow, then some error messages appear in red. These errors were not caused by anything you did, they are intentional.

The compiler finishes the compilation and prints the concluding message:

```
compiling finished
linkfile not available
press x to quit, r to restart
```

Press the `[x]` key to return to the CCP. With `[r]` you could restart the compiler to compile a different source file.

If there are errors during compilation or if you have mistakenly entered incorrect parameters (such as source file or link file), you can stop the compiler at any time by pressing `[STOP]+[RESTORE]`. This produces a non-maskable interrupt (NMI).

This compilation has a few errors, therefore you must return to the editor to correct them. Reload the C-editor (insert the system diskette first).

Once you are in the editor, insert your work diskette (Version V2 only) and load the file `error.e` from the work diskette or from the RAM disk, as appropriate. All of the error and status messages

are contained in this file. Copy this text into the extra text area. To do this, use the transfer command ([F5] [t]) and mark the entire document as a block. After you have ended the block input with [RETURN], set the destination. Put the destination line in the extra text with the [HOME] key. End the destination input with [RETURN] and the file `error.e` is now moved to the extra text area. Then load the source file `sample.c` into the file text area.

Now you can correct the source text with the help of the error messages in the extra text area. Each error has the line number in which the error occurred. The extra text contains the following errors:

```
?expression syntax error in 0013
?statement syntax error in 0022
?declaration syntax error in 0036
```

Lines 13 and 22 are displayed in red and contain errors. To eliminate these errors, delete these lines. The error in line 36 results from the errors in lines 13 and 22 and disappears once they are removed.

Once you have corrected the errors, save the modified source file ([F5] [s]). The message `replace y/n?` appears. Since the file already exists the editor asks if it should be replaced. Press [y] so that the old file is replaced by the corrected file.

You now have a corrected source file and can compile it again. Exit the editor and start the compiler with `cc [RETURN]`.

```
source file name: h:sample.c [RETURN]
link file name: h:o.o [RETURN]
error file name: h:error.e [RETURN]
```

For Version 2, you must use `a :` as the device identifier.

If a link file with the same name already exists, it is overwritten. This time the compiler runs through the source text without error. The termination message this time is:


```
compiling finished
linkfile available
press x to quit, r to restart
```

The link file is now available. You can therefore proceed to the linking. Exit the compiler with the [x] key and return to the command processor.

5.3 Linking

Several functions are used in the compiled sample program which must be explicitly made available under C. To make these available, the link file of the sample program must be linked to the library(ies) containing these functions. A library is a link file which contains the appropriate functions already compiled.

Before you start linking, check if all the link files which you want to link are on one diskette. The library is called `libc.l`. It should be on your work diskette. It was copied to this diskette in Chapter 1. If it is not there, you must copy the file onto your work diskette with the `copy` command.

If both link files, `o.o` and `libc.l`, are now on the work diskette or the RAM disk, load the loader (insert the system disk first). The linker is called `cl` and is loaded with [c1] [RETURN].

The linker displays the linker header and requests you to enter the final program filename. First insert your work diskette if you are working with Version V2. Enter the name of the sample program without the `.c` extension:

```
program file name: h:sample [RETURN]
```

The device identifier indicates where the file is to be saved, on Version V2 use `a:` instead of `h:`.

The linker then asks you to enter the name of the required link files. Enter the two link files, `libc.l` and `o.o`. The first input contains `libc.l`, but without a device identifier. This means that you can accept the default only if `libc.l` is on drive `a`. In Version V3 the

default already has the device identifier h: because this file is usually on the RAM disk.

You can use the [DEL], [CLR], and [RETURN] keys for editing.

The [DEL] key deletes the previous character, [CLR] clears the entire input field and [RETURN] ends the input.

The order of the link files is not important. You can change the order around as you like, the same C program always result. A good habit is to enter the libraries first.

```
link file h:libc.l [RETURN]
link file h:o.o [RETURN]
link file [RETURN]
```

After the two link files are entered, press [RETURN]. This ends the link file input.

After this you can set the upper limit of the C program memory. If you don't need any memory for other applications, you can accept the default. The maximum C program memory available is 50K for V2 and 51K for V3.

```
memory top page $e9 [RETURN]
```

The default here is \$d0 for Version 2.

Next you are asked to enter either c or b (this happens only in Version V2.) You can accept the default letter c. The c means that the C program can be started only from the CCP. The linker option b creates a C program which can be started from BASIC. We designate a program as a C-version if it is to be started from the CCP and as a B-version if it is to be started from BASIC. The C-version has the advantage that you don't have to exit the C system in order to run the program. The B-version has the advantage that the Super C system doesn't have to be loaded to run the program. This option is not available in Version 3.

```
(c=ccp/b=basic) c [RETURN]
```

Now the linker begins to link the files. Status messages are printed in grey, errors in red. The linker requires two passes. The start and end of each pass is displayed by the linker. In addition the link file from which the linker is currently reading is indicated in yellow.

Here is an example of what the linker displays in Version 3.

```
program file name: h:sample [RETURN]
```

```
link file h:libc.l [RETURN]
```

```
link file h:o.o [RETURN]
```

```
link file [RETURN]
```

```
memory top page $e9 [RETURN]
```

```
pass 1
```

```
link file h:libc.l
```

```
link file h:o.o
```

```
end of pass 1
```

```
pass 2
```

```
link file h:libc.l
```

```
link file h:o.o
```

```
end of pass 2
```

If the linking was error free, the concluding message reads:

```
linking finished
```

```
program file available
```

```
press x to quit, r to restart
```

Exit the linker with the [x] key. With [r] you could start the linker again.

A source listing of the sample.c program is found in the appendix.

5.4 Executing

The file `sample` contains the finished C program. To start this program type:

```
a:sample [RETURN]
```

The CCP loads the C program `sample` and starts it automatically.

With version V3 you compiled the program to the RAM disk so you would naturally have to specify the device identifier `h:` to load the program with Version 3. It is a good idea to copy the finished program from the RAM disk to a diskette at this time.

Once the program is started, it clears the screen and waits until a key is pressed. It then displays the message "Character: ". When you press a key the program displays the `char` constant of the pressed key in the C notation. If you want to designate this key in a C program, you can use this `char` constant value. In the next line the ASCII value of the key in decimal, hexadecimal, and octal is displayed. These numbers are also represented in C notation.

The hexadecimal numbers have a leading `0x` (or `0X`), the octal numbers have a leading zero. The decimal numbers are in normal notation, but may not have a leading zero because they will then be interpreted as octal numbers.

Since the program consists of an infinite loop, you can end it only by pressing `[STOP]+[RESTORE]`. The following messages appear when these keys are pressed:

```
?nmi interrupt
press x to quit, c to continue,
r to restart
```

With `[x]` you can end the program, with `[c]` you can continue the program, and with `[r]` you can restart it. Try these three options. The two options `[r]` and `[c]` have some peculiarities which we will explain in detail in the reference section (Part II Chapter 6 and Part II Chapter 7.2). Strange results can occur if the `[STOP]+[RESTORE]` are pressed during an output operation.

6. Introduction to C

In the last chapters you were introduced to program development with the SUPER C language compiler. In this chapter you will become better acquainted with the C language. We will use some example programs which you can and should enter, since a programming language is best learned through examples.

This chapter is only an introduction, more detailed and specific information can be gathered from the C language description in the reference section (Part II Chapter 8).

Experienced C programmers can skip this introduction and continue with Part II Chapter 1 of the reference section.

6.1 Overview

6.1.1 The first program

The first program which you should enter looks like this:

```
#include "stdio.h"

main()
{
    printf("\nYour first\nprogram\n");
    getchar();
}
```

Compile this program and link it with the library `libc.l`. Run the finished program and you should see:

```
Your first
program
```

This text remains until you press a key. You have now seen what the program does.

The first line of the source file contains an `include` command (`#include <stdio.h>`) which makes it possible to use the standard input output functions from `libc.1`. If this line is in the program, you must always link the `libc.1` module to the program.

The remainder of the program is a function definition. A C program consists of functions. The function `main` is the primary function. This function is called when executing a C program. With the end of this function, the program also ends.

`main ()` is called a function header. It tells the compiler that a function with the name `main` will be defined. The instructions which are to be executed in the function are enclosed in braces (`{,}`). This is called a block. The braces are similar to `BEGIN` and `END` in Pascal.

The function block contains the instructions which are to be executed when the function is called. In our case there are two instructions in the block. In the first instruction, a function by the name of `printf` is called and then a function called `getchar`. Both functions are found in `libc.1`.

You can pass data to a function for it to process. These data are called arguments. `printf` requires such an argument. The argument for `printf` is a string. `printf` outputs this string on the screen.

You probably noticed the characters `\n`, which were not printed, in the string. The `\` is called the escape character. A letter may follow it which together with the escape character represents a character. `\n` represents the carriage return and causes the next output to appear at the start of the next line.

Calling the function `getchar()` requires no parameters and causes the computer to wait for a key press. That is execution does not leave the function `getchar` and return to the function `main` until a key is pressed. There the end of the block is reached (`}`) and the program is finished.

Most instructions are terminated with a semicolon. This also applies to the last instruction in a block (in contrast to Pascal).

6.1.2 Objects

An object is a storage (memory) area used in a program. Data can be stored in this storage area. Such objects must be first created before they can be used. To do this you use declarations. A declaration which creates an object is also called a definition. The object is assigned a type and a storage class by the declaration. But the most important thing is that the object receives a name through the definition. With this name it can be accessed in the program.

The type of an object determines the length and the interpretation of its contents.

Here is our next example where we define various objects.

```
#include "stdio.h"

main()
{
    double e,pi;
    int a,b;
    e =2.7182818;
    pi=3.14159265358973;
    a =2;
    b =4;
    print ("e=%g\npi=%g\n",e,pi);
    print ("a=%d\nb=%d\n",a,b);
    getchar();
}
```

This is a very simple C program which outputs the following:

```
e = 2.7182818
pi= 3.14159265358973
a = 2
b = 4
```

The first lines of the program should be familiar to you. `main() { ... }` defines the main function. Within a block you can make declarations. These must come at the beginning of the block:

```
double e, pi;
```

declares two objects of type *double*. The two objects have the names `e` and `pi`. The type *double* indicates that floating-point numbers with double precision can be stored in this object. In this case up to 11 places can be stored.

If you want to define several objects of the same type, they can be separated with commas. A declaration is, like most other instructions, terminated by a semicolon.

```
int a, b;
```

is a similar definition. Here two objects, `a` and `b`, are defined as integers by `int`. Only whole numbers (integers) can be stored in objects of this type.

The next four program lines are instructions in which the defined objects are assigned values. The object identifier must always stand on the left side of the equals sign. Such an identifier is called an *lvalue*. To the right of the equals sign is the value which is to be stored in the object. In the four program lines, all objects are assigned the right number value.

The last lines of the program contain instructions which make calls to the functions `printf` and `getchar`.

`printf` now has more than one argument, however. The first argument is always a character string.

```
"e=%g\npi=%g\n"
```

All the characters are printed up to the `%g` characters. These are called format instructions. They cause another argument of `printf` to be printed. `%g` requires an argument of type *double*. The value of this argument is printed as text. The `\n` character causes the next output to appear at the start of the next line.

The second `printf` instruction is constructed similarly. Here the format character is `%d`, which requires an argument of type `int` and prints it in decimal.

`getchar` waits for a key press before the program is ended.

You have now been introduced to objects. The objects in this example were all defined without specification of a memory class. You will later see what consequences this has. We will say only that these objects exist only within the block in which they were defined.

The number values which occur are constants. The floating-point constants are always of type `double`. Integer constants are of type `int`, as long as they are not too large.

You do not necessarily specify constants on the right of an assignment. On the right side there can be an identifier or a complicated expression.

```
pi=e;
```

6.1.3 Loops

Up to now our example programs have been processed sequentially, meaning that the individual instructions were always executed in order, then after the last instruction the function and the program was ended. In the many applications this is not satisfactory. This is why there are *loop* instructions, which make it possible to repeat certain instructions.

The next example is a program which prints a table of Celsius and Fahrenheit degrees.

```
#include "stdio.h"
/* Table of Celsius to Fahrenheit
   for c=-50,-40, ..., 50 */

main()
{ int start,end,step;
```

```
double fahr, celsius;
start=-50;
end=50;
step=10;

celsius=start;
while(celsius<=end)
{   fahr=(9.0/5.0)*celsius+32.0;
    printf("%4.0f %7.1f\n",celsius, fahr);
    celsius=celsius+step;
}
getchar();
}
```

The compiler ignores everything between the `/*` and `*/` characters. Between these characters are the comments for the program, use comments liberally in all your programs.

A set of objects are defined. `start` and `end` represent the first and last numbers in the table. `step` specifies the step width with which the Fahrenheit degrees will be calculated. `celsius` represents the current Celsius value, `fahr` the current Fahrenheit value. `start`, `end`, and `step` are then assigned the required values. `celsius` is initialized with the value of `start`. `celsius` now has the value for the first conversion.

Next is the `while` instruction. A condition enclosed in parentheses must follow `while`. In this case a comparison is made to see if `celsius` is less than or equal to (`<=`) `start`. If this condition is fulfilled, the body of the loop is executed. This is a block in this case. In this block there are instructions which are to be repeated as long as the condition is fulfilled. In this case the loop is executed until `celsius` is larger than `end`. Then the end of the table has been reached.

At the beginning of the loop the Fahrenheit value is converted to the Celsius value. On the right side of the assignment there is a complicated expression which performs the conversion. The Celsius and Fahrenheit values are printed opposite each other with the `printf` function.

At the end of the loop the Celsius value is incremented by the step width. The program tests to see if the loop condition is still true. If so, the loop is repeated. If the condition is no longer true, program execution continues after the loop instruction, behind the loop block.

Now to some program details.

```
celsius=lower;
```

Here the value of `lower`, an `int` value, is assigned to the object `celsius`. The value of `lower` is automatically converted to type `double`. For each C assignment the right side is always adapted to the type of the left side. When possible, the numerical value remains the same.

In the conversion formula we see the division `9.0/5.0`. Here `double` constants are used. If we used `int` constants and write `9/5`, the result would be 1 because integer division would have been performed. If you want division by a `double` value, at least one of the operators must be of this type.

The format instruction of `printf` has been changed somewhat. `%f` means that a `double` number without exponent will be printed. Numbers can be placed between the `%` and `f`. `%4.0f` means that the `double` number will be printed with a text of at least four characters, with zero places after the decimal. The decimal point and the sign of the number must be taken into account when calculating the minimal text width. With this format instruction, only numbers of up to 2 digits can be printed. If the numbers are larger, the field becomes larger than 4 characters and this destroys the output format. If the number is smaller, the text is filled with spaces until it is 4 characters wide. The instruction `%7.1f` specifies a `double` number without exponent with an output width of at least seven characters and one place after the decimal.

6.1.4 Symbolic constants

The previous conversion program can be easily rewritten for other values. But imagine a considerably more complex program. Changing all of the constants would be a great deal of work and would also be a source of errors if a constant were forgotten. To avoid this, modern programming languages have symbolic constants. A name is defined as a constant. Wherever this name appears in the program it is replaced by the constant it was defined as.

Our next example uses symbolic constants:

```
#include "stdio.h"
#define START (-50)
#define END 50
#define STEP 10

main()
{ double celsius, fahr;

  for(celsius=START; celsius<=END;
      celsius+=STEP)
  { fahr=(9.0/5.0)*celsius+32.0;
    printf("%4.0f %7.1f\n",celsius,fahr);
  }
  getchar();
}
```

This program produces the same result, but it looks quite different. The variables `start`, `end`, and `step` are missing. Constants were defined for them instead. This is done with the command `#define`. This command must always be at the start of a line. After such a definition the specified name can be used like the constant following it.

The `while` instruction was replaced by a `for` instruction. After `for` are three expressions in parentheses. The first expression corresponds to the initialization of the loop, the second represents the loop condition, and the third is the continuation of the loop. This continuation is executed every time the body of the loop has ended and before the condition is tested.

The only unknown element for you may be the += operator.

```
celsius+=STEP           corresponds to  
celsius=celsius+STEP
```

`celsius` need be evaluated only once, however, which means that the assignment is performed faster.

6.1.5 Arrays

In this section you will become acquainted with arrays. Let's take a look at the following program:

```
#include "stdio.h"  
  
main()  
{ static int numbers[10];  
  int i;  
  char c;  
  
  for(i=0; i<50; i++)  
  {   c=getchar();  
      if(c<='9' && c>='0')  
          numbers[c-'0']++;  
  }  
  for(i=0; i<10; i++)  
  printf("Digit%d:%d times\n",i,numbers[i]);  
  getchar();  
}
```

This program defines `numbers` as an array with ten elements. The elements have type `int`. The number of times a certain key is pressed will be counted in these elements. The key word `static` stands before the declaration. It represents a memory class. Here `static` is used because the objects of this memory class are automatically set equal to zero, that is, the array contains only values zero at the start of the program.

In C, array elements are counted starting with zero, meaning that a ten-element array has elements 0 to 9. In these we will count how many times the digit keys 0 to 9 are pressed.

As temporary storage an `int` object `i` and a `char` object `c` are defined. The type `char` creates objects which can accept one character from the character set.

The first instruction creates a loop. In it the variable `i` runs from 0 to 49. `i++` is the continuation of the loop. This expression has the same effect as `i+=1` (`i` is incremented by one).

In the loop body there is a block with two instructions. First the function `getchar` is called, which waits for a key to be pressed. It not only waits, it also returns the code of the pressed key. This value is represented by the function call. Here the value is assigned to the object `c` (`getchar(c)`).

Next is an `if` instruction. Its body is executed only if the condition after `if` is true.

'0' and '9' are character constants, the values of which equal the code of the enclosed character `c`. This code can be different from computer to computer. The C-64 and C-128 used a modified ASCII character set. The digits are coded in order, however.

The `if` condition checks to see if the character read has a code less than or equal to the code of the character '9' and if the code is greater than or equal to the code of '0'. The two conditions are combined with a `&&` operator, which makes the whole condition true only if both individual conditions are fulfilled (logical AND).

Since the codes for the digits are in increasing order, the condition is fulfilled only for characters which are digits. In this case

```
numbers[c-'0']++
```

is executed. `c-'0'` returns the digit as the value, for '0' the value 0 and for '9' the value 9. The array is indexed with this value, meaning that the element with the number `c-'0'` is selected.

This element is incremented through `++`. The corresponding array element is incremented for each digit key.

The `for` loop is executed 50 times, which means that you must press 50 keys before the loop will be exited.

The next instruction is again a `for` loop, which prints a list. `i` runs through the values 0 to 9, and something like the following will be printed:

```
Digit 0: 2 times
Digit 1: 15 times
etc.
```

The last `getchar` waits for a key and the table is displayed until then. Once you have started the program you must press keys. After 50 keys the table appears indicating how often you pressed the digit keys.

6.1.6 Character arrays

If you program with BASIC you are acquainted with character strings. But they were really only string constants, that is, strings with predetermined sequences of characters. There is no type for changeable character strings in C. Strings are stored in arrays of type `char`. The result of this is that the length of the string is limited by the length of the array, but only by the length of the array. The end of a string in an array is designated by a character with the code zero. This end code is created by the compiler for string constants.

```
#include "stdio.h"

main()
{   char name[41];

    gets(name, 40);
    printf("\n%s\n", name);
    getchar();
}
```

In this program a character array with 41 elements is defined. Since one character is required for the end of the string, you can store up to 40 characters. `gets` is a standard function for reading strings. The first argument is the name of an array in which the string is to be stored. The second argument specifies the maximum number of characters to be read. The function causes the cursor to appear on the screen and allows you to enter a string. The input works like BASIC, that is, you must end it with `[RETURN]`.

In the function `printf` the control character `%s` expects an array name as an argument. The string in the array is printed as text, in this case, what you had entered.

`getchar` is again used to wait for a key so that you can view the output.

6.2 Expressions and declarations

6.2.1 Names

The names which are connected to objects through declarations may not match any C key words. These are reserved names which have a certain meaning in the program text, such as `int` as a type name.

A name must start with a letter. After the first letter may come digits. The underline character `_` counts as a letter.

You should choose variables names which suggest the purpose and contents of the variable and which are sufficiently unique so that a minor typing error does not result in a different valid variable name.

6.2.2 Types

In C there are a group of simple data types. The simple types are in contrast to the more complex types like arrays.

You have already become acquainted with the data types `char`, `int`, and `double`. In addition there are the following:

`float` is like `double`, but with lower precision. In the Super C system this type has a precision of 6 digits.

`short int`, also abbreviated to `short`, can store only whole numbers like `int`.

`long int`, also abbreviated to `long`, can also store only integers. The three integer types differ only in their value range, that is, the size of the largest representable number. The value range of `short` is guaranteed to be less than or equal to that of `int` and the value range of `long` is guaranteed to be greater than or equal to that of `int`. In Super C `short` and `int` are the same size while `long` is twice as large and requires twice as much memory.

The memory required is also called `SIZE`.

All integer types (including `char`) can be represented without a sign by placing `unsigned` before the type name. The contents of such as object are then interpreted without a sign as positive. `unsigned int` can also be abbreviated to `unsigned`.

6.2.3 Constants

You have already used `int` and `double` constants. The compiler recognizes a `double` constant by a decimal point and/or an exponent in the constant. An exponent is designated by the letter `e` or `E` and the corresponding exponent.

$$1e5 = 100000.0 = 100E3 = 1E+5 = 0.1e6$$

All of these constants have the same value.

`int` constants are integers. If you exceed the value 32767, they can no longer be stored in objects of type `int` (this can be different for other C compilers). In this case the constant becomes type `long`. If you wish to make an integer constant `long`, this can be done by placing `l` or `L` after it.

$$15L \quad 21 \quad 0L \quad 40000$$

n:name[,id] (new)

The contents of the RAM disk are erased and a new RAM disk with the disk name `name` will be created. You can specify the `id`, but it has no significance. Before you give this command, be sure that the right device identifier precedes `com` so that you erase the correct disk.

b-a: dr tt ss (block-allocate)

`dr` stands for drive; here you can specify only 0. `tt` designates the track and `ss` the sector of the block which is to be allocated.

b-f: dr tt ss (block-free)

This command frees the specified block. The arguments are the same as for `b-a`.

b-r ch dr tt se (block read)

`ch` designates the channel which was opened for the direct access. `dr`, `tt`, `se` select the block which will be loaded into the buffer of channel `ch`.

b-w ch dr tt se (block write)

Writes the buffer to the specified block.

b-p ch ps (block pointer)

The block pointer of channel `ch` will be set to the value `ps`.

u1 ch dr tr se

Modified read (like `b-r`, but the first block byte will not be taken as a pointer).

u2 ch dr tr se

Modified write (like `b-w`, but the first block byte will not be taken as a pointer).

3.0 The C editor

The C editor has the name `ce` and is loaded like a transient command.

In version V3 there are actually two editors, `e4` and `e8`, one for the 40-column screen and one for the 80-column screen. `ce` loads the right editor for the current screen. Both editors are completely compatible. `e8` can display 70-columns per line at the same time so that text is not shifted left and right as it is on the other version.

An argument can be passed to the editor. The argument is a filename with device identifier. This file will be loaded after the editor is started.

```
a:ce b:test.c
```

3.1 Character sets and text display

The editor has two text areas, a file text area in which you edit the C source text, and an extra text area in order to store text temporarily. There are about 43K bytes available for both text area together.

In contrast to BASIC, the cursor in the C editor does not blink and the repeat function works for all keys. The keyboard layout has been changed slightly from BASIC so that you can enter C-specific characters. The editor is in the C character set. The keyboard layout is shown in the appendix.

Super C V2

The keys `[SHIFT]+[CBM]` can be used in V2 to switch between the C character set and the CBM lower case set.

Super C V3

The keys [ESC] [1] enable the C character set and [ESC] [2] enables the Commodore character set. Within the Commodore set [SHIFT]+[CBM] can be used to switch between the upper and lower case sets. The switch mechanisms are the same as in the CCP.

The screen of the C editor indicates the cursor position in the first line. The first number shows the column, the second the line, in which the cursor is found. Messages and errors are displayed in the first lines and the command names are shown.

The second screen line contains the filename in the file text and the message extra in the extra text. The condition of the clock is displayed on the right side. The current clock time can be set in the CCP with the command set.

The third line indicates the tabs. * means that the tab is set. The remaining lines, 4 to 25, contain the actual text field.

A document consists of individual lines which have a set maximum length (40-80 characters). If the line length is greater than 40 characters, the remaining characters outside the screen are displayed by horizontal scrolling of the screen (not necessary in e8). Each line of a document has its own color, which you can set with the color keys ([CBM]+[1] to [CBM]+[8] and [CTRL]+[1] to [CTRL]+[8]). The last line of a document cannot be written. It makes it possible to insert additional lines. If you try to move the cursor beyond this line or to write on it, the editor responds last line.

The document which you enter is immediately stored, without you having to press the [RETURN] key.

If an operation would make the document too long so that it would no longer fit in the available memory, the operation will not be performed and the editor will respond overflow.

There are control characters and commands available for editing a document. Control characters are available during text input with only one key press. The commands, on the hand, are more complex editor functions which usually require additional

parameters. Commands are preceded by the command key [F5]. After this you select a command by entering the corresponding key for the command. Parameter inputs may follow the command. There are five input types for parameters. These five input types will be discussed in the following sections.

3.2 Control keys

⇔ The cursor is moved with the cursor left/right keys, moving the screen left or right. The cursor stops at the end of a line.

↑↓ The cursor moves up or down and the screen scrolls as required.

[RETURN]

The cursor jumps to the start of the next line.

[SHIFT+RETURN]

The cursor jumps to the end of the previous line.

[TAB] ⇐ (left-arrow) The cursor jumps to the next tab position (*).

[SHIFT+TAB]

([SHIFT] and the left-arrow key) The tab marker in the column in which the cursor is currently found changes (set or cleared).

F1 Page down. The text at the 22nd line after the cursor line is displayed.

F2 Page up. The text at the 22nd line before the cursor line is displayed.

F3 Search for text beginning at the current cursor position.

A search is made for the previously-defined search string see command `r = replace` for the input of the search text). The editor looks for the search string after the F3 key is pressed. This can take up to two seconds for long strings. If the editor finds an occurrence, the string is displayed with the cursor at the first character of the string.

The editor jumps from one occurrence to the next with each subsequent press of the F3 key. If no more occurrences are found, the editor displays the **last line** of the document.

The search process can be stopped with the [STOP] key. The cursor is positioned to the line and column at which the search had advanced to.

F4 Replace with query. The next occurrence of the search string is searched for and displayed in reverse. The question `replace y/n?` appears in the command line. If you press `y` (yes), the text is replaced by the previously defined replace string (see the command `r=replace`). Press `n` if you do not want to replace the string.

After you answer the question the editor continues with the search. You can halt the search and query with [STOP] and the editor returns to the text input.

If a line becomes longer than the set maximum length as the result of a replacement, the editor halts the replacement and displays the message **error overflow in line**. The cursor stands at the occurrence whose replacement would have made the line too long. The same applies for replace without query with F6.

F5 Command key. All commands start with this key. The message enter command appears in the first screen line. The corresponding command is called by pressing a certain key.

F6 Replace without query. All occurrences of the search string, from the cursor position on, are replaced with the replace string automatically and without query. Replace can be halted with the [STOP] key.

If an overflow in line occurs, the same procedure is followed as for F4 (replace with query).

F7 Insert lines. A blank line is inserted before the cursor line. The color of the line is copied from the preceding line.

F8 Delete lines. The cursor line is deleted and the remaining text moves up.

[HOME]

Switch text areas. The display is toggled between the file area and the extra text area.

[CLR] Start of text. The text is displayed starting at the beginning of the document.

[STOP] Interrupts all command inputs, halts the printing, loading, reading the directory, searching (F3), and replacing (F4, F6). Basically, everything but saving can be halted with [STOP].

3.3 Parameter inputs

If you have selected a command, you must usually enter parameters. The inputs the various commands require will be described in Section 3.3. The five different types of parameter inputs are explained in the following sections. All five can be interrupted with the [STOP] key which returns you to text input.

3.3.1 Key input

No cursor appears for this type of input. The editor waits for certain keys. The message in the first line of the screen indicates which keys you may select from. The keys at the end of the message are separated by a / character (for example: replace y/n?). Except for the given keys and the [STOP] key, no other keys have any effect.

3.3.2 Input a number

Only the digit keys 0-9 and the control keys [DEL], [RETURN], and naturally [STOP] are accepted during a number input. The input range is limited to a certain number of digits. At the end of the field the cursor stops and no more digits are accepted.

[RETURN] ends the input. If no digits are entered, the input is not ended. [DEL] deletes the last character entered. [STOP] halts the input.

3.3.3 Input a string

The input is limited to a certain number of characters. No more characters are accepted at the end of the field except for [DEL]. All printable characters from the keyboard are allowed as input.

[DEL] erases the last character entered, [STOP] interrupts the input. [RETURN] ends the input. All characters from the start of the input field to the character before the cursor belong to the entered string.

3.3.4 Block input

For this input the first screen line contains the message **marking out range**. In this input type you can determine a block which is displayed in reverse type. Various operations can then be performed on this block. A block is a contiguous section of lines. The block can be edited with the following keys:

- ⇔ The cursor is moved to the right or left. The cursor itself cannot be seen, but its position is indicated on the position display. These keys have only the function of shifting the screen right or left during the block input.
- ⇓ The size of the block is increased.
- ⇑ The size of the block is decreased.

[RETURN] ends the input. The limits of the block are now set.

[STOP] interrupts the input. The editor returns to the text input.

3.3.5 Destination input

For this input the first line of the screen contains the message **fixing target**. The target line is displayed in reverse text. The destination line appears in normal text in a line which appears in the middle of a marked block of text. After the block input, the target line is the line directly after the reverse block and cannot immediately be recognized. You will see the target line if you move it.

You can move the target line with the following control keys:

⇐ Changes the cursor column. Scroll the screen left or right during the destination line input.

↑↓ Moves the target line up or down.

F1 (page down) The destination line is moved 22 lines down.

F2 (page up) The destination line is moved 22 lines up.

[HOME]

Switch text areas. This control key is possible only with the **transfer** command.

g The **g** key calls the command **goto**. You can enter the number of a line as the target line.

[RETURN]

ends the input if the target line does not lie within the previously marked block. Otherwise the editor displays **no target line** and the target must be reentered.

[STOP] interrupts the input.

3.4 Commands

The message `enter` command appears in the first screen line when the [F5] key is pressed. The editor expects the user to press a key which selects a command. All keys except for the possible command keys and [STOP] are ignored. [STOP] interrupts the input.

In the description of the commands the input types for the parameters are indicated as follows:

key input	<key>
number input	<number>
character string	<string>
block input	<block>
destination input	<dest>

The input type is not indicated on the screen, it is only used to inform you what kind of input you should make. In most cases this will be clear anyway.

The key which calls the given command is set apart from the paragraph. Indented and printed in different type are the messages which appear during the command.

b **bytes free**

A message appears in the status line of the screen that displays the amount of memory space remaining to the editor.

h **hunt**

Enter the search string for the search function (F3) and (F4 or F6).

hunt:<search string>

r **replace**

Enter the search string for the replace function (F4 or F6). The first character string which you enter is the search string and the second is the replace string.

```
hunt:<search string>
rplc:<replace string>
```

e **erase**

Delete blocks of text. You must first mark the block.

```
erase:marking out block <block>
erase:are you sure y/n? <key y,n>
```

After marking a confirmation question appears. The key [n] for no prevents the deletion. The key [y] for yes deletes the block. The text is displayed at the deleted block following the deletion.

t **transfer**

Copy a block from one point to another. You must first mark the block and then set the target (destination) line. The target of this command can also lie in the other text.

```
t'fer:marking out range <block>
t'fer:fixing target <dest>
```

After the input of the target line, a copy of the block is inserted in front of the target line. The screen then displays the document after the copied text. If the document becomes too long, the transfer command will not be executed. The editor responds **overflow** and the screen shows the text at the select target line.

m move

Move a block from one location to another. You must first mark the block and then set the target. The block is inserted before the target line.

```
move:marking out range <block>
move:fixing target <dest>
```

c color

Enter the number of a color (0-15) and mark a block. The block is then colored in the selected color. The screen then displays the document starting with the colored text block.

```
color:<number 0-15>
color:marking out range <block>
```

l load

Enter the name of a text file to be loaded into working memory. Any text in memory will be erased. The extra text area remains unchanged.

```
load
file: <string>
```

If the message **file format error** appears when loading, the format is incorrect for the C_EDITOR. The editor changes the text so that it is readable. Information may be lost through this process, however. The message **overflow** indicates that the text no longer fits in memory. This command can be used from the file area.

- s save**
Save the document with the name displayed in line two of the screen. If a file with this name already exists on the diskette, the following question appears:

save replace y/n? <key y,n>

If you answer with [y], the existing file will be replaced by the new one. An [n] halts the saving process and you are back in text input. This command can be used in the file area.

- f filename**
Change the name of the document in the file area. This command can be used only in text file area.

file: <string>

- k kill**
Erases the document in the file area. The extra text remains unchanged. This command can be given only in the text file area.

kill: are you sure y/n? <key y/n>

The confirmation question protects the memory from unintentional erasure. The [n] key stops the command.

- i input disk error**
Reads the disk drive error channel and displays the contents on the status line.

d **directory**

Displays the directory of the diskette and inserts it in the text at the current cursor line. You can give a specifier with the directory command (such as *=prg, test*, test*=usr). More about the function and syntax of these specifiers can be found in the disk drive manual. If you enter nothing and just press [RETURN], the entire directory is displayed.

directory:<string>

It is best to read the directory into the extra text because it will not disturb anything there.

If you want to read the directory of a drive other than drive a, you must give at least a device identifier as a specifier. For example:

b:

lists the contents of drive b.

x **exit**

Exit from the editor and return to the CCP.

exit: are you sure y/n? <key y/n>

The [n] key interrupts this command.

n **new text**

Erase and set new parameters for a new document. The line length for the new document is set here. It cannot be changed later.

new: length of line <number 40-80>

The line length of the file text also applies to the extra text. If lines in the extra text are longer than the new line length, the remainder of the line is no longer accessible.

file:<string>

Next, the filename is entered. The file text is erased and now has the new line length. With a line length of 40 the screen is no longer shifted horizontally. If you don't want the screen to scroll, you can prevent it from doing so by specifying a line length of 40. 80 columns at a time are displayed in the e8 editor under version V3. If you set fewer than 80 characters, you will not be able to move the cursor beyond the set position.

g goto
Goto (jump) a given line number.

goto : <number>

p print
Print the document on the printer (device no. 4).

print : input defaults y/n? <key y,n>

With the n key the parameters last specified are used. After the editor is started the following parameters are in effect:

secondary address 0, cbm, extra, lines per page 72, offset 0

You can change these parameters with [y] . If you press [y], the following inputs appear:

print : sec. address <number 0-15>

You can set the secondary address with which the text will be sent to the printer here.

print : cbm or ascii c/a? <key c,a>

If you select [c] for cbm, the text is output in the CBM character set. With a, the text is output in the ASCII character set.

If you have selected cbm:

print:normal subst extra n/s/e?
<key n,s,e>

- e:** The characters of the ASCII character set are represented using programmable characters on the Commodore printer. This print mode requires more time.
- s:** (subst=substitute) The ASCII characters are replaced with suitable graphics characters from the Commodore character set.
- n:** The text is output to the printer without conversion.

If you selected ascii:

print:line feed on y/n? <key y,n>

[y] causes the editor to send carriage return/line feed combinations instead of just carriage return which n produces.

print: epson printer y/n? <key y,n>

With [y] the editor sends the code sequence for the American character for Epson printers before printing (\$1b,\$52,\$00). With [n], the sequence is not set. The following inputs are common to both types.

print: lines per page <number>

With this input you set the page length. For the American standard of 11 inch paper, this would be 66 lines per page.

print:offset <number>

This number specifies the number of spaces that the printing will be indented from the edge of the paper.

The input of the date is again common, independent from whether you changed the parameters or not.

print: date <string>

Here you can enter the date which will be printed beneath the text name. If you entered the date before, the editor skips this question. C

You can stop the printing with [STOP]. It may be that you have to hold the [STOP] key down longer than usual before the editor reacts. C

3.5 Error messages

- illegal text:** The command selected may not be used in extra text (new, save, load, kill, filename).
- overflow:** The text storage is full. The function will not be executed.
- overflow in line:** The line became longer than the maximum line length when replacing. The replacement is halted.
- no target line:** The target line lies within a marked text block. This is not allowed and the input of a target is not ended.
- file format error:** The loaded file does not have the necessary text format. The file is probably not a text file at all. The editor forces the text to the required format, but information can be lost in the process.
- If a FATAL ERROR occurs in the compiler, the corresponding error file can generate this error when it is loaded. No information is lost if this happens.
- last line:** You tried to write on the last line or you tried to move the cursor past the last line.
- i/o error:** A input/output error occurred or the device being accessed is not turned on.

4. C compiler

The name of the compiler is `cc`. It is loaded from the master disk and started like a transient command. The `cc` belongs to the system components and will automatically be loaded using the fast-loading procedure. It makes no difference what disk drive is used (1541/71). The loading time varies between 10 and 13 seconds. Note, however, that `cc` is about 25K long.

4.1 Start without arguments

After the compiler has been loaded, the compiler header appears along with the message:

```
source file name:
```

The name of the source file which the compiler is to compile must be entered here. This input is done without quotation marks and without spaces in front of the name. A device identifier can be given in front of the filename. If this is missing, drive a will be selected as usual.

The following control characters can be used during the input.

[DEL]	delete the last character
[CLR]	erases the entire input field.
[RETURN]	ends the input

Next the compiler requests the link file names.

```
link file name:
```

If you use the extensions given in Section I.1.3, such as `.c` for the source file name, `cc` will print the name with `.o` added as a suggestion for the link file name so that you only have to press [RETURN]. If you do not want to take the suggestion, because you want a different name or because the link file is supposed to go to a different device, you can erase the input with [CLR].

After entering the link file, `cc` expects the input of the error file name:

```
error file name:
```

Here the device identifier of the source file with the name `error.e` is given as a default. Naturally, you can change this name or accept it by pressing [RETURN].

The compiler now starts compiling. The source files currently being processed are displayed in grey type. If an `#include` file is ended, a `#` is printed. The names of functions which `cc` is compiling appear in yellow.

Errors are printed by the compiler in red during compilation. The error messages are also collected in the error file at the same time. The error file is opened only if an error occurs. This also erases a previous file by the same name. This error file can be read with the C-editor and contains other status information from the first error on so that it should not be hard to trace an error occurring in an `#include` file.

At the conclusion of the compilation `cc` responds

```
compiling finished
linkfile (not) available
press x to quit, r to restart
```

The link file will be available depending on whether an error occurred or not. If it is not available, there will be a file on the disk with the link file name, but this is only a fragment which contains only the code up to the error and cannot be linked.

By pressing the [x] you will be returned to the CCP. With [r] the compiler starts over and is ready to compile another source program.

All source programs which the compiler needs for compilation must be on the disks in the designated drives before the input of the error file name is ended. It is not possible to change disks during the compilation.

4.2 Start with arguments

Arguments from the command line can be passed to the C compiler. The first three arguments refer to the source, link, and error files:

```
cc: a:test.c a:test.o a:error.e
```

In this example the compiler is loaded and started with the arguments. The compiler puts the arguments in place of the inputs and begins compilation immediately.

You can use this only if you have at least two disk drives or a RAM disk. Users with only one drive must insert the appropriate program disk after starting without arguments.

If fewer than three arguments are given, the compiler will ask for the remaining arguments.

If more than three arguments are given, the compiler will start the linker. First the program will be compiled as usual. If an error occurs, the usual message will appear. If the compilation was error-free, the following message appears:

```
compiling finished
linkfile available
loading linker
```

The linker will be loaded from the same drive as that the compiler was loaded.

The fourth argument will be used as the program file name in the linker. All other arguments apply as link files for the linker. The file created by the compiler will automatically be specified as the last link file. The call:

```
cc a:test.c a:test.o a:error.e a:test h:libc.l
```

first compiles `test.c` to the file `test.o`. Then `test.o` will be linked with the library `libc.l` which is located on the RAM disk. The linked program file will be called `test` on drive `a`. The transient command `c1` (Section I.1.8.8) makes use of this option.

4.3 Compiler error messages

The compiler outputs error messages to the screen and to the specified error file at the same time. This file can be read with the editor and used to find and correct the errors.

The error file will be opened with the first error and also contains all status messages which otherwise appear only on the screen.

Behind the error message is the number of the line in the source file in which the error occurred. Often an error will lead to other errors which will disappear after the first error is corrected.

After the compiler discovers an error, it searches for the next semicolon or brace and continues compilation at this point. This naturally causes parts of the program to be skipped, which the compiler assumes cannot be compiled properly because of the error. Skipping these sections can lead to other errors.

Some errors cause the compilation to stop because it no longer seems to be possible. These are called FATAL ERRORS. Such errors are not put in the error file because output to the error file may cause new errors (such as bus errors).

Here is a list of all of the compiler error messages:

?FLOPPY ERROR (FATAL ERROR)

- FLOPPY ERROR stands for a disk drive error message. In any case the compilation will be terminated. The cause of the error can be determined from the error text. The device identifier of the corresponding device will also be given in the error text.

?DEVICE X: NOT PRESENT (FATAL ERROR)

- The output device with device identifier X: is not available.

?ILLEGAL COMMAND

- no preprocessor command recognized
- no string follows #include

?RUN END OF LINE

- Terminating " character is missing from a string

?STRING TOO LONG

- string constant consists of more than 254 characters
- macro definition longer than 254 characters
- argument of a macro definition is longer than 254 characters

?TOO MANY CONCATS

- More than seven files chained with `#include`

?EXPECTING IDENTIFIER

- no name follows `#ifdef`, `#ifndef`, `#define`
- parameter of a macro definition is not a name
- a `struct`, `union`, or `enum` name is expected for a `struct/union` component whose type is `struct`, `union`, or `enum`
- names are expected in `enum` specifier
- a `struct`, `union` name is expected for a parameter declaration with type `struct/union`

?COND. COMPILER ERROR

- more than 8 nested conditions
- more than one `#else` in the `#if - #endif`
- `#else` without `#if`, `#ifdef`, `#ifndef`
- `#endif` without `#if...`
- expression after `#if` contains an error
- the expression evaluation is interrupted through `#if`. `#if` is possible only outside an expression or constant.

?RUN END OF FILE

- end of program although preprocessor command `#if` not closed with `#endif` yet
- end of program although the argument of a macro call is still expected
- declaration was not closed
- block structure still open

?MACRO EXISTS

- The macro to be defined already exists.

?STACK OVERFLOW (FATAL ERROR)

- no space for new macros for #define
- no space for the entry of a new declaration
- recursion by initialization too large (about 40)
- constant buffer overflowed, cannot be emptied

?MACRO NOT DEFINED

- #undef was used on an undefined macro.

?ILLEGAL NOTATION

- improper char constant (not exactly one character)
- More than one decimal point of exponent in a double number
- exponent is incomplete

?ILLEGAL MACRO CALL

- the call requires the specification of arguments
- the call has too many or too few arguments

?ILLEGAL OPERATOR

- A character was recognized which cannot be evaluated as an operator (like \$, #, @).

?OVERFLOW ERROR

- a double constant which was too large was read
- an enum constant is too large
- overflow during constant evaluation

?DIVISION BY ZERO

- Division by zero in a constant division.

?DECLARATION OVERFLOW

- more than 60 nested arrays or pointers
- more than 60 parameters in a function definition

?EXPECTING SUBSCRIPT

- more than one dimension contains no subscript
- the first dimension does not contain a subscript

?SIZE OVERFLOW

- Object is longer than 32767 characters.

?DECLARATION SYNTAX ERROR

- If the compiler encounters a block structure after a global declaration, it cannot evaluate this as a declaration. Since the block structure would be recognized as an error only because of a previous error in a declaration, this will be skipped. The compiler responds with this error to indicate this. If the block structure were not skipped, a host of secondary errors would occur.
- improper declarator
- no name in declarator
- no , or ; as the end of a declarator
- no type was given for a struct/union component
- no } as the end of an enum specifier
- neither type nor storage class for parameter definitions or local declarations

?DECLARATION SEMANTIC ERROR

- auto or register as the storage class for a global declaration
- typedef cannot define functions
- components or parameters declared as function
- an attempt was made to define arrays of functions or to define functions which return arrays, functions, structures, or variants
- an attempt was made to define a component as a structure or a variant whose type agrees with that of the struct or union being defined (recursion)
- declaration of a local function

?IDENTIFIER ALREADY DEFINED

- the name is already declared as extern, local name or as a type name, struct, union, enum, or component name or as an enum constant
- the name to be declared already exists, but with a different declaration
- the names to be defined is already known, but with a different declaration

?EXCEPTION ERROR

- This error normally does not occur

?IDENTIFIER NOT DEFINED

- name is not defined, although no specifier is given
- the struct, union name in a component (for struct/union subdeclaration) is not defined

?DECLARATION INCOMPATIBLE

- Name is defined, but is not a struct, union name.

?EXPECTING IDENTIFIER

- Neither a specifier nor a specifier name was given.

?TYPE CONFLICT

- no int-convertible type of address in enum
- char cannot be initialized with addresses
- auto addresses are not allowed as constants
- switch expression is not of integral type
- general improper type for operator, such as double for %
- structures and variants can be combined only with . -> &
- right operand of . or -> must be a component
- function call without corresponding declaration

?INITIALIZER TOO LONG

- more elements or components were initialized in as array or structure initialization than are possible
- A string is longer than the char array

?ILLEGAL INITIALIZER

- initializers for functions and variants is not allowed
- initializers for extern declarations are not allowed

?PARAMETER MISMATCH

- the declaration of the parameters does not agree with the order in the parameter list
- occurs as a secondary error if a parameter declaration is incorrect

?TOO MANY STATEMENTS NESTED

- More than 16 instructions (blocks are also instructions) were nested.

?TOO MANY BLOCKS NESTED

- More than 8 blocks were nested.

?STATEMENT SYNTAX ERROR

- occurs as a secondary error from erroneous instructions when a block is terminated
- no name behind `goto`
- no `;` behind `break` or `continue`
- conditions not parenthesized (for `if`, `for`, `while`, `switch`)
- expressions behind `for` are not separated by `;` (two `;`'s are necessary)
- no `while(..);` follows `do`

?LABEL NOT DEFINED

- The name behind `goto` is not defined as a label.

?EXPRESSION SYNTAX ERROR

- general syntax error: incorrect parentheses, multiple constants or names in a row (if `;` was forgotten)
- use of a keyword in an expression (only `sizeof` allowed)
- improper parenthesization within the `CAST` declarator
- a name which designates no object was used
- operator stands alone in front
- `a :` must follow every `?`

?ILLEGAL STATEMENT

- `break` or `continue` not allowed here.

?TOO MANY CASES

- More than 42 case labels were given within a `switch` block.

?CASE WITHOUT SWITCH

- Label defined outside a `switch` block.

?EXPRESSION SEMANTIC ERROR

- Only one `CAST` per simple expression is allowed.

?EXPRESSION OVERFLOW

- expression consists of more than 63 elements
- expression consists of more than 28 names or constants
- CAST storage exceeded

?NO CONSTANT ERROR

- The expression does not return a constant, although this is required.

?EXPECTING L-VALUE

- the first operand of an assignment is not an lvalue
- the first operand in front of . is not an lvalue
- an lvalue is expected for the unary operator &
- ++, -- may be used only on lvalues

?EXPECTING ADDRESS

- An address is expected for the unitary * operator.

?NMI INTERRUPT

- Generates an interrupt; all files are closed.

5. Linker

The C linker has the task of converting compiled source files, called link files, into an executable machine language program. With the C linker you can link up to 10 separately compiled source files into one program file which contains an executable C program.

The order of the link files is in principle irrelevant. As long as you specify the same link files, the same C programs result. You should, however, give the libraries as the first link files so that they are the start of the program.

Note that the C linker does not make any declaration checks. You can, for example, define an object as a structure in one file and declare it as a function in another. If both objects have the same name and are available externally, the linker will link references to both objects without recognizing their different declarations.

All link files which you link must be on the inserted disks during the linking process. It is not possible to change disks during linking. You can use the `copy` command to copy files.

5.1 Start without arguments

For inputs in the linker, the same control characters apply as for the compiler. The input is ended with [RETURN]. With [DEL] (delete) you can erase the last input character. The key [CLR] clears the entire input field.

First you enter the name of the program file in which the linked program will be stored. A existing file with this name will be erased.

```
program file
```

The filename can be naturally be specified with a device identifier.

Now the linker asks for the individual link files which are to be linked together.

```
link file libc.l
```

The name of the standard library `libc.l` is printed as the default for the first input since this file is usually linked to every program. In version V3 this default has the device specifier `h:`, so that the library will be read from the RAM disk.

You can specify a maximum of 10 files. After the tenth name the linker automatically moves on to the next input. If you want to link fewer than 10 files, you can terminate the link file input by not entering anything behind link file and just pressing [RETURN]. You must have at least one link file, however, or the linker will ignore your attempt to terminate the input.

The next input reads:

```
memory top page $d0
```

A default value is given for the input of the top of memory and you can usually accept this default. This input is done in hexadecimal and requires exactly two digits. In version V2 the default is `$d0` and in V3 it is `$e9`. The input will be repeated if an incorrect response was given.

The memory top page designates the first page (page=256 bytes) which is no longer available for C programs. For V2 and the default value of `$d0`, the boundary of the C program memory is `$d000`. This means that your C program reaches from `$0801` to `$cfff` (50K). In the V3 version the memory reaches from `$1c00` to `$e900` (51.25K).

Specification of a memory top page allows you to protect memory from C programs. You can then use the memory at the memory top page to `$cfff` or `$e8ff` for your own applications, such as common storage for C programs which are chained and loaded in succession.

```
linker option:  
(c=ccp/b=basic) c
```

This linker option is only possible in version V2. The option `c` means that your program can be started only from the CCP. The C program is designated as a C-version. The `b` option allows your C program to be started from BASIC.

After the parameter input the linker starts to link the files. The linker announces the individual passes through the link files. If errors occur, they will be printed in red.

The linker can be stopped at any time, even during parameter input, by pressing the [STOP]+[RESTORE] keys (=NMI).

You then see the message:

```
nmi interrupt
press x to quit, r to restart
```

With the [x] key you exit the linker and go back to the CCP. With [r] you start the linker again. If you enter incorrect parameters, generate an NMI and select [r] for restart. You can then repeat the parameter input.

5.2 Start with arguments

Like the compiler, arguments can also be passed to the linker. The first argument is the program file. All other arguments are link files. A maximum of 11 arguments are possible.

If the linker receives only one argument, it asks for the input of link files. If at least two arguments are passed, the linker automatically starts the link process with the specified arguments. The defaults are taken for the memory top page and the linker option.

```
cl a:test a:libc.l a:libgraph.l a:test.o
```

This call automatically starts the linking process. The two libraries are linked with the file test.o to produce the program file test.

5.3 Error messages

The following messages can occur during the linking:

pass 1/2

The linker is starting the first/second pass through the link files.

end of pass 1.2

The first/second pass is finished

link file

The linker is reading from the specified link file.

linking finished

The linking ended without error.

linking aborted

The linking cannot be continued because of error.

incorrect linkage

An error occurred in the linking.

program not available

The program file is not available because of errors. It was erased.

In addition, the following error messages are possible:

no reference to

The specified name is not defined in any link file.

no clear reference to

The specified name is defined in at least two link files.

no external declaration of ...

The specified name is not declared as extern. This error does not prevent the correct creation of a C program.

no linkfile format

The format of the file read is not right. The file in question is probably not a link file.

overflow in symbol table

The desired link file combination cannot be linked because the capacity of the available memory for the extern names is too small.

static variables out of range

The C program does not fit in the available C program memory because the static variable area exceeds the memory top page. (The static variable area includes all static and external variables which are not initialized).

program out of range

The C program does not fit in the available C program memory.

linkfile incompatible

One of the link files is not compatible with the current linker version. This error can occur only if you use link files which were compiled with other Super C systems. This is also the case, for example, if you want to link link files from the V2 compiler with the V3 linker.

device X: not present

The device designated by device identifier X: is not turned on.

floppy error on X:

An disk error occurred on device X:. The error message will be printed. This error leads to termination of the linker.

nmi interrupts

This message appears when [STOP]+[RESTORE] are pressed.

exception error ?

This error does not occur under normal circumstances.

6. C programs

6.1 Start

C programs can be linked as C versions or as B versions (V2 only). The C versions can only be started through the CCP. To do this, enter the program name with the device identifier in the CCP. In principle C programs are loaded like transient commands. The program device, for example, is written in C.

B versions can be started under BASIC. To do this, load the program with the following command:

```
load "program name",8,1
```

The B version starts itself automatically after loading. B versions can also be loaded in the CCP. They will destroy the CCP. After the program ends you will be back in BASIC.

B versions can be created only in V2. The `autoexec` mechanism, which makes it possible to load and start a user program after boot-up, serves as a replacement in V3.

B and C versions run in basically the same environment. This environment, memory division and usage, will be described in Section II.6.4.

If a C program is ended in the normal manner, that is, the main block is ended, you will be back in the CCP for the C-version or BASIC for the B-version. The C program can be restarted only by loading it again.

One can pass arguments from the command line to C-versions:

```
a:tester alpha beta
```

In the C program, a parameter list must be defined for the function `main()` for the evaluation of these arguments. This is normally done as follows:

```
main(argc, argv)
  int argc;
  char *argv[];
{ ...
}
```

`argc` then contains the number of arguments, whereby the filename is counted as well. `argc` is therefore at least 1. The first argument is an array of pointer to the arguments:

```
argv[0] points to "a:tester"
argv[1] points to "alpha"
argv[2] points to "beta"
argc is equal to 3
```

A maximum of 40 arguments can be passed in V2, or 63 arguments in V3 (for more information, see II.1.4).

No arguments can be passed to B-versions. The variable `argc` will have the value zero. So you can test in the program if a B-version is present.

6.2 Operating modes

Run-time errors can occur while a C program is running. In BASIC such errors always lead to termination (this can be circumvented in the new BASIC 7.0). In C you can set whether or not an interruption will occur with operating modes.

The operating modes are `ERRON`, `ERROFF`, `NMION`, and `NMIOFF`, which can all be set with standard functions by the same name. Combinations of these result in four operating modes:

```
ERRON  + NMION (default)
ERROFF + NMION
ERRON  + NMIOFF
ERROFF + NMIOFF
```

The question of whether or not a program will be interrupted depends not only on the operating mode, but also on the error

number of the run-time error (run-time errors and their numbers are given in the next chapter).

1. Error numbers 1 to 63 are switched off with `ERROFF` and on with `ERRON`, that is, no interruption will occur if `ERROFF` is set.
2. Error numbers 64 to 127 are turned off by `NMIOFF` and on by `NMION`.
3. Error numbers 128 to 255 always lead to a termination of the program.

All occurring errors place the error number in a special register which the function `qerror()` can read. If the interruption is turned off, the program can react to the error itself.

An interruption prints the error text. For example:

```
?division by zero
press x to quit, c to continue,
r to restart
```

You can get back to the CCP with the [x] key and thereby end the program.

With [c] the program will be continued as if no interruption occurred. The number of the error still stands in the error register, however, so that `qerror()` functions without problems. If run-time errors occur when reading data from the screen, the data to be read may be destroyed by the error message so that the program does not continue properly when [c] is pressed. Eventually other errors will occur.

The program is restarted with [r]. Note that all external and static variables will retain their values unchanged. Initializations are not performed. Also, static and external variables will not have their defined initial value of zero. Open channels will not be closed.

Error number 64 to 127 are manipulated with `NMION/OFF`. The name comes from NMI (Non-Maskable Interrupt), which is generated by pressing [STOP]+[RESTORE]. This generates a

run-time errors with number 127 and the text `?nmi interrupt`. You can enable and disable the interruption of a C program by the NMI. If the program is continued after the interruption with `[c]`, the error register will contain the number 127. If the NMI is disabled with `NMIOFF`, the error register will not be changed by an attempt to generate an NMI. You should never interrupt an input or output operation in a C program with NMI or peculiar effects may occur if the program is continued.

Error numbers 128 to 255 cannot be affected. Nor do they lead to a normal interruption. Only the error message appears. For example:

```
?stack overflow
```

and the computer waits for a key press. The program is then ended. This involves an error which makes it impossible for the program to continue.

If you look at the following list of errors, you will see that not all of the numbers are used. You have enough room for your own run-time errors, which you can create with the function `error()`. You can then select by the error number in which operating mode an interrupt is to be performed. You should, however, keep error numbers 12 to 32 free for future extensions.

6.3 Run-time errors

The following error messages are possible in a C program. The error numbers are given in parentheses. Note the significance of the error number for the reaction of the program (whether an interrupt is performed or not).

?too many files (1)

No more than 10 file descriptors can be used at a time. This error occurs on an attempt to open more than 10 files at once.

?i/o#2 (2)

This message corresponds to the BASIC error message FILE OPEN and cannot occur in C because the file descriptors are not selected by the user.

?illegal filedescriptor (3)

The specified file descriptor is not being used, that is, either the file has been closed already or it has not been opened yet.

?i/o#4 (4)

This message corresponds to the BASIC error message FILE NOT FOUND. It can occur in C only if a file is to be read from cassette. In this case the function open() searches for this file on the cassette. If open() finds an EOT mark, this message appears.

?device not present (5)

A device is not reachable on the bus. It is probably turned off. This message can appear when opening files and when performing input and output operations.

Note: The message does not occur in the V2 version (because of an error in the C-64 ROM) if a file is opened on the serial bus without specifying a name and a read operation is performed after this. If the addressed device is not present, the open () function will be executed without the message ?device not present appearing. If a read operation is performed on this channel, the computer will hang in an endless loop, which can be ended only with an NMI.

?not input file (6)

This error can occur in C only if a file on tape was opened with a secondary address other than zero (=write access) and an attempt was then made to read from the file.

?not output file (7)

This error occurs only if a file is opened on the keyboard and an attempt was made to write to this file.

?i/o#8 (8)

Corresponds to the BASIC error message MISSING FILENAME and cannot occur in C.

?i/o#9 (9)

Corresponds to the BASIC error message ILLEGAL DEVICE NUMBER. In C this error occurs only if the pointer to the tape buffer is less than \$0200 (V2) or \$0400 (V3).

?break (10)

This message appears in C only when the [STOP] key is pressed during a cassette routine. Continuation with [c] is not recommended.

?illegal format (11)

When reading with a scanf function the data read does not match the expected format.

?run eof (12)

The input of a scanf function was ended with an EOF signal although additional data were expected.

?illegal quantity (13)

An uncalculable argument was passed to a mathematical function. For example, the argument for sqrt and log must be positive.

?division by zero (33)

An attempt was made to divide by zero. After [c] (continue), the expression has the value zero.

?overflow (34)

A double operation exceeded the range. After continue the expression has the value zero. The overflow is ignored for integer operations.

?nmi interrupt (127)

Appears when [STOP]+[RESTORE] is pressed.

?stack overflow (129)

The run-time stack of the C program exceeds the available program storage or too much memory was requested by alloc.

6.4 Memory layout

The memory layout is of interest only to those programmers who want to access memory locations directly via pointers.

Super C V2

The addresses are all given in hexadecimal

\$0000-\$03ff	Memory for system variables
\$0400-\$07ff	CCP
\$0800-\$cfff	C program storage
\$d000-\$d7ff	I/O range
\$d800-\$dfff	Color RAM
\$e000-\$e3ff	Video RAM
\$e400-\$ffff	Operating system

The C program memory can be shortened in the linker. The remaining memory is then freely available.

Super C V3

The addresses are all given in hexadecimal:

Bank 0

\$0000-\$03ff	Memory for system variables
\$0400-\$07ff	Video RAM
\$0800-\$1bff	CCP
\$1c00-\$e8ff	C program storage
\$e900-\$ffff	RAM disk driver

Bank 1 (2 and 3 for memory expansion)

\$0000-\$ffff	RAM disk storage
---------------	------------------

The complicated memory management of the C-128 does not allow the I/O range (\$d000-\$d7ff) and the two color RAMs (\$d800-\$dfff) to be accessed directly. Functions from the C library do make this possible, however.

As with V2, the program storage for C programs can be restricted in the linker.

7.0 The Library functions

There are several libraries and header files on the system diskette. The libraries contain pre-programmed link files which you include in your own C programs. These link files have an extension `.l` (`.l` = library). Header files have an extension `.h`. They are source codes that can be inserted into your C programs with `#include`. Header files declare library functions or define constants and macros. There is no need for you to repeatedly define individual functions. Set them up as a series of definitions in a header file and `#include` that file in your programs.

The header files and library files available on Super C are:

<code>stdio.h</code>	-	<code>libc.l</code> and <code>libcs.l</code>
<code>math.h</code>	-	<code>libmath.l</code>
<code>graphic.h</code>	-	<code>libgraph.l</code>
<code>ctype.h</code>		

`stdio.h` does the work of two libraries. Both of these libraries are similar to one another; you'll probably work with `libc.l` most of the time. This library takes up quite a bit of memory due to the size of the `printf` and `scanf` functions. If you have no use for these functions in your programs, you can use the `libcs.l` library, which excludes `printf` and `scanf`.

`ctype.h` is a header file which contains macro definitions, and has no corresponding file in the library files.

7.1 Standard C Libraries

`stdio.h` is the name of the standard library header file (see Appendix for a listing of this file). `stdio.h` contains the following constants set by `#define` besides those found in `libc.l` and `libcs.l`.

<code>STDIO</code>	as file descriptor for the standard input/output
<code>NULL</code>	as 0
<code>CR</code>	as carriage return <code>\$0d</code>
<code>CRSUP</code>	as code for cursor up
<code>CRSDOWN</code>	as code for cursor down
<code>CRSRIGHT</code>	as code for cursor right
<code>CRSLEFT</code>	as code for cursor left
<code>HOME</code>	as code for cursor home
<code>CLR</code>	as code for clear home
<code>REVERSON</code>	as code for RVS
<code>REVERSOFF</code>	as code for RVS off
<code>NIL</code>	as 0, pointer to nothing
<code>EMPTY</code>	as an empty string

The `getchar()` function is also defined in `stdio.h`. This function returns a character (type `char`) which is read from the standard input. It will wait for a keypress unequal to zero.

To give you a better understanding of the standard functions, here are descriptions of these functions.

7.1.1 **erron(), erroff(), nmion(), nmioff(), qerror(), error(), exit()**

```
void erron()
```

The operating mode **ERRON** is enabled with **erron**, which has the effect that run-time errors with numbers 1-63 create an interruption (see Sec. 6 - C programs). The operating mode **ERRON** is the initial mode.

```
void erroff()
```

Enables the operating mode **ERROFF**. This has the effect that run-time errors with numbers 1-63 are masked, that is, no interruption is made. The only result is that the number of the error is placed in the run-time error register.

```
void nmion()
```

The operating mode **NMION** is enabled. All run-time errors with numbers from 64 to 127 and the NMI itself [**STOP+RESTORE**] lead to interruptions. This is the initial mode (see also Sec. 6 C programs).

```
void nmioff()
```

The operating mode **NMIOFF** is enabled. All run-time errors with numbers from 64 to 127 and the NMI itself are masked, meaning that no interruption occurs. The run-time errors 64-127 are only placed in the error register of the run-time system. An attempt to issue an NMI [**STOP+RESTORE**] during **NMIOFF** does not change the error register.

```
int qerror()
```

Returns the value of the error register of the run-time system. This contains the number of the last run-time error encountered, even if this error caused an interruption and the program was continued with **c**. After calling the **qerror** function the error register is zero. You can also use this function to set the error register to zero.

```
void error(string, fnr)
    char *string;
    int fnr;
```

string is the pointer to some error text, fnr is the associated error number.

The error function creates a run-time error whereby the given error number and enabled operating mode determine whether an interruption will be created or only the error register will be loaded. (see Section 6)

```
void exit()
```

exit() ends the C program and closes all of the open files. As soon as the user presses a key, exit() returns to the CCP or to BASIC.

7.1.2 open(), close()

All of the following input/output functions can trigger runtime errors (which can be suppressed with ERROFF). The error register is changed in any case. Most functions present additional results in an error state.

```
file open(prim, sec, name[,buffer])
    int prim, sec;
    char *name;
```

This function opens a file with a device address of prim. (device number, rather than device identifier), a secondary address of sec and a filename of name. Filenames can have up to 255 characters. If no filename is sent, an empty string must be given.

The secondary address states the operating mode of the device being used (e.g. 15 for the disk error and command channel).

The function opens a file with the given parameters. open() returns a file descriptor as the result. This corresponds to the logical file number in BASIC. The file descriptor has the type file in

`stdio.h`. Since the file descriptor of an opened file is required for all further operations, it must be stored in a variable of type file.

```
file id;  
fd=open(8,2,"testfile,s,r");
```

These commands are similar to the BASIC command:

```
OPEN fd,8,2,"testfile,s,r"
```

You must give the logical file number in BASIC, while in C, the `open` function looks for a free file number. If you are not acquainted with the `open` command in BASIC, see your manual for a detailed description.

When an error occurs `open()` returns the file descriptor of 0 or `STDIO`. This will be the case in any standard i/o errors. All file descriptors normally given by `open()` are unequal to 0.

The argument put into brackets in the above example is optional, and is necessary only when the device number is 1 or 2. Device 1 opens a cassette file and the address of the cassette buffer must be present in the argument `buffer`. This buffer memory must be at least 192 bytes long. Super C version V3 must put this area between `$1C00` and `$C000`. Version V2 must have the cassette buffer between `$0800` and `$D000`.

Device 2 is used for the RS-232 interface. The given buffer must be 512 bytes long. Here is an example for 300 baud, 3-wire handshake, full duplex and no parity.

```
fd=open(2,0,"\6\200",buf);
```

Most of the time the argument `buffer` can be omitted.

```
file close(fd);  
file fd;
```

This function closes a file and therefore must be given the file descriptor of the file. If this result is zero, an error has occurred.

7.1.3 `putc()`, `fputc()`, `getc()`, `fgetc()`

```
int putc(c, fd)
    char c; file fd;
```

```
int fputc(c, fd)
    char c; file fd;
```

Both of these functions are essentially the same. The `c` character is output with the file descriptor `fd`. The result returned will be 1 in normal circumstances, or 0 in an error state.

```
char getc(c, fd)
    file fd;
```

```
char fgetc(c, fd)
    char c; file fd;
```

Both of these functions are essentially the same; they read a character from a file. If the file descriptor is equal to 0 or `STDIO`, no cursor appears. The value is `'\0'` when no key is pressed, and a value corresponding to the keypress is returned. Errors return a -1.

7.1.4 `getchar()`, `putchar()`

```
char getchar();
```

```
int putchar(c)
char c;
```

These functions work like `getc` and `putc`. They let you work with both the keyboard and screen. `getchar` waits for a keypress. `getchar` is defined in `stdio.h`, `putchar` is a parameter macro.

7.1.5 gets(), fgets(), puts(), fputs()

```
char *gets(line,n)
    char *line; int n;

char *fgets(line,n,fd)
    char *line; int n;
    file fd;
```

These functions read in a string. `gets` reads from the keyboard (cursor visible), `fgets` reads from the given file. It reads characters until it reads a `'\0'` or `'\n'`. The string read will be stored at the `line` address and a `'\0'` will be at the end of the `line[n]` area, along with the `'\n'` character. The result will be returned in the address `line`.

```
int puts(line)
    char *line;

int fputs(line,fd)
    char *line; file fd;
```

Both functions output a string until a closing `'\0'` (note: the `'\0'` is not output). `puts` writes to the screen, `fputs` to a file. The result returned is the number of characters actually handled.

7.16 fgetf(), fputf()

```
int fputf(line,n,fd)
    char *line; int n;
    file fd;
```

This function writes `n` characters from the address `line` into the appropriate file, without regard for the `'\0'` character. The result returned is the number of characters actually written.

```
int fgetf(line,n,fd)
    char *line; int n;
    file fd;
```

This function reads up to *n* characters from the file in memory at the address *line*. Thus, the result returned is the number of characters actually read. This can be *n* in an error state, or when an EOF is found.

EOF can usually be read by all *get* functions. When EOF occurs during reading, the variable is unequal to zero, otherwise equal to zero.

The *ST* variable can be used here as in BASIC (see your computer manual).

Both version are defined in `stdio.h`.

7.1.7 `fopen()`, `fclose()`

```
file fopen(name, mode)
    char *name, *mode;
```

This function opens a file on the disk. The first argument points to the filename with device identifier. *mode* points to a string which can have the following contents:

"r" for reading a file
"w" for writing a file
"a" for appending data to an existing file

The opened files always refer to the type SEQ. If you want to process files of type PRG or USR, add *,p* or *,u* to *mode*.

```
fopen("b:test", "r,u");
```

opens the USR file *test* on device *b* for reading.

As with *open*, the file descriptor returned as a result must be stored.

The secondary address, which specifies the disk channel, is equal to the file descriptor. This is important if you want to open additional channels to a disk with `open`.

```
file fclose(fd)
file fd;
```

This function is identical to `close()`. It serves only to achieve a certain compatibility with UNIX.

7.1.8 `strlen()`, `strcmp()`, `strncmp()`

```
int strlen(str)
char *str;
```

The function returns the length of the string to which `str` points. The `'\0'` character is not counted, that is:

```
str[strlen(str)]=='\0'.
```

```
int strcmp(str1, str2)
char *str1, *str2;
```

The two strings are compared lexically. The result returned:

```
-1 for str1<str2
0 for str1=str2
1 for str1>str2
```

```
int strncmp(str1, str2, n)
char *str1, *str2;
int n;
```

This function works like `strcmp`. At most `n` characters are compared with each other.

7.1.9 strcat(), strncat(), strcpy(), strncpy()

```
char *strcat(str1, str2)
char *str1, *str2;
```

The string `str2` will be appended to the string `str1`. The string `str1` must have enough free memory space for this. `str1` will be returned as the result.

```
char *strncat(str1, str2, n)
char *str1, *str2;
int n;
```

This function works like `strcat`, but it appends a maximum of `n` characters of `str2` to `str1`.

```
char *strcpy(str1, str2)
char *str1, *str2;
```

The string `str2` will be copied into the string `str1`. The ending zero is not copied along. This means that if `str2` is shorter than `str1`, that `str2` will overwrite only the first characters of `str1`.

If `str2` is longer, the string may not have a terminating zero any more. In Super C this is recognized and a terminating zero is added. This is an extension, however, which cannot be expected of other systems.

The result is the address of `str1`.

```
char *strncpy(str1, str2, n)
char *str1, *str2;
int n;
```

This function copies like `strcpy`, but a maximum of only `n` characters of `str2` will be copied.

7.1.10 strchr(), strrchr()

```
char *strchr(str,c)
char *str; char c;
```

The function searches in the string `str` for the character `c`. The search starts from the beginning. If the character is found in the string, `strchr` returns the address of the first occurrence of the character. If the character is not found, the pointer `NIL` will be returned.

```
char *strrchr(str,c)
char *str; char c;
```

This function searches for the specified character like `strchr`. The search begins at the end of the string, however, and proceeds toward the front.

If the character does not occur in the string or occurs only once, both functions return the same value.

7.1.11 cursor(), exec()

```
void cursor(line,pos)
int line,pos;
```

The cursor is set to the line given by `line` and the column given by `pos` on the current screen.

```
void exec(string)
char *string;
```

This function ends the C program and executes the CCP commands contained in `string`. This allows resident commands of the CCP to be called, though this is not particularly useful.

You can, however, load a new C program with this command and start it. The arguments for this are passed in `string`.

One example would be a menu program in C:

```
#include "stdio.h"
main()
{
    int i;
    while()
    {
        printf("1. tester1\n");
        printf("2. testprg2\n");
        printf("3. END\n");
        scanf("%d",&i);
        switch(i)
        {case 1: exec("a:tester1 arg0 arg1");
         case 2: exec("a:testprg2 arg0");
         case 3: exec("");
        }
    }
}
```

Every user program could end with the line `exec("a:menu")`, which would reload the menu program.

If you want to pass more than arguments to the loaded programs, you must set the memory top page down when linking the individual programs. You can then use the resulting area as a common data pool. The access must be done via pointers.

7.1.12 `cmove()`, `move()`

```
void cmove(target,n,source)
char *target;
int n;
char *source;
```

The function moves `n` bytes starting with the memory location `source` to the memory location `target`. Overlappings are checked.

This function can, for example, be used for assigning complex types:

```
char s[100],t[100];
...
cmove(s,sizeof(t),t);
```

In this example, the array `t` is assigned to the array `s`.

```
void move(target, n, source, mem)
char *target; int n;
char source; int mem;
```

The function `move` is available only under V2. It works like `CMOVE`, but you can in addition set the memory configuration. This configuration corresponds to the processor port in the C-64. You can find the significance of these bits in the appropriate literature.

`mem=52` means, for example, that the memory layout of 64K RAM is enabled during the copy. This allows you to change character sets in V2 which are located at `$d000-$dfff`. `mem=53` is the memory layout during the C program. The `move` routine should be fairly close to the start of the memory (link libraries first). If the function itself becomes covered by ROM through the memory switch, the computer will hang up.

7.1.13 `alloc()`, `free()`

```
char *alloc(size)
int size;
```

`alloc()` prepares memory space for objects. These objects do not have names and can be accessed only via pointer values. They serve, for example, for list management or as temporary storage. The length of the required object is passed as the argument. `alloc()` returns as the result the pointer value to the object. This pointer value is defined as a pointer to `char`. If other types are required, the address must be converted by means of a `CAST` to another type.

The argument `size` may have only positive values from 0 to 32767, or an `overflow` error will result. If larger objects are required, two `alloc()` calls are necessary, whereby the second call returns the base address of the entire object.

If not enough space is available for an object, a `?stack overflow` error will be displayed. Remember that the `alloc()` function limits the storage for the C stack. This stack contains local variables and data for function calls.

```
char *free(size)
    int size;
```

`free()` represents the reverse function of `alloc()`. It releases objects defined with `alloc()`. Objects must be released in the reverse order in which they were generated. The argument `size` may not be larger than 32767 or the run-time error `?overflow` error will be created.

7.1.14 `settime()`, `gettime()`

```
char *settime(string)
    char *string
```

The function expects a string in the form "HH:MM:SS" where HH are the hours, MM the minutes, and SS the seconds of the clock time to be set.

The function sets the internal clock to the given time and then reads the clock time again. It returns an address to the read time (see `gettime`).

```
char *gettime()
```

The function reads the time of the built-in clock. It returns an address to a string in which the time is stored.

The string has the following format:

```
" :HH:MM:SS.s"
```

HH signify the hours, MM the minutes, and SS.s the seconds. The resolution amounts to one tenth of a second. This string can be read with the function `sscanf`, for instance.


```
scanf (gettime (), "%d:%d:%d", &h, &m, &s);
```

h, m, and s contain the hours, minutes, and seconds. If you want to read the tenths as well, you must change the last %d to %lf and declare s as double.

7.1.15 keys()

```
int keys(string)
char *string;
```

This function places the characters of the string into the keyboard buffer. This can be used to provide defaults for screen inputs, for example.

keys() returns the number of characters which were put in the keyboard buffer.

7.1.16 call()

```
long call(p)
char *p;
```

The function calls a machine language program at location p. The call is done as a subroutine. The instruction RTS causes a jump back to C.

Additional arguments can be passed to the function. The memory locations \$20,\$21 of the zero page contain a pointer to the start of these arguments.

Memory locations \$20 to \$30 may be used during the assembly language program.

A result of the function may be placed in memory locations \$42 to \$45. This is a long value. The value must be stored with the lowest-order byte first.

The only possibility for placing an assembly-language program is in the area in the C program storage which the linker creates by limiting the program storage area.

This function should be used only by those familiar with assembly and machine language.

7.1.17 fast(), slow() (V3 only)

```
void fast()  
void slow()
```

These functions switch to the FAST or SLOW mode of the C-128. The processor clock is set to either 2MHz or 1MHz. You can find more under the CCP commands of the same names. These functions are not available in V2.

7.1.18 window() (V3 only)

```
void window(lcol,tline,rcol,bline)  
int lcol,tline,rcol,bline;
```

The screen area of the standard output will be set to the given window. `lcol` means the left column, `tline` the top line, `rcol` the right column, and `bline` the bottom line of the window. The line numbers run from 0 to 24 and the column numbers from 0 to 39 or 79.

The window area will be set back to the whole screen by pressing [HOME] twice.

7.1.19 vdcin(), vdcout() (V3 only)

```
char vdcin(reg)  
int reg;
```

The register `reg` of the VDC chip will be read and returned as the result. The function takes care of the appropriate handshake.

You can find more about the registers of the VDC chip in the appropriate literature, such as the book *C-128 Internals* by Abacus.

```
void vdcout(reg,c)
    int reg; char c;
```

This function writes register `reg` of the VDC chip with the character `c`. The function takes care of the appropriate handshake.

7.1.20 `io\in()`, `io\out()` (V3 only)

```
void io\out(adr, val [,colram] )
    char *adr; char val;
    [int colram;]

char io\in(adr [,colram] )
    char *adr;
    [int colram;]
```

A memory location in the I/O range (`$d000-$dfff`) is selected with `adr`. The memory location is written with the value `val` with `io\out()`. With `io\in()` the contents of the memory location are read and returned as the result.

If the area `$d800` to `$dfff` is selected (color RAM), the argument `colram` must be specified in addition. This selects between the two color RAMs in the C-128. If `colram` is 0, the color RAM for text display is selected, else the color RAM for graphics will be selected if `colram` is 1.

7.1.21 `is80()`

```
int is80()
```

This function returns 1 if the current screen is the 80-column monitor, else a 0 is returned.

7.1.22 Formatted output

```
void printf(control, arg1, ..., argn)
    char *control;
    ....
    { /* ..... */ }

void fprintf(fd, control, arg1, ..., argn)
    file fd;
    char *control;
    ....
    { /* ..... */ }

void sprintf(string, control, arg1, ..., argn)
    char *string, *control;
    ....
    { /* ..... */ }
```

These three functions can be used for formatted output. The function `printf()` prints to the standard output (screen), `fprintf()` to the file `fd`, and `sprintf` prints to the string `string`. If you select `fd=0` with `fprintf()`, you have the same function as `printf()`.

The formatted output is controlled by the character string `control`. `control` consists of normal characters which are printed without change, and format instructions which control the conversion of the arguments `arg1, ..., argn`. The `printf` function uses the string `control` in order to interpret the arguments following it. If fewer arguments are provided than format instructions in `control`, or the data types of the format instructions do not agree with those of the parameters passed, the `printf` function outputs nonsense.

Each format instruction starts with the `%` character and ends with a character which designates the conversion. The following may be used:

A minus sign, which directs the converted argument to be left justified.

A decimal digit string, which indicates the minimum field width. If this is absent, the default value 0 is used. If the converted argument is shorter than the minimum field width, it is padded with blanks. If this string of digits starts with a 0, the remaining positions up to the minimum field width are filled with zeros (0) instead of spaces. This padding is performed such that the output is right or left justified as specified.

A period, which precedes another string of digits.

A string of digits, which indicates the maximum number of digits which will be output, or which sets the number of places after the decimal for the conversions *e* and *f*. If this number and the period are missing, the default value 6 will be used and the length of the argument to be converted for all other conversions will be supplemented by this amount (conversions *d*, *o*, *x*, *u*, *c*, *s*). A digit string is expected if the period is entered. 0 will be assumed if this is missing.

The letter *l*, which designates the corresponding argument as long (concerns the conversions *d*, *o*, *x*, and *u*).

Each of the above specifications is optional. The number inputs are converted modulo 256. At most the first 254 characters of a string can be printed with the `printf` function. The output of a format instruction may not comprise more than 255 characters or incorrect results will be obtained.

The following characters control the conversion:

- d** The argument is represented as a signed decimal number. The argument must be of type *int*, or of type *long* if an *l* is contained in the format instruction.
- u** The argument is represented in decimal without a sign. The type of the argument must be unsigned *int* or unsigned *long* if an *l* appears in the format instruction.
- o** The argument is represented as an octal number without sign or leading zero. The type of the argument is the same as that for *u*.

- x The argument is represented in hexadecimal without sign and without leading 0x. The type of the argument is the same as that for *u*.
- c The argument is represented as a single character. The type of the argument must be *char*.
- s The argument is represented as a character string. The type of the argument must be *char ** (pointer to *char*).
- e The argument must be of type *float* or *double* and is output in decimal in the following format:

[-]m.nnnnnnE[+-]xx

With this conversion the second string of digits in the format instruction represent the number of places after the decimal. The default is 6 places.

- f The argument must be of type *float* or *double* and is output in decimal in the following format:

[-]mmm.nnnnnn

The second string of digits determines the number of places after the decimal. The default here is six. If more places after the decimal are specified than the number of significant digits present, the following digits become zero (this also applies to the conversion with *e*).

- g The argument must be *float* or *double*. The conversion is made as per *f* or *e*, whichever of the two is shorter. The representation is selected so that only the significant digits are shown. If both conversions are the same length, *e* is chosen.

If the conversion character is not one of those found above, the character itself is output. The % character can be printed with %%.
 C

Example 1:

```
double dbl=3.456E+1;

printf("%0f\n%e\n%12g\n%12.1f\n%-012.2e\n",
        dbl,dbl,dbl,-dbl,dbl
        );
```

The following output appears on the screen:

```
34.560000
3.456E+01
      34.56
     -34.6
3.46000E+01
```

Example 2:

```
static char s[9]="Example";
printf(":%s:\n:%9s:\n:%.5s:\n:%9.5s:
\n:-9.5s\n", s,s,s,s,s);
```

The following output appears on the screen:

```
:Example:
:  Example:
:Examp:
:      Examp:
:Example  :
```

7.1.23 Formatted input

```
int scanf( control, arg1, arg2...)
    char *control;
    ...
    { /* ... */
    }
```

```
int sscanf( string, control, arg1, arg2...)
    char *string;
    char *control;
    ...
    { /* ... */
    }
```

```
int fscanf( fd, control, arg1, arg2...)
    file fd;
    char *control;
    ...
    { /* ... */
    }
```

These three functions make formatted input of data possible. The function `scanf()` reads from the keyboard (standard input), `sscanf()` from a character string, and `fscanf()` from a file.

A control string is passed as an argument to all three functions. The input is interpreted by means of this control string.

The `scanf` function requires additional arguments in order to store the data read in. These arguments are all pointer values which must point to objects in which the data read can be stored.

The following characters may be in a control string:

Spaces and line separators, which will be skipped

Other characters (except for `%`) which are then expected in the input (after an arbitrary number of spaces or line separators)

Format elements which begin with the `%` character. A `*` character and/or a string of digits and the format character

which indicates the type of the data read in eventually follows them.

A format element determines the interpretation of the input and the type of the object to which the input is to be assigned. A pointer to the object must follow control as an argument.. If the format element contains a * character, the assignment is suppressed and no pointer argument is required.

An field is defined as a sequence of characters which contains no blanks. An input field extends to the next blank or to the optional field width given by the digit string, or to the character which no longer fits the given format.

The following format characters are possible:

- d An integer decimal number is expected as input. A pointer to *int* should be given as an argument.
- h like d
- o An octal integer is expected as input. The argument should be an *int* pointer. The digits 8 and 9 are interpreted as octal 10 and 11. The octal number is read with or without leading zero.
- x A hexadecimal number is expected as input (with or without leading 0x). An *int* pointer should be passed as an argument.
- c A single character is read as input and a pointer to *char* is expected as argument. In this case the next input character is assigned, even blanks. If a digit string comes before the c, the next spaces are read as with the other format elements.
- s A string of characters is read in. More information can be found in the next section, 7.1.23.1.
- e A decimal floating-point number is read as input. The argument should be pointer to float. A decimal point as well as an exponent may be present in the input. The

exponent consists of the character **E** or **e**, an option sign, and a string of digits.

f like **e**

The letter **l** may stand before the conversion characters **d**, **o**, or **x**, in order to show that the corresponding pointer argument points to a *long* object. Before **e** and **f** the letter **l** indicates that the pointer type is *double*.

If a conversion is interrupted by a character which can not be interpreted, it is applied to the next field. Such a character is lost if no further input field is required in one call of **scanf** or **fscanf**. This is because the input and output in operating system of the C-64 are not buffered.

```
Example: int x;
         float d;
         double e;
         scanf ("%o%e%le", &x, &d, &e);
```

The cursor appears and you enter:

```
44.123 2.5
```

The following data is assigned:

```
36 is placed in x,
0.123 is placed in d,
2.5 is placed in e
```

If you enter the same input with only the following evaluation:

```
scanf ("%d", &x);
```

the decimal point is lost as a separator. (**scanf** differs from the normal standard functions in this regard).

If an input from the keyboard is not completely evaluated, the remainder of the input is lost and printing is done to the screen. Because the standard input does not send an EOF signal, the input

is repeated until all arguments are served. The `sscanf` function creates an EOF signal if it encounters the end of the input string.

The `scanf` functions return the number of correctly-read data as the result.

7.1.23.1 Reading strings

The reading of a string is done at the start of the next input field. The reading is not interrupted by blanks. The number of characters read in is determined by the field width given, but does not go beyond an EOF signal. In addition, the reading can be interrupted by a boundary character. This boundary character is normally assigned the code for RETURN. As a general rule, the string is read only up to a RETURN character. The boundary character always belongs to the string read.

The boundary character can be determined by the user. A . (period) character must appear in front of the `s` and then the boundary character.

Caution is recommended when reading strings from the standard input without specifying a field limit. Since the standard input does not send an EOF signal, the input is stopped only by the boundary character.

7.1.23.2 Error messages

If the input is ended by an EOF signal although the `scanf` function expects more data, the error ?RUN EOF (10) is given. The error message ?ILLEGAL FORMAT (11) is generated if a certain character was expected in the input but a different character was read instead. This also applies for the input of numbers. At least one digit must be present for these.

7.1.23.3 sscanf and fscanf

The function `sscanf` reads the input from a string. Another argument is passed before the control string, namely the input string.

The function `fscanf` reads from a file. A file descriptor originating from the opening of the appropriate file must be additionally passed as an argument.

A good example is reading the error message from the disk:

```
int f1, f2, f3;
char ft[30];
file floppy=open(8, 15, EMPTY);
```

```
fscanf(floppy, "%d, %, s%d, %d", &f1, ft, &f2, &f3);
```

`fscanf` reads from the error channel. First, a decimal number is stored in `f1`. Then a comma must follow in the input, or an illegal format error will result. A string is then read and stored in `ft`. The string is interrupted at the first comma. Then two `int` numbers are read.

If the error messages reads, for example,

```
65, NO BLOCK, 10, 14
```

the following assignments will be made:

```
f1=65;
ft="NO BLOCK, ";
f2=10;
f3=14;
```

7.2 The graphics library

The graphics functions of Super C create four-color graphics on the 40-column monitor with a resolution of 160x200 points. Each of the four colors can be chosen from a palette of 16 colors. A point can have one of four conditions. These conditions are called color classes. You can assign each color class its own color with a corresponding function. The color class zero has a special significance; it is the color class which corresponds to the background color of the screen. Points of this color cannot be seen. You must therefore distinguish between color and color class. If you change the color of a color class, all points in the graphic which possess this color class will change.

A color class is of type `int`, whereby one of the lowest two bits are relevant (0-3). This allows the four color classes to be distinguished. A color is of type `int`, whereby only the lowest four bits are relevant (0-15). This allows 16 colors to be displayed.

You do not need to pass the color number to the corresponding function, however. In `graphic.h` there are macros defined whose names correspond to the available colors. These macros will then be replaced by the corresponding color number in the C program:

```
#define black 0
#define white 1
...
#define lgrey 15
```

In addition, the external declarations of the graphics functions and some system variables for the graphics are made in `graphic.h`. If you want to use the graphic functions in a C program, insert this with the following instruction in your source text:

```
#include "graphic.h"
```

Remember that this filename may have a device identifier. For users of Super C V3, this will usually be `h:` because this file is found on the RAM disk after boot-up.

When linking the compiled program, you must then link the graphics library `libgraph.l` as well.

The possible points of the graphic screen are addressed through a Cartesian coordinate system. The coordinates are of type `int`. The visible drawing plane is a rectangle bounded by the points (0,0) and (159,199). The lower left screen point is the point (0,0), the right upper (159,199). In all functions you can also specify points which lie outside the drawing surface. The effect of the function is then partially visible at best. Points which lie outside the drawing plane always have color class zero (background) when read. Write accesses to points outside the drawing plane will be ignored.

Graphics and text are independent of each other. This means that you can display text on the 40-column screen while you construct a graphic, or vice versa. Only the background and border colors are the same for text and graphics (color RAMs are different).

7.2.1 `graphic()`, `graphon()`, `graphoff()`, `isgraph()`

```
void graphic()
```

With this function, which requires no arguments and does not return anything, you can allocate the memory space required for graphics. This memory space goes from the upper end of the C program storage on. The exact length is determined by the architecture of the video chip. If there is not enough space for the graphics because your C program is too long, the error `?stack overflow` will be printed.

As you may already know, you can request memory with the function `alloc()`. This memory is taken from the upper end of the C storage down. The function `graphic()` does not take into account whether you have already allocated memory with `alloc()` or not, because the graphic memory can lie only an one specific location.

In order to eliminate complications, you should call `graphic()` at the start of the C program and then allocate the required memory

space. Since the graphics also use `alloc()` indirectly to allocate the memory, you can also release the graphic storage with the `free()` function.

```
void graphon()
```

The graphics mode on the 40-column monitor is enabled.

```
void graphoff()
```

This function is the opposite of `graphon()`. The graphics mode is switched off and the text mode is enabled.

```
int isgraph()
```

`isgraph()` returns 1 (true) if the graphics mode is currently enabled. If the text mode is enabled, `isgraph()` returns 0 (false).

7.2.2 `backgr()`, `clrmap()`, `colors()`, `setcol()`

```
void backgr(colex, colbk)
    int colex, colbk;
```

This function is the same in text and graphics modes. With it the border and background color (color class 0) can be set (40-column screen only). The two arguments `colex` and `colbk` are colors, this means you can use the macros defined in `graphic.h` for the colors or pass the appropriate code.

```
backgr(dgrey, black);
```

would set a dark grey border and a black background. The argument `colbk` is the color of color class 0.

```
void clrmap(cnr)
    int cnr;
```

`clrmap()` clears the graphic storage. A color class with which the graphic memory will be filled can be passed as an argument. Normally one uses the value 0 for the background color class.

```
void colors(col1, col2, col3)
    int col1,col2,col3;
```

With this function you set the three foreground colors. The arguments are color values. You can therefore use the color macros or specify the corresponding number. The three arguments determine the colors of color classes 1 to 3.

```
colors(red, green, blue);
```

assigns red to the color class 1, green to color class 2, and the color blue to color class 3. All points in the graphic which are stored in one of the color classes change their color.

```
void setcol(cnr)
    int cnr;
```

Here the color class in which most of the graphics functions draw can be set. If you specify the color class 0 here, it means that points will be set in the color of the background, which amounts to erasing them.

7.2.3 dot(), dotin(), bdot()

```
void dot(x,y)
    int x,y;
```

With `dot()` you can set the individual points of the graphic. The color class of the points is set by `setcol()`. You can erase points by first selecting the color class 0 with `setcol()`. If the point lies outside the drawing surface, `dot()` has no effect.

```
int dotin(x,y)
    int x,y;
```

`dotin()` is the reverse of `dot()`. `dotin()` returns the color class of the point (x,y). If the point lies outside the drawing area, 0 will be returned.


```
int bdot(x,y)
    int x,y;
```

bdot () has almost the same effect as **dot ()**. The point will only be set, however, if the point previously had the color class 0, the background color. **bdot ()** returns the color class of the point (x,y) before it was overwritten. With the help of **dot ()** and **dotin ()** one could rewrite **bdot ()** in the following manner:

```
int bdot(x,y)
    int x,y;
{
    int z;
    if ((z=dotin(x,y))==0)
    {    dot(x,y);
        return 0;
    }
    return z;
}
```

The effect of **bdot ()** is this: An object which you draw with **bdot ()** can be seen wherever there was previously background in the graphic. This creates the impression that the object was drawn behind the foreground.

An example:

```
#include "stdio.h"
#include "graphic.h"

main()
{   int i,j;
    graphic(); graphon();
    clrmap(0); colors(red, green, blue);
    setcol(1);
    for (i=80; i<=110; ++i)
        for (j=80; j<=110; ++j)
            dot(i,j);
    setcol(2);
    for (i=70; i<=140; ++i)
        for (j=85; j<=130; ++j)
            bdot(i,j);
    getchar();
}
```

First a red rectangle is drawn and then a green one. The green rectangle would cover the red, but it will be drawn with `bdot()`, creating the impression that the green rectangle was drawn behind the red one.

7.2.4 line(), bline(), mline(), oline()

```
void line(x1,y1,x2,y2)
  int x1,y1,x2,y2;
```

With this function you can draw lines in the current color class. The line begins at the point (x1,y1) and ends at the point (x2,y2). One or both of the points can lie outside of the drawing surface. The line is then only partially visible. If the line is drawn in color class 0, all points which lie on the line will be erased.

```
void bline(x1,y1,x2,y2)
  int x1,y1,x2,y2;
```

bline() works like bdot(). The line will be drawn from (x1,y1) to (x2,y2), but only the points which previously lay in the background.

The example program of bdot() can be formulated with line() and bline() as follows:

```
#include "stdio.h"
#include "graphic.h"
main()
{
  int i;

  graphic(); graphon();
  clrmap(0); colors(red, green, blue);
  setcol(1);
  for (i=80; i<=110; ++i)
    line(i, 80, i, 110);
  setcol(2);
  for (i=70; i<=140; ++i)
    bline(i, 85, i, 130);
  getchar();
}
```

```
char *mline(x1,y1,x2,y2,put)
  int x1,y1,x2,y2;
  char *put;
```

First `mline()` has the same function as `line()`—it draws a line from (x_1, y_1) to (x_2, y_2) . In addition, all of the overwritten points are placed in memory at the location `put`. The color class of the points is stored. Since a point can have four different color classes, you need 2 bits of storage per point. The memory required for the entire line can be calculated so:

```
if ((dx=x1-x2)<0) dx=-dx;
if ((dy=y1-y2)<0) dy=-dy;
if (dx<dy) dx=dy;
memavail=((dx+1)*2+6)/8;
```

You can get the memory space for such lines in three ways. You can declare appropriate objects which can store such lines. You can allocate the memory space dynamically with `alloc()`. The third possibility is to use unused graphic memory. In all cases you must be sure that the memory area actually suffices for storing the line. We will go into the unused graphic memory later.

`mline()` returns a pointer which points behind the memory in which the overwritten line points were stored. In contiguous memory the result of `mline()` can serve as the `put` argument for another call.

```
char *oline(x1,y1,x2,y2,get)
    int x1,y1,x2,y2;
    char *get;
```

The function `oline()` is the opposite of `mline()`. `get` must point to memory in which a line was stored with `mline()`. `oline()` draws the stored line on the screen. The coordinates should be the same as they were when calling `mline()` or the result will not make much sense.

`oline()` returns an address to the end of the memory area of the line.

Here is a small example for `mline()` and `oline()`. The following function draws a line from a set starting point (x_1, y_1) . You can move the end point across the screen with the help of the cursor keys.

```
drawline(x1,y1)
  int x1,y1;
{  int x2,y2;
   char buffer[100];

   x2=y2=0;
   setcol(3);
   mline(x1,y1,x2,y2,buffer);
   while ((c=getchar())!='\n')
   {   oline(x1,y1,x2,y2,buffer);
       switch(x)
       {   case CRSUP: y2+=10; break;
           case CRSDOWN: y2-=10; break;
           case CRSRIGHT: x2+=10; break;
           case CRSLEFT: x2-=10; break;
       }
       mline(x1,y1,x2,y2,buffer);
   }
}
```

7.2.5 setplot(), plot()

```
void setplot(x,y)
  int x,y;
```

setplot() sets the starting point for the plot function to the specified coordinates.

```
void plot(x,y)
  int x,y;
```

plot() draws a line from the set starting point to the specified coordinates. The specified destination point becomes the starting point for the next call of plot(). The starting point can be changed with setplot().

7.2.6 shape(), fill()

```
void shape(x1,y1,x2,y2)
    int x1,x2,y1,y2;
```

The rectangle designated by the diagonal end points (x1,y1) and (x2,y2) will be filled with the current color class. The specified rectangle need not necessarily lie within the drawing plane. For example;

```
shape(-10,-20,30,40);
```

The following calls are identical:

```
shape(-10,40,30,-20);
shape(30,40,-10,-20);
shape(30,-20,-10,40);
```

```
void fill(x,y)
    int x,y;
```

This function fills a background surface which is bounded by points of a different color class. The point (x,y) must lie within the surface to be filled. If the point lies outside the drawing plane or is not a point of the color class 0 (background), fill() does nothing.

The figure which will be filled can naturally have bizarre shapes--it need only be bordered.

7.2.7 pushobj(), plotobj(), fplotobj(), bplotobj()

```

char *pushobj(x1,y1,x2,y2,put)
    int x1,y1,x2,y2;
    char *put;

```

`pushobj()` has an effect like that of `mline()`. The specified coordinates determine a rectangle. This will then be placed at the memory location `put`. The points of the rectangle are stored line by line from top to bottom and from left to right. The color class zero will be stored for points outside the drawing surface. Two bits are required per point. The required memory space can be calculated as follows:

```

if ((dx=x1-x2)<0) dx=-dx;
if ((dy=y1-y2)<0) dy=-dy;
memneed=++dx * ++dy;
memneed=(memneed*2+6)/8;

```

As with `mline()` you must always make sure that the memory space provided suffices for the rectangle because otherwise the program and data will be overwritten.

`pushobj()`, like `mline()`, returns the address of the first memory location no longer needed.

```

char *plotobj(x1,y1,x2,y2,get)
    int x1,y2,x2,y1;
    char *get;

char *fplotobj(x1,y1,x2,y2,get)
    ...

char *bplotobj(x1,y1,x2,y2,get)
    ...

```

These three functions are the counter parts to `pushobj()`. With them you can merge a stored graphic object back into the graphic. The argument `get` is the address of the memory in which `pushobj()` placed the object. It thereby corresponds to the argument `put` of the corresponding `pushobj()` call. The points

(x1,y1) and (x2,y2) are interpreted as the diagonal end points of a rectangle. To prevent the graphic object from becoming distorted, the rectangle must have the same dimensions as for the call to `pushobj()`. The position of the rectangle can naturally be selected differently. You should save the dimensions when storing with `pushobj()` in order to call the corresponding `plotobj()` functions correctly. These functions return as result the address behind the object read so that if objects are stored sequentially in memory, the address of the next object will be returned.

`plotobj()` draws the stored object on the screen exactly as it was stored.

`bplotobj()` draws the object only at the places in the graphic where the color class 0 (background) is present. The object will thereby be drawn behind the current graphic.

`fplotobj()` draws only the points of the object which do not have the color class 0. The object is thereby drawn in front of the current graphic. At the places where the object has the color class 0, the previous graphic appears, that is, the object is drawn in the foreground. Here is an example of these functions:

```
#include "stdio.h"
#include "graphic.h"
main()
{   char obj[1200]; /* 50*90*2-8=1200 */
    graphic(); clrmap(0);
    colors(red,green,blue);
    graphon();
    setcol(1);shape(50,60,90,140);
    setcol(2);shape(52,64,88,136);
    setcol(1);shape(63,86,77,114);
    setcol(0);shape(65,90,75,110);
    pushobj(45,55,95,145,obj);

    plotobj(-10,40,40,130,obj);
    plotobj(-10,-10,40,80,obj);
    bplotobj(65,90,115,180,obj);
    fplotobj(15,0,65,90,obj);
    getch();
}
```


7.2.8 mask, bmap, mapv()

```

char mask;
char *bmap;
char mapv(x,y)
    int x,y;

```

mapv() returns the byte in which the point (x,y) is located. The variable mask is set at the same time. In it is the value in which only the two bits are zero which in the result of mapv() represent the point(x,y). In addition, the pointer bmap is set after mapv() which points to the location in the bit map in which the result of mapv() stands.

If you are familiar with the architecture of the bit map storage (multi-color graphics mode), you can manipulate the graphic directly with this function. Normally you do not need this function.

The function dotin() could look like this in C:

```

dotin(x,y)
    int x,y;
{
    char c;
    c=mapv(x,y);
    switch(mask ^ 0xff)
    {
        case 0xc0: c>>=2;
        case 0x30: c>>=2;
        case 0x0c: c>>=2;
    }
    return c & 0x3;
}

```

7.2.9 Layout of the graphics memory

Super C V2

\$8c00-\$8fff	Video RAM of the graphics (color storage)
\$9000-\$9bff	Unused (free for graphic objects)
\$9b00-\$9bff	Stack for fill
\$9c00-\$9fff	Temporary color RAM (for graphic/text switch)
\$a000-\$cfff	Graphics bit map
\$d000-\$dfff	I/O range
\$e000-\$e3ff	Video RAM of the text

The free RAM from \$9000 to \$9bff results from the architecture of the C-64. This area will always be addressed as character ROM by the video chip, so graphics here are not possible. The video RAM for the graphics must therefore lie below \$9000. You can use the free memory for graphics objects. Make sure, however, that these objects fit in the memory provided or your graphics will be overwritten.

We cannot go into the significance of the individual memory locations for the graphics. We recommend the relevant literature for the C-64.

Super C V3

\$c000-\$dfff	Bit map of the graphics
\$e000-\$e3ff	Video RAM for graphics
\$e400-\$e4ff	Stack for fill
\$e500-\$e8ff	Unused (free for graphic objects)

Temporary color RAM is not required because the C-128 has two switchable color RAMs.

When using the unused memory, make sure that the area above \$e900 is not overwritten. This will destroy the RAM disk driver.

7.2.10 Demo program

On your master disk there is a demonstration program called `cdemo.c`. In this program you will find extensive applications of the graphics routines. An analog clock, among other things, is implemented.

The Super C V2 owners have a compiled version on their diskettes. On Super C V3, the disk storage sufficed only for the source text. You can start the demo with the following commands:

Master disk in drive a:

```
h:com s:*
a:cc cdemo.c h:o.o h:error.e h:cdemo libc.l libgraph.l
a:slow
h:cdemo
```

The last command starts the demo. Note: The contents of the RAM disk will be erased.

7.2.11 Storing the graphics

We want to present two useful routines here. The functions load and save a graphic. You can pass the name to the function along with a device specifier:

```
static col[3]; /*contains the color classes*/

void writegr(name)
    char name[];
{file fd;
  fd=open(name,"w");/*name no RETURN at end */
  fputc(col,sizeof(col),fd);
  if (ST)
      error("graphic file exists",20);
  if (fputc(bitmap,8000,fd)!=8000)
      error("write error",21);
  fclose(fd);
}
```

```
void readgr(name)
  char name[];
{
  file fd;
  fd=fopen(name,"r");
  fgetf(col,sizeof(col),fd);
  if (ST)
    error("graphic file not found",22);
  if (fgetf(bitmap,8000,fd)!=8000)
    error("read error",fd);
  colors(col[0],col[1],col[2]);
}
```

7.3 Math library

The math library requires the header file `math.h`. It makes the appropriate function declarations. In addition the constants PI (3.14...) and E (2.718...) are defined as macros.

The mathematical library is called `libmath.l`. It must be linked to the program in the linker when using the mathematical functions.

7.3.1 `sin()`, `cos()`

```
double sin(d)
  double d;
```

```
double cos(d)
  double d;
```

The functions calculate the sine and cosine of the specified arguments. Note that the arguments must have the type `double`. `sin(1)` is not allowed. It must be `sin(1.0)`.

The arguments must be given in radians.

7.3.2 tan(), atan()

```
double tan(d)
    double d;

double atan(d)
    double d;
```

The functions calculate the tangent and the arctangent of the the argument.

7.3.3 abs(), sgn(), rnd()

```
double abs(d)
    double d;

double sgn(d)
    double d;

double rnd(d)
    double d;
```

`abs()` calculates the absolute quantity of the arguments.

`sgn()` calculates the sign function for the argument. The result will be -1 for a negative argument, 1 for a positive argument, and 0 if the argument is zero.

`rnd()` creates random numbers between 0 and 1. The number one is never reached. The argument determines the method of creation. For negative arguments the argument will be enlisted to form a random sequence. For positive arguments a random number results which is dependent only on the last random number. The use is similar to the BASIC function `rnd`.

7.3.3 `sqr()`, `sqrt()`

```
double sqr(d)
    double d;
```

```
double sqrt(d)
    double d;
```

The function `sqr` (square) calculates the square of the argument. The function `sqrt` (square root) calculates the square root of the argument. Note that the root of only positive numbers can be taken. Otherwise an `?illegal quantity error` will appear.

7.3.4 `log()`, `exp()`

```
double log(d)
    double d;
```

```
double exp(d)
    double d;
```

`log()` calculates the logarithm base e . This number is defined in the macro `E`. The logarithm can be used only on positive numbers.

You can calculate logarithms of other bases like so:

```
log(x) / log(b)
```

where x is the argument and b is the base.

`exp(x)` calculates the number e to the x power. The exponentiation to other bases is done by:

```
exp(log(b) * x)
```

where b is the base and x is the exponent. This construction is also suited for general exponentiation. The calculation is admittedly rather inaccurate so that only the accuracy of float can be guaranteed.

7.4 ctype.h

This header file does not correspond to any library. In it only a set of macros are defined which can be used like functions.

```
int isupper(c)
    char c;
```

Returns the value 1 (true) if the argument is an upper case letter, else 0 (false).

```
int islower(c)
    char c;
```

Returns the value 1 if the argument is a lower case letter, else 0.

```
int isalpha(c)
    char c;
```

Returns the value 1 if the argument is a letter, else 0. The underscore character is counted as a letter.

```
int isdigit(c)
    char c;
```

Returns the value 1 if the argument is a digit, else 0.

```
int isspace(c)
    char c;
```

Returns the value 1 if the argument is a space, a shifted space, a tab, a new line character, or a shifted new line character.

```
char tolower(c)
    char c;
```

This function converts upper case letters to lower case letters. If the argument is not an upper case letter, the argument will be returned unchanged.

```
char toupper(c)
char c;
```

This function converts lower case letters to upper case letters. It reacts like `tolower`.

8.0 C language description

8.1 Introduction

In this chapter we will discuss the entire range of the C language and the Super C language compiler. Differences between this compiler and the language as described by Kernighan and Ritchie will be pointed out. In general however, most compilers are quite compatible, including this one. C programs can be directly transported except for a few details which usually result from the different hardware configurations.

8.2 Text conventions

The source text of a C program consists of six classes: names (identifiers), keywords, constants, strings, operators, and separators. Spaces, line separators, and comments belong to the separators. This is skipped during the compilation. They serve only to separate neighboring words, constants, etc., where the compiler cannot recognize the relationship without a separation. In each case the compiler tries to interpret the longest string of characters possible as a word, constant, etc.

8.2.1. Comments

Comments begin with `/*` and end with `*/`. They cannot be nested.

8.2.2 Names

An identifier, or name, begins, as in almost every language, with a letter and can then consist of an arbitrarily long sequence of letters or digits. The `_` (underscore) character also counts as a letter. Upper and lower case are distinguished and may be mixed in a name.

The Super C compiler use only the first eight letters to differentiate between names, however. For external names, which must be processed by the LINKER, the same conventions apply. In other compilers this can be different.

8.2.3 Keywords

These are names which have a predefined significance. They may not be used as identifiers:

auto	break	case	char	continue
default	do	double	else	entry
enum	extern	float	for	goto
if	int	long	register	return
short	sizeof	static	struct	switch
typedef	union	unsigned	void	while

No distinction between upper and lower case is made for keywords. AUTO is accepted as auto just as is aUtO. The keywords entry, fortran, asm have no meaning in Super C.

8.2.4 Constants

8.2.4.1 Integer constants

Integer constants are whole-number constants. They consist of a sequence of digits. It is interpreted as a decimal number and has the type `int`. If a digit string starts with 0, the digits following it are interpreted as an octal number. The digits 8 and 9 are interpreted as octal 10 and 11 and are thus allowed.

If a digit string begins with 0x or 0X, the following digits are treated as hexadecimal number. Here the letters a-f or A-F apply as the values 10-15. In Super C, all integer constants are automatically converted to the type `long` if their decimal value is greater than 32767. If an l or L stands behind the integer constant, the constant is always converted to type `long`.

8.2.4.2 Char constants

A char constant consists of a character enclosed in single quotes, such as 'a'. The value of the constant is the value from the character set of the C-64, here 65. The following symbols also count as single characters:

<code>\b</code>	backspace	in Super C:	DELETE	\$14
<code>\t</code>	tab	in Super C:	SPACE	\$20
<code>\n</code>	line separator	in Super C:	CARRIAGE RETURN	\$0d
<code>\r</code>	carriage return	in Super C:	SHIFT RETURN	\$8d
<code>\e</code>	escape	in Super C:	ESCAPE	\$1b
<code>\\</code>				
<code>\'</code>				
<code>\"</code>				
<code>\0</code>	\$00			
<code>\ddd</code>	d are octal digits, returns the value of the constant Oddd, for example <code>\24</code> corresponds to <code>\b</code> in Super C (see character set table)			

All characters in the character set can be accessed with `\ddd`. If a character other than the ones given here is placed after the escape code character, the escape code symbol is ignored.

8.2.4.3 Floating-point constants

A floating-point constant consists of a sequence of digits which represent the integer portion of the constant, followed by a decimal point and a sequence of digits for the fractional portion. Finally comes the exponent, given with `e` or `E` and a sequence of digits with an optional sign. Either a decimal point or an exponent must be present for the compiler to recognize the number as floating-point. Floating-point constants have the type `double`.

8.2.5 Strings

As already mentioned, a *string* is a string of characters. It consists of a sequence of characters enclosed in double quotes. The number of characters in a string constant can vary between 0 and 254 in Super C. A string is viewed as an array of characters with storage class `static` and initialized with the given characters. The compiler automatically appends a `\0` character at the end of the string in order to recognize this.

All of the escape code symbol combinations in Section 8.2.4.2 can also be used within a string. If an escape code symbol stands at the end of the line in the source text, it is ignored and the compiler skips the end of the line, meaning that the string can be continued on the next line.

8.2.6 Example

Here are some examples and their interpretations:

<code>2</code>	<code>-></code>	<code>2</code>	<code>int</code>
<code>2L</code>	<code>-></code>	<code>2</code>	<code>long</code>
<code>010</code>	<code>-></code>	<code>8</code>	<code>int</code>
<code>0xffff</code>	<code>-></code>	<code>65535</code>	<code>long</code>
<code>1.5</code>	<code>-></code>	<code>1.5</code>	<code>double</code>
<code>1.5E2</code>	<code>-></code>	<code>1500.0</code>	<code>double</code>
<code>1.5e-2</code>	<code>-></code>	<code>0.015</code>	<code>double</code>
<code>.5</code>	<code>-></code>	<code>0.5</code>	<code>double</code>
<code>1e5</code>	<code>-></code>	<code>100000</code>	<code>double</code>
<code>"\""</code>	<code>-></code>	<code>string</code>	<code>"\0</code>
<code>"abc\n"</code>	<code>-></code>	<code>string</code>	<code>abc\n\0</code>

8.3 Object names

To clarify this term, we must first clarify the term "object." By object we mean a certain contiguous area of memory with a specific length within a C program. In BASIC an object is comparable to a variable.

As a rule each object has a name. With this name you can access that object, by writing something to it or reading something from it. An object in C has two attributes: the storage class and the type. The location and lifetime of an object are determined by its storage class. The type of the object determines the interpretation of the value from the memory area of the object.

In order to inform the compiler what storage class and what type the object has, the name of the object must be declared. If an object is created at a declaration, then it is called a definition.

8.3.1 Storage classes

There are four storage classes in C: `auto`, `static`, `extern`, and `register`. Objects with the storage class `auto` or `register` are local. They exist only as long as execution in the block in which they were defined continues. When the block is exited, the objects are erased. The compiler tries to place `register` objects in hardware registers in order to make faster access possible. If all hardware registers are used, `register` variables are automatically converted to `auto`. In Super C, `register` variables are always converted to `auto` variables because the processor has no registers free.

Variables defined as `static` are accessible only in the block in which they were defined. These objects remain, however, and retain their old values when execution returns to the same block.

Objects declared as `extern` remain available throughout the program. External variables can also be used by separately compiled program fragments. `Static` objects which are defined outside of a block are also available throughout the entire program, but are available only in the program in which they were defined.

8.3.2 Types

The following types are available in C:

`char` objects can accept a character from the character set. The value of a character is always positive after its definition. `char` objects can also be assigned integer numbers.

Other integral types are `short int`, `int` and `long int`. `short int` can be abbreviated `short` and `long int` to `long`. Longer types may not have a smaller value range than shorter. For this reason all types can be implemented with the same size on a compiler. In Super C, `short` and `int` are the same and `long` is twice as large. All integer types can also be defined as unsigned, meaning that their value will be always be interpreted as positive. unsigned `char` can be defined, but is not different from `char` in Super C because the definition of all characters of the character set is positive and the set fills the entire value range of a `char` variable.

`float` and `double` are floating-point types. In Super C `double` is twice as large as `float`.

The type `void` can only be declared for the result of functions. This means that the function returns no type, meaning that it is a procedure in the Pascal sense.

The type `enum` indicates an enumeration type (see Section 8.8.10).

Arrays can be created of all types. An array contains several objects of the same type (array elements).

One can define a pointer to a certain object.

Functions can be programmed which return simple types as results.

You can declare structures (`struct`) which contain a group of objects of various types, or variants (`union`) which contain one object of a group of various types.

These constructions can also be nested.

8.3.3 Hardware-specific type data

The special type properties of Super C are listed in the following table. This can naturally be different in other compilers. The only guarantee is that the value range of *short* \leq *int* \leq *long* and that of *float* \leq *double*.

Type (written out)	Abbrev.	Value range	Size
short int	short	-32768 to +32767	2
int	-	-32768 to +32767	2
long int	long	-2147483648 to 2147483647	4
unsigned short int	unsigned short	0 to 65535	2
unsigned int	unsigned	0 to 65535	2
unsigned long int	unsigned long	0 to 4294967295	4
char	-	a character from the character set or 0 to 255	1
unsigned char	-	a character from the character set or 0 to 255	1
float	-	+/-9.09E-77 to +/-6.78e+74 accurate to 6 or 7 places	4
long float	double	+/-9.09E-77 to +/-6.78e+74 accurate to 16 places	8

8.4 Objects and L-values

An object is, as mentioned, a memory area. An L-value is an expression which denotes an object. The simplest L-value is a name which is defined. In C however an expression can also yield an L-value. This is done with pointers. If **E**, for example, contains a pointer to the type *int*, ***E** is an L-value and refers to the *int* object to which E points.

8.5 Conversion of a type

Various type conversions are performed depending on the operators.

8.5.1 Integer values between each other

The conversion of integer values between each other is done so that the sign is retained when converting to a longer integer value. The most-significant bits are cut off when converting to a smaller type.

Converting a signed integer value to an unsigned value succeeds only through different interpretation. Negative values are represented in two's complement in Super C.

8.5.2 Floating-point values between each other

Floating-point calculations occur only in the type *double* in C. *float* values are automatically converted to *double*. If a floating-point value is assigned to a *float* variable, it is first converted to *float*. This is done by rounding the mantissa.

Converting from *float* to *double* is done by appending zero-bits.

8.5.3 Floating-point and integer values

The manner in which floating-point and integer values are converted among each other depends on the compiler. The only guarantee is that if the floating-point number has a reasonable number, it can be converted. If the floating-point number cannot fit in the integer number, however, the result is not guaranteed.

8.5.4 Addresses and integer values

The conversion of an integer value to an address and back is performed without change. Only the type of the value changes. This conversion is not performed automatically.

8.5.5 The standard conversions

The "standard conversions" are performed by most of the operators:

1. char or short operands are converted to int, float to double operands.
2. if one of the two operands is double, the other is converted to double and the result is double.
3. if one of the operands is long, the other operand is converted to long and the result is long.
4. if one of the operands is unsigned, the other operand is converted and the result is unsigned.
5. if both operators are of type int, the result is also int.

8.6 Syntax notation

For a better understanding of the next section, we offer a C grammar. At the start of each grammar definition stands a name which is defined. Usually, several alternatives follow with which the name can be replaced. Letters and characters in bold face must not be changed. Names in normal type can be replaced by the corresponding definition of a name. An alternative stands in each line within a definition.

Sections which are enclosed in square brackets [] can be omitted. Sections in braces { } can be repeated.

8.7 Expressions

An expression consists of operands and operators. $a+b$ is an expression. a and b are the operands of the operator $+$.

A distinction is made between unary and binary operators. Unary operators operate on only one operand, binary on two. A binary operator stands between the two operands.

Each operator has a set precedence to determine the order in which the operators are executed. If operators having the same precedence stand on the same line, the processing direction determines the order of evaluation (left to right or right to left).

Apart from the precedence, the order of processing is not defined, meaning that it is up to the compiler to determine how expression fragments will be nested in order to make optimizations, even if the expression fragments create side effects through assignments, etc. Associative and commutative operators can be switched arbitrarily, even when explicit parentheses are present. A specific order of evaluation can be guaranteed only by assigning (temporary) variables.

The handling of errors during the evaluation of an expression depends on the compiler in question. In general, an overflow in an integer operation is ignored.

8.7.1 Simple expressions

A simple expression (*operand*) is, for example, a name or constant (including string constant). First the syntactic definition:

```
operand:
  name
  constant
  string
  ( expression )
  operand ( [argument list] )
  operand [expression]
  operand . name
  operand -> name
```

```
argument list
  assignment { , assignment }
```

A name is usually an L-value. If it refers to a function or array, however, it is to be treated as a constant which represents the address of the function or the array. A name from an `enum` specifier is only a constant. The name of a structure or variant, on the other hand, is an L-value.

An expression enclosed in parentheses is a simple expression. Because the parentheses have highest precedence, the expression within the parentheses is evaluated first. The compiler can remove the parentheses in associative expressions such as `a+(b+c)`, however.

If a parenthesized argument list follows a simple expression, the whole thing is handled as a simple expression, a function call. The left part then represent the address of the function. In the simplest case this is the name of the function. The list in the parentheses contains the arguments which are to be passed to the function. The arguments can themselves be expressions.

The use of a free comma (not parenthesized) is not allowed because it is found in the above definition assignment. If the type of such an expression is `char` or `short` it is converted to `int`. The type `float` is converted to `double`. The argument list can also be empty. A function call is not an L-value.

If an expression in square brackets follows a simple expression, this is again a simple expression. The left part then represents the address of an array. In the simplest case this can be the name of the array. The whole thing is the selection of an array element. The expression must have an integer value. The whole expression is an L-value. Internally, the simple expression $a[b]$ is converted to $*(a+(b))$. To understand this you must first understand the operators $*$ and $+$.

The arguments are passed to the function exclusively by copying the value (*call by value*). The parameters of the function are simply assigned the values of the arguments. The function parameters can be changed as desired without changing the original arguments. This also applies to pointer values (addresses). The object can be changed from the function via the address, however.

The order in which the arguments are evaluated is not defined. Watch out for side effects, such as with assignments in arguments.

Functions can also be called recursively, meaning that a function calls itself. An argument of a function can be a call to the same function.

If a simple expression is followed by a . (period) character or by an arrow (\rightarrow from a minus sign and the greater-than character), it is treated as a reference to a structure or variant. This is a simple expression. If a . (period) is present, the expression on the left should refer to a structure or union. If an arrow is present, an address of a structure or union should be on the left. The right portion must always be the name of a structure or union component. The whole expression represents the selected component as object and is therefore an L-value. $A\rightarrow B$ is internally replaced by $(*A).B$.

8.7.2 Unary Operators

Unary operators are evaluated from left to right. None of the operators yield an L-value except for `*`.

unary:

- operand
- operand ++
- operand --
- `* unary`
- `& unary`
- `- unary`
- `! unary`
- `~ unary`
- ++ unary
- unary
- (type spec) unary
- sizeof unary
- sizeof (type spec)

The operand of the unary operator `*` must be an address or a pointer. The result is an L-value which refers to the object to which the address points.

The unary operator `&` requires an L-value as operand. The result is the address of the object referred to. This operator is to a degree the opposite of the `*` operator.

The unary operator `-` returns the negative value of its operand. With integer values the negative is computed using two's complement. This also applies for **unsigned** values. There is no unary `+` operator in C.

The `!` operator returns the logical negation. The logical value zero is false, the logical value of all other values is true. If the operand is zero, `!` returns the value 1; if the operand is not zero, `!` returns zero.

The `~` operator inverts the individual bits of an integer value and thereby computes its one's complement. The operand must have an integral type.

The operators `++` and `--` add or subtract 1 from their operand (increment, decrement). The operands must be L-values. The result of the expression depends on whether the operator is placed before or after the operand. If the operator is in front, the result is the value of the object after the increment or decrement, while if the operator is behind, the object is incremented or decremented after the evaluation.

Converting a value from one type to another is done with the `cast`. A type specifier in parentheses stands in front of the operand. The operand is then converted to the given type. An example of the type specifiers is found in Section 8.8.12.

The `sizeof` operator returns the size of the operand. Applied to an L-value, one receives the length of the designated object. If the operator is applied to other values, one receives the length of the type of the value. A type can be directly given by placing a type specifier in parentheses. The length is measured in bytes. The operation represents an `int` constant with the length as the value.

8.7.3 Multiplication, Division

The operators `*` / `%` fall into this category. They are processed from left to right and the standard conversions are performed.

multiplication:

- unary
- multiplication `*` unary
- multiplication / unary
- multiplication `%` unary

The binary `*` operator denotes multiplication. It is commutative and associative.

The `/` operator denotes division, the `%` operator the remainder of the corresponding division. On most compilers the remainder has the same sign as the dividend. If the divisor is not zero, $(A/B) * B + A \% B - A$ is equal to zero.

The `%` operator may be used only on integer values.

8.7.4 Addition, subtraction

The operators + and - are evaluated from left to right. The standard conversions are performed. Addresses and pointers can also be combined.

addition:
multiplication
addition + multiplication
addition - multiplication

+ denotes addition, - subtraction. The + operator is commutative and associative so that rearrangement by the compiler are possible.

A pointer value and an integer value can be added. It is then assumed that the pointer points to an array. The result is an address which points as many elements farther as the integer value is large. If A is an array, A+1 is the address to element 1 (second element) of the array.

An integer value can also be subtracted from a pointer value. As a result one receives an address which points the appropriate number of elements previous. The pointer value must always be on the left.

Two pointer values can be subtracted from each other. The result is the number of array elements between the addresses. A necessary condition for a reasonable result is that both pointers point in the same array. This is not checked by the compiler.

8.7.5 Shift operations

The shift operators << and >> are evaluated from left to right. The two operands must be of integral type. The result has the type of the left operand.

shift:

addition

shift << addition

shift >> addition

The value of $A \ll B$ is the bit pattern of A shifted B bits to the left. Zero-bits are shifted in on the right. $A \gg B$ is, correspondingly, the bit pattern of A shifted right. If A is an unsigned value, zero-bits are shifted in from the left. It is dependent on the system whether zero-bits or the sign bit will be shifted in from left if the value is signed. Sign bits are shifted in on the Super C compiler.

8.7.6 Comparisons

Comparisons are evaluated from left to right. This property is mentioned as a warning before use. $A < B < C$ does not yield the expected result. The comparison $A < B$ returns the result 0 for false, 1 for true. Then a comparison is made to see if C is greater than 0 or 1.

comparison:

shift

comparison < shift

comparison <= shift

comparison > shift

comparison >= shift

The operators < (less than), <= (less than or equal), > (greater than), and >= (greater than or equal) return 0 for false and 1 for true. The result type is always *int*. The standard conversions are performed before the comparison.

Pointer values may also be compared whereby there machine addresses are used. Such comparisons are only portable to other systems when both pointers point in the same array.

8.7.7 Equivalence comparisons

The compare operators == (equal) and != (not equal) behave like the compare operators above. They have a lower precedence, however, so that the following expression makes sense: $A < B == C > D$ returns the value 1 if $A < B$ and $C > D$ are both false or both true.

equivalence:
 comparison
 equivalence == comparison
 equivalence != comparison

Pointer values may also be compared with integer values. This is not portable, however. The only guarantee is that the pointer value will never be equal to the integer value 0 if the pointer actually points to an object. Pointers which are not supposed to point to any object can be assigned the value 0. The constant NIL is defined as an address to no object in the standard declarations of Super C. You are warned against an access to such an address since this processor register can be changed, leading to a system crash. Before each address it should be ascertained that the pointer value does not equal NIL.

8.7.8 Bit operations

The operators **&** (and operation), **~** (exclusive or), and **|** (or) combine their operands bit by bit. The operands must be integer values. The standard conversions are performed.

bitwise-and:
equivalence { **&** equivalence }

bitwise-xor:
bitwise-and { **~** bitwise-and }

bitwise-or:
bitwise-xor { **|** bitwise-xor }

The bit operators are commutative and associative and can be rearranged by the compiler.

If a and b are corresponding bits of the left and right operands, then:

a AND b is 1 if both bits a and b are 1
a OR b is 1 if at least one of the two bits is 1
a XOR b is 1 if a and b are different (not both 1 or both 0)

8.7.9 Logical operations

There are two logical operations in C, **&&** (AND) and **||** (OR). The operands are guaranteed to be evaluated from left to right. The result of the **&&** operator is 1 if both operands are non-zero, else the result is 0.

log-and:
bitwise-or { **&&** bitwise-or }

The second operand is evaluated only if the left operand is not zero.

The result of the **||** operator is zero if both operands are zero, else it is one. The second operand is evaluated only if the first is zero.

log-or:

log-and { || log-and }

The operands can be completely different types, but they must permit a comparison to zero. The result type is *int*.

8.7.10 Condition evaluation

selection:

log-or

log-or ? selection : selection

The first expression is evaluated. If its value is not zero, the second expression is evaluated, otherwise the third. Only one of the last two operands is evaluated. The result is the value of the evaluated expression. The standard conversions are performed on the last two expressions if possible, in order to get the same result type in both cases. Otherwise the result types must be two addresses which point to objects of the same type.

8.7.11 Assignments

All assignment operations are evaluated from right to left. The left operand of an assignment must be an L-value. The type of the result is always that of the left operand. The result is the value assigned.

assignment:

selection

unary = assignment

unary *= assignment

unary /= assignment

unary %= assignment

unary += assignment

unary -= assignment

unary >>= assignment

unary <<= assignment

unary &= assignment

unary ~= assignment

unary |= assignment

With the simple assignment `=` the value of the right operand is converted to the type of the left and then assigned to the object to which the L-value refers.

The result of a complex assignment of the form `A op= B` is the same as that of the assignment `A = A op (B)`. `A` is evaluated only once however. The left operand may be a pointer with `+=` and `-=`.

A C compiler allows assignments of pointer values to integer objects and vice versa, as well as assignments of pointer values which point to objects of different types. This assignment is done purely by copying the value and may not be portable to other machines. The only guarantee is the portability of assigning the constant zero (NIL) to a pointer value.

8.7.12 Lists

Two expressions separated by a comma are evaluated from left to right. The result is the value of the right expression.

expression:
assignment { , assignment }

In a situation in which the comma has another meaning, such as in an argument list or in initializations, the comma operator can be used only in parenthesis. Thus the following function call

```
f ( 4 , ( a=3 , a*2 ) , 6 )
```

has the arguments 4, 6, and 6.

8.8 Declarations

A declaration determines how names will be processed by the compiler. The name is connected to a type and a storage class in a declaration. The compiler can then recognize what type the object is and to which the name refers. If an object is created in a declaration it is called a definition.

Declarations with the storage class **extern** do not reserve any memory space. They serve only to make objects known prior to their definition or to refer to an object which is defined in another separately compiled file.

A C program consists of a sequence of global declarations. The definition of the function **main** must be found in one of several separately compiled program segments. The execution of the C program begins and ends with this function.

Names can also be declared locally, meaning that they are declared within a block in a function definition.

```
c-program:
    { global }

global:
    function-definition
    global-definition ;
    declaration ;
    type declaration ;
local:

    local-definition ;
    declaration ;
    type-declaration ;
```

8.8.1 Storage classes

There are three storage classes for definitions in C: **auto**, **static**, and **register**, which were already described in section 8.3.1.

storage-class:

auto
register
static

The **&** operator cannot be used on objects of storage class **register**. As a rule, the **register** storage class is used to make programs faster and shorter. The microprocessor on the C-64 does not allow us to make use of this storage class, however. If no storage class is given, **auto** is assumed inside a block. Outside a block the declaration is assumed to be a global definition.

8.8.2 Types

The following may be used as type names:

type-name:

[unsigned] [short] int
[unsigned] [long] int
[unsigned] short
[unsigned] long
[unsigned] char
[long] float
double
void
struct-union-type-name
enum-type-name
typedef-name

A declaration may contain only one type name. If the type name is missing, **int** is assumed.

8.8.3 Data definitions

Data definitions serve to create data objects. The definitions contain storage class and type specifiers and a sequence of declarators. Each declarator contains a name which is to be declared. The defined objects can be initialized to a certain value in the definition. Local objects can be initialized only with simple types.

global-definition:

static [type-name] i-declarators
type-name [**static**] i-declarators

local-definition:

storage-class [type-name] i-declarators
type-name [storage-class] i-declarators

declaration:

extern [type-name] declarators
type-name **extern** declarators

Declarations declare a sequence of names in declarators. They cannot be initialized. A corresponding data definition must be located in some part of the C program.

8.8.4 Type declarations

type-declaration:

typedef [type-name] declarators
type-name **typedef** declarators
struct-union-type-name

The names contained in the declarators are declared as type names (**typedef-name**). The type represented is what was declared.

A **struct-union-type-name** also applies as a type declaration in case a **struct-name** or **union-name** is defined in it. This definition assigns a specific configuration of components to the name.

8.8.5 Functions

function-definition:

```

static [ type-name ] f-declarator par-declaration block
type-name [ static ] f-declarator par-declaration block

```

Functions can have the storage class **static** or they may be global. A function definition consists of one function declarator, the parameter declaration, and the function block.

8.8.6 Declarators

Declarators serve to declare a name. The name is used in declarator as it could be used in an expression. If the name is used in an expression exactly as in the declarator, the expression has the same type as the type name given in the declaration. This may seem peculiar, but is absolutely unambiguous.

declarator:

```

{ * } declarator
( declarator )
declarator ( )
declarator [ [ constant ] ]
name

```

It is easy to see that the simplest declarator is a name:

```
type name;
```

defines **name** as an object of type **type**. If **name** is supposed to be a pointer to an object of type **type**, a ***** character must be added in front of **name**:

```
type * name;
```

One can see that if the expression ***name** is used in an expression, its type is **type** because the expression refers to the object to which **name** points.

If an array is to be declared, it looks like:


```
type name[constant]
```

name is then a vector with as many elements as the constant indicates. **name** alone is the constant address to the start of this array and not an L-value.

Functions are declared by placing parentheses after the name:

```
type name()
```

name is now a function which returns a value of type **type**. The definition of a function is discussed in the next section. A name which is defined as a function represents the constant address of the function.

These various declarators can be nested in order to declare more complex types. Parenthesis have a higher precedence than the * character. The declarator can also be parenthesized to change the precedence.

Let us take a look at the following declarations:

```
int (*f)(), *g(), *h[5];
```

f is defined as a pointer to a function which returns a value of type **int**. **g** is a function which returns a pointer value to an **int** object. **h** is an array with five elements which are all pointers to objects of type **int**. Experience has shown that it can be very difficult to determine the type from a declaration at the start.

The following syntax definitions finish up the normal declarations:

i-declarators:

```
declarator [=initializer] {,declarator [=initializer]}
```

declarators:

```
declarator {, declarator}
```

8.8.7 Function declarator

A function declarator is only slightly different from a normal declarator. Instead of a name, a name with a parameter list must be given.

f-declarator:

```
{ * } f-declarator
( f-declarator )
f-declarator ( )
f-declarator [ [ constant ] ]
name ( name-list )
```

name-list:

```
[ name ]
name { , name }
```

The parenthesization of the name list identifies the name as a function. The name list can also be empty. It specifies the parameters.

8.8.8 Parameter declaration

par-declaration:

```
{ register [ type-name ] declarators ; }
{ type-name [ register ] declarators ; }
```

The parameter declaration declares the types of the parameters in the order in which they occur in the name list of the function declarator. The objects generated can be used like **auto** or **register** objects. They are initialized with the values of the arguments when the function is called.

Parameters of type **char** are converted to **int**, type **float** becomes **double** automatically. Parameters of type *array* become type *pointer* because the array can be used like a pointer as a parameter; it's an L-value.

8.8.9 Structures and unions

Structures and unions are declared like other objects. A special type-name is used for them:

struct-union-type-name:

```
struct [struct-name] {{ c-declaration }}  
struct struct-name  
union [union-name] {{ c-declaration }}  
union union-name
```

c-declaration:

```
type-name c-declarator {, c-declarator} ;
```

c-declarator:

```
declarator  
[ declarator ] : constant
```

The component declarations in braces are call **struct** or **union** specifiers. A **struct** or **union** name can always be given. If a specifier follows it, the name is defined by the specifier. Only the name need be given for a new declaration.

A component is declared like a normal declaration. The option in italics to declare bit fields as components is not possible in Super C.

A structure or variant may be declared as a component. If the structure or variant is of the same type as that being declared, only pointer may be defined.

A specifier is not allowed within a component declaration. The specifier must be defined outside the structure with its own **struct** name.

8.8.10 Enumeration type

The enumeration type `enum` has its own type name.

enum-type-name:

```
enum [enum-name] { enumerator {, enumerator } }  
enum enum-name
```

enumerator:

```
name [= constant]
```

The specifier can be defined via a name as with structures. The constants of the enumeration type are enumerated in the specifier. The constants are numbered from 1 on. If a constant is given explicitly in an `enum`, it is accepted. The next `enum` constants will be defined beginning with the next highest value.

Objects of the enumeration type behave like `int` objects. They serve only to make a program more readable and understandable. The programmer must ensure that an object of the enumeration type is assigned a value from the specifier. The compiler does not check this.

The defined constants can be used in the program text like `int` constants.

8.8.11 Initializations

initializer:

```
assignment  
constant  
{ initializer {, initializer} }
```

Simple types are initialized by appending an equals sign and a constant to their declarator. Complex types like arrays and components are initialized by a list of constants enclosed in braces. This procedure can be nested as desired.

```
int x [3] [3]= { {0,1,2} ,  
                {3,4,5} ,  
                {6,7,8} };
```

This definition initializes a two-dimensional array with three elements in each dimension. The values 0,1, and 2 are assigned to the elements `x[0][0]`, `x[0][1]`, and `x[0][2]`, and so on.

The list for arrays and structures need not be complete. If fewer elements than necessary are given, the rest are automatically initialized with zero.

If all elements or components are initialized, one can eliminate the nested listing. The above definition can also look like:

```
int x [3] [3]= { 0,1,2,3,4,5,6,7,8 };
```

The compiler assigns the values to the elements or components in order.

Functions and variants cannot be initialized. Only simple types of `auto` objects can be initialized. In contrast to other initializations, however, entire expressions can be initialized (assignment in the syntax definition).

Static and global objects are automatically initialized to zero if no other initializer is given. `auto` objects without initializer have an undefined value.

8.8.12 Abstract declarators

Abstract declarators serve to specify a type in a CAST.

```

type-spec:
  type-name [a-declarator]

a-declarator:
  { * } a-declarator
  ( a-declarator )
  a-declarator ( )
  a-declarator [ [constant] ]

```

An abstract declarator does not contain a name. The compiler can always determine where the name would have stood, so this construction is unambiguous.

```
int *()
```

is, according to this, a function which returns a pointer to *int*.

8.9 Statements

Statements are normally executed in sequence; the execution path is indicated if this is not the case.

```

statement:
  label statement ;
  block
  expression ;
  while( [expression] ) statement
  do statement while( [expression] ) ;
  for( [expression];[expression];[expression] ) statement
  switch( expression ) block
  if( expression ) statement [ else statement ]
  break;
  continue;
  return [expression] ;
  goto name ;
  ;

```

```
label:
  name : [ label ]
        case constant : [ label ]
        default : [ label ]
block:
  { { local } { statement } }
```

The most common form of a statement is the expression. It normally consists of assignments or function calls.

8.9.1 Blocks

A entire block can also be a statement. Local definitions can again be used in a block. This then applies only within the block. A block is usually used to gather several instructions together, such as behind a loop.

8.9.2 while statement

The **while** statement has the form:

```
while ( expression )
  statement
```

The statement is repeated until the value of the expression is zero. The expression is always evaluated before the statement. If the expression is omitted, the loop is infinite.

8.9.3 do statement

The **do** statement has the form:

```
do
    statement
while( expression );
```

The statement is repeated until the expression is zero. The expression is always evaluated after the statement. Here the statement is executed at least once, whereas it may never be executed with **while**.

8.9.4 for statement

The **for** statement has the following form:

```
for ( expression1; expression2; expression3)
    statement
```

It can be directly converted to a **while** statement:

```
expression1;
while( expression2 )
{
    statement
    expression3;
}
```

All three expressions can be omitted. The semicolons must remain in the parentheses, however. If the second expression is omitted, the loop is infinite.

8.9.5 if statement

An **if** statement can have an option **else** section:

```
if( expression )  
    statement
```

or:

```
if( expression )  
    statement  
else  
    statement
```

The expression is evaluated in both cases. If the value of the expression is not zero, then the statement behind the **if(...)** is executed. If the value is zero, the first statement is skipped and the statement behind **else** (if present) is executed. If several **if** instructions are nested, an **else** is always paired with the last **if**.

8.9.6 switch statement

```
switch( expression )  
    block
```

The **switch** statement causes the execution of the program to branch to one of several instructions. First, the expression is evaluated. It must return an integer value. In Super C, addresses can also be given. **case** labels can stand in the block. Behind each of these labels is a constant. If the constant agrees with the value of the expression, execution continues behind that label. A constant should be found only once behind a **case** label. The constants can also be constant expressions. If no constant matches the value of the expression, execution continues behind the **default** label. If this is not present, the whole block is skipped.

In contrast to other languages, execution starts after the matching label and continues to the end of the block. A **break** statement can be used to prevent this. The **default** label need not come at the end of the block.

The block can contain variables. These will not be initialized, however.

8.9.7 break statement

The last **do**, **while**, **for**, or **switch** statement can be exited with a **break** statement. The execution of the program continued after the interrupted statement.

8.9.8 continue statement

The **continue** statement refers to the last **do**, **for**, or **while** statement. In these loops, **continue** causes a jump the location which determines whether the loop will be repeated or not.

8.9.9 return statement

The **return** statement causes execution to return from a function call. Execution continues after the function call. An expression may stand behind **return**. The expression is converted to the type given in the definition of the function.

If the program execution reaches the end of function block, the compiler supplies a **return** statement without expression.

8.9.10 Labels

A label may be placed in front of any statement. This label consists of a name and a colon. The names is thereby defined as a label and can be jumped to with **goto**.

8.9.11 goto statement

With the **goto** statement one can jump to label. The execution of the program then continues behind this label. Such a statement requires that the name be defined within the same block.

The use of labels as well as **goto**'s is not recommended. They tend to destroy the advantages of structured programming. Also, one should avoid jumping into a block because local definitions will not be performed. No variables are present and therefore also not initialized.

8.9.12 Empty statement

An **empty** statement consists of only a semicolon ;. They are mostly used to place a label at the end of block. For example:

```
...  
label: ;  
}
```

The empty statement is also used to create loops which are not supposed to repeat any statement.

8.10 Scope

By the scope of an object we mean the range of its validity. A distinction is made between two scopes: the scope on which a name is bound, and the scope on which an object is bound.

8.10.1 Scope of a name

By this term we mean the range of the program in which a declared name is tied to its declaration. Static global names apply over the entire source file. Global names declared without storage class apply also to other source files bound to the one in which they are declared and in which a corresponding declaration is made.

Externally declared names refer to a global definition and make this name known globally.

Local predeclarations can be made within a block. Local predeclarations work like global predeclarations in Super C. They serve only to designate once more which global objects will be used in the block. Several declarations of the same name with the same type do not hurt.

All other local names apply only within the defined block. Note that global and also local names can be covered up by declarations in a "deeper" block. The most recent valid declaration always applies within a block.

Another characteristic applies in Super C. All names must normally be declared in C. If one wants to use objects before their definition, they must be predeclared. This is normally only done with global objects. In Super C, static objects can also be predeclared with the storage class **extern**. If you want to prevent objects from applying outside their source files, you may not predeclare these objects.

Note that the compiler can look for global definitions and predeclarations only within one source file. If a name is used in a global definition in one file and a declaration with the same name but different type in another file, the compiler will never discover this. The linker binds these files together without an error message, but the program will probably not work correctly.

There are normally two classes of names in C: first, all **struct**, **union**, **enum**, and component names, and second, all other names. This rule is not implemented in Super C, however. This is not a problem, since it is not a good idea to use the same name for more than one thing in a program.

8.10.2 Scope of an object

By the scope of an object we mean the range in which memory space exists for the object in the program.

For global definitions, the memory space applies over the whole program. If a static object is defined in each of two files which are bound together into one program, they are treated as two separate objects whose memory space exists over the whole program. The memory space only addressible in the file in which it is defined because of the scope of a static name.

Local static objects are retained over the entire program. Only **auto** and **register** objects are created at their declaration and then erased again as soon as the block in which they were defined is left.

8.11 Preprocessor

A C compiler is equipped with something call a preprocessor. The preprocessor alters the source text according to specific rules before it is sent to the actual compiler. This does not change the source text on the diskette. The preprocessor is built into the compiler in Super C and it operates on the text as soon as it is read by the compiler.

All preprocessor commands occupy a separate line in the source text. The first character of a preprocessor line must be a **#** character. The effect of a preprocessor command applies until the end of the source file and is not dependent on the scopes of C declarations.

8.11.1 Macros

Names can be defined as macros with the preprocessor. If these names appear in the program text following, they will be replaced with a replacement string.

```
#define name replacement_string
```

The defined macro name has precedence above all scopes, meaning that it is first checked whether a name is defined as a macro. This also applies for keywords.

A macro definition can also be made with parameters.

```
#define name(name1,name2,...) replacement_string
```

The macro replacement is similar to a function call. An argument list as with a function must follow the defined macro name in the program text. The parenthesis (of the argument list must come directly after the macro name or the preprocessor will recognize it as a macro without parameters. The name and the list are replaced by the replacement string. First, however, all of the names in the replacement string which match the parameter names are replaced with corresponding argument strings from the call. Note that no names may occur in the argument strings which match those in the parameters.

The C preprocessor does not have command of the C language, however. It replaces the text without recognizing its relationship and its meaning. C macros must be used carefully and with consideration.

The macros serve to define program constants and small "functions." A macro call is the concern of the compiler and does not take up any time at the program run time. Complex macro definitions are better realized with functions because these require less space in the C program. The replacement text is recompiled at each macro call.

```
#undef name
```

causes a defined macro to be erased.

8.11.2 Chaining files

Multiple source files can be combined with a preprocessor command.

```
#include "filename"
```

This preprocessor line will be replaced by the entire source text filed under the name **filename** when the program is compiled.

Additional **#include** calls may be found in this file. The files may be nested up to six deep in Super C. As many files as desired can be combined by placing such instructions one after the other in the same file, however.

Chained text files count as one source text. The chaining is not to be confused with the binding of several separately compiled files.

In other C systems the filename can also be enclosed in **<** and **>**, which causes a different search procedure to take place. This command is not necessary because of the size of the floppy.

8.11.3 Conditional compilation

In C, program sections can be selected for compilation. This allows the same source text to be used for various program versions. The selection of the text range to be compiled is done with an **if** statement.

```
#if constant  
#ifdef name  
#ifndef name
```

are the selection instructions. The text following these instructions is selected if the constant after **#if** has a value other than zero, if the name after **#ifdef** is defined as a macro, or if the name after **#ifndef** is not defined as a macro.

In this case the text behind the selection instructions is compiled up to a command:

```
    #endif  
or:  
    #else
```

The last command indicates that there is an **else** portion which is skipped. The **else** portion must be concluded with **#endif** at some point.

If the logical value in a selection statement is false (if the number is zero, etc.), the program section behind the selection statement is skipped. If an **else** portion is present, this is compiled.

The constant after **if** can be a constant expression. The conditional compilation instructions can be nested, up to eight levels in Super C.

8.11.4 Line numbering

In more complex systems the line numbering and source file name can be influenced through the command:

```
#line constant name
```

This is not necessary in Super C and is not implemented.

8.12 Implicit declarations

Certain specifications within a declaration can be omitted. These are then supplemented by default values.

If the storage class is not given in a global definition, it means that the definition applies over the whole program. If no type is given, **int** is assumed.

If no storage class is given in a local declaration, **auto** is assumed. One exception is the declaration of a function which is

assigned the storage class **extern** in local declarations and is thereby only predeclared. If only the storage class is given in a local declaration, **int** is assumed as the type. Both specifications, storage class and type, can not be omitted in a local declaration because the declarator will otherwise be recognized as an expression.

If the compiler does not recognize a name, if the name is not declared, it is automatically predeclared as a global name with the type **int** or as a function which returns type **int**. This should not be overused in larger programs for reasons of style.

8.13 Operations on different data types

8.13.1 Structures and unions

A structure or union cannot be used for all operations. One can select a component with the operators **.** and **->**. The address of a structure or union can be determined with the **&** operator.

In many implementations, structures can be assigned to structures of the same type or passed to functions as arguments. A function may also be able to return a structure as a result. This is not possible in Super C.

In all compilers, pointer to structures and unions can be passed to functions as arguments, of course.

With structures it is possible to avoid the usual type checking. The right operand of the operators **.** and **->** need not refer to the declaration of the left operand; any component declaration is valid. The left operand need only be an L-value and it will be used as a structure or union. With the **->** operator the left operand can be a pointer value. Caution is urged with these constructions. They are not portable.

8.13.2 Functions

Only two things can be done with functions: they can be called or their address can be determined.

The name of a function standing alone in the program represents the address of the function. One can pass functions as arguments, for instance.

```
int a()
{...}
main()
{ ...      b(a);
  ...}
int b( fp )
  int (*fp) ();
  { ...
    (*fp) (...);
    ...}
```

The address of the function **a** is passed to function **b**. Function **a** can be called in **b**.

8.13.3 Arrays, pointers

The identifier of the array alone is always converted to a pointer value to the first element in the array. The name is thereby a constant and not an L-value. The index operator [] is converted to addition. $a[b]$ is converted to $(*(a+(b)))$. **a** is a pointer value and **b** an integer value. The addition works in the conversion such that $(a+(b))$ points to an array element which is **b** elements removed from the first. The * operator generates an L-value from the address. The whole expression corresponds to that which is expected when one uses $a[b]$. This operation is commutative, although it does not look it.

This applies correspondingly for multi-dimensioned arrays. If one has an array;

```
int a[5][4];
```

for example, `a` is first an array. The elements of this array are again arrays. `a[3]` is an array and is treated as such. The elements of this array are `int` elements. The index 3 in this expression means that element three of the array `a` is being handled. The elements are stored line by line in the memory of the object `a`, meaning that the last index varies the fastest. The first element is the array `a[0]`, then the array `a[1]`, and so on.

If the `*` operator is applied to an array, the expression refers to the first element (element 0) of the array. Note that when the number of array elements is given in the declaration of an array, the elements are counted starting at zero.

8.13.4 Conversion of pointer values

A pointer value can be converted to an integer value. In Super C the type `unsigned int` is used. The conversion returns the memory address in Super C.

An integer value can be converted to a pointer value. This is different from machine to machine since larger computers require that the address of an object be divisible by the `SIZE`. This problem does not exist in Super C. In any case it is guaranteed that a conversion from a pointer value to an integer value and back again results in the original value.

8.14 Constant expressions

Constant expressions can be used, for example, after `case`, after `#if`, in an `enum` specifier, and in initialization.

Constant expressions consist of constants and character strings which can be combined with the operators:

`sizeof` - `~`

and all of the binary operators except for assignment and logical operations.

+ - * / % & | ^ << >> == != <= >= ? :

Parentheses can also be inserted. Calling of functions is not allowed.

The addresses of already declared global or static objects can be used as constants with the & operator. Array and function names with indices and argument lists are also handled as constant addresses.

8.15 Portability

Not only the value range of the various types need be noted when transporting programs from one machine to another. The following processing methods are open to the C compilers and, in order to promote portability, should not be used excessively.

In Super C the order of the bytes within an object is always stored from *low* to *high*, the least-significant byte first. The actual processing of **register** objects are handled as **auto** in Super C. The order of the evaluation of arguments need not proceed strictly left to right.

8.16 Differences from standard compilers

Although the Super C compiler understands almost all elements of C, there are a few differences between it and some other compilers which must be mentioned here.

Some compilers understand certain original language elements such as **=+** instead of **+=**. This was changed in later versions. The Super C compiler does not recognize these earlier constructions. If a corresponding program is to be compiled, it must fit or be made to fit the modern standard.

No lists of **auto** can be initialized in Super C. Each initialized **auto** variable must be concluded with **;**:

```
auto int x=5;
auto int y=4;
```

The two name classes for structure names and other names are not realized in Super C. This is not really a problem though, since one should not use the same name for two different things.

Super C also offers possibilities which other systems do not offer. Do not use these in programs which are to transported to other machines.

Addresses can be given as **case** constants. The specification of a boundary character is possible when reading strings with **scanf**, **sscanf**, and **fscanf**.

Some compilers do not recognize the construction **while ()** as an infinite loop.

8.17 Differences from the C-Compiler 64

The language scope of the Super C compiler has hardly changed from from that of its predecessor Super C-64. Only some standard functions and macro definitions have been changed in order to realize better compatibility to UNIX:

```
stdio.h was called stdio.c
EOF was called EOI in C for the C-64
fgets was called gets
fputs was called puts
fgetf was called getf
fputf was called putf
```

The function **inkey** was removed.

The function **CMOVE** is now called **cmove**.

The return values of the functions **strcpy**, **strcat**, **free**, and **gets** have changed.

PART III. Appendix

1.0 Keyboard V2

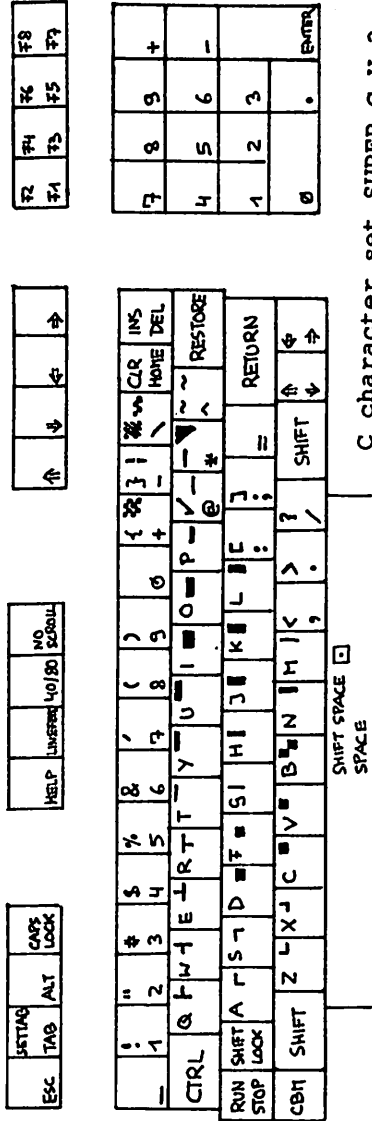
SETAB TAB 1	!"	#	\$	%	&	'	()	*	+	{		~	INS HOME DEL	F2 F1
CTRL	@	1	2	3	4	5	6	7	8	9	0	-	=	RESTORE	F4 F3
RUN STOP LOCK	Q	W	E	R	T	Y	U	I	O	P	[]	;	RETURN	F6 F5
CBM	SHIFT	Z	X	C	V	B	N	M		<	>	?	/	SHIFT	F8 F7
SHIFT SPACE															
SPACE															

C character set SUPER C V-2

SETAB TAB 1	!"	#	\$	%	&	'	()	*	+	{		~	INS HOME DEL	F2 F1
CTRL	@	1	2	3	4	5	6	7	8	9	0	-	=	RESTORE	F4 F3
RUN STOP LOCK	Q	W	E	R	T	Y	U	I	O	P	[]	;	RETURN	F6 F5
CBM	SHIFT	Z	X	C	V	B	N	M		<	>	?	/	SHIFT	F8 F7
SHIFT SPACE															
SPACE															

CBM character set SUPER C V-2

1.2 Keyboard V3 C character set



2.2 CTRL-codes V2

The following CTRL codes can be accessed by keypress and within a program. We have included the corresponding key sequence and the print codes in octal.

000		End character of strings.
003	[STOP]	Stop-key
005	[CTRL]+[2]	white
010	[TAB]	Tab (editor)
011	[SHIFT]+{TAB}	Set. clear tab (editor)
015	[RETURN]	RETURN key, '\n'
016	[CTRL]+[n]	C-character set switch
021	[C-DOWN]	Cursor down
022	[CTRL]+[9]	Reverse on
023	[HOME]	Home
024	[DEL]	Delete
034	[CTRL]+[3]	red
035	[C-RIGHT]	Cursor right
036	[CTRL]+[6]	green
037	[CTRL]+[7]	blue
201	[CTRL]+[1]	orange
203	[SHIFT]+[STOP]	
205	[F1]	F1
206	[F3]	F3
207	[F5]	F5
210	[F7]	F7
211	[SHIFT]+ [F1]	F2
212	[SHIFT]+ [F3]	F4
213	[SHIFT]+ [F5]	F6
214	[SHIFT]+ [F7]	F8
215	[SHIFT]+[RETURN]	
216		CBM key
220	[CTRL]+[1]	white
221	[C-UP]	Cursor up
222	[CTRL]+[0]	Reverse off
223	[SHIFT]+[HOME]	CLR
224	[SHIFT]+[DEL]	Insert
225	[CBM]+[2]	brown
226	[CBM]+[3]	lt. red
227	[CBM]+[4]	dk. grey
230	[CBM]+[5]	lt. green

231	[CBM]+[6]	lt. green
232	[CBM]+[7]	lt. blue
233	[CBM]+[8]	lt. grey
234	[CTRL]+[5]	lt. purple
235	[C-LEFT]	Cursor left
236	[CTRL]+[2]	yellow
237	[CTRL]+[4]	cyan

2.3 CTRL-codes V3

The following CTRL codes can be accessed by keypress and within a program. We have included the corresponding key sequence and the print codes in octal.

000		End character of strings.
002	[CRTL]+[b]	Set bottom window (80 column only)
003	[STOP]	Stop-key
005	[CTRL]+[2]	white
007	[CRTL]+[g]	Bell
011	[TAB]	Tab (editor)
012	[LINE-FEED]	Line-feed
015	[RETURN]	RETURN key, '\n'
016	[CTRL]+[n]	C-character set switch
021	[C-DOWN]	Cursor down
022	[CTRL]+[9}	Reverse on
023	[HOME]	Home
024	[DEL]	Delete
030	[SHIFT]+[TAB]	set, clear tab
033	[ESC]	ESCAPE
034	[CTRL]+[3]	red
035	[C-RIGHT]	Cursor right
036	[CTRL]+[6]	green
037	[CTRL]+[7]	blue
201	[CTRL]+[1]	orange
203	[SHIFT]+[STOP]	
204	[HELP]	HELP
205	[F1]	F1
206	[F3]	F3
207	[F5]	F5

210	[F7]	F7
211	[SHIFT]+ [F1]	F2
212	[SHIFT]+ [F3]	F4
213	[SHIFT]+ [F5]	F6
214	[SHIFT]+ [F7]	F8
215	[SHIFT]+[RETURN]	
220	[CTRL]+[1]	white
221	[C-UP]	Cursor up
222	[CTRL]+[0]	Reverse off
223	[SHIFT]+[HOME]	CLR
224	[SHIFT]+[DEL]	Insert
225	[CBM]+[2]	brown
226	[CBM]+[3]	lt. red
227	[CBM]+[4]	dk. grey
230	[CBM]+[5]	lt. green
231	[CBM]+[6]	lt. green
232	[CBM]+[7]	lt. blue
233	[CBM]+[8]	lt. grey
234	[CTRL]+[5]	lt. purple
235	[C-LEFT]	Cursor left
236	[CTRL]+[2]	yellow
237	[CTRL]+[4]	cyan

2.4 ESC-Codes V3

Escape sequences consist of pressing the ESC key and an additional character. This sequence can be activated by keypresses or in program codes. The ESC character is accessible in SUPER-C with '\e', but this is not compatible with most C systems. For "portable" code use '\33' which is the equivalent of ESCAPE.

[ESC], [1]	enable C character set
[ESC], [2]	enable CBM character set
[ESC], [@]	erase screen from cursor position to end
[ESC], [a]	Auto insert mode on
[ESC], [b]	set bottom of window
[ESC], [c]	Auto insert mode off
[ESC], [d]	Delete one line
[ESC], [e]	cursor off flash
[ESC], [f]	cursor on flash
[ESC], [g]	enable bell
[ESC], [h]	disable bell
[ESC], [i]	insert one line
[ESC], [j]	jump to start of line
[ESC], [k]	jump to end of line
[ESC], [l]	scrolling off
[ESC], [m]	scrolling on
[ESC], [n]	REVERSE off (80 column)
[ESC], [o]	INSERT, QUOTE m RVS off
[ESC], [p]	erase to end of line
[ESC], [q]	erase to start of line
[ESC], [r]	REVERSE on (80 column)
[ESC], [s]	Solid block cursor (80 column)
[ESC], [t]	set top of window
[ESC], [u]	underline cursor (80 column)
[ESC], [v]	scroll up one line
[ESC], [w]	scroll down one line
[ESC], [x]	switch 40/80 column screen
[ESC], [y]	all TABs normal
[ESC], [z]	erase all TABs

3. Function Overview

This list should give all the functions in the SUPER-C libraries. Keep in mind the following:

- (like V1) when the function is identical to the older C-64 version.
- (similar to V1) the functions are similar to the older C-64 version.
- (V3 only) available in version V3 only.

```

void erron()                (like V1)

void erroff()              (like V1)

void nmion()               (like V1)

void nmioff()              (like V1)

int qerror()               (like V1)

void error(string,fnr)     (like V1)
    char *string;
    int  fnr;

void exit()                 (like V1)

file open(prim,sek,name,buffer (similaiar to V1)
    int prim,sek;
    char *name;
    char *buffer; (optional)

file close(fd)             (like V1)
    file fd;

```

`int putc(c, fd)` (like V1)
 `char c;`
 `file fd;`

`int fputc(c, fd)`
 `char c;`
 `file fd;`

`char getc(fd)` (like V1)
 `file fd;`

`char fgetc(fd)`
 `file fd;`

`char getchar()` (like V1)

`int putchar(c)` (like V1)
 `char c;`

`char *gets(line,n)`
 `char *line;`
 `int n;`

`char *fgets(line,n,fd)` (similar to V1 gets)
 `char *line;`
 `int n;`
 `file fd;`

`int puts(line)`
 `char *line;`

`int fputs(line,fd)` (similar to V1 puts)
 `char *line;`
 `file fd;`

`int fgetf(line,n,fd)` (like V1 getf)
 `char *line;`
 `int n;`
 `file fd;`

int fputf(line,n,fd) (like V1)
char *line;
int n;
file fd;

file fopen(name,mode)
char *name, *mode;

file fclose(fd)
file fd;

int strlen(str) (like V1)
char *str;

int strcmp(str1,str2) (like V1)
char *str1,*str2;

int strncmp(str1,str2,n)
char *str1,*str2;
int n;

char *strcat(str1,str2) (like V1)
char *str1,*str2;

char *strncat(str1,str2,n)
char *str1,*str2;
int n;

char *strcpy(str1,str2) (similar to V1)
char *str1,*str2;

char *strncpy(str1,str2,n)
char *str1,*str2;
int n;

char *strchr(str,c)
char *str, c;


```
char *strchr(str,c)
    char *str, c;

void cursor(line,pos)      (like V1)
    int line, pos;

void exec(string)
    char *string;

void cmove(target,n,source) (like V1CMOVE)
    char *target;
    int n;
    char *source;

void move(target,n,source,mem) (nur V2) (wie V1)
    char *target;
    int n;
    char *source;
    int mem;

char *alloc(size)         (like V1)
    int size;

char *free(size)          (like V1)
    int size;

char *settime(string)
    char *string;

char *gettime()

int keys(string)
    char *string;

long call(p)
    char *p;

void fast()                (V3 only)
```

```
void slow()                (V3 only)

void window(lcol,tline,rcol,blne) (V3 only)
    int lcol,tline,rcol,blne;

char vdcin(reg)            (V3 only)
    int reg;

void vdcout(reg,c)        (V3 only)
    int reg;
    char c;

void io>>out(adr,val,colram) (V3 only)
    char *adr,val;
    int colram; (optional)

char io>>in(adr,colram)    (V3 only)
    char *adr;
    int colram; (optional)

int is80()                (V3 only)

void graphic()

void graphon()

void graphoff()

int isgraph()

void backgr(colex,colbk)
    int colex,colbk;

void clrmap(cnr)
    int cnr;

void colors(col1,col2,col3)
    int col1,col2,col3;
```

```
void setcol(col)
    int col;

void dot(x,y)
    int x,y;

int dotin(x,y)
    int x,y;

int bdot(x,y)
    int x,y;

void line(x1,y1,x2,y2)
    int x1,y1,x2,y2;

void bline(x1,y1,x2,y2)
    int x1,y1,x2,y2;

char *mline(x1,y1,x2,y2,put)
    int x1,y1,x2,y2;
    char *put;

char *oline(x1,y1,x2,y2,get)
    int x1,y1,x2,y2;
    char *get;

void setplot(x,y)
    int x,y;

void plot(x,y)
    int x,y;

void shape(x1,y1,x2,y2)
    int x1,y1,x2,y2;

void fill(x,y)
    int x,y;
```

```
char *pushobj(x1,y1,x2,y2,put)
    int x1,y1,x2,y2;
    char *put;
```

```
char *plotobj(x1,y1,x2,y2,get)
    int x1,y1,x2,y2;
    char *get;
```

```
char *fplotobj(x1,y1,x2,y2,get)
    int x1,y1,x2,y2;
    char *get;
```

```
char *bplotobj(x1,y1,x2,y2,get)
    int x1,y1,x2,y2;
    char *get;
```

```
char mapv(x,y)
    int x,y;
```

```
double sin(d)
    double d;
```

```
double cos(d)
    double d;
```

```
double tan(d)
    double d;
```

```
double atan(d)
    double d;
```

```
double abs(d)
    double d;
```

```
double sgn(d)
    double d;
```

```
double rnd(d)
    double d;
```

double sqr(d)
double d;

double sqrt(d)
double d;

double log(d)
double d;

double exp(d)
double d;

int isupper(c)
char c;

int islower(c)
char c;

int isalpha(c)
char c;

int isdigit(c)
char c;

int isspace(c)
char c;

char tolower(c)
char c;

char toupper(c)
char c;

4. Listing of the header files

V2: stdio.h

```
1 /* library headerfile for 'libc.l' and 'libcs.l' */
2 /* C-Compiler V2 Super-C */
3
4 #define STDIO 0
5 #define NULL 0
6 #define CR '\n'
7 #define CRSUP '\221'
8 #define CRSDDOWN '\21'
9 #define CRSRIGHT '\35'
10 #define CRSLEFT '\235'
11 #define DELETE '\b'
12 #define INSERT '\224'
13 #define HOME '\23'
14 #define CLR '\223'
15 #define REVERSON '\22'
16 #define REVERSOFF '\222'
17
18 #define NIL 0
19 #define EMPTY ""
20 #define MAXINT 32767
21 #define MAXLONG 2147483647L
22
23 #define ST (*(char*)0x90)
24 #define EOF (ST & 0x40)
25
26 #define putchar(X0) putc(X0,STDIO)
27 #define cmove(X1,X2,X3) move(X1,X2,X3,0x35)
28
29 typedef int file;
30
31 extern file open(),close(),fopen(),fclose();
32 extern int putc(),getc(),fgetc(),fputc();
33 extern char *gets(),#fgets();
34 extern int puts(),fputs(),fgetf(),fputf();
35 extern void erron(),erroff(),nmaion(),nmioff();
36 extern void error(),exit();
```

PAGE: 2 stdio.h
DATE: 4/21/86

```
37 extern int  qerror();
38 extern void cursor(),move(),exec();
39 extern int  strlen(),strcpy(),strncap();
40 extern char *strcat(),*strncat(),*strcpy(),*strncpy();
41 extern char *strchr(),*strrchr();
42 extern char *alloc(),*free();
43 extern char *settime(),*gettime();
44 extern int  keys();
45 extern long call();
46
47 extern void printf(),sprintf(),fprintf();
48 extern int  scanf(), sscanf(), fscanf();
49
50 char (*screen)[40] = 0xe000;
51 char (*color ) [40] = 0xd000;
52 char (*charram1)[8]= 0xd000;
53 char (*charram2)[8]= 0xd000;
54
55 char getchar()
56 ( char c;
57
58   while((c=getc(STDIO))!=0);
59   return c;
60 }
61 .
```

V3: stdio.h

```

1 /* library headerfile for 'libc.l' and 'libcs.l' */
2 /*      C-Compiler V3 Profi-C/Super-C      */
3
4 #define STDIO 0
5 #define NULL 0
6 #define CR  '\n'
7 #define CRSUP  '\221'
8 #define CRSDOWN  '\21'
9 #define CRSRIGHT  '\35'
10 #define CRSLEFT  '\235'
11 #define DELETE  '\b'
12 #define INSERT  '\224'
13 #define HOME  '\23'
14 #define CLR  '\223'
15 #define REVERSON  '\22'
16 #define REVERSOFF  '\222'
17
18 #define NIL  0
19 #define EMPTY  ""
20 #define MAXINT  32767
21 #define MAXLONG  2147483647L
22
23 #define ST  (*(char*)0x90)
24 #define EOF  (ST & 0x40)
25
26 #define putchar(X0)  putc(X0,STDIO)
27
28 typedef int file;
29
30 extern file open(),close(),fopen(),fclose();
31 extern int  putc(),getc(),fgetc(),fputc();
32 extern char *gets(),*fgets();
33 extern int  puts(),fputs(),fgetf(),fputf();
34 extern void  erron(),erroff(),naion(),naioff();
35 extern void  error(),exit();
36 extern int  qerror();
37 extern void  cursor(),cmove(),exec();
38 extern int  strlen(),strcpy(),strncap();
39 extern char *strcat(),*strncat(),*strcpy(),*strncpy();

```


PAGE: 2 stdio.h

DATE: 4/21/86

```
40 extern char *strchr(),*strrchr();
41 extern char *alloc(),*free();
42 extern char *settime(),*gettime();
43 extern char io_in(),vdcin();
44 extern int is00(),keys();
45 extern void io_out(),vdcout(),fast(),slow(),mindow();
46 extern long call();
47
48 extern void printf(),sprintf(),fprintf();
49 extern int scanf(),sscanf(),fscanf();
50
51 char (*screen)[40] = 0x0400;
52 char (*color ) [40] = 0xd000;
53 char (*charraa1)[8]= 0x1000;
54 char (*charraa2)[8]= 0x1000;
55
56 char getchar()
57 ( char c;
58
59 while((c=getc(STDIO))!=0);
60 return c;
61 }
62
```

V2: graphic.h

PAGE: 1 graphic.h

DATE: 4/21/86

```

1 /* headerfile for 'libgraph.l' */
2 /* C-Compiler V2 Super-C */
3
4 extern void graphic(),graphon(),graphoff();
5 extern void backgr(),colors(),clrmap(),setcol(),fill();
6 extern int dot(),dotin(),bdot();
7 extern int line(),bline(),shape();
8 extern int isgraph(),setplot(),plot();
9 extern char mapv(),*mline(),*oline();
10 extern char *plotobj(),*pushobj();
11 extern char *bplotobj(),*fplotobj();
12
13 extern char mask, *bmap;
14
15 #define black 0
16 #define white 1
17 #define red 2
18 #define cyan 3
19 #define purple 4
20 #define green 5
21 #define blue 6
22 #define yellow 7
23 #define orange 8
24 #define brown 9
25 #define pink 10
26 #define dgrey 11
27 #define grey 12
28 #define lgreen 13
29 #define lblue 14
30 #define lgrey 15
31
32 char (*video)[40]=0x8c00, /* Video ram in Graphic-Mode
33      *bitmap=0xa000, /* Bit map in Graphic-Mode
34      *vic=0xd000; /* Base address Video Interface Chip */
35
36

```

V3: graphic.h

PAGE: 1 graphic.h
DATE: 4/21/86

```

1 /* headerfile for libgraph.l */
2 /* */
3
4 extern void graphic(),graphon(),graphoff();
5 extern void backgr(),colors(),clrmap(),setcol(),fill();
6 extern int dot(),dotin(),bdot();
7 extern int line(),bline(),shape();
8 extern int isgraph(),setplot(),plot();
9 extern char mapv(),*mline(),*oline();
10 extern char *plotobj(),*pushobj();
11 extern char *bplotobj(),*fplotobj();
12
13 extern char mask, *bmap;
14
15 #define black 0
16 #define white 1
17 #define red 2
18 #define cyan 3
19 #define purple 4
20 #define green 5
21 #define blue 6
22 #define yellow 7
23 #define orange 8
24 #define brown 9
25 #define pink 10
26 #define dgrey 11
27 #define grey 12
28 #define lgreen 13
29 #define lblue 14
30 #define lgrey 15
31
32 char (*video)[40]=0xe000, /* Video ram in Graphic-Mode */
33 *bitmap=0xc000; /* Bit map of Graphic-Mode */
34
35
36

```

V2/V3: ctype.h

PAGE: 1 ctype.h

DATE: 4/21/86

```
1 /#      headerfile 'ctype'      */
2 /* C-Compiler V2/V3 Super-C */
3
4 #define isupper(X) (X)='A' & X<='Z')
5 #define islower(X) (X)='a' & X<='z')
6 #define isalpha(X) ((X)='A' & X<='Z')|(X)='a' & X<='z')|X=='_')
7 #define isdigit(X) (X)='0' & X<='9')
8 #define isspace(X) (X==' ' |X=='\t' |X=='\n' |X=='\r')
9
10 #define tolower(Y) (isupper(Y) ? Y & 0x7f : Y)
11 #define toupper(Y) (islower(Y) ? Y | 0x80 : Y)
12
13 /*      header file 'ctype'      */
14
```

V2/V3: math.h

PAGE: 1 math.h

DATE: 4/21/86

```
1 /*      headerfile for 'libmath.l'  */
2 /* C-Compiler V2/V3 Profi-C/Super-C */
3
4 #define PI 3.14159265359
5 #define E 2.71828182846
6
7 extern double sin(),cos(),tan();
8 extern double atan(),abs(),sgn();
9 extern double sqr(),sqrt(),rnd();
10 extern double log(),exp();
11
12
```

5. Listing "text.c"

PAGE: 1 text.c
DATE: 4/21/86

```
1 We can display 16 colors
2 on the screen. The source text does
3 not produce them. The colors
4 are used to high-light the most important
5 lines of the source text.
6
7
8
9 Color list:
10 CBM+1 black
11 CBM+2 white      ^^black^^
12 CBM+3 red
13 CBM+4 cyan
14 CBM+5 violet
15 CBM+6 green
16 CBM+7 blue
17 CBM+8 yellow
18 CTRL+1 orange
19 CTRL+2 brown
20 CTRL+3 light red
21 CTRL+4 dark gray
22 CTRL+5 gray
23 CTRL+6 light green
24 CTRL+7 light blue
25 CTRL+8 light gray
26
27
28
29 The Editor can display two character sets:
30
31 the BASIC-character set and a
32 special C-character set
33
34 !"#%&'()*+,-./0123456789;<=>?
35 @abcdefghijklmnopqrstuvwxyz[\]^_
36 `ABCDEFGHIJKLMNPOQRSTUVWXYZ{|}~`
```

PAGE: 2 text.c

DATE: 4/21/86

37 ---and 2 x 32 Graphic characters----

38

39 With [SHIFT]+[CBM] (64) or [ESC] [1/2] (128) it is possible to switch

40 between the two character sets.

41

42 The character set includes the following

43 special C-characters: \ ^ _ (!) ^

44

45 When in the CBM-character mode, it will

46 be difficult to find the special C

47 characters, please use the C- character mode.

48

49 The Characters: _ is done with left arrow key

50 ([SHIFT] + [0] on 64)

51 and the Left-arrow key (64 only) is the

52 Editor TAB key. TAB SET and release is

53 done by using [SHIFT]+[Left-arrow] (64 only) [SHIFT] + [TAB] (128 only)

54 To obtain the character ! use the

55 [C]+[-].

56

57

58 These lines stand at the start of our goal.

59 Our goal is to mark a block once.

60 Set the goal line of the previous line.

61

62

63

64 This line is the end of the Block *****

65 This is the beginning of the text block that we will erase

66

67

68 GREEN

69

70 Block to be erased only

71

72 has

PAGE: 3 text.c
 DATE: 4/21/86

```

73
74             one
75
76             Number
77             1
78             2
79             3
80             4
81             5
82             6
83             7
84             8
85             9
86 Block to be 10
87   erased    11
88
89            12
90            13
91            14
92            15
93            16
94            17
95            18
95 These are the last lines of the block
96 that we will erase with the erase command >>> line 95 <<<
97 This Text is after the block we *****
98 wish to erase.
99
100
101 This line is before the blue text block
102 Here begins the Block for moving.
103
104     2. . 0.0.20
105     +1.) }}} +1.
106     1'=f=---=f -
107
108
    
```

PAGE: 4 text.c
DATE: 4/21/86

109 This is the last line of the blue block.
110 This line is not in the blue text block
111
112 This is the next line to the 'last line'
113

5. Listing "sample.c"

PAGE: 1 sample.c
DATE: 4/21/86

```

1 #include "stdio.h"
2 #define CASE(Z) case 'Z': printf("\Z' ");break
3
4 main()
5 ( char c;
6
7  putc(CLR,STDIO);
8  puts("Display the value for key pressed\n",STDIO);
9
10 while()
11 {
12     c=getchar();
13     char c; /*This line is incorrect*/
14     printf("Character: ");
15
16     if((c & 0x7f) >= 0x20)
17         if(c=='\ ' || c=='\ ' || c=='\ "')
18             printf("\Zc' ",c);
19         else
20             printf("%c' ",c);
21     else
22 } /*This line is incorrect*/
23     switch(c)
24     {
25         CASE(n);
26         CASE(t);
27         CASE(f);
28         CASE(r);
29         CASE(b);
30         default: printf("\Z'o' ",c);break;
31     }
32
33
34     printf("\nASC-Code: Z3d 0XZ02x 0Z-3o\n\n",c,c,c);
35 }
36 }

```

Index

address, 81, 82
alloc(), 175
arguments, 72
arithmetic, 52, 83
arrays, 45, 81, 84, 85, 87, 88, 250
assignments, 56
atan(), 205
auto, 76, 210, 213
autoexec

B-version, 34
backgr(), 191
bdot(), 192
bit operations, 55, 226
bline(), 195
block, 61, 129, 239
block commands (editor), 21
boot, 5, 116
bplotb(), 199
break, 66, 66

C-command processor, 131
C-program, 140, 159
c-system, 117
c1, 117, 118
call(), 177
case, 251
CAST
cc, 30
CCP, 6
ce, 15
char, 47, 182, 211, 230
cl
close(), 166
clrmap(), 191
cmove(), 174
colors(), 191
com (resid.), 7, 110

commands

editor, 15-26
RAM disk, 121
CCP, 155
processor, 144
resident, 6
transient, 8, 111

component, 91

conditional evaluation, 57

constants, 49

control string, 125

control structures, 60, 150

continue, 67, 242

copy, 10, 112

cos(), 204

cursor(), 173

default, 241

#define, 246

declarations, 74, 86, 90, 229, 234

declarator, 233, 234

decrement, 54

device (prg), 8, 105, 111

dir (resid), 6, 109

directory (editor), 135

do, 66, 240

dot(), 192

dotin(), 192

double, 40, 182, 230

editor commands, 123

else, 248

#else, 248

end (resid.), 111

#endif, 248

enum, 236, 251

EOF, 97, 187

erase (editor), 21, 132

err (resid.), 6-9, 139

erroff(), 156, 165

erron() 156, 165

error() 156, 165

exit(), 165
exp(), 173, 206
extension, 106
extern, 213

f, 114
fast (resid.), 110
fast(), 178
fclose(), 170
fgetc(), 168
fgetf(), 169
fgets(), 169
filename (editor), 105, 134
fill(), 198
float, 182, 211, 230
fopen(), 170
for, 44, 65, 240
format instructions, 40, 180
fplotobj(), 199
fprintf(), 168
fputc(), 168
fputf(), 169
fputs(), 169
free(), 175
fscanf(), 186, 188
functions, 69, 92, 95

getc(), 168
getchar(), 38, 168
gets(), 169
gettime(), 169
goto (editor), 22
goto, 68, 240
graphic(), 190
graphoff(), 190
graphon(), 190

hunt (editor), 24

if, 61, 241
#IF, 247
#ifdef, 247

#ifndef, 247
#include, 38, 247
increment, 54
initializations, 76, 236
int, 185, 237
io_in(), 179
io_out(), 179
is80(), 179
isalpha(), 190
isdigit(), 190
isgraph(), 190
islower(), 190
isspace(), 190
isupper(), 190

keys(), 128, 177
kill (editor), 134

labels, 68, 242
library, 33, 163, 164, 189
line(), 195
linker option, 150, 162
linkfile, 27
load (editor), 20, 133
log(), 206
long, 230
loops, 41
lram, 10, 13, 115
lvalue, 40, 219

macro, 79, 80
mapv(), 201, 245
mathematical functions, 222, 223
memory classes, 51, 162, 202
mline(), 195
monitor, 109
move (editor), 22, 133
move(), 174

names, 48, 232
new (editor), 15, 135
NMI, 157

nmioff(), 100 , 156, 165
nmion() 100 , 156, 165

object, 39
 global, 51, 73
 local, 51, 76
oline(), 195
open(), 166
operand, 219
operators, 53
 arithmetic, 52
 logical, 52
 precedence, 58shift-

parameter, 128, 234
Pascal, 1, 38
plot(), 197
plotobj(), 199
pointer, 1, 81,85,87,88,250
preprocessor, 245
print (editor), 136
printf(), 38, 180
program file, 27, 96
program structures, 69
pushobj(), 199
putc(), 168
putchar(), 168
puts(), 169

RAM disk, 13, 106, 119, 121
register, 76, 213
replace (editor), 132
return, 242

save (editor), 19
scanf(), 184,188
screen colors, 18
set (resid.),108, 110
setcol(), 191
setplot(), 197
settime(), 176
shape(), 198

short, 230
sin(), 204
size, 251
sizeof, 251
slow (resid.), 111
slow(), 178
sourcefile, 27
sprintf(), 180
sqr(), 206
sqrt(), 206
sram, 115
sscanf(), 184
starting Super C, 5, 103, 104
static, 45, 213
stdio (prg), 13, 37, 98
strcat(), 172
strchr(), 173
strncmp(), 171
strcpy(), 172
string, 129, 187, 212
strlen(), 171
strncat(), 172
strncmp, 172
strncpy(), 173
strrchr(), 173
struct, 235
structure, 89,90,92,93,249
switch, 63,241
symbolic constants, 44
sysgen, 115, 116

tan(), 205
text field, 16, 209
time (resid.), 110
tod (resid.), 110
tolower(), 208
toupper(), 208
transfer (editor), 23, 130,132
types, 48, 118,214
type conversions, 54, 230
typedef, 95

#undef, 246
union, 231, 249
UNIX, 1, 106, 253
unsigned, 251

variant, 89, 94
vdcin(), 178
vdcout(), 178

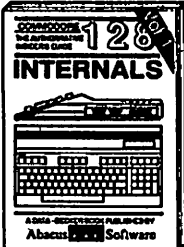
while, 42, 64, 239
window(), 178

C

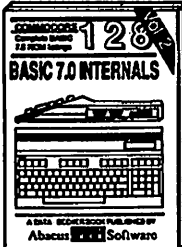
C

C

C-128™ AUTHORITYATIVE and C-64™ BOOKS



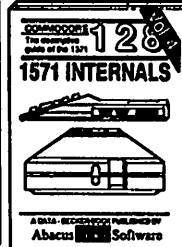
Detailed guide presents the 128's operating system, explains graphics chips, Memory Management Unit, 80 column graphics and converted ROM listings. 600pp \$19.95



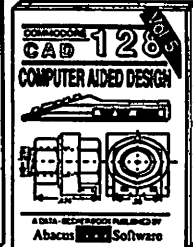
Get all the inside information on BASIC 7.0. This exhaustive handbook is complete with commented BASIC 7.0 ROM listings. Coming Summer '84. 312pp \$19.95



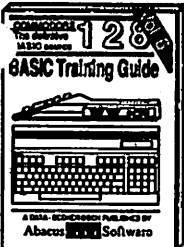
Filled with info for everyone. Covers 80 column hi-res graphics, windowing, memory layout, keyboard routines, splices, software protection, autostarting. 300pp \$19.95



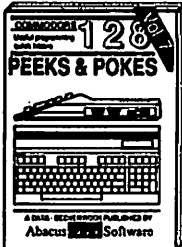
Insider's guide for novice & advanced users. Covers sequenced & relative files, & direct access commands. Describes DOS routines. Commented listings. 618pp \$19.95



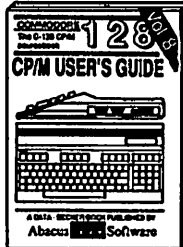
Learn fundamentals of CAD while developing your own system. Design objects on your screen to dump to a printer. Includes listings for V4 with Simon's Best. 300pp \$19.95



Introduction to programming; problem analysis; thorough description of all BASIC commands with hundreds of examples; monitor commands; utilities; much more. 160pp \$19.95



Presents dozens of programming quick-hitters. Easy and useful techniques on the operating system, stacks, zero-page, pointers, the BASIC Interpreter and more. 160 pp



Essential guide for everyone interested in CPM on the 128. Simple explanation of the operating system, memory usage, CPM utility programs, subroutines & more. 118pp \$9.95



ANATOMY OF C-64 Insider's guide to the '84 Internals. Graphics, sound, I/O, kernel, memory maps, more. Complete commented ROM listings. 300pp \$19.95

TRICKS & TIPS FOR C-64 Collection of easy-to-use techniques: advanced graphics, improved data input, enhanced BASIC, CPM, more. 279pp \$19.95

SCIENCE/ENGINEERING ON C-64 In depth intro to computers in science. Topics: chemistry, physics, biology, astronomy, electronics, others. 320pp \$19.95

Adventure Gamewriter's Handbook Step-by-step guide to designing and writing your own adventure games. With automated adventure game generator. 200pp \$14.95

ANATOMY OF 1541 DRIVE Best handbook on 1541 drive. All. Many examples and CPM utility commented 1541 ROM listings. 600pp \$19.95

1541 REPAIR & MAINTENANCE Handbook describes the disk drive hardware. Includes schematics and techniques to keep 1541 running. 200pp \$19.95

CASSETTE BOOK C-84VIC-28 Comprehensive guide; many sample programs. High speed operating system fast file loading and saving. 223pp \$14.95

PEEK & POKED FOR THE C-64 Includes in-depth explanations of PEEK, POKE, USR, and other BASIC commands. Learn the "hacker's" tricks to get the most out of your '84. 389pp \$14.95

MACHINE LANGUAGE C-64 Learn 6510 code with fast programs. Many samples and listings for complete assembler, monitor, & simulator. 200pp \$19.95

ADVANCED MACHINE LANGUAGE Not covered elsewhere: video controller, interrupts, timers, clocks, I/O, real time, extended BASIC, more. 210pp \$14.95

IDEAS FOR USE ON C-64 Themes: auto expenses, calculator, recipe list, stock lists, diet planner, window advertising, others. Includes listings. 399pp \$12.95

Optional Databases for books For your convenience, the programs contained in each of our books are available on diskette to save you time entering them from your keyboard. Specify name of book when ordering. \$14.95 each

GRAPHICS BOOK C-64 Best reference covers basic and advanced graphics. Splices, animation, Hires, Multicolor, Lightpen, 3D-graphics, FPO, CAD, protocols, curves, more. 320pp \$19.95

PRINTER BOOK C-84VIC-28 Understand Commodore, Epson-compatible printers and 1520 plotters. Packed: utilities; graphics dump; 30-ppt; commented 15P0901 ROM listings, more. 320pp \$19.95

COMPLEX BOOK C-84C-128 All you need to know about complex: how they work; designing and writing your own; generating machine code. With working example compiler. 390pp \$19.95

Abacus Software

P.O. Box 7219 Dept. H8 Grand Rapids, MI 49510 - Telex 709-101 - Phone (616) 241-5510

Optional diskettes available for all book titles - \$14.95 each. Other books & software also available. Call for the name of your nearest dealer. Or order directly from ABACUS using your MC, Visa or Amex card. Add \$4.00 per order for shipping. Foreign orders add \$10.00 per book. Call now or write for your free catalog. Dealer inquiries welcome—over 1400 dealers nationwide.

C-128 and C-64 are trademarks of Commodore Business Machines Inc.

SUPER SOFTWARE

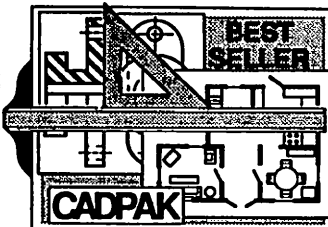
BASIC Compiler



Give your BASIC programs the speed and performance they deserve

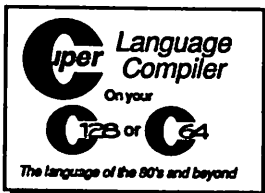
The complete compiler and development package. Speed up your programs 5x to 35x. Many options: flexible memory management; choice of compiling to machine code, compact p-code or both. '128 version: 40 or 80 column monitor output and FAST-mode operation. '128 Compiler's extensive 80-page programmer's guide covers compiler directives and options, two levels of optimization, memory usage, I/O handling, 80 column hi-res graphics, faster, higher precision math functions, speed and space saving tips, more. A great package that no software library should be without. **128 Compiler \$59.95**
64 Compiler \$39.95

optimization, memory usage, I/O handling, 80 column hi-res graphics, faster, higher precision math functions, speed and space saving tips, more. A great package that no software library should be without. **128 Compiler \$59.95**
64 Compiler \$39.95



Remarkably easy-to-use interactive drawing package for accurate graphic designs. New dimensioning features to create exact scaled output to all major dot-matrix printers. Enhanced version allows you to input via keyboard or high quality lightpen. Two graphic screens for COPYING from one to the other. DRAW, LINE, BOX, CIRCLE, ARC, ELLIPSE available. FILL objects with preselected PAT-TERNS; add TEXT; SAVE and RECALL designs to/from disk. Define your own library of symbols/objects with the easy-to-use OBJECT MANAGEMENT SYSTEM—store up to 104 separate objects. **C-128 \$59.95**
C-64 \$39.95

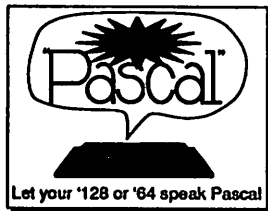
TERNS; add TEXT; SAVE and RECALL designs to/from disk. Define your own library of symbols/objects with the easy-to-use OBJECT MANAGEMENT SYSTEM—store up to 104 separate objects. **C-128 \$59.95**
C-64 \$39.95



The language of the 80's and beyond

For school or software development. Learn C on your Commodore with our in-depth tutorial. Compile C programs into fast machine language. C-128 version has added features: Unix™-like operating system; 80K RAM disk for fast editing and compiling Linker combines up to 10 modules; Combine ML and C using CALL; 51K available for object code; C-128 \$59.95
C-64 \$39.95

Fast loading (8 sec. 1571, 18 sec. 1541); Two standard I/O libraries plus two additional libraries—math functions (sin, cos, sqrt, etc.); 8 20+ graphic commands (line, fill, dot, etc.). **C-128 \$59.95**
C-64 \$39.95



Let your '128 or '64 speak Pascal

Not just a compiler, but a complete system for developing applications in Pascal with graphics and sound features. Extensive editor with search, replace, auto, renumber, etc. Standard J & W compiler that generates fast machine code. If you want to learn Pascal or to develop software using the best tools available—SUPER Pascal is your first choice. **C-64 \$59.95**

OTHER TITLES AVAILABLE:

COBOL Compiler

Now you can learn COBOL, the most widely used commercial programming language, and **write COBOL** on your 64. COBOL is easy to learn because it's **easy to read**. COBOL Compiler package comes complete with Editor, Compiler, Interpreter and Symbolic Debugger. **C-64 \$39.95**

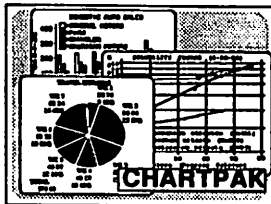
Personal Portfolio Manager

Complete portfolio management system for the individual or professional investor. Easily manage your portfolios, obtain up-to-the-minute quotes and news, and perform selected analysis. Enter quotes manually or automatically through Warner Computer Systems. **C-64 \$39.95**

Xper

XPER is the first "expert system" for the C-128 and C-64. While ordinary data base systems are good for reproducing facts, XPER can derive knowledge from a mountain of facts and help you make expert decisions. Large capacity. Complete with editing and reporting. **C-64 \$59.95**

C-128 and C-64 are trademarks of Commodore Business Machines Inc. Unix is a trademark of Bell Laboratories



Easily create professional high quality charts and graphs without programming. You can immediately change the scaling, labeling, axis, bar filling, etc. to suit your needs. Accepts data from CalcResult and MultiPlan. C-128 version has 3X the resolution of the '64 version. Outputs to most printers. **C-128 \$39.95**
C-64 \$39.95

PowerPlan

One of the most powerful spreadsheets with integrated graphics. Includes menu or keyword selections, online help screens, field protection, windowing, trig functions and more. PowerGraph, the graphics package, is included to create integrated graphs and charts. **C-64 \$39.95**

Technical Analysis System for the C-64 **\$59.95**
Ada Compiler for the C-64 **\$39.95**
VideoBasic Language for the C-64 **\$39.95**

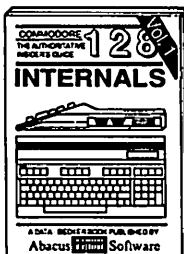
Abacus Software

P.O. Box 7219 Dept. M9 Grand Rapids, MI 49510 - Telex 709-101 - Phone (616) 241-5510

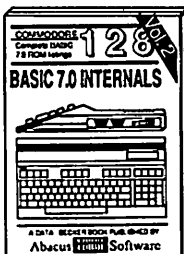
Call now for the name of your nearest dealer. Or to order directly by credit card, MC, AMEX or VISA call (616) 241-5510. Other software and books are available—Call and ask for your free catalog. Add \$4.00 for shipping per order. Foreign orders add \$12.00 per item. Dealer inquires welcome—1400+ nationwide.

Top shelf books

from Abacus



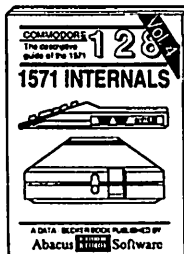
Detailed guide presents the 128's operating system, explains graphics chips, Memory Management Unit, 80 column graphics and commented ROM listings. 800pp \$19.95



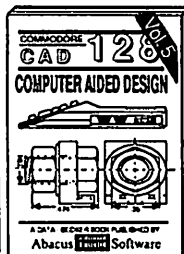
Get all the inside information on BASIC 7.0. This exhaustive handbook is complete with commented BASIC 7.0 ROM listings. Coming Summer '86 319.95



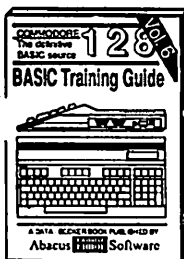
Filled with info for everyone. Covers 80 column hi-res graphics, windowing, memory layout, kernel routines, sprites, software protection, autostarting. 300pp \$19.95



Insiders' guide for novice & advanced users. Covers sequential & relative files, & direct access commands. Describes DOS routines. Commented listings. \$19.95



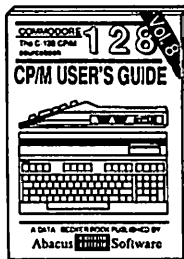
Learn fundamentals of CAD while developing your own system. Design objects on your screen to dump to a printer. Includes listings for '84 with Simon's Basic. 300pp \$19.95



Introduction to programming; problem analysis; thorough description of all BASIC commands with hundreds of examples; monitor commands; utilities, much more. \$16.95



Presents dozens of programming quick-hitters. Easy and useful techniques on the operating system, stacks, zero page, pointers, the BASIC interpreter and more. \$16.95



Essential guide for everyone interested in CPM on the 128. Simple explanation of the operating system, memory usage, CPM utility programs, submit files & more. \$19.95



ANATOMY OF C-64 Insider's guide to the '84 Internals. Graphics, sound, I/O, kernel, memory maps, more. Complete commented ROM listings. 300pp \$19.95

TRICKS & TIPS FOR C-64 Collection of easy-to-use techniques: advanced graphics, improved data input, enhanced BASIC, CPM, more. 272pp \$19.95

SCIENCE/ENGINEERING ON C-64 In depth intro to computers in science. Topics: chemistry, physics, biology, astronomy, electronics, others. 320pp \$16.95

Adventure Gamewriters Handbook Step-by-step guide to designing and writing your own adventure games. With automated adventure game generator. 200pp \$14.95

ANATOMY OF 1541 DRIVE Best handbook on floppy disks. All. Many examples and tips. Commented 1541 ROM listings. 600pp \$19.95

1541 REPAIR & MAINTENANCE Handbook describes the disk drive hardware. Includes schematics and techniques to keep 1541 running. 200pp \$16.95

CADGETTE BOOK C-64/VIC-20 Comprehensive guide; many sample programs. High speed operating system test file loading and saving. 225pp \$14.95

PEEKs & POKEs FOR THE C-64 Includes in-depth explanations of PEEK, POKE, USR, and other BASIC commands. Learn the "inside" tricks to get the most out of your '84. 200pp \$14.95

MACHINE LANGUAGE C-64 Learn 6510 code write fast programs. Many examples and listings for complete assembler, monitor, & simulator. 200pp \$14.95

ADVANCED MACHINE LANGUAGE Not covered elsewhere! - video controller, interrupts, timers, clocks, I/O, real time, enhanced BASIC, more. 210pp \$14.95

IDEAS FOR USE ON C-64 Themes: auto expenses, calculator, recipe file, stock lists, diet planner, window advertising, others. Includes listings. 200pp \$12.95

Optional Diskettes for books For your convenience, the programs contained in each of our books are available on diskette to save you time entering them from your keyboard. Specify name of book when ordering. \$14.95 each

GRAPHICS BOOK C-64 - best reference covers basic and advanced graphics. Sprites, animation, Hires, Multicolor, lightpen, 3D-graphics, I/O, CAD, programs, curves, more. 350pp \$19.95

PRINTER BOOK C-64/VIC-20 Understand Commodore, Epson-compatible printers and 1520 plotter. Packed; utilities; graphics dumping; 3D-plot; commented MP5001 ROM listings, more. 330pp \$18.95

COMPILER BOOK C-64/C-128 All you need to know about compilers: how they work; designing and writing your own; generating machine code. With working example compiler. 300pp \$18.95

C-128 and C-64 are trademarks of Commodore Business Machines Inc.

Abacus Software

P.O. Box 7219 Dept.M9 Grand Rapids, MI 49510 - Telex 709-101 - Phone (616) 241-5510

Optional diskettes available for all book titles - \$14.95 each. Other books & software also available. Call for the name of your nearest dealer. Or order directly from ABACUS using your MC, Visa or Amex card. Add \$4.00 per order for shipping. Foreign orders add \$10.00 per book. Call now or write for your free catalog. Dealer inquires welcome—over 1400 dealers nationwide.