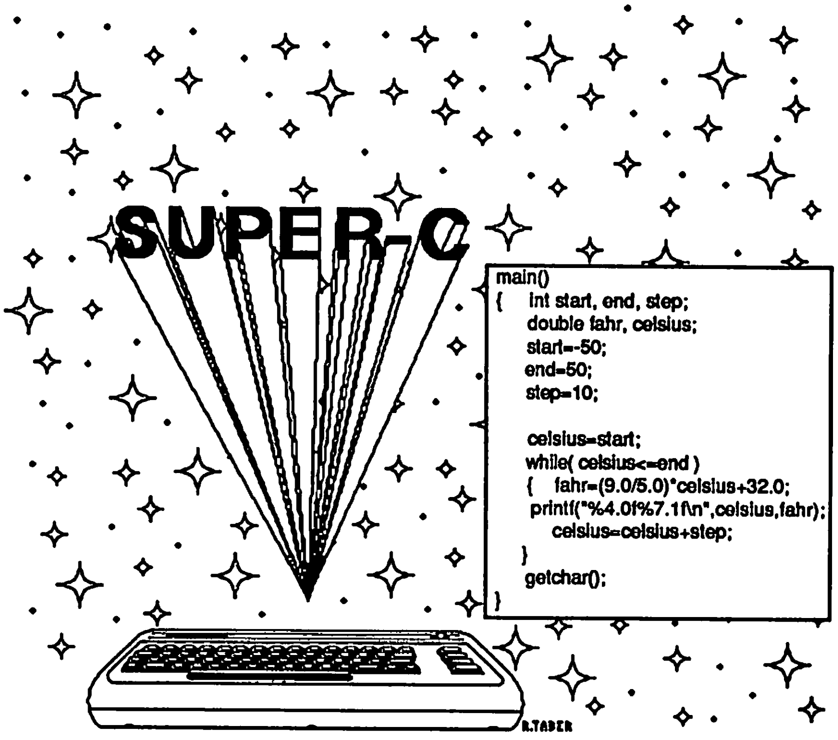
SUPER-C

for the

C-128 or C-64

by F.J. Hauck and T. Eirich



```
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();
}
```

A Data Becker product from

Abacus Software

233277

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 then 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

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 the is 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: textfile
```

`[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 [illegible]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] [l]). 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 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 the 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 exitting 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:

file type	extension	example
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 identifer.

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 `[cl] [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.l`. If this line is in the program, you must always link the `libc.l` 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.l`.

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 in 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 place 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 programs 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 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 exitted.

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    21    0L    40000
```

Integer constants which have a leading zero are evaluated as `octal`, meaning that the compiler interprets the number in the base 8 system.

```
    077 (octal) = 63 (decimal)
```

Write all of your decimal numbers without leading zeros or they will be regarded as octal.

Integer constants can also be read as hexadecimal by placing `0x` or `0X` in front of the number. The digits 10 to 15 represent the letters `a` to `f` or `A` to `F`.

```
 0x3f (hex)  =  077 (octal)  =  63 (decimal)
```

We have also used character constants. They contain a character enclosed in single quotation marks:

```
    'a'     'X'     '\n'     '\0'
```

The value of such constants is the code of the character in the character set. This value is converted into the type `int` so that calculations can also be performed on it. Combinations with the escape character \ can also be used as character constants. `'\n'` represents the code for `[RETURN]`.

```
    '\n'   =   13   =   0x0d
```

`'\0'` is the code 0or null, which is used as the end character for strings. Up to three digits can come behind the \, which are interpreted as octal digits. The value of these octal numbers is then the code of the character:

```
    '\101'   =   'a'   =   65
```

Another constant is the string. The characters in it are placed in memory. At the end is the end character `'\0'`:

```
"string\n"   ->   's','t','r','i','n','g','\n','\0'
```

A string constant can be used like an array name. Two strings constants which look alike are in reality two different constants.

Strings and characters are also different:

```
"a"      'a'
```

The first is a string, which contains a `\0` character at the end, while `'a'` is the value of the code of the letter `a`.

6.2.4 Memory classes

Up to now we have defined objects only within the function block. If no memory class is specified, the memory class `auto` is assumed. This memory class has the effect that the objects are available only within the block and are discarded when the block is exitted.

Objects which are defined within a block are called `local`. Local objects with the memory class `static` are also available only within the block. But the objects retain their value throughout the program until the block is accessed again. An advantage of these objects is that they automatically contain the value zero at the start of the program.

`global` objects may also be declared. These are declared outside a function. If no memory class is given, the object applies over the entire program.

The memory class `static` can also be specified for `global` objects. But if several separately-compiled C programs are linked together, `static global` objects from one file cannot be accessed by the others.

As a general rule, all objects must be declared before they can be used in C. Names which the compiler does not recognize through declaration are assumed to be `global` and be of type `int` or a function returning an argument of type `int`, so that such objects do not necessarily have to be declared.

6.2.5 Arithmetic operators

Arithmetic operators are the basic types of calculations + - * /. The meaning of the operators should be clear to everyone: two numbers are added, subtracted, multiplied, or divided. Important in C is the type of the result. There are things called standard conversions which are used for these operators for the many ways of combining types.

1. `char` or `short` operands are converted to `int`, `float` operands are converted to `double` operands.
2. If one of the two operands is `double`, the other is converted to `double` and the result will be `double`.
3. If one of the operands is `long`, the other operand and the result will be `long`.
4. If one of the operands is `unsigned`, the other operand and the result will be `unsigned`.
5. If both operators are of type `int`, the result will also be `int`.

`%` also belongs to the arithmetic operators. The result is the remainder after division. The standard conversions are performed for this operator as well. Only integer types are allowed as operands.

6.2.6 Comparisons, logical operators

Some comparison operators have already been used. They return an `int` value as the result, 0 for false and 1 for true.

```
    <    <=    >    >=    !=    ==
```

The operators mean: less than, less than or equal, greater than, greater than or equal, not equal, equal.

All of the simple types may be compared with each other. The standard type conversions are performed first.

A logical value can be negated with the `!` operator (logical NOT).

`!(a<b)` corresponds to `a>=b`

The `!` operator can be applied to all types. The operand is checked to see if its is zero. The result is then 1 (true) else 0 (false).

Two conditions can be combined with `&&` or `||`. The operands do not have to be conditions, however. They are only compared to zero and then receive their value true or false.

`&&` returns 1 (true) if both operators are non-zero (logical AND), else 0 (false).

`||` returns 1 (true) if one of the operators is non-zero (logical OR), else 0 (false).

These operators are guaranteed to be evaluated from left to right. The second operand will not be evaluated if the result can be determined from the first, that is, if the first operand of `&&` is 0 or not 0 for `||`.

6.2.7 Type conversions

Type conversions are performed automatically in some cases, such as the standard type conversions. Type conversions also underlie the argument of a function call. `char` and `short` are converted to `int` and `float` is converted to `double`.

Type conversions can also be forced, however. This is done through something called a `CAST`. The type name of the result is placed in parentheses and this `CAST` is placed in front of the type to be converted.

```
(char) pi
```

The value of the object `pi` will be converted to type `char`. The conversion is always done so that values with "smaller" types are converted to "larger" types, without changing the value. A conversion in the other direction can change the value if the value does not fit in the value range of the destination type.

6.2.8 Increment and decrement

C has an increment command ++ and a decrement command -- to increment or decrement an object by 1.

```
i++      ++i
```

increments the object `i`. Both expressions have the same effect. But in C every expression has a value, even assignments and increment and decrement operations. In the first case the expression has the value of `i` before the increment, while the other has the value of `i` after it has been incremented. The `--` operator can be used for decrement (subtract one) in the same manner.

```
i--      --i
```

It should be noted that these operators have side effects if they are not alone.

```
numbers[i++]+i
```

The above expression indexes the array numbers with the value i. But i will be incremented and affects the expression following. Often the order of evaluations in C is not predetermined, in order to give the compiler free room for optimizations. The compiler may reverse the expression above.

```
i+numbers[i++]
```

The side effect now has a different result since the first i is not incremented. So watch out for side effects! You can avoid this effect by using objects which will be changed by side effects only once in the expression.

For integer types the increment and decrement operators are faster than the corresponding assignment.

6.2.9 Bit operations

In C there are also operators which change the bit pattern of a value. Such operators can be used only on integer types.

First there are operators which combine two values bit by bit. The standard type conversions are performed.

& bitwise AND operation:
Result bit 1 if both operand bits are 1, else 0

| bitwise OR operation:
Result bit 0 if both operand bits are 0, else 1

^ bitwise exclusive OR:
Result bit 1 if both operands equal, else 0

The second group of bit operations are the shift operators. They shift the bit pattern of a value.

```
1<<2
```

With the << operator the bit pattern of the left operand is shifted as many times to the left as the right operand specifies. The above expression therefore has the result 4. 0-bits are shifted in from the right. A shift to the left corresponds to a multiplication by 2.

```
4>>2
```

The >> operator shifts the bit pattern to the right. The result is 1 here. The operation corresponds to a division by 2 for one shift. If the left operand is `unsigned`, 0-bits are shifted on the left. But if the operand is not of `unsigned` type, sign bits are shifted in Super C, so that `-4>>2` returns the result `-1`. This is different for some compilers however and they always shift in 0-bits. The result can therefore be different from machine to machine (this is legitimate since Kernighan and Ritchie proposed both versions).

The type of these shift operations is always that of the left operand. If the right operand is negative or too large, the result is undefined.

6.2.10 Assignments

The assignment through the = operator has already been used and is a fundamental part of every program. The assignment assigns the right operand to the object which the left operand denotes. The left operand must therefore designate an object; it must be an `lvalue`. 1+2 is not an `lvalue` according to this.

The type of the right operand is converted to the type of the left operand before the value is assigned. In C an assignment has a value. This value can also be used further. The value of an assignment is the converted value of the right operand.

```
#include "stdio.h"

main()
{   char c;
    while ((c=getchar()) != '\n')
        putchar(c);
}
```

In the program above the key just pressed is assigned to `c` in the loop condition. The value of the assignment, the pressed key, is compared to the `[RETURN]` character. If the key pressed was `[RETURN]`, the loop is ended. Otherwise the key pressed is printed with the function `putchar`.

Note that the assignment must be enclosed in parentheses since the compiler will otherwise perform the comparison first and then assign its result to `c`.

There are short forms for assignments if the value to be assigned is to be combined with the `lvalue`. `a = a op (b)` can be written as `a op= b`. Operators permitted are:

```
*=   /=   %=   +=   -=   ^=   &=   |=   <<=   >>=
```

`x*=y+1` will be converted to `x=x*(y+1)`, meaning that the precedence of operators doesn't apply. The entire right operand will be combined. The `lvalue` is evaluated once in this short form.

```
numbers[i++]+=1
```

causes `i` to be incremented only once.

If the value of an assignment is used further, side effects similar to those for increment and decrement should be watched for, which can be caused by changing an object.

6.2.11 Conditional evaluation

C offers the ability to perform conditional evaluation. It consists of three parts and two operators:

```
a ? b : c
```

The value of the expression `a` determines if the expression `b` or `c` will be evaluated. `b` will be evaluated if `a` is not equal to 0, else `c` will be evaluated. The value of the entire expression is the value of the expression finally evaluated.

```
1 ? 2 : 0
```

always returns the value 2.

```
    i ? 2 : 0
```

returns the value 0 if `i` is equal to 0, else 2.

```
    x ? i++ : j++
```

If `x` equals 0, `j` will be incremented. The value of `j` before the increment is the value of the expression. Otherwise `i` will be incremented and the value of the expression is `i` before the increment. Only one of the two objects is ever incremented.

If the result types of the two possible result expressions are different, a standard type conversion is performed on both in order to get the same result type in both possible cases.

6.2.12 Precedence and order of operators

Multiplication and division operations are performed before addition and subtraction. In C all of the operators have a preset precedence which determines which operator will be performed first. If several operators of the same precedence are in a row, the order of an operator determines whether it will be evaluated from left to right or from right to left.

In the following table, operators on the same line have the same precedence. The first line has the highest precedence, meaning that the operators on that line will be executed first. The last line has the lowest precedence.

Operators	Order
`() [] . ->`	from left
`++ -- * & - ! ~ (CAST) sizeof`	from right
`* / %`	from left
`+ -`	from left
`<< >>`	from left
`< <= > >=`	from left
`== !=`	from left
`&`	from left
`^`	from left
`|`	from left
`&&`	from left
`||`	from left
`? :`	from right
`= *= /= %= += -= >>= <<= &= ^= !=`	from right
`,`	from left

The associative and commutative operators `+ * ^ | &` can be rearranged by the compiler. This cannot be prevented even with parentheses. For all other operators the order, whether the left operand or the right operand will be evaluated first, is not set. The operators `&&` and `||` are exceptions to this. Their operands are guaranteed to be evaluated from left to right.

6.2.13 Additional operators

Operators which you don't recognize in the list above will be discussed later in the tutorial section. Otherwise you will find an exact description in the reference section.

One operator should be mentioned yet. With the `,` operator you can split an expression into two parts, both of which will be executed. The value of the expression is the value of the right part.

```
    if (t=0, s+1) ...
```

The condition of the `if` instruction is just `s+1`. But first `t` will be set equal to zero.

6.2.14 Program text

In principle a program can be entered format-free. The only important thing is that the compiler, as one would expect, reads the text line by line, from left to right. Whether you write

```
main(){int i;for(i=0;i<10;i++)printf("%d\n",i);}
```

or

```
main()
{    int i;
     for (i=0; i<10; i++)
       printf("%d\n",i);
}
```

is entirely up to you. But you see that you can get a clean, understandable program if you follows certain rules.

- indent sub-statements and dependent program sections
- write brackets which belong together in the same column
- insert blank lines to make things easier to read
- don't overload one line with text

In the C editor you have the ability to change the color of program sections. Beginners may wish to use this but don't over use it. A rainbow-colored program is also hard to read. Use the supplied programs as examples. On Version 3 when using a monochrome monitor it is best to avoid color.

6.3 Control structures

A C program consists of functions. In the function blocks there are instructions which the program executes. Such instructions are executed sequentially, one after the other. In order to be able to leave this rigid scheme, a programming language offers control structures. All control structures are instructions themselves. But they contain other instructions, whose execution is not necessarily sequential. Sub-instructions can be repeated or skipped entirely (loops, branches).

Essentially, control structures change the sequential execution of instructions. This change is usually made conditional, that is, the processing is changed according to the value of certain data.

6.3.1 Block

A block is a group of statements enclosed in braces `{}`. Local objects, which are available only in this block, may be defined at the start of a block. A block is itself an instruction, so that blocks can also be nested.

A block serves to group instructions together. In a loop, for example, you can repeat only one instruction. If the loop is to contain several instructions, you combine these into a block and you have one instruction.

The block is an exception to the instructions, since it is not terminated with a semicolon, but with the `}` brace.

6.3.2 if instruction

In an `if` instruction a sub-instruction is executed only if a certain condition is fulfilled.

```
if (c=='a')
     printf("Letter: a");
```

Only when the condition `c=='a'` is fulfilled, true, will the instruction following it be executed. This instruction could also be a block, of course.

The parentheses following `if` need not necessarily contain a condition. The expression is simply evaluated to see if the value is 0 (false) or not 0 (true).

The `if` instruction can also be extended with an section.

```
    if (c=='a')
        printf("Letter: a");
    else
        printf("another character");
```

The instruction behind `else` is executed whenever the condition is false. **Either** the instruction behind `if` **or** the instruction behind `else` is executed.

You can program a branch to one of two different statements with `if...else`. To branch to one of several instructions, `if...else` instructions can be chained by placing another `if` instruction in the `else` portion of the previous `if` instruction.

```
    if (c=='a')
        printf("Letter: a");
    else
        if (c=='b')
            printf("Letter: b");
        else
            printf("Another character");
```

Note that this whole thing is *one* instruction, although it consists of several nested sub-instructions. In order to increase readability, the normal indentation can be eliminated:

```
    if (c=='a')
        printf("Letter: a");
    else if (c=='b')
        printf("Letter: b");
    else
        printf("Another character");
```

`if-else` chaining has the disadvantage that it requires a good deal of writing. The conditions must be somewhat different in each instruction, but must be reprogrammed. Furthermore, all compilers place some limit on the number of nested statements. For this reason most higher-level programming languages have a way of programming larger branches:

6.3.3 switch instruction

A switch instruction can branch to one of up to 43 statements in Super C. The branch is made based on the result of an expression:

```
switch(c)
{
    case 'a': printf("Letter: a");
              break;
    case 'b': printf("Letter: b");
              break;
    case 'c':
    case 'd': printf("Letter: c or d");
              break;
    default:  printf("another character");
              break;
 }
```

The expression after `switch` is here just the object `c`. Its value is the basis for the branch. After `switch(..)` follows a block containing the various instructions. If a certain instruction should be executed based on a certain result, a `case` label must be placed before the instruction:

Behind `case` is a constant. If the result of the expression matches the constant, the instruction following it will be executed. Not only the following instruction, but all instructions following the matching label. In some cases this can be very useful. To prevent it, you place a `break` instruction after the branch to exit the block.

You can place several `case` labels in a row. One special thing is the `default` label. If it comes before an instruction in the block, this instruction will be executed if there is no `case` label which matches the result of the expression. The `default` label need not be at the end of the `switch` block. If there is no `default` label, no instructions are executed if no `case` label is found.

The result of the expression must have an integral type. Floating-point values are not allowed. The same applies for the `case` constants.

You may be surprised by the last `break` instruction in the block. This is actually superfluous, since the block will also be ended without this instruction. But you should still write this `break` because the danger exists that it will be forgotten if a new branch is later added. The instruction of this new branch will then be executed along with the `default` instruction.

6.3.4 while instruction

The `while` instruction is a loop. It repeats a sub-instruction as long as a condition is fulfilled.

```
while (i<10)
    i++;
```

As with the `if` instruction, the condition is in parentheses. The loop instruction then follows, and it will be repeated as long as the condition is fulfilled, as long as the expression in the parentheses is not equal to zero.

With loops, make sure that the loop condition will become false at some point, or the loop will never end.

Endless loops, called infinite loops, can be programmed by omitting the loop condition.

```
while()
    INSTRUCTION
```

Such infinite loops can sometimes be very useful. Omitting the loop condition has even been propagated by Kernighan and Ritchie, but we have found that many compilers will not allow it. In order to write portable programs, write:

```
while(1)
    INSTRUCTION
```

6.3.5 for instruction

The `for` instruction is a special `while` instruction. It is not only provided with a loop condition, but also an initialization expression and a continuation expression.

```
for (i=0; i<10; i++)
    putchar(string[i]);
```

This is usually used as it is in BASIC or Pascal, that is, a variable is set to an initial value (initialization: `i=0`). The variable runs up to a certain value (loop condition: `i<10`). The variable is incremented at the end of each repetition (continuation: `i++`). But a `for` instruction is not tied to one variable. One can use three expressions for the initialization, condition, and continuation.

```
for (c=getchar(), i=0; i<10; putchar(c), i++);
```

The initialization of the loop is:

```
c=getchar(), i=0
```

This is *one* expression. The `,` operator divides the expression. A key is read and its code is stored. `i` is set to zero. The condition checks to see if `i` is less than 10. The continuation consists of two parts. The key pressed is printed and `i` is incremented. The sub-instruction of the loop is just a semicolon. This is an empty instruction. The entire statement waits for a key to be pressed and then prints the character 10 times.

You can see that you can program much more complex `for` loops in C than in other languages.

If the condition is omitted, you get an infinite loop. The initialization and the continuation can also be omitted.

```
for (;;);
```

is the smallest `for` loop, an infinite loop.

6.3.6 do instruction

The do instruction is a new type of loop. With the while and for loops the condition is tested at the beginning of each repetition, including the first time the loop is ever executed. It can occur that the loop is never executed at all.

The do instruction does not check the loop condition until the end of the loop instruction. Such loops are used when the loop instruction is to be executed at least once.

```
do
{   printf("Input: ");
    gets(string, 20);
}
while(string[1]=='\0');
```

The loop starts with the key word do. The loop instruction follows, here a block with two instructions. At the end of the loop is the loop condition behind while. The instruction prints "Input: " and then reads a string of at most 20 characters into the array string, which must naturally be defined with at least 21 elements. The loop condition tests if the second element of the array (element 1) is equal to the end character of the string. If this is the case, the first element (element 0) is a [RETURN] character and no more characters were read. Then the loop and the input is repeated. Otherwise the loop ends.

```
do ; while();
```

If you omit the loop condition, the result is an infinite loop.

6.3.7 break instruction

This instruction was already discussed in connection with switch. It causes a loop or a switch instruction to be exitted. The break instruction works only within loops or switch blocks. It always refers to the last loop or switch instruction, if several of these are nested.

```
    break;
```

The break instruction causes the loop to be exitted immediately and execution to continue after the entire loop. In a switch instruction the block is exitted.

```
    for (i=0; i<20; i++)
    {
        string[i]=getchar();
        if (string[i]=='\n');
            break;
    }
    string[i]='\0';
```

In this program fragment the variable i runs from 0 to 19. It waits for a key to be pressed and then assigns the key to the ith element of a character array string. If the element read is a [RETURN] character, a break instruction is executed. This exits the loop.

A maximum of 20 characters are read, but not beyond a [RETURN] character. In the last instruction a '\0' character is appended to the end of the string.

You could also integrate the test for the [RETURN] key in the loop condition. This would make the program harder to understand, however.

6.3.8 continue instruction

The continue instruction is not used very often. It applies only to loops and not to switch instructions. continue directs execution back to the end of the loop instruction.

With a while or do loop the loop condition is immediately tested and the loop instruction repeated if the condition is fulfilled. With a for loop, a continuation is first executed, then the loop condition is tested.

The continue instruction is usually used to skip complicated instructions in the loop.

```
for (i=i; i<20; i++)
{   getchar();
    if (c=='\n')
        continue;
    /* complicated computation */
}
```

The "complicated computation" indicated by the comment is skipped if the key read is [RETURN].

6.3.9 goto instruction and labels

Labels can be placed in front of any instruction. They consists of a name and a colon:

```
name : instruction
```

The name is thereby defined as a label. A jump can be made to such a label with a goto instruction, that is, program execution can be made to continue behind the label.

```
goto name;
```

Such jump instructions should not be used. There is no situation in which a jump cannot be replaced by the existing set of control structures. Jumps are to be strictly avoided in structured programming in order to preserve the structure and readability of the program. Only in rare cases is a jump useful, such as when an error occurs within several nested loops. Only the innermost loop can be exitted with break, while goto can be used to exit all of them.

A jump cannot be made into another function. You can jump out of blocks. You should avoid jumping into blocks, however. If objects are defined in such a block, they will not be defined and will not be available.

6.4 Program structures

6.4.1 Functions

Up to now we have defined only one function, main. Programs usually consist of more than one function, however. As we already know, functions can be passed arguments, with which they can perform computations. A function can return a result value. If a function is not supposed to return a result, it is also called a procedure. Procedures are defined with the type void.

```
void nextline()
{    putchar('\n');
}
```

The function nextline prints a [RETURN] character. It does not return a result. It can be called in any other function.

```
main()
{   printf("Demonstration");
    nextline();
    printf("of the function");
    nextline();
    printf("nextline");
}
```

For functions for which no type has been specified, as with main here, the compiler assumes type int and assumes that the end result has this type. main does not return a result, however, since the end of main also means the end of the program. The type void is often left off of main through sheer laziness.

Functions can return all simple types. Take a look at the following function, power, which has two arguments. The first argument is a double value x, the second is an int value y. The function power is to return the value of x to the power y.

```
double power(x,y)
  double x;
  int y;
{ /* ... */
}
```

The function is assigned the type `double`, the result type. In the function parentheses is a list of names. These are the names of the parameters. The parameter names must be declared. This is done in the usual manner, but without storage class. Note that there may be no semicolon between the first and second lines, but a semicolon must come at the end of each of the parameter declarations.

The parameter declarations must be made in the order in which the parameter names are listed.

What do these parameters mean? The parameters are local variables just like those defined within the block. They contain the value of the arguments used when calling the function. The complete function `power` looks like this:

```
double power(x,y)
  double x;
  int y;
{   if (y==0)
        return 1;
    if (y<0)
        return 1/power(x,-y);
    else
        return x*power(x,y-1);
}
```

Let's go through the instructions of `power` step by step. When the function is called the values of the arguments are stored in the parameters. If, for example, you want to calculate 5 squared, call the function `power`.

```
power(5.0,2);
```

Here we see a peculiarity of C. The types of the arguments must match the types of the parameters. You must write 5.0 so that the first argument has the type `double`.

What happens in power? x has the value 5.0 and y has the value 2. If y were equal to 0, the result of power would be 1. This is achieved through a return instruction. It can be anywhere in a function block. If it is executed, the function is exitted. An expression which calculates the result of the function may come behind return. 1/power(x,-y) calculates the same thing as power(x,y). But now the exponent is positive. The function power therefore calls itself. This is called recursion. x and -y here are arguments whose values are assigned to x and y in the new call. The parameters x and y have only the same names as those in the call. In reality the parameters are recreated like auto objects at each new call so that they cannot disturb each other.

If y is positive, the function is exitted with the result x*power(x,y-1). In our call:

```
        5.0*power(5.0,0)
```

power is called again. But in this call power sets the result clearly to 1, so that the result of the second call is:

```
        5.0*1.0 -> 5.0
```

The recursion runs back again. The result of the second call is then used in the first call.

```
        5.0*5.0 -> 25.0
```

25.0 is then the end result of the call power(5.0,2).

It is often difficult for the beginner to understand the structure of a recursive function. Remember that the parameters of a new function call are different from those of the old call. They have the same names, but are different objects. The result of a function is represented by the call. power(5.0,1) is represented by the result 5.0.

Recursion is often easier to understand than linear programs. This allows the function power to be implemented with quite few statements.

There doesn't have to be an expression behind `return`. But then the result of the function is not defined. One should exit a function of type `void` with `return` without an expression. A function is also ended after the last instruction in the function block has been executed. You know this from `main`. You can also end a C program through a `return` instruction in the function block of `main`.

6.4.2 Arguments

The are some characteristics of passing arguments to functions in C which you should be aware of. The arguments are always evaluated. If the type is `char` or `short`, the argument will be converted to `int`, just as `float` will be converted to `double`.

The types of the arguments must match the types of the parameters. This is not checked by the compiler. It remains the responsibility of the programmer. If the types do not match, you can force agreement with a `CAST`.

If you don't notice that the types do not match the function will probably return erroneous results.

Parameters declared with the type `char`, `short`, or `float` will be converted to `int` or `double`, as appropriate.

Parameters in C are passed only by value (call by value), which means that the value of the argument is assigned to the parameter. This parameter can be used like a local variable.

```
main()
{    double a;
     int b;
     ...
     power(a,b);
     ...
}
```

In this example the value of the objects `a` and `b` are passed to `power`. The contents of these objects cannot be changed by the function, however, not even by changing the corresponding parameters.

If you want to change objects outside of the call, you must use pointers. This will be discussed in the following sections.

6.4.3 Global definitions

It was mentioned briefly before that you can define global objects. These definitions are programmed outside of the function blocks.

```
int maximum;

main()
{    ...
     i=maximum+1;
     ...
}
```

maximum is a global object with the type int. You may not specify a storage class. A global object remains valid throughout the entire program and is not discarded. The name of such an object can be used in the entire program. If a local object with the same name is defined, the definition of the global object will be "hidden" by the local object, meaning that the local object will always be accessed.

Global objects are used for storing data which is to be made accessible to several functions. They can also be used to save parameters or to return several results per function. Functions can exchange data among themselves with global objects.

Global objects, defined without storage class, can also be used by a separately-compiled program if both link files are linked together. To prevent this outside access the memory class static can be placed before the definition. The object still retains its value over the whole program, but it can be used only in the file in which it was defined.

A function definition is also a global definition since functions can also be thought of as objects. Functions can also be declared as static, meaning that they can be used only with in the file. To do this, static is placed in front of the function header. main may not be defined as static.

6.4.4 Declarations

If you want to use global objects from a different, separately-compiled program section, these objects must be declared so that the compiler knows what type they are. No memory space is reserved by a declaration. The compiler codes the declaration in the compiled program such that the declared object will be connected with its definition during linking.

Declarations can be made global or local, whereby in this compiler system the declaration remains valid over the whole program in both cases. They are designated with the storage class extern.

The modules with the standard functions are nothing more than separately-compiled programs. These functions must be declared before they can be used. This is done by the program line:

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

The file stdio.h is a source file in which these declarations are made. The #include command inserts this source file into the program.

```
     extern void printf();
     extern char *gets();
```

The functions printf and gets, which you already recognize, must be declared. You see that no parameter list and no function block are specified. Such declarations must always come before the first use, but they can be either local or global.

```
extern void printf();
main()
{   extern void printf();
    ...
    printf("...");
    ...
}
```

Local declarations are used in order to clarify which functions are used within the block. An object can be declared more than once, but the declarations must agree.

Naturally, objects other than functions can also be declared. With functions, declarations within a source file are also interesting. If you have two functions which call each other, one function must logically come before the other. When the first function is compiled, the compiler does not yet recognize the second. This must therefore be declared:

```
double alpha()
{
    extern long beta();
    ...
    beta();
    ...
}

long beta()
{   ...
    alpha();
    ...
}
```

It would be better programming style to declare each function in order. In summary: If an object is accessed by its name, it must be known to the compiler through either a declaration or a definition.

If the compiler encounters an undeclared name, it assumes the object to be of type `int` and with `global` storage class (not `static`). The compiler would also assume this in the previous example if `beta` had not been declared. An error message would *not* appear.

Global, non-static, `int` objects or functions returning type `int` do not have to be declared. In spite of this, it is recommended that you do so to preserve the understandability of the program.

6.4.5 Local definitions

The local definitions are found within a block. If no storage class is specified, the compiler adds `auto`. Objects of this storage class are generated at the start of the block and are discarded at the end. If the block is called recursively, new objects are generated which have only the name in common with the old objects.

The `static` local objects are handled differently. These are defined with the storage class `static`. The object retains its value throughout the entire program, but is accessable only in the block in which it was defined. Logically, recursive calls also refer to the same object.

There is a third storage class which can be used for local definitions: `register`. Such definitions work like `auto` definitions, but the compiler tries to place these objects into special processor registers so that they can be accessed faster. Unfortunately, neither the C-64 nor the C-128 has such registers available. If such definitions can no longer be placed in registers, whether they are all used up or because there are none, these definitions are handled like `auto`.

6.4.6 Initializations

In contrast to many other languages, you can initialize objects during their definition, meaning that these objects are pre-assigned a certain value. Such initializations save time over assignments and are easier to follow.

Initializations are made by placing an = sign behind the declarator, followed by the appropriate value.

```
main()
{
    static int maximum = 50;
    auto double minimum = power(5.0,2);
    ...
}
```

Static and global objects can be inititalized only with constant values. maximum contains the value 50 after the initialization. If static and global objects are not initialized, the compiler automatically sets the value to zero.

Local auto objects are not automatically set to zero. Entire expressions can be assigned to auto objects so that the above initialization is possible. minimum contains the value of the function call power(5.0,2).

There is another important difference between auto objects and others. auto objects are initialized upon each new function call. global and all static objects are initialized all together at the start of the program. Their value is initialized only once. Declarations with the storage class extern cannot be initialized.

Arrays can be initialized element by element. auto arrays cannot be initialized, however.

```
char name[20] = {'a','n','n','a','\0'};
main()
...
```

A list of elements is enclosed in braces and placed after the = character. The array has 20 elements. Here only the first five are initialized. If fewer elements than necessary are found in such a list, the missing values are filled with zero. There is a short form of the above initialization. Character arrays can be initialized with strings, meaning that the elements of the string are placed in the character array.

```
char name[20] = "anna";
```

performs the same initialization.

The specification of the dimension can also be omitted in the definition of an initialized array. The dimension is then automatically as large as the number of initialized elements.

```
int month[] = {0,31,28,31,30,31,30,31,31,30,31,30,31};
```

The array month is declared with 13 elements. The elements 1 to 12 contain the days of the months 1 to 12. Element 0 is not used, but it must appear in the list.

```
char name[]="anna";
```

The array name is here dimensioned with 5 elements. Always remember the '\0' character.

Arrays can also have multiple dimensions and can be initialized as such.

```
int month[][13]={ {0,31,28,31,30,31,
                  30,31,31,30,31,30,31},
                    {0,31,29,31,30,31,
                  30,31,31,30,31,30,31} };
```

This is a two-dimensional array, which means that the elements of the array are again arrays, whose elements are of type int. The initialization is recursively, meaning that one has a list of two elements. These are again lists, whose elements are initialized.

For multi-dimensional arrays, the specification of the first dimension can be omitted, as in the example. Here month[2][13] is declared based on the initialization given above. The sense of the above definition is the following: By the first dimension we decide if the year is a leap year or not, and with the second dimension we select the month. The result of an access to the array returns the number of days of the month:

```
month[1][2];
```

accesses February in a leap year and returns value 29.

If fewer than necessary are given in a sublist, the rest are filled with zero. To initialize all the elements use one list:

```
int
month[][13]={0,31,28,31,30,31,30,31,31,30,31,30,31,
             0,31,29,31,30,31,30,31,31,30,31,30,31
};
```

The compiler automatically recognizes the structure and assigns the first thirteen elements to the array `month[0]` and the next thirteen to `month[1]`.

6.4.7 Macros

You have already become acquainted with macros under the name symbolic constants. You have assigned constants a name which can then be used in the whole source file as the constant. This concept is not limited to just constants. You can assign a name to any desired piece of text. Wherever this name occurs in the source file, the text string is inserted instead. The replacement string must be separated from the name by a space.

```
     #define NL putchar('\n')
```

In the program following this you could use the following expression:

```
     NL;
```

This causes a `[RETURN]` character to be printed. `NL` will be replaced by `putchar('\n')`.

Note that the # character must always be at the start of a line. Commands which begin # belong to something called the preprocessor. They are not directly part of C, but have an effect only on the source file. Such a preprocessor command can come in the middle of a source file, but requires its own line.

A name defined with `#define` is called a macro. Such macros can also be used like functions, that is, you can pass arguments to them.

```
    #define PRINT(x) printf("%d",x)
    #define PRINT2(x,y) PRINT(x),PRINT(y)
```

The macro `PRINT` can now be called with an argument like a function. The value of the argument is not important, but the argument text is inserted in the replacement text wherever the parameter name is located. If, for example, you call the macro like this:

```
    main()
    {    ...
         PRINT(2*3);
         PRINT(3*i-j);
         PRINT2(5*4-a,b);
    }
```

the calls will be replaced with:

```
    main()
    {    ...
         printf("%d",2*3);
         printf("%d",3*i-j);
         printf("%d",5*4-a),printf("%d",b);
    }
```

The following must be noted when making such a definition. The parenthesis, `(`, must follow immediately behind the macro name or it will be interpreted as a normal replacement text. The parameter names must be chosen so that the same name does not occur in the arguments, or it may continue to be replaced in the arguments as well.

Macro names are often written in upper case so that they can be recognized as macros. This is a matter of style, however, and can differ from programmer to programmer.

An defined macro name applies to the end of the source file. The name can no longer be declared because the name will be replaced in the declaration text as well. If the name in front of a macro definition is already declared, the name will always be interpreted as a macro.

```
#undef PRINT
```

This is a preprocessor command which erases a previously-defined macro. From this line up to the end of the file, the macro defined with `PRINT` will no longer be available.

6.4.8 Chaining files

The C compiler allows source files to be chained together. A special preprocessor line takes care of this.

```
#include "prg_part 2.c"
```

The contents of the file `prg_part 2.c` will be inserted in place of this preprocessor command. We have used this command before to insert the file `stdio.h`. In this file all the functions in the standard modules are declared and various macros are defined.

Additional `#include` commands may appear within a file inserted with `#include`. Such chained source files result in only one file for the compiler because the preprocessor affects only the source text. Chained files are not to be confused with two separately compiled files.

6.5 Pointers, addresses, and arrays

One of the more powerful advantages of C is the *pointer*. It is an object like any of the others. The special part is the value range of a pointer. The content of a pointer object is an address. This address usually points to another object. You can access an object via this address without having to use its name.

The difference between a pointer and an address is something like the difference between an `int` object and and `int` constant.

6.5.1 Pointers

A pointer is declared by placing an asterisk (*) in front of the name.

```
    int *p;
```

defines p as a pointer whose address points to an int object. Take a look at the following example program.

```
    main()
    {   int a,*p;

        p=&a;
        *p=2;
    }
```

An int object and a pointer to int are defined. The & operator appears in front of an lvalue and returns the address of the designated object. &a is the constant address of the object a. This address is assigned to the pointer.

The * operator precedes an address or a pointer and makes its operand an lvalue of the designated object. *p has the same effect as the name of the object whose address is stored in p. This object is here assigned the value 2. Instead of *p we could also have written a.

Further consequences:

> *&a corresponds to a
> (the * and & operators are evaluated from right to left, *(&a))
>
> &*p corresponds to p
> (&*p is not an lvalue but only an address)

In summary:

A pointer is declared by placing an * in front of the name. The pointer can contain only addresses which point to an object of the declared type.

The * operator requires an address or a pointer. The entire expression represents the object to which the address points. This construction is an `lvalue`.

The `&` operator requires an `lvalue` and returns the constant address of the object.

6.5.2 Address arithmetic

Computations can be performed with addresses and pointers in C as well. This is made possible by pointer arithmetic:

```
int array[6];
int *p;
...
p=&array[4];
p=p+1;
*p=5;
```

First the address of the array element 4 is assigned to the pointer `p`. The pointer is incremented by one. But actually the pointer is not incremented by 1 but by 2. The pointer arithmetic causes the summand 1 to be multiplied by the `SIZE` of the designated object, 2 in this case. This has the result that `p+1` returns an address which points to exactly one object beyond the current one, or `array[5]`. `*p=5` means the same thing as `array[5]=5`. The addition with 1 is independent of the type of the array. If the elements were of type `double`, 6 would have been added to the address since the `SIZE` of type `double` is 6. This addition only makes sense when the new address still points in the same array, because only then is it guaranteed that the objects will be right behind each other.

Instead of `p=p+1` we could also have written `p+=1` or even `p++` or `++p`. Furthermore, the last two lines could have been replaced with the following:

```
*++p=5;
```

The effect would be the same. The operators * and ++ have the same precedence and are processed from the right, meaning that first the ++p is executed and then the *.

You can use subtraction exactly as addition. The new address points to an object a corresponding number previous.

```
    p=&array[5]-4;
```

p points to the object array[1].

Two addresses or pointers can be subtracted one from the other. A precondition for a correct result is that the two addresses point within the same array.

```
    &array[5]=&array[1]
```

The result will be divided by the SIZE of the type, which means that such an operation is independent of the type of the array. It returns the number of objects between the two addresses.

It is from this pointer arithmetic that the indexing of an array element is derived. The access of an array element a[b] is internally converted to *(a+(b)).

The name of an array alone represents the address of the first element in the array. The name itself is not an lvalue. But if you adds to it the number of the desired element, you get its address because of the pointer arithmetic. The * operator makes the expression an lvalue so that *(a+(b)) has exactly the same effect as a[b].

The following consequences result from this:

&array[0]	corresponds to	array
&array[1]	corresponds to	(array+1)
array[2]	corresponds to	*(array+2)

Further consequences result for the indexing by [ ]. Since these brackets are converted to an addition according to the scheme above, their use is not restricted to arrays.

```
int array[6];
int p*;
p=array+3;
p[2]=5;
```

p is assigned the address of array element 3. p[2] is converted to *(p+2) and thereby represents the object array[5]. You can see that pointers can be used arrays and vice versa.

6.5.3 Pointers and arrays as arguments

It has already been mentioned that arrays cannot be passed as arguments. Pointers or addresses of all objects can be passed as arguments. To pass an array to a function, you pass just its address.

```
int name[41];
...
gets(name,40);
```

This fact was already used in an earlier example program. The function gets receives as an argument the address of the array name. name alone represents the this address. The corresponding parameter declaration of gets would have to look like this:

```
char *gets(string, length, filenr)
  char string[];
  int length;
  ...
```

The specification of the dimension is not of interest and can be omitted here. In reality string does not represent an array, because the object to which the address of the array is assigned is a pointer which can be used like an array within the function block. The parameter declaration could also be:

```
char *string;
```

Passing arrays is done via the address (call by reference). This procedure has the result, however, that the array can be changed by the called function. This is in contrast to passing other types where only the value is passed.

If you want to change other objects within a function, you simply pass an address:

```
main()
{   double a,b;
    ...
    swap(&a, &b);
    ...
}

swap(x,y)
  double *x,*y;
{   double z;
    z=*x;
    *x=*y;
    *y=z;
}
```

Calling the function swap passes the addresses of a and b so that the contents of the two objects can be exchanged.

6.5.4 Complex declarations

Up to now you have seen only declarations with simple declarators. Declarators are the part after the type and storage class which contains the name. Such a declarator could like this up to now:

```
name
name[...]
name(...)
*name
```

In the first case the declared object is of the specified type, in the second case it is an array whose elements are of this type. In the third case it is a function which returns a value of the specified type, and in the fourth case it is a pointer which can point to objects of this type.

At first it may appear that the declarators have been chosen somewhat at random. But it holds for all declarators that when you use them in the declaration, a result comes about whose type is that of the declaration. You can convince yourself that this is so. In keeping with the title, let's construct some complex declarations:

```
int (*alpha)[5], *beta[5], (*gamma)();
```

Parentheses can also be inserted in order to change the precedence of the operators used. From Section 6.2.12 you know that all parentheses are evaluated before the * operator.

Look at the three declarators. First the * operator is used on `alpha`, with the result that `alpha` is a pointer. Then the index brackets are evaluated, meaning that `*alpha` is an array or `alpha` is a pointer to an array with 5 int elements.

For the second declarator the index brackets are evaluated first. `beta` is an array whose five elements are all pointers to `int` objects.

`gamma` is then a pointer to a function which returns an `int` value as the result.

You see that arbitrarily complex declarators can be used. These are, however, seldom needed.

6.5.5 Pointer arrays

The above declaration of `beta` was such a pointer array, that is, the elements of the array are pointers. Such pointers must first be selected by an index from the array and can then be used like pointers.

The use of pointer arrays of type `char` is of interest to us. It has already been mentioned that string constants can be used like array names. A string constant is a constant address to the specified string and can therefore also be used like an address. For example, you can initialize a pointer array of type `char` with strings.

```
#include "stdio.h"

main()
{   static *string[13]= {NIL,
                           "January\n",
                           "February\n",
                        ...
                           "December\n"};
    int i;

    for (i=1; i<13; i++)
        puts(string[i]);
    getchar();
}
```

In this program the array `string` is initialized with the month name. In reality the compiler places the string somewhere in the program and initializes the address to the string. The element 0 is set to NIL. This is the address to "nothing." NIL must be used carefully. In no case may an object be accessed via the address NIL. NIL serves only to indicate that such an access is not allowed.

The program passes the address of the `i`th string to the function `puts` (put string) and prints this on the screen. `getchar` waits for a key so that the output is not immediately erased again.

Keep this initialization separate from the initialization of character arrays through string constants. Here only the address is initialized. But for character arrays the contents of the string are placed in the array.

6.5.6 Pointers and multi-dimension arrays

Take a look at the following definitions:

```
int alpha[5][5];
int *beta[5];
```

In the first case we have a two-dimensional array and in the second a pointer array. The beginner will probably find it difficult to keep the two constructions apart. Both can be used in the same manner:

```
    alpha[2][2];
    beta[2][2];
```

or:

```
    *alpha[1];
    *beta[1];
```

You must differentiate between them, however. `alpha` is an array which actually consists of 25 `int` elements. `beta`, on the other hand, consists of five elements. But these are all pointers. `beta` does not generate a single `int` object. An element of `beta` can only point to an `int` object.

The advantage of arrays of pointers is that a pointer of the array can point to a subarray of unlimited length while still allowing it to be accessed like a two dimensional array. The different pointers can point to different length arrays, while the number of elements in a two-dimensional array is predetermined. The disadvantage is that the subarrays must be declared separately and the whole construction requires more memory space because the pointer objects must be added.

We saw that pointer arrays can access arrays of varying lengths in Section 6.5.5. The list of month names, which was assigned to the array strings, can be though of as subarrays.

```
    strings[12][0];
```

access the letter D in the month December, and so on. The second dimension is variable and is dependent on the initialization.

6.6 Structures and variants (struct/union)

Structures exits in every high-level programming language. In Pascal and related languages they are called RECORDs. A structure is a type. Objects of this type consist of several subobjects. You can select these subobjects as you can select an array element. The difference from an array is that a structure can contain subobjects of differing types.

6.6.1 Declarations of structures

Let's assure that you want to create a type in which you can store the data. To do this you would use a structure:

```
struct date { int day;
              int month;
              int year;
              char monthname[4];};
```

This whole construction can be used like a type name. `struct` is a keyword for the type structure. `date` is a `struct` name. The declarations enclosed in braces represent the subobjects of the structure. These component declarations are called the struct specifier.

There are several possibilities for declaring an object of type structure. No objects are defined in the above example. A `struct` name is defined. The specifier is assigned to this name so that you can omit the specifier in future declarations:

```
struct date birthday;
```

`birthday` is an object which consists of the above components.

We could also have defined this object along with the definition of date:

```
struct date { ...
                ... } birthday;
```

If you need a specifier only once, you don't have to define a `struct` name:

```
struct { int day;
           ... } birthday;
```

The part in front of `birthday` is one type name and so must stay together.

Complex declarators can also be used in declarations of the type structure and these can be declared in a list without having to repeat the type name:

```
     struct date birthday, *p, personal[50];
```

An object birthday, a pointer p to objects of type date, and an array which consists of 50 structures of type date are defined.

The declarators can also be defined in definitions. This does not work for the storage class auto, however. The initialization of the individual components is done with a list, similar to arrays.

```
     struct data birthday= {10,8,1965, "Aug"};
```

If fewer elements than components are specified, the rest are filled with zeros.

```
struct data personal[50]= { { 26,5,1939,"May"},
                            { 10,9,1935,"Sep"},
                            ...                 };
```

These lists can be nested again. The sublists can always be omitted if all subobjects are defined. The compiler then divides the elements of the list in order according to the array elements and components.

Some compilers allow you to define bit fields as components. This is not possible with Super C.

6.6.2 Access to components

Components in C are accessed with the . operator.

```
     birthday.day
```

The first operand is the name of the structure, the second is the selected component. The entire expression is an lvalue and can be used like any other lvalue. The type is the type of the component.

```
birthday.monthname
```

is naturally not an `lvalue` but an address to a character array with a maximum of 4 characters.

If you have a pointer to a structure, the access is possible as usual:

```
    (*p).year
```

`*p` must be put in parentheses because the . operator has precedence. This construction has its own operator, ->, which is used quite often.

```
    p->year
```

has the same effect.

If you have an array of structures, an element is selected and then the component:

```
    personal[5].monthname[3]
```

or:

```
    (personal+5)->monthname[3]
```

6.6.3 Functions and structures

Structures cannot be passed as arguments to functions, but addresses of structures can. The `&` operator can be used on structures for this. Also, a function cannot return a structure as a result, but it can return an address:

```
int monthdays(p)
  struct date p*;
{   static month[13]= {0,31,28,31,... };

    if (p->month==2)
        return(28+leapyear(p->year));
    else
        return(month[p->month]);
}
```

```
int leapyear(year)
  int year;
{      return(year%4=0  &&  year%100!=0  ||
year%400==0);
}
```

The function monthdays returns the maximum number of days in the month of the date to which p points. The familiar list of month days is used for this. In the case of February the result is 28+leapyear. leapyear is a function which returns 1 if the year passed to it is a leap year, else 0. To do this the function requires the year of the date as the argument.

The complicated condition in the return instruction in leapyear can best be read as:

If the year is *either* divisible by 4 *and* not divisible by 100 *or* it is divisible by 400, then the year is a leap year.

This makes the condition correct according to the Gregorian calendar in which a leap year occurs every four years, but not on whole centuries. Centuries which are divisible by 400 are leap years, however.

When the condition above is true it returns 1, otherwise 0. It returns exactly the result needed in the calculation.

The function monthdays can be called as follows:

```
     i=monthdays(&birthday);
```

6.6.4 Recursive structures

Structures can possesses structures as components. The component structure may not have a specifier, however. It must be previously defined with a struct name.

The same structure which is currently being defined cannot be declared as a substructure. It is allowed to declare pointers to the same structure as components.

We can define a tree structure with structures:

```
struct node { struct node *left;
              struct node *right;
              char nodename[20];};
```

Each node has pointers to two other nodes. The "tree" branches off to the right and left. Such tree are used to keep names in alphabetical order, for instance.

6.6.5 Variants

Variants are declared exactly like structures, but with `union` instead of `struct`. A variant is a special type. It can contain only *one* of the declared components, that is, the entire object can be used like one of the components. The storage space required is as large as the largest component.

Variants are used where one wants to store objects of various types and an object of a constant size is needed. If, for instance, you would want to define an object which can store a C constant:

```
    union cconst { int ivalue;
                   long lvalue;
                   double dvalue;
                   char *pvalue; };
```

you would define a variant. It can store an `int`, `long`, or `double` constant or an address to `char`.

The variants are accessed just like structures. The component which is intended must be specified.

```
    union cconst k, *p;
    ...
    k.ivalue=5;
    *p->pvalue='a';
```

The object k is large enough to store the largest component. This is independent of the system and is therefore easily portable.

You must ensure yourself that the variant is read as it was stored. If the components are changed on access, the result is not defined.

Variants can also be defined in structures and vice versa. In Super C, however, it is not possible to declare a specifier within another specifier. Specifiers of substructures of subvariants must be defined outside with their own names.

A variant can also be designated as a structure whose components are all stored at the start of the object or which have the relative address 0. Variants, like structures, cannot be passed as arguments and also cannot be the result of a function. Variants cannot be initialized.

6.6.6 Type definitions

You can also define new data types in C. These are not really new, but are combinations of the existing types.

For such a definition you specify the "storage class" `typedef`. The compiler then recognizes that an object is not being defined but a type. A name is declared which can then be used as a type name. It represents the type with which it was declared.

```
    typedef int length;
```

`length` can now be used as a synonym for `int`:

```
    length len;
    static length l[20];
```

Another example:

```
    typedef char *string;
    string lines[5];
```

`lines` is an array with 5 pointers to `char`.

```
    typedef struct { double re,im; } complex;
```

Here the type complex is declared, which in reality is a structure and must be used as such.

```
complex x;
x.re=5.5;
x.im=-0.5;
```

6.7 Programming environment

Now that you have become acquainted with the essential C components, you should in this section learn the particular features of this C system. The input and output functions are not contained in the compass of the language. These are directly related to the hardware in question.

6.7.1 Files

You know from BASIC how files are opened. The functions open and close are used for this in C:

```
open(8,15,"");
```

opens the error channel (15) of the disk drive (device 8). The filename must always be given--here it is an empty string. You have no doubt noticed that the logical file number is missing. This is not required in C. A similar instrument is the file descriptor. The file descriptor is used just like the file number in order to access a file. A file descriptor is an object which should be defined with the type file.

```
file fchannel;
fchannel=open(8,15,"");
```

The result of the function is the file descriptor for the opened file. The result of open must be stored or you will be able to neither use the file nor close it.

The file is closed with:

```
     close (fchannel);
```

The type `file` is defined with `typedef` in the file `stdio.h` and is usually not available in C.

You have already become acquainted with the functions `puts` and `gets`. There are corresponding functions by the names of `fputs` and `fgets`, which do not operate on the screen or keyboard, but read from or write to a file.

```
     fputs("n0:program disk,cc\n",fchannel);
```

This sends a format command to the disk and erases the diskette.

```
     fgets(string,40,fchannel);
```

reads the first 40 characters of the error message and stores them in the array string.

6.7.2 EOF

In order to recognize the end of a file there is an EOF flag, End Of File. This flag is realized with a macro which has the value 64 if EOF occurred or the value 0 if not. In order, for example, to get the error message from the disk drive, you read characters until EOF is encountered.

```
     #include "stdio.h"

     main()
     {    file fchannel=open(8,15,"");
          char c, status=0;

          while (!status)
          {    c=getc(fchannel);
               status=EOF;
               putchar();
          }
          close(fchannel);
          getchar();
     }
```

The EOF value must be placed in a temporary variable because it may be changed by other input/output functions like `putchar`.

`putchar` outputs a character on the screen. `getc` reads a character from the specified file.

6.7.3 STDIO

`STDIO` is a special file descriptor. It can be specified anywhere a file descriptor is required as an argument. No open call is required for `STDIO`, however. Outputs are directed to the screen by `STDIO`, inputs are read from the keyboard:

```
#include "stdio.h"

main()
{   static char command[40];
    file fchannel=open(8,15,"");

    fgets(command, 39, STDIO);
    fputs(command, fchannel);
    close(fchannel);
}
```

A string is read from the standard input and is printed in the error channel as a command.

`STDIO` is defined in `"stdio.h"` and represents the value 0.

6.7.4 Additional functions

The standard modules contain a number of other functions whose exact descriptions you can find in the system section.

Important and useful are the functions `printf` and `scanf` which make formatted output and input possible. These two functions are relatively large. There are therefore contained only in module `libc.l`. Otherwise the modules `libcs.l` and `libc.l` contain the same functions.

The declaration file `stdio.h` can be used for both modules. The declaration of `printf` and so on when using `libcs.l` does not create an error as long as these functions are not actually called.

6.7.5 Error handling

The error handling in C is system dependent and are therefore not necessarily portable. In this system you can turn the error messages on or off so that errors can also be processed in the program. If the error messages are enabled, the following message might appear:

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

You can end the program with the [x] key, restart it with [r], or continue executing with [c].

Caution is advised in the last two cases. No `static` or `global` objects are initialized or set to zero when the program is restarted.

If the program execution is continued, other errors may occur because the value of a division by zero is set to zero.

The error messages can be turned on and off with the procedures `erron()` and `erroff()`. The default condition is `erron()`.

6.7.6 Interruption

In BASIC you can interrupt the program with the keys [STOP] and [RESTORE]. This is also possible in C and is sometimes useful, for exiting an infinite loop, for instance. The message:

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

is printed. You have the same options as for the error messages. Here the same caution is advised. There is also the possiblity that the NMI interruption through [STOP]+[RESTORE] can come during input/output operations. It may occur that undesired side effects will result from the continuation of the program with the [c] key.

This interruption can be turned on and off independent of the error messages. The procedures nmion() and nmioff() are available for this purpose. nmion() is the initial condition.

Part II

System Guide

[illegible]

1. The command processor

1.1 Start, NMI, and RESET

Super C V3

The master disk must be placed in drive a,
press the RESET key or
enter the command BOOT in BASIC.
In both cases the system will be booted.

After the booting the file `autoexec` will be executed. This is a file created by the editor. The first line of the text contains a command which will be executed.

On the master disk the command in `autoexec` is

```
lram stdio.p
```

The prompt is light red at the start of version V3, since the `lram` command is resident.

When starting the screen will be selected by means of the 40/80 column key.

Super C V2

The master disk can be in any drive. Load the CCP with

```
-load "c-system",8,1 or
-load "c-system",8
 run
```

The specified device number must be changed if you are loading from device 9. Super C V2 can also be booted in the C-128 like V3. The 64 mode is automatically enabled. `autoexec` is not executed in V2.

An NMI (non-maskable interrupt) can be generated by pressing the [STOP]+[RESTORE] keys. This places the CCP back in the command mode. A loaded `copy`, `lram`, `sram`, or `sysgen` command is no longer resident. The screen in V3 will be selected according to the 40/80 column key.

The NMI is used to bypass certain sticking points of the `CCP`. The NMI should only be used then. The NMI should be avoided during input and output operations.

Super C V3

The C-128 has a RESET button. This can be used to restart the processor if the computer crashes for some reason. After pressing the RESET button the `CCP` will hopefully respond. If an erroneous program caused the crash, it may have destroyed part of the `CCP` or the RAM disk driver. You will notice this if the `CCP` or the RAM disk do unusual things or certain commands cause the computer to crash.

If this is the case, even the RESET button will not help. You can try to call the command `c-system` in the `CCP` in order to restore the `CCP` and the RAM disk driver. This will erase the contents of the RAM disk, however. If the `CCP` will no longer execute this command, the only thing left is to turn the computer off.

When the RESET key is pressed, the contents of the RAM disk are *not* lost. All connected devices are reinitialized so that device addresses set with device will be erased. The RAM disk will be set back to the device address h.

Super C V2

If version V2 is used on the C-128, the RESET button has the same effect as if you had turned the computer off and then back on.

1.2 Device identifiers and filenames

The device identifier specifies which drive the filename refers to. The letters a to h signify the device addresses 8 through 9.

```
a  b  c  d  e  f  g  h
----------------------
8  9 10 11 12 13 14 15
```

As a general rule a device identifier consists of the device letter and a colon. For example:

```
b:
```

The following device identifiers are also possible for dual drives:

```
b0:  b1:  b:0:  b:1:
```

in which 0 and 1 specify the drive.

A filename consists of a device identifier and a name. The name should not contain any of the following characters:

```
: = , * ?
```

The characters ? and * are wildcards and can be used only if the drive allows (not with write accesses). The * character also has a special meaning in some CCP commands.

If a filename is specified without a device identifier, it refers to device 8 or a. This is not the case if a command name is given without a device identifier, however. Then the name refers to the device which the prompt specifies, even if this is erased from the input line.

1.3 Extensions

The end of a filename is usually provided with a period and a letter in order to indicate the type and use of the file.

There is a convention for this which is derived from the UNIX operating system:

`.c`	C program as text file
`.h`	header file as text file (contains declarations)
`.e`	error file as text file (created by the compiler)
`.o`	object file as link file (created by the compiler)
`.l`	library as link file (created by the compiler)
`.b`	finished, executable program as B version (to be started from BASIC, V2 only)
`.p`	file package, contents of the RAM disk (created by `sram`)
no extension	finished program executable in the CCP

Typically the name before the extension remains the same. You would for example, write a text file with the name `test.c` with the editor and then compile this into the file `test.o` which is then linked with the standard functions in the linker, resulting in the executable program `test`.

1.4 Passing arguments

The CCP reads a line and evaluates it as a command. The input line is first divided into arguments. The space is used as the separator, for example:

```
a: copy b:test.c to h:test.h
 <arg0>  <arg1>  <arg2> <arg3>
```

First a check is made to see if `<arg0>` is a resident command. If so, it is processed. Otherwise the file designated by `<arg0>` is loaded and started. The arguments and the number of arguments are passed to this program.

The maximum length of the input line is 80 characters for V2 and 160 characters for V3. The maximum number of arguments for V2 is 40 and 63 for V3. If a space is to appear within an argument, it must be entered as a shifted space. This will be displayed as a space with a small dot in the middle (in the C character set). The shifted space will be converted to a space after the evaluation.

Parameters from the input line can also be passed to C programs. The following or a similar declaration must be made in the function `main`:

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

`argc` contains the number of arguments. Argument 0 is included in the count. In the above example `argc` would equal 4. Leading and trailing spaces are removed from the argument text.

```
arg[0] points to "a:copy"
arg[1] points to "b:test.c"
arg[2] points to "to"
arg[3] points to "h:test.h"
```

1.5 Character sets

Super C V3

After booting, after NMI or RESET and after the end of each command the C character set is automatically set (on both screens). In addition, the C character set can be set on the current screen by pressing or printing (within a C program, for instance) the sequence

[ESC] [1]

The Commodore character set is obtained on the current screen by the sequence:

[ESC] [2]

[SHIFT]+[CBM] can be used here to switch between upper and lower case, as is usual with BASIC.

[SHIFT]+[CBM] switches between upper and lower case.

Super C V2

In version V2, two character sets can be displayed: the C character set and the Commodore lower case character set. The C character set is set by the CCP as with V3. [SHIFT]+[CBM] can be used to switch between the two character sets.

The switch with [SHIFT]+[CBM] can be done by printing the control characters `'\16'` and `'\216'`.

`'\16'` switches to lower case (or C character set in V2)
`'\216'` switches to upper case (or CBM character set in V2)

1.6 Monitors

Two monitors can be connected to the C-128 at once, an 80-column monitor and a 40-column monitor. Both are supported by Super C version V3.

[ESC] [x]

switches the input to the CCP to the other monitor. In the fast mode the 40-column screen is no longer displayed, but it can still be addressed.

The 40/80 column key sets the screen on booting, RESET, NMI, and when ending commands.

In version V2 only the 40-column monitor can be used.

1.7 Resident commands

In the following discussion, if mention is made of the device identifier of a command, we mean the device identifier in front of the command name or, if this is missing, the identifier specified by the prompt. The device identifier of the following commands:

```
a:b:dir      b:dir
```

is in both cases `b:`.

`dir`
`dir   <arg1>` This command lists the directory of the disk in the device specified by the device identifier preceding `dir`. A specifier may be given as an argument. For example:

```
b:dir test*
```

displays all files whose names begin with test.

```
a:dir 1
```

lists the directory of device a drive 1 (for a double drive).

The error message `file not found` appears if the device is not available or if the directory is not readable.

err reads the error message of the device specifed by the device identifier.

com <agr1> The drive executes the text of `<arg1>` as a command. The command reads the error message after execution of the command. You can learn about the various commands in the disk drive manual. For all delete commands, note which drive the command will be sent to.

`file not found` will be printed if the device is not accessable.

time(in V3)
tod (in V2) outputs the current clock time. This clock time is set to zero upon booting, so the time that first appears will be the time since you started working with Super C. The clock can be set with the command set.

set <time> sets the clock time to <time>. <time> must be in the form `HH:MM:SS`, whereby

`HH` specifies the hours (00-23),
`MM` specifies the minutes (00-59),
`SS` specifies the seconds (00-59)

If you enter a syntactically-incorrect time, tod or time outputs a totally irrational time.

fast (V3 only) switches to the 2 MHz or FAST mode. The processor then runs twice as fast as before. The RAM disk in particular benefits from this speed, while the transfer from normal disks increases only slightly.

In the FAST mode the 40-column screen is not visible (it would slow things down too much), though it can still be accessed as usual.

`slow (V3 only)` switches back to the SLOW mode. The processor clock is set to 1 MHz. The 40-column screen becomes visible again.

`end` ends Super C. The effect is if you had turned the computer off and then back on again. In version V3 you should first take the master disk out of the disk drive or the computer will boot the C system again since the C-128 performs a boot operation when turned on.

1.8 Transient commands

For transient commands the device identifier of the command always indicates the drive from which the command will be loaded:

```
b: a:device
```

loads `device` from `a`.

1.8.1 device

This program makes it possible to change the device address of disk drives from software. The following appears:

```
device a to b.
```

The cursor appears first on the `a`. You can set the device identifier of the drive whose device identifier you want to change with the letters `a` to `h`. You confirm this device address with [`RETURN`]. The cursor then goes to the `b` and you can set the device address which you want to have for the selected drive in the future. When you press [RETURN] the address will be changed. With [RUN/STOP] you can exit the input and return to the CCP.

When changing the device address, device indicates the DOS version of the disk drive. device can be used on all known Commodore drives and for the RAM disk. If you have an exotic drive, with which device does not work, you can change device accordingly. The C source file is stored on the master disk under the name device.c.

1.8.2 copy

```
copy
copy <source> to <dest>
copy <source> <dest>
```

With copy you can copy files of type USR, SEQ, and PRG between various devices. copy is loaded only on the first call. For all following calls copy is resident. This holds until another transient command is loaded. If you have only one drive you can proceed as follows: First call copy without any arguments so that copy will only be loaded into memory. Then remove the master disk from the drive and insert the disk from which you want to copy.

<source> and <dest> are filenames with device identifiers. <source> specifies the name and device of the program to be copied. <dest> specifies the device and the name of the copy. The name of the original will be used if the name of the copy is specified with *.

```
copy a:test h:*
```

Device identifiers can naturally be omitted for drive a. If you have a dual drive, the following command is also possible:

```
copy b1:test a0:*
```

If the program determines that `<dest>` and `<source>` address the same device, the original will first be loaded into memory and the message:

```
   quit to save!
```

will be displayed. You now have the opportunity to change the disks in this drive. Then press any key and `copy` will save the loaded file. If you have only one disk drive and want to copy `test` from diskette X to diskette Y: Insert disk X.

```
   copy test *
```

results in the message `quit to save!` Remove disk X and insert disk Y and press a key. When the `CCP` responds again, test has been copied.

The copy procedure can be interrupted with `[STOP]`. The following errors are possible:

`file not found`
- when copy was first called the program file was not found
- incorrect device identifier in the arguments
- <source> not found
- addressed device not present

`illegal command`
- illegal number of arguments
- * was improperly used as a destination argument

`break`
- termination through [STOP]

The prompt will be displayed in red instead of yellow while `copy` is resident. The device identifier for `copy` is only important for the first call. Arguments can also be passed on the first call (only of interest for users with several disk drives).

The RAM disk does not have a `copy` command for copying within the RAM disk. Use copy for this.

1.8.3 f

```
f <arg0> <arg1> <arg2> ...
```

`f` is a special command for fast-loading commands and programs. This fast loading works in both versions with the 1541/71 disk drives. In V3, however, the `f` command has no function with 1571 disk drives since these drives load quickly enough.

The arguments of `f` are first the command name with device identifier. All other arguments are passed to the command or program to be loaded. The device identifier in front of `f` determines from where `f` will be loaded. You can try out `f` on the master disk with `device`, for instance:

```
a:f a:device
```

If you want to use `f` for your application programs and you have only one disk drive, you should copy `f` to your work disk.

If, for example, you want to load the copy program quickly for this copy procedure, enter:

```
a:f a:copy a:f a:*
```

first loads `copy` quickly and then copies `f`.

For short programs `f` is not very efficient. But for longer programs it is possible to increase the speed of the load by a factor of three.

System components (editor, linker, and compiler) automatically recognize if a slow device is present and then load fast. `f` therefore has no effect. The 40-column screen will be dark while `f` is loading because the transfer would be slowed by memory accesses to the screen.

1.8.4 lram (V3 only)

```
lram
lram <file>
```

lram loads files saved with sram which contain the contents of the RAM disk. lram is loaded only on the first call and then remains resident like copy until the next command is loaded. This is only important for users with only one drive. They can first load the command, change diskettes, and then execute the command by calling lram with an argument.

lram expects a filename which designates a file which was stored with sram. This is loaded into the RAM disk. The old contents of the RAM disk are erased by this. lram determines the device number of the RAM disk itself.

1.8.5 sram (V3 only)

```
sram
sram <file>
```

sram saves the contents of the RAM disk to a file. This is expected as the argument. You can also enter a device identifier with * as the argument. Then the contents of the RAM disk will be saved under its name. This name appears in the first line of the directory.

Just like lram, sram remains resident after the first call. The contents of the RAM disk are not disturbed. sram determines the device number of the RAM disk itself.

1.8.6 sysgen

```
sysgen
gen <text>
```

sysgen serves to construct user disks which are to contain a command processor. These diskettes can then be booted like the master disk. After booting the CCP and the RAM disk are then available. You can then set up a autoexec file. This file can be created with the editor. It is possible, for instance, to load a user-created menu program or another user program after booting.

sysgen is called without arguments. After loading, the prompt will be red. Another command by the name of gen is now available.

The device identifier of gen specifies the drive whose disk is to be outfitted with a boot mechanism and CCP. If gen is called without arguments, the message booting c-system v3... will appear during booting. If you want your own message to appear, enter this as the argument. Remember that spaces must be entered as shifted spaces. In addition, the characters \n for new line and \t for tab are accessable in the argument text. For example:

```
a:gen user disk 1\n(contains test programs)\n
```

Later during booting, the message will appear:

```
   booting user disk 1
   (contains test programs)
```

gen uses track 1 sector 0 to create a boot block. If this is already occupied by programs, gen cannot be used. You can erase the user disk with com, whereby gen remains resident in memory. gen also saves the file c-system which contains the CCP and the RAM disk driver.

Possible errors:

file not found

for sysgen:	sysgen not found
	c-system not found

no block

for gen:	block 1,0 already occupied.

1.8.7 c-system

```
   c-system
```

`c-system` is the file which contains the CCP (and the RAM disk in V3). If the CCP or the RAM disk driver should be overwritten by a runaway C program, you can try to load `c-system` by entering it as a command in the CCP.

If the CCP still functions enough so that the loading process is performed correctly, you will then have an error-free CCP available. This will erase the contents of the RAM disk.

1.8.8 cl

```
   cl <x:file> <link1> ...
```

`cl` is a command which calls the compiler and linking process. This is done by passing arguments to the compiler. `cl` expands some of the arguments required so that you can save a good deal of typing.

Super C V2

`cl` calls the compiler as follows.

```
   cc <x:file>.c <x:>o.o <x:>error.e <x:file>
    <x:>libc.l <link1> ...
```

The compiler will load file from the same drive as `cl`. The name specified by `<x:file>` determines the name and device identifier of the source text and the finished program. The file `o.o` will be used as the link file and it will be stored on the same device as the source file. The library `libc.l` is automatically linked to the user file. Additional libraries may be specified.

It is possible to use this command only with two drives. If, for example, the master disk is in drive a and the program disk in drive b, then you can compile the program `test.c` on the program disk in the following manner:

```
   a:cl b:test
```

The compiler and linker will be loaded automatically and the finished file `test` will be created on drive b, assuming that no errors occur during compiling and linking.

Super C V3

The `cl` command in version V3 runs similar to that under V2. The object file and the error file are both placed on the RAM disk, however. The library `libc.l` is also taken from the RAM disk.

```
   a:cl b:test
```

compiles `test` from drive b to the RAM disk. The finished program is again stored on drive b. If you have only one drive, you will also usually take the source file from the RAM disk. In this case `cl` works like it does in V2.

1.8.9 type

```
   type <file2> ...
```

The `type` command expects at least one filename. The file should be a text file created by the editor. `type` prints all text files which are passed as arguments one after the other on the screen.

`can't open` appears if the file is not present.

If, for example, you want to know what text is stored in the `autoexec` file on the master disk, enter:

```
   a:type a:autoexec
```

2. RAM disk

2.1 Deviations from Commodore DOS

The RAM disk is available only in Super C V3 (C-128). In this chapter we will outline only the differences from Commodore DOS. Since it is not possible to explain the function of the Commodore DOS in the framework of this manual, we refer you to the drive user's manual or to the appropriate literature.

The second RAM bank in the C-128 and the eventual memory expansions up to 256K are used for the RAM disk. The RAM DOS is loaded when the system is booted and remains available the entire time. The device identifier of the RAM disk is `h`.

You can use the RAM disk like a normal Commodore disk drive. Except for the following points, there is no difference between a normal disk drive and the RAM disk:

- The RAM DISK cannot use relative files.
- The memory commands (`m-r`, `m-e`, `m-w`) do not exist.
- The user vectors `u0`, `u2` to `u8` do nothing and cannot be set. (`u1` and `u2` retain their significance: modified block read/write).
- `u9`, `u:` generate a RESET of the RAM disk. The disk contents are retained.
- The command `&` for USER files does not exist.
- The command `p` for selecting a record of a relative file is missing.
- The command `b-e` does not exist.
- The command `c` for copying and appending files does not exist.
- The command `$` always transfers all the directory (no specifier allowed).
- The ID specification in the command n for new has no effect.
- Track and sector division is different from the Commodore drives.
- The construction of the BAM as well as the directory is different from the Commodore format.

- In the RAM disk the device address can be changed by the command u: or u9. Simply enter the device identifier behind the command:

```
h:com u:b
file not found
```

The device address is now changed from h to b. The error message appears only because com tries to read the error channel, but device h no longer exists at this point (it became device b).

2.2 Track and sector division

If you do not want to manipulate the RAM disk with direct access, this chapter is not important for you.

The track and sector division is shown in the following table:

Track	Sectors
1	0-11
2-15	0-15
16	0-12

up to here without memory expansion (239 blocks free)

Track	Sectors
17	0-11
18-31	0-15
32	0-12

up to here with 64K memory expansion (488 blocks free)

Track	Sectors
33	0-11
34-47	0-15
48	0-12

up to here with 128K memory expansion (737 blocks free)

The BAM block is in track 1 sector 0. 9 blocks (from 1,1 to 1,9) are allocated for the directory. The directory can accept a maximum of 99 entries.

2.3 The RAM disk commands

This description should give you a glimpse of the commands. It is neither comprehensive nor complete by any means. For an exact study, read your disk drive manual.

`i[dr] (initialize)`
Initializes the disk. The drive specification is optional. You can specify only 0 since 1 will result in "`drive not ready`". The command does not work in the RAM disk

`v[dr] (validate)`
Puts the directory and the sector chaining in order. The drive specification behaves as for `i`.

`s:name[=type] [,name[=type]...] (scratch)`
Files can be erased with this command. You can specify multiple names (up to four). The names can contain * and ?. In addition you can restrict the types of the files by specifying the type behind the name.

Example:

```
h:com s:lib*=seq
01, files scratched,00,00
```

Since there are no sequential files in the RAM disk with begin with `lib`, none are erased.

```
h:com s:lib*
01, files scratched,03,00
```

Now all library files are deleted.

`r:namenew=nameold (rename)`
The file `nameold` will be renamed `namenew`. Note the direction of the name change.

`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 is 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 occurence 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 preceeding 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 contigious 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 existsing 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 a 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 lengh 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.

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.

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 `cl` (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. COMPILE 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 to 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 recongized 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.

?EXRESSION 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 ERRQR`
- 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 occoured 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 it, 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 furture 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 is 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 addreses 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 the 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:

```
stdio.h      -     libc.l and libcs.l
math.h       -     libmath.l
graphic.h    -     libgraph.l
ctype.h
```

`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`.

`STDIO`	as file descriptor for the standard input/output
`NULL`	as 0
`CR`	as carriage return $0d
`CRSUP`	as code for cursor up
`CRSDOWN`	as code for cursor down
`CRSRIGHT`	as code for cursor right
`CRSLEFT`	as code for cursor left
`HOME`	as code for cursor home
`CLR`	as code for clear home
`REVERSON`	as code for RVS
`REVERSOFF`	as code for RVS off
`NIL`	as 0, pointer to nothing
`EMPTY`	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(), nimon(), nimoff(), 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 it, 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 aquainted 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 is 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)
    charc; 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 by 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 where 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.

```
    sscanf(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 provided 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 location $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 %%.

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) is the graphics mode is currently enabled. If the text mode is enables, `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 (x1,y1) to (x2,y2). 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 (x1,y1). 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 are data will may 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);
    getchar();
}
```

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-\$8ffff	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 */
    fputf(col,sizeof(col),fd);
    if (ST)
        error("graphic file exists",20);
    if (fputf(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 consists 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:

`\b`	backspace	in Super C:	DELETE $14
`\t`	tab	in Super C:	SPACE $20
`\n`	line separator	in Super C:	CARRIAGE RETURN $0d
`\r`	carriage return	in Super C:	SHIFT RETURN $8d
`\e`	escape	in Super C:	ESCAPE $1b
`\\`	\		
`\'`	'		
`\"`	"		
`\0`	$00		
`\ddd`	d are octal digits, returns the value of the constant 0ddd, for example `\24` corresponds to `\b` 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 intialized 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:

`2`	`->`	`2`	`int`
`2L`	`->`	`2`	`long`
`010`	`->`	`8`	`int`
`0xffff`	`->`	`65535`	`long`
`1.5`	`->`	`1.5`	`double`
`1.5E2`	`->`	`1500.0`	`double`
`1.5e-2`	`->`	`0.015`	`double`
`.5`	`->`	`0.5`	`double`
`1e5`	`->`	`100000`	`double`
`"\""`	`->`	`string`	`"\0`
`"abc\n"`	`->`	`string`	`abc\n\0`

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. The 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 <= int <= long* and that of *float <= 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	1
`unsigned char`	-	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 are 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 arbitarily, 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 of the array. A name from an ***enum*** specifier is only a constant. The name of a structure of 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 (-> 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->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. The 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<<B is the bit pattern of A shifted B bits to the left. Zero-bits are shifted in on the right. A>>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 is 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. The 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
    typdef-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 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 by 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 an 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 it 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, of 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 an 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 identifer 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 correponds 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 progams 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 recongnize 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 infinte loop.

8.17 Differences from the C-Compiler 64

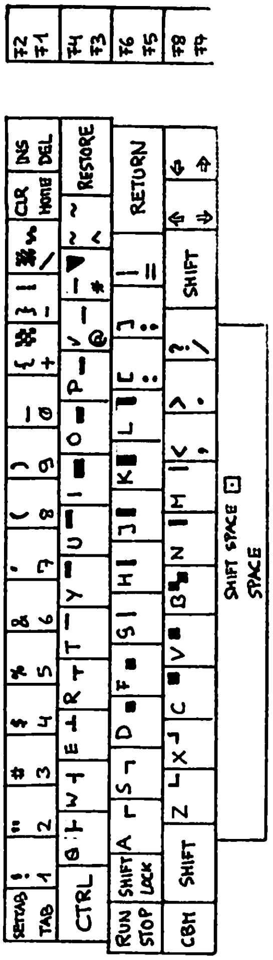
The language scope of the Super C compiler has hardly changed from from that of its predecessor Super C-64. Only some standard functions and macro definitions have been changed in order to realize better compatibility to UNIX:

`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`

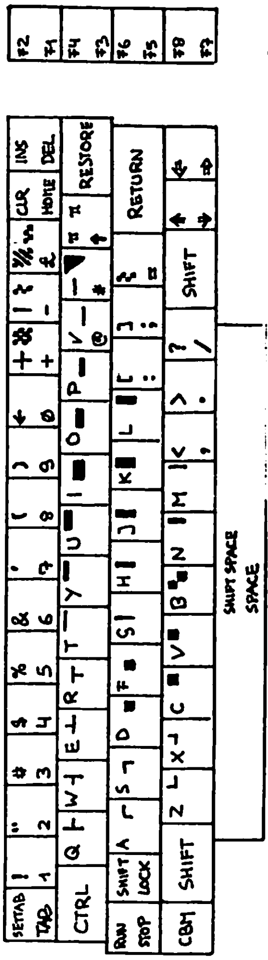
The function `inkey` was removed.
The function `CMOVE` is now called `cmove`.
The return values of the functions `strcpy`, `strcat`, `free`, and `gets` have changed.

PART III. Appendix

1.0 Keyboard V2

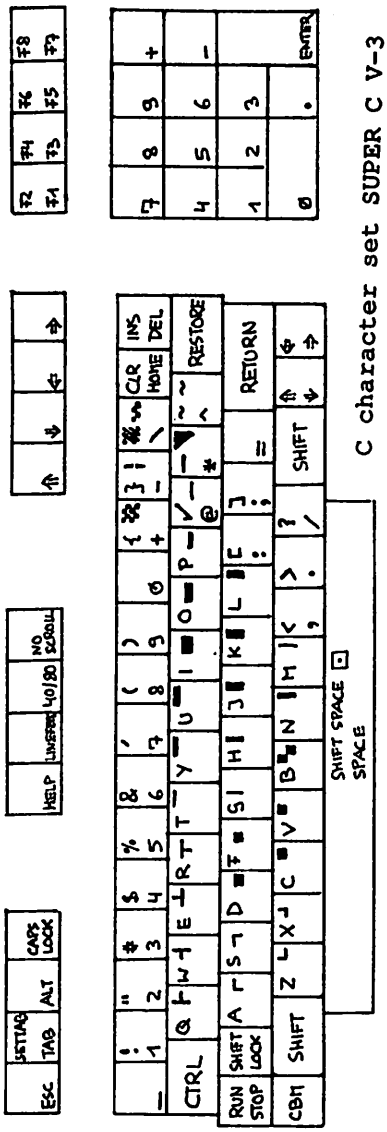


C character set SUPER C V-2



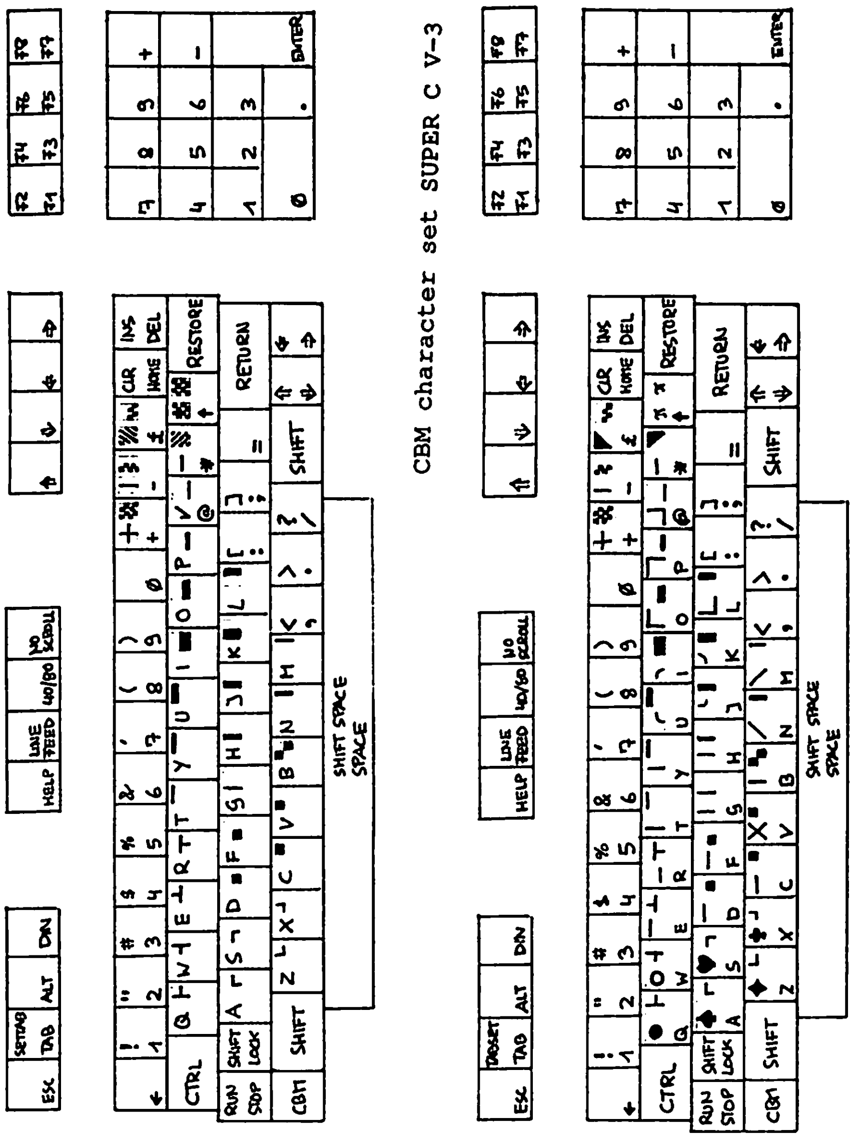
CBM character set SUPER C V-2

1.2 Keyboard V3 C character set



C character set SUPER C V-3

1.3 Keyboard V3 CBM character set



2.1 Character codes

	C-set	CBM-lower	CBM-upper		C-set	CBM-lower	CBM-upper		C-set	CBM-lower	CBM-upper		C-set	CBM-lower	CBM-upper
040				120	p		P	240	▨			320	P		▨
041	!			121	q		Q	241	▨			321	Q		▨
042	"			122	r		R	242	▨			322	R		▨
043	#			123	s		S	243	▨			323	S		▨
044	$			124	t		T	244	▨			324	T		▨
045	%			125	u		U	245	▨			325	U		▨
046	&			126	v		V	246	▨			326	V		▨
047	'			127	w		W	247	▨			327	W		▨
050	(			130	x		X	250	▨			330	X		▨
051	)			131	y		Y	251	▨		◩	331	Y		▨
052	*			132	z		Z	252	▨			332	Z		▨
053	+			133	[			253	▨			333	{	⊞	
054	,			134	\	£		254	▨			334	|	▨	
055	-			135	]			255	▨			335	}	▨	
056	.			136	^	↑		256	▨			336	~	▨	π
057	/			137	_	←		257	▨			337	◥	▨	◥
060	0			140	▨			260	▨			340	▨		
061	1			141	A		▨	261	▨			341	▨		
062	2			142	B		▨	262	▨			342	▨		
063	3			143	C		▨	263	▨			343	▨		
064	4			144	D		▨	264	▨			344	▨		
065	5			145	E		▨	265	▨			345	▨		
066	6			146	F		▨	266	▨			346	▨		
067	7			147	G		▨	267	▨			347	▨		
070	8			150	H		▨	270	▨			350	▨		
071	9			151	I		▨	271	▨			351	▨		◩
072	:			152	J		▨	272	▨		□	352	▨		
073	;			153	K		▨	273	▨			353	▨		
074	<			154	L		▨	274	▨			354	▨		
075	=			155	M		▨	275	▨			355	▨		
076	>			156	N		▨	276	▨			356	▨		
077	?			157	O		▨	277	▨			357	▨		
100	@			160	P		▨	300	▨			360	▨		
101	a		A	161	Q		▨	301	A		▨	361	▨		
102	b		B	162	R		▨	302	B		▨	362	▨		
103	c		C	163	S		▨	303	C		▨	363	▨		
104	d		D	164	T		▨	304	D		▨	364	▨		
105	e		E	165	U		▨	305	E		▨	365	▨		
106	f		F	166	V		▨	306	F		▨	366	▨		
107	g		G	167	W		▨	307	G		▨	367	▨		
110	h		H	170	X		▨	310	H		▨	370	▨		
111	i		I	171	Y		▨	311	I		▨	371	▨		
112	j		J	172	Z		▨	312	J		▨	372	▨		□
113	k		K	173	{	⊞		313	K		▨	373	▨		
114	l		L	174	|	▨		314	L		▨	374	▨		
115	m		M	175	}	▨		315	M		▨	375	▨		
116	n		N	176	~	▨	π	316	N		▨	376	▨		
117	o		O	177	◥	▨	◥	317	O		▨	377	~	▨	π

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	[CRTL]+[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	[CRTL]+[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 progrom 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]	curosr 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, QUOTEm 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 olderC-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)       (similaiar to V1 gets)
     char *line;
     int n;
     file fd;

int puts(line)
     char *line;

int fputs(line,fd)           (similaiar 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)       (similaiar to V1)
     char *str1,*str2;

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

char *strchr(str,c)
     char *str, c;
char *strrchr(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,bline)  (V3 only)
     int lcol,tline,rcol,bline;

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

```
/* library headerfile for 'libc.l' and 'libcs.l' */
/*     C-Compiler V2 Super-C             */

#define STDIO  0
#define NULL   0
#define CR     '\n'
#define CRSUP  '\221'
#define CRSDOWN    '\21'
#define CRSRIGHT   '\35'
#define CRSLEFT    '\235'
#define DELETE     '\b'
#define INSERT     '\224'
#define HOME   '\23'
#define CLR    '\223'
#define REVERSON   '\22'
#define REVERSOFF  '\222'

#define NIL    0
#define EMPTY  ""
#define MAXINT 32767
#define MAXLONG    2147483647L

#define ST     (*(char*)0x90)
#define EOF    (ST & 0x40)

#define putchar(X0)    putc(X0,STDIO)
#define cmove(X1,X2,X3) move(X1,X2,X3,0x35)

typedef int file;

extern file open(),close(),fopen(),fclose();
extern int  putc(),getc(),fgetc(),fputc();
extern char *gets(),*fgets();
extern int  puts(),fputs(),fgetf(),fputf();
extern void erron(),erroff(),nmion(),nmioff();
extern void error(),exit();
```

```
PAGE:   2       stdio.h
DATE: 4/21/86

extern int  qerror();
extern void cursor(),move(),exec();
extern int  strlen(),strcmp(),strncmp();
extern char *strcat(),*strncat(),*strcpy(),*strncpy();
extern char *strchr(),*strrchr();
extern char *alloc(),*free();
extern char *settime(),*gettime();
extern int  keys();
extern long call();

extern void printf(),sprintf(),fprintf();
extern int  scanf(), sscanf(), fscanf();

char (*screen)[40] = 0xe800;
char (*color )[40] = 0xd800;
char (*charram1)[8]= 0xd000;
char (*charram2)[8]= 0xd800;

char getchar()
{  char c;

   while((c=getc(STDIO))==0);
   return c;
}

```

V3: sthdio.h

```
/* library headerfile for 'libc.l' and 'libcs.l' */
/*      C-Compiler V3 Profi-C/Super-C          */

#define STDIO  0
#define NULL   0
#define CR     '\n'
#define CRSUP  '\221'
#define CRSDOWN    '\21'
#define CRSRIGHT   '\35'
#define CRSLEFT    '\235'
#define DELETE     '\b'
#define INSERT     '\224'
#define HOME   '\23'
#define CLR    '\223'
#define REVERSON   '\22'
#define REVERSOFF  '\222'

#define NIL    0
#define EMPTY  ""
#define MAXINT 32767
#define MAXLONG    2147483647L

#define ST     (*(char*)0x90)
#define EOF    (ST & 0x40)

#define putchar(X0) putc(X0,STDIO)

typedef int file;

extern file open(),close(),fopen(),fclose();
extern int  putc(),getc(),fgetc(),fputc();
extern char *gets(),*fgets();
extern int  puts(),fputs(),fgetf(),fputf();
extern void erron(),erroff(),nmion(),nmioff();
extern void error(),exit();
extern int  qerror();
extern void cursor(),cmove(),exec();
extern int  strlen(),strcmp(),strncmp();
extern char *strcat(),*strncat(),*strcpy(),*strncpy();
```

```
PAGE:  2      stdio.h
DATE: 4/21/86

extern char *strchr(),*strrchr();
extern char *alloc(),*free();
extern char *settime(),*gettime();
extern char io_in(),vdcin();
extern int  is80(),keys();
extern void io_out(),vdcout(),fast(),slow(),window();
extern long call();

extern void printf(),sprintf(),fprintf();
extern int  scanf(), sscanf(), fscanf();

char (*screen)[40] = 0x0400;
char (*color )[40] = 0xd800;
char (*charram1)[8]= 0x1000;
char (*charram2)[8]= 0x1000;

char getchar()
{  char c;

   while((c=getc(STDIO))==0);
   return c;
}

```

V2: graphic.h

```
PAGE:  1      graphic.h
DATE: 4/21/86

/*   headerfile for 'libgraph.l'    */
/*  C-Compiler V2 Super-C   */

extern void graphic(),graphon(),graphoff();
extern void backgr(),colors(),clrmap(),setcol(),fill();
extern int  dot(),dotin(),bdot();
extern int  line(),bline(),shape();
extern int  isgraph(),setplot(),plot();
extern char mapv(),*mline(),*oline();
extern char *plotobj(),*pushobj();
extern char *bplotobj(),*fplotobj();

extern char mask, *bmap;

#define black   0
#define white   1
#define red     2
#define cyan    3
#define purple  4
#define green   5
#define blue    6
#define yellow  7
#define orange  8
#define brown   9
#define pink   10
#define dgrey  11
#define grey   12
#define lgreen 13
#define lblue  14
#define lgrey  15

char (*video)[40]=0x8c00,     /* Video ram in Graphic-Mode
     *bitmap=0xa000,          /* Bit map in Graphic-Mode
     *vic=0xd000;             /* Base address Video Interface Chip */


```

V3: graphic.h

```
PAGE:  1      graphic.h
DATE: 4/21/86

/* headerfile for libgraph.l                                          */
/*                                                                    */

extern void graphic(),graphon(),graphoff();
extern void backgr(),colors(),clrmap(),setcol(),fill();
extern int  dot(),dotin(),bdot();
extern int  line(),bline(),shape();
extern int  isgraph(),setplot(),plot();
extern char mapv(),*mline(),*oline();
extern char *plotobj(),*pushobj();
extern char *bplotobj(),*fplotobj();

extern char mask, *bmap;

#define black   0
#define white   1
#define red     2
#define cyan    3
#define purple  4
#define green   5
#define blue    6
#define yellow  7
#define orange  8
#define brown   9
#define pink   10
#define dgrey  11
#define grey   12
#define lgreen 13
#define lblue  14
#define lgrey  15

char (*video)[40]=0xe000,    /* Video ram im Graphic-Mode  */
     *bitmap=0xc000;         /* Bit map of  Graphic-Mode   */



```

V2/V3: ctype.h

```
PAGE:  1      ctype.h
DATE: 4/21/86

/0       headerfile 'ctype'        */
/* C-Compiler V2/V3  Super-C */

#define isupper(X) (X>='A' & X<='Z')
#define islower(X) (X>='a' & X<='z')
#define isalpha(X) ((X)>='A' & X<='Z')|(X)>='a' & X<='z')|X=='_')
#define isdigit(X) (X>='0' & X<='9')
#define isspace(X) (X==' '|X==' '|X=='\t'|X=='\n'|X=='\r')

#define tolower(Y) (isupper(Y) ? Y & 0x7f : Y)
#define toupper(Y) (islower(Y) ? Y | 0x80 : Y)

/*       header file 'ctype'        */

```

V2/V3: math.h

```
PAGE:  1      math.h
DATE: 4/21/86

/*    headerfile for 'libmath.l'    */
/* C-Compiler V2/V3 Profi-C/Super-C */

#define PI 3.14159265359
#define E  2.71828182846

extern double sin(),cos(),tan();
extern double atan(),abs(),sgn();
extern double sqr(),sqrt(),rnd();
extern double log(),exp();


```

5. Listing "text.c"

```
PAGE:   1      text.c
DATE: 4/21/86

We can display 16 colors
on the screen.  The source text does
not produce them.  The colors
are used to high-light the most important
lines of the source text.



Color list:
CBM+1  black
CBM+2  white          ^^black^^
CBM+3  red
CBM+4  cyan
CBM+5  violet
CBM+6  green
CBM+7  blue
CBM+8  yellow
CTRL+1 orange
CTRL+2 brown
CTRL+3 light red
CTRL+4 dark gray
CTRL+5 gray
CTRL+6 light green
CTRL+7 light blue
CTRL+8 light gray



The Editor can display two character sets:

the BASIC-character set and a
special  C-character set

 !"#$%&'()*+,-./0123456789:;<=>?
@abcdefghijklmnopqrstuvwxyz[\]^_
`ABCDEFGHIJKLMNOPQRSTUVWXYZ{|}~
```

```
PAGE:   2      text.c
DATE: 4/21/86

---and  2 x 32 Graphic characters----

With [SHIFT]+[CBM] (64) or [ESC] [1/2] (128) it is possible to switch
between the two character sets.

The character set includes the following
special C-characters: \ ^ _ { ¦ } ~

When in the CBM-character mode, it will
be difficult to find the special C
characters, please use the C- character mode.

The Characters: _ is done with left arrow key
( [SHIFT] + [@] on 64)
and the Left-arrow key (64 only)is the
Editor TAB key. TAB SET and release is
done by using [SHIFT]+[Left-arrow] (64 only) [SHIFT] + [TAB] (128 only)
To obtain the character ¦ use the
[C=]+[-].


These lines stand at the start of our goal.
Our goal is to mark a block once.
Set  the  goal line  of the  previous line.



This line is the end of the Block ***************************************
This is the beginning of the text block that we will erase


                        GREEN

Block to be erased      only

                        has
```

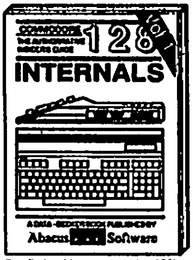
```
PAGE:   3      text.c
DATE: 4/21/86

 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                     12
 89                      13
 90                       14
 91                        15
 92                         16
 93                          17
 94                           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'=1=-=-=1 -
107
108
```

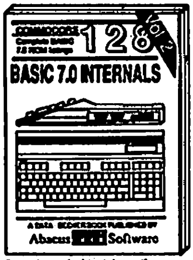
```
PAGE:   4       text.c
DATE: 4/21/86

 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"

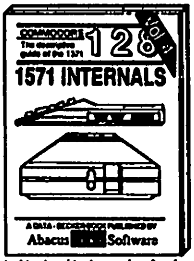
```
PAGE:  1      sample.c
DATE: 4/21/86

#include "stdio.h"
#define  CASE(Z) case '\Z': printf("'\\Z'   ");break

main()
{  char c;

   putc(CLR,STDIO);
   puts("Display the value for key pressed\n",STDIO);

   while()
   {
       c=getchar();
       char c;  /*This line is incorrect*/
       printf("Character: ");

       if((c & 0x7f) >= 0x20)
           if(c=='\\' || c=='\'' || c=='\"')
               printf("'\\%c'    ",c);
           else
               printf("'%c'    ",c);
       else
   } /*This line is incorrect*/
           switch(c)
           {
               CASE(n);
               CASE(t);
               CASE(f);
               CASE(r);
               CASE(b);
               default: printf("'\\%o' ",c);break;
           }


       printf("\nASC-Code: %3d   0X%02x   0%-3o\n\n",c,c,c);
   }
}
```

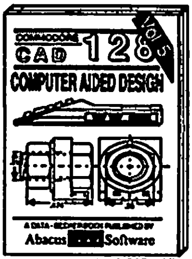
Index

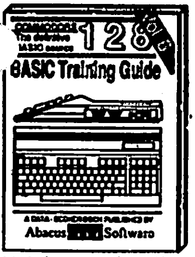

COMMODORE 128
INTERNALS
Vol 1
A DATA BECKER BOOK PUBLISHED BY
Abacus Software

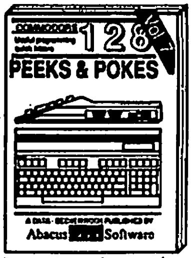

COMMODORE 128
BASIC 7.0 INTERNALS
Vol 2
A DATA BECKER BOOK PUBLISHED BY
Abacus Software

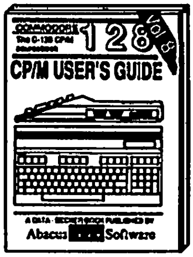
COMMODORE 128
TRICKS & TIPS
Vol 3
BEST SELLER
A DATA BECKER BOOK PUBLISHED BY
Abacus Software


COMMODORE 128
1571 INTERNALS
Vol 4
A DATA BECKER BOOK PUBLISHED BY
Abacus Software


COMMODORE CAD 128
COMPUTER AIDED DESIGN
Vol 5
A DATA BECKER BOOK PUBLISHED BY
Abacus Software


COMMODORE 128
The definitive BASIC source
BASIC Training Guide
Vol 6
A DATA BECKER BOOK PUBLISHED BY
Abacus Software


COMMODORE 128
Useful programming quick hitters
PEEKS & POKES
Vol 7
A DATA BECKER BOOK PUBLISHED BY
Abacus Software


COMMODORE 128
The C-128 CP/M
CP/M USER'S GUIDE
Vol 8
A DATA BECKER BOOK PUBLISHED BY
Abacus Software