Super C

the 4part C compiler

comprising:

- Easy to use Editor.
- 6502 Machine language compiler producing maximum 53k of object code.
- Linker allowing seven separately compiled modules into one executable program.
- Disk manager.

- **Plus comprehensive manual with a SYSTEM Guide for your day-to-day reference giving an exact description of the C language.**

FIRST SOFTWARE LTD

A Data Becker Product from First Software Limited

SUPER C LANGUAGE COMPILER
for the C-64 & C-128

By Thomas Eirich & Franz Hauck

A Data Becker Product

Published by

FIRST SOFTWARE LTD

Unit 20B, Horseshoe Road
Horseshoe Park
Pangbourne, Berks

ISBN 0 948015 11 x

INTRODUCTION

The programming language C has been in use since about 1972. It was developed by Dennis Ritchie for the Unix (*) operating system. With the spread of Unix, C also enjoys a great popularity. The **Super C** language compiler now makes it possible to program in C on the Commodore 64 and 128.

Like Pascal, C is a structured programming language. You can program small problems as functions (comparable to subroutines in BASIC) and then call functions using specific data. This programming technique results in readable and maintainable programs. By building libraries of functions, new programs can incorporate them to create easier solutions to a wide assortment of problems.

C is characterized by its many different data types. There are a total of twelve basic types, which can be combined into vectors (ARRAYs), structures (similar to the RECORD in Pascal), and variants (UNIONs). We should not forget the pointer which is of great importance in C. C supports pointers through clever pointer arithmetic. This moves C into the area of assembly languages. Like assembly languages, certain problems can be solved with the use of some tricks involving pointer arithmetic, without assuming any characteristics of the computer being used. This means that these programs will run on any C machine without significant changes.

The **Super C** compiler is a four-part system. An editor makes it easy to enter programs. The programs are translated into machine language by the compiler. The compiler is a complete version of the C language except for bit fields. The linker binds separately compiled programs together. The fourth component of the system is a disk manager which can be used to copy files.

This user's manual is divided into two major sections, a User's Guide and a System Guide. The User's Guide is designed to introduce you to the system. If you do not know how to program in C, you will find an introduction to the basics of the language in this section. The System Guide serves as a reference work for your day-to-day work with the **SUPER C System**. Here you will also find an exact description of the C language.

Even if you already know C, you should start with the User's Guide. At the appropriate points we will let you know which parts you can skip.

Franz J. Hauck
Thomas Eirich

(*) UNIX is a Trademark of Bell Laboratories.

Table of Contents

I. User's Guide

II. System Guide

III. Appendix

PART I. Users's Guide

1.0 Introduction

In this User's Guide we'll introduce you to the **SUPER C Compiler System**. The initial topics to be covered are the C-LOADER and the disk manager. In a way, the C-LOADER forms the basis of the entire system. The disk drive can be easily controlled with the disk manager, C-COPY. In addition, C-COPY is required to make the C system ready to use.

The functions of the editor, with which programs are entered, are explained in Section 4. There the sequence of operations required for program creation are discussed. At the close of the User's Guide you'll find an introduction to C. With the knowledge from the previous sections you can then use the sample programs.

Text which is entered into the computer or which is printed by the computer will be printed in a **bold type style.** Control characters will be enclosed in brackets, such as [RETURN] for example.

2.0 C-LOADER

To start the SUPER C System, you must insert the distribution diskette into your disk drive. We'll call this diskette the master disk. It is write and copy-protected and cannot accept any additional programs.

Once you have inserted the master disk, load the C-LOADER and start it. To do this, enter:

```
LOAD"C*",8  [RETURN]
```

[RETURN] indicates that you are to press the RETURN key. The computer loads the C-LOADER and responds with the following:

```
SEARCHING FOR C*
LOADING
READY.
```

Start the program by entering the following:

```
RUN  [RETURN]
```

The C system responds with a menu that looks like this:

```
(C) Copyright 1985 by DATA BECKER
          written by
  Thomas Eirich and Franz Hauck

SUPER-C for the C-64/C-128
  X: to basic

  a: c-copy          c: c-compiler
  b: c-editor        d: c-linker

  u: user file
```

This menu is the starting point for working with C. All parts of the SUPER C System are started from here. When parts such as the editor are exited, the user is returned to this menu. Compiled C programs can also be loaded and executed from this menu with the user selection.

By pressing the X key (shift + x) you are returned to BASIC. This erases the C-LOADER and it must be reloaded if it is to be used again.

The current menu selection is displayed in reverse. You can change a selection by using the cursor-up key or cursor-down key. The current menu item is activated by pressing the [RETURN] key. The menu selections can also be made by pressing the corresponding letter **u, a, b,c,** or **d.**

The selection **u (user)** loads a user program, programs written and compiled in C.

The other selections load and execute the corresponding parts of the SUPER C System. It is assumed that the master disk is in the disk drive in order to load that portion of the SUPER C System.

Select **a: c-copy** using the cursor-up and cursor-down keys and press **RETURN**. The drive will start to load C-COPY, the disk manager. After a short time you will see a list of commands. Now press:

```
x [RETURN]
```

This is the command to exit the C-COPY program. You'll find yourself back at the menu and can select another program. You can see how easy it is to select a part of the SUPER C System and get back to the menu without having to leave the C system.

3.0 C-COPY

Load the disk manager again by pressing the **a** key. C-COPY responds with a list of possible commands:

```
@ gets error message from floppy
< loads a file into memory
> saves a loaded file on disk
# fixes unit
/ displays directory
x return to loader
. sends a command to floppy
```

3.1 Error messages

You can display the current error message from the drive with the command @. The commands in C-COPY must always be followed by the RETURN key. Enter:

```
@ [RETURN]
```

The floppy disk error message now appears, hopefully `00,OK,00,00`.

3.2 Directory

You can display the directory by entering:

```
/ [RETURN]
```

The directory of the master disk is now displayed. A specifier can be given behind the / character. You then get only a part of the disk directory. For example, to see only the programs beginning with 'c', the command would be:

```
/c* [RETURN]
```

3.3 Send commands to the disk drive

All commands which you ordinarily send to the drive with PRINT# statements can be sent with the . (period) character in C-COPY. For this example, replace the master diskette with a diskette which can be erased. You'll need such a diskette later, so let's prepare it for later use.

We want to format this diskette with C-COPY. To do this, insert the disk into the disk drive. Now enter the following command:

```
.n:pgmdisk,cc  [RETURN]
```

The disk will be erased and is given the name "pgmdisk" and the ID "cc". If you are not familiar with the command **n** for NEW, you should refer to the appropriate section in your VIC-1541 User's Guide.

When you are familiar with the disk commands, you can access these with the . to do such things as rename or erase files. The corresponding commands are used exactly as in BASIC. The command text need only be given without quotation marks behind the . character.

3.4 Copying

To copy files you must first load them into the computer's memory and then save them again. C-COPY has a load and a save command to do this.

Insert the master disk again and give the following command:

```
<stdio1.l  [RETURN]
```

C-COPY loads the file **stdio1.l** from the master disk and responds with:

```
file loaded
```

Insert the newly formatted diskette named "pgmdisk" and enter the save command:

```
>stdio1.l  [RETURN]
```

C-COPY responds with:

```
file saved
```

Now the file named **stdio1.l** has been copied from the master disk to your program diskette. If the name of the copy is to be the same as the original, the filename can be omitted from the save command. You then need only enter the command >. File types PRG, SEQ, and USR are automatically recognized by C-COPY.

If you have two drives, you can also change the preset device number 8 for the drive. If you want to copy from drive #8 to drive #9, enter the following command before saving:

```
#9  [RETURN]
```

The save command now works on device number 9. You can switch it back with #8.

3.5 Copying the standard files

To be able to work with the C compiler, some files must be copied from the master disk to the program disk. These include two sample programs and the standard library which must be bound to your C programs with the linker. This contains functions for input and output such as **open, close,** and others. In C, these functions do not automatically belong to the language. They must always be bound to your programs.

Copy these files:

```
stdio1.l  (if you did the previous example, this file is already copied)
stdio2.l
stdio.c
text.c
sample.c
```

Once you have this procedure behind you, your C system is ready to use. Exit C-COPY with the command

```
x  [RETURN]
```

and you'll find yourself back in the menu.

4.0 C-EDITOR

From the menu press the letter **b** to load the C-EDITOR after inserting the master diskette.

4.1 New - create new file

For practice, enter a small document. You begin a new document with the editor command **new**. To do this, press the command key **F5**. This message appears in the first line of the screen:

```
enter command
```

This means that the editor is waiting for you to press a key which determines the command. Press the **n** key for **new**. To confirm the command choice, the name of the command always appear in the first line on the screen. For the command **new**, you are asked to enter the length of the lines.

```
new: length of line Δ
```

You can enter any number between 40 and 80 for the length of the line. This allows you to set the maximum line length of your document. This length cannot be changed later. When entering the length you can enter only the digits 0 to 9 and the [DEL] key and [RETURN] keys. [DEL] erases the last character entered and [RETURN] ends the input. Try the [DEL] key with this input once and enter the number 63. Now erase 63 by pressing the [DEL] key twice and enter 80. Finally, end the input by pressing [RETURN].

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

The cursor is now located behind the word **file:** on the second screen line. Here the editor expects a character string, the filename of the document. Enter:

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

Again, [DEL] erases the last character entered. The other control keys have no effect. Let's assume that you want to enter **testfile** instead of **textfile**. You must delete all of the characters up to and including the **x** and then re-enter the rest of the name.

Press [RETURN] to end the input.

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

Now you have a document with a line length of 80 and the filename `testfile`.

4.2 Inserting and deleting lines

After the **new** command, the cursor is positioned in the text field. The text field is lines 4 through 25 of the screen.

When you now press letter or number keys, you see that the message `last line` is displayed in the first line of the screen. The characters entered do not appear on the screen, but the cursor moves forward.

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

When a document is started it has only one line - the last line. You cannot write on this line, nor can you position the cursor beyond it. In both cases the editor responds with `last line`.

To enter text, insert a few blank lines. To do this, press the **F7** key several (six) times. The **F7** key is used to insert blank lines.

The text field now has the dimensions of 6 lines by 80 characters. In this text field you can position the cursor at any point using the cursor control keys. Each key that you enter is immediately echoed on the screen. You don't have to press [RETURN] to record the entered text.

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

Use the cursor keys and the [SHIFT]+[RETURN] key to move around within the text field. You'll notice that the screen shifts to the left when you move the cursor beyond column 20. When you move the cursor to the left, the screen moves to the right. Since the screen can display only 40 characters per

line, the characters outside the screen display area are brought into view by shifting the screen left or right.

Move the cursor to line one, column one. You can also use the control key [SHIFT]+[HOME] to position to line one, column one.

Now enter the following text. You can also use the control keys [DEL] and [INS], [SHIFT]+[DEL], when entering the text. These keys have the same function here as they do in BASIC.

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

Position your cursor somewhere in the second line and press the **F7** key to insert a blank line. All lines from the cursor line forward are moved down one line and the cursor is positioned in the blank line. Now you can insert more text. If one line is not enough, press the **F7** key several times.

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

Use the **F8**, [SHIFT]+**F7**, key to delete lines. The cursor line disappears and all of the following lines are moved up one line. Press **F8** again and the second line of the document disappears. It has been erased and all of the following lines are moved up.

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

The editor permits each line to have its own color. Set the colors with the color control keys ([CBM]+[1] through [CBM]+[8] and [CTRL]+[1] through [CTRL]+[8]). The cursor line is changed to another color.

Position the cursor to line one and press [CTRL]+[6]. The cursor line becomes green. Move the cursor to line two and press the keys [CBM]+[4]. The line is dark grey. The color of the last line cannot be changed--it is always red.

For line insertions (**F7**), the color of the line moved down is kept. If you press **F7** in line two, the blank line has the color dark grey.

Assuming you want to insert seven red lines between line one and line two, you do not have to insert seven lines and then color these seven red. You need only insert one line, set its color to red and then insert the other six. These then all have the color red.

You can now experiment with the colors and inserting and deleting lines before moving on to the next section.

4.3 Saving text

Now the text should be saved. Be sure to insert a (formatted) work diskette into the drive. You must first name your text before you can save it. Select the command **file** by pressing the command key **F5** and the the **f** key. The command **`file:`** appears in the second line followed by the cursor. Type in a name for your document and then press [RETURN]. Now you may select the command **save** by pressing the command key **F5** and then the **s** key. The command **`save:`** appears in the first line and the editor saves the text. After the document is saved the cursor appears at its former position in the text field.

Repeat the save (**F5 s**). After the drive runs for a short time, the following question appears behind the command name:

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

The file which you want to save already exists on the diskette. The editor is now asking you if the file should be replaced. If so, press the **y** key. With **n** for no, the command is cancelled and you are returned to the input mode.

4.4 Loading text

A document should now be loaded. Insert the master diskette from which we'll load a sample program.

Press the command key **F5** followed by the **l** key for **load**. The command is displayed in the first line and the cursor is positioned behind `file:` in line two of the screen. You can now enter a filename. Enter the filename `text.c`:

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

After pressing the [RETURN] key, the file is loaded. Any existing document in memory is replaced by the new file (text.c).

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 to examine the document, but do not change it. Try out the horizontal scrolling. Since this document contains enough lines, we can also use the vertical scrolling by moving the cursor beyond the top or bottom lines of the screen.

You can also use [RETURN] to move to the start of the next line, or [SHIFT]+[RETURN] to jump to the end of the previous line. In addition you can use two additional control keys to move the cursor through the text quickly. The **F1** key is the **page down** key. The screen displays the document from the 22nd line following the cursor line. The **F2** key, [SHIFT]+**F1**, is the **page up** key. Here the 22 lines preceding the cursor line are displayed on the screen.

4.5 Block commands

The block commands act on multiple lines of text. A block is a group of lines which are manipulated 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, right, up, down, [RETURN], [STOP]).

4.5.1 Delete block

Erase lines 64 through 95 from the example document **text.c**. Do this by using the block-delete command. (You can also delete lines individually with the control key **F8**).

Move the cursor to line 64. The status (first) line of the display indicates the cursor's location within the document. The first number indicates the column, the second the line. Call the **erase** command with the key sequence **F5**, followed by the **e** key. The editor indicates the command name and the block-input mode in the first line.

```
erase: marking out range
```

In block-input mode the lines which mark the block are displayed in reverse. You can increase or decrease the size of the block by using the cursor up/down keys.

Increase the size of the block with the cursor down key until the reverse line field reaches line 99. But since we want to erase only up to line 95, decrease the block size with cursor up until the reverse block includes only up to line 95. The status line will help you determine the line numbers.

End the block input with the [RETURN] key. A confirmation now appears:

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

The key **n** cancels the command and you are returned to the input without deleting the block. The key **y** erases the block. After deletion, the text behind the erased range is displayed. You can convince yourself that the block was erased by moving the cursor up a few lines.

You can also cancel the block input by pressing the [STOP] key.

4.5.2 Move block

With the command for the block move you can move blocks from one position in the document to another. Reload the document **text.c** by pressing the **f5** followed by **l** and then the filename **text.c**. In the sample document, move the line range 101 to 113 inclusive to 58.

Move the cursor to line 101. Call the **move** command (**F5**+ [**m**]). As you can see from the first line (`marking out range`), you are in the block-input mode. You must now mark the range which you want to move. This marking is done in the same way as described under "4.5.1 Delete block" by using the cursor keys.

Once you have ended the input with [RETURN], the message **fixing target** appears. The editor is asking for the destination (target) for the block move. This target line is displayed in reverse. In a block which is already displayed in reverse, the target line is displayed normally.

You must set the destination line at the line before that which the block is to be inserted. You can use the following control keys to do this:

⇑ ⇓ cursor up/down moves the destination line up and down.

The keys **F1** and **F2** page the destination line down and up, respectively.

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

[RETURN] ends the destination input. The destination line may not lie within the previously marked block. In this case, the editor will display the message, **no target line**, and will await a new target.

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

The marked block is moved and the document is displayed beginning with the text inserted. If you move the cursor one line up you will see the line that precedes the destination line.

You can also use the command **goto** after the command key **F5**. Press **F5** followed by **g**. The command name **goto line:** appears in the status line. You can enter a number with a maximum of three digits following the prompt. The cursor then jumps to the start of this specified line, but will not go past the last line of the document.

4.5.3 Copy block

Copying a block is very similar to moving a block. When copying, the marked block is duplicated before the destination line. It allows you to duplicate the block within the text. The command for copying a block is called **transfer**. The procedure for using it is the same as the **move** command,

except for one addition. When setting the target line one additional control key is active, the [HOME] key.

The editor has two text areas, the **file** area in which we have been working and the **extra text** area. The two text areas are indicated by the second status line. The file area is displayed as `file:` ***filename***. The extra text area contains additional text that is stored temporarily.

The [HOME] key switches the display between the **file** area and the **extra text** area. You can edit the extra text exactly as you would the file text, except for a few commands which are not allowed in extra text. Among these commands are **load** and **save**. Exchange the text and try to call the command **load** ([HOME] **F5 l**). The editor responds with the error message `illegal text`.

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

You can move text from one area to the other with the **transfer** command. Do this by pressing the [HOME] key when the editor is asking for the target.

For practice, move an arbitrary piece of text to the extra text area. After the command key **F5** select the **t** key for the command **transfer**. Now mark a block of text. When the editor asks for the target, switch the display to the extra text area by pressing the [HOME] key. You cannot move the target line within the extra text since the extra has only one line at the moment, the last line.

As soon as you have entered the target with [RETURN], the text block will be duplicated in the extra text area.

Using the extra text area you can insert text in other files, for example. If you copy a block of text into the extra text area and then switch back to file text area and load a different file, you can then insert the extra text in the file text with the **transfer** command.

Another very useful application of the extra text will be presented in Chapter 5.

4.6 Search and replace

You can search for text within the document and optionally replace the text with new text. Before searching and replacing text you must set a search string and an optional replacement string.

4.6.1 Searching

The search string is entered with the command **hunt**. Press the command key **F5** followed by the **h** key. The screen displays `hunt:`. Enter the search string followed by [RETURN]. If you make a mistake when entering the string, you can erase the last character entered with the [DEL] key. As soon as you press [RETURN] the search string is terminated and you find yourself back in the normal text input mode. Enter the search string **violet.**

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

Now you have set the search string. You can begin searching starting at the current cursor position by pressing the **F3** key. The cursor is placed at the first character of the string if it is found in the text.

You can go to the start of the document with [CLR] and search with **F3**. After this, the cursor is positioned to line 14 at the start of the string. Press **F3** again. The editor looks for the next occurence of the search string. Since this string does not occur in the document again, the last line of the document is displayed.

You should become aquainted with the search function by using strings which occur more often (such as "the").

4.6.2 Replacing

To replace text, the search and replace strings are entered with the command **replace**. Press the command key **F5** followed by **r**. The screen displays `hunt:`. Enter the search string followed by [RETURN]. Then the screen displays `rplc:`. Enter the replacement string followed by [RETURN]. Enter the following:

`hunt: violet [RETURN]` and `rplc: purple [RETURN]`

The text **violet** will be replaced by **purple** when you press the corresponding control key in the text input.

You can preform the replacement in two different ways. In the first, the text from the cursor position forward is completely searched and every possible replacement is made. In the second, the editor displays the position of the search string and asks if it should be replaced or not. The control key for the complete replacement is **F6** (SHIFT+**F5**), the key for the ask-before-replace is **F4** (SHIFT+**F3**).

First perform a replace with query. Go to the start of the document by pressing the [CLR] key and initiate the replace with query by pressing the **F4** key. The editor starts to search for string **violet** at the current cursor position. If it finds an occurence, the text is displayed in reverse. The question **replace y/n**? appears in the status line.

Press **y** to replace the string. Press **n** to bypass this text. In both cases the editor resumes its search once the question has been answered.

To replace without query, enter the command **replace** and enter the new search and replacement strings. For example, **the** as the search string and **THE** as the replacement string. Then go to the start of the document by pressing the [CLR] key and start the replace without query by pressing the **F6** key.

The editor replaces the string throughout the entire document without query. After all replacements the cursor is positioned to the last line.

The replacement process can be stopped with the [STOP] key. The cursor will then appear at the point in the document that the editor had reached in the process.

The following applies for replacing with or without query: If the replacement causes the maximum line length to be exceeded, the editor stops the operation without having made the replacement. The cursor is positioned at the start of the string and the error message **overflow in line** appears in the first screen line. You must then decide what to do with the text.

For practice try to create an **overflow in line** error. You must select a replacement string which is longer than the search string.

4.7 Return to the menu

You can stop all operations (search, all inputs and queries, loading, printing, etc.) by pressing the [STOP] key. A complete description of all functions can be found in the System Guide, section 3.

You can exit the editor with the command **exit**. Press the command key **F5** followed by the **x** key. The status line displays **`EXIT: are you sure y/n?`**. Press **n** to return to text input mode. Press **y** to return to the menu. In this case, the contents of the document are lost. Don't forget to save your text before exiting from the editor.

5.0 The first C program

In this section we'll create a simple C program. Starting with the editor and proceeding to the compler, linking and starting the program you'll see how to work with the **Super C** language compiler using the sample program.

Three files are create by the SUPER C System: the source file, the link file, and the program file. The source file is created with the C-EDITOR and contains the program text. The C-COMPILER converts this source file into a link file. The link file is then bound with other link files to create the program file by the C-LINKER. The program file then contains the executable C program.

To distinguish these three files from each other, we recommend that you denote source files with a **.c** at the end of the filename and link files with **.l**. The filename without a suffix is then the program file. Let's assume that you want to write a C program named ***test***. The source file would then have the name ***test.c*** and the link file would be called ***test.l***. This rule is only a suggestion; you can of course use any system you like to keep the file types organized.

If you are not now in the menu of the LOADER, load the C-LOADER from the master disk and RUN it. Then load the EDITOR.

5.1 Editing

You create the program text with the C-EDITOR. You already know how to use the editor fromSection 4. We will not go into that here. Since you must first learn how to use C, you do not need to edit a source file here. For this purpose we have already prepared a source file on the master disk. Load the source file called **sample.c** with the editor (**F5 l sample.c** [RETURN]).

You can look at the program text, but don't change it. Save the file on a work disk (**F5 s**) if you have not done this already.

Imagine that you created this source file with the C-EDITOR and want to compile it. All source files which are to be compiled must be on one disk. In this case, these are the source files **sample.c** and **stdio.c.** The file **stdio.c** is chained with the file **sample.c**. In the first line of **sample.c** you see the line:

```
#include "stdio.c"
```

This line means that the text of the file **stdio.c** will be inserted at this point. The purpose of **stdio.c** is explained later.

Before you run the compiler, you must be sure that all source files which the compiler requires are stored on one diskette. If a file is missing, the compilation process will be stopped with the message **file not found.**

Check to see that **stdio.c** and **sample.c** are on your work disk. To do this, switch to the extra text area [HOME] and load the directory of your work disk (**F5 d** [RETURN]).

The directory is inserted into the extra text area. Normally these two files are on your work disk since you copied them from the master disk in section 3 in the description of C-COPY.

Exit the EDITOR ([F5 x [RETURN]) and load the C-COMPILER from the LOADER (first insert the master disk).

5.2 Compiling

The C-COMPILER responds with the compiler message screen and asks for the name of the source file. First insert your work disk. Enter **sample.c** [RETURN].

When entering the filename the only control keys active are [DEL] ,[CLR] (SHIFT + HOME) and [RETURN.] [DEL] deletes the last character entered, [CLR] erases the entire input field. [RETURN] terminates the input.

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

After a short time, a second question appears, this time for the name of the link file. The link file is the file in which the converted program will be saved. If your source file ends with **.c,** the compiler supplies the filename extended with **.l.** It is recommended that you give your link files a common name to limit the number of file names on your disk.

Erase the input field with [CLR] (SHIFT + HOME) and enter the collective name (for example, l.l):

```
link file name:l.l [RETURN]
```

After this input the compiler has all the information it needs to begin the compilation.

The compiler outputs `stdio.c` in grey characters. This means that `stdio.c` has been read from the source file. The yellow message `'inkey'` tells you that the compiler is compiling the function **inkey**. After this comes a grey # character. This indicates that the source file `stdio.c` has been completely read and the COMPILER is reading the remainder of `sample.c`. After the compiler responds with `'main'` in yellow, some error messages will appear in red. These errors are not your fault, they are intentional.

The compiler ends 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 LOADER. With `r` you could restart the compiler to compile another source file. This saves having to reload the compiler.

If you see that the compilation is senseless or if you have entered incorrect parameters (source file or link file), you can interrupt the compiler at any time by pressing [STOP+RESTORE].

Since errors were present in the compilation, you must return to the editor and correct them. Load the C-EDITOR (insert the master disk).

Once you are in the EDITOR, insert your work disk and load the file **error-c**. The compiler places all of the error messages in this file. Copy this file into the extra text area. To do this use the **transfer** command (**F5 t**) and mark the entire file text as a block. After you have ended the block input with [RETURN], set the target. Place the target line to the extra text area with the [HOME] key and press [RETURN]. The file **error-c** wil be in the extra text area. Load the source file **sample.c** in the file text area.

Now you can correct the source file with the help of the error messages in the extra text area. Each error has a line number associated with it, indicating the line 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
```

We intentionally inserted lines 13 and 22 into the program text. They are highlighted in red and contain errors. Delete these lines to eliminate these errors. The error in line 36 is a error resulting from the errors in lines 13 and 22 and will be eliminated once these are removed.

Once you have deleted lines 13 and 22, save the source file (**F5 s**). After a short time the message `replace y/n?` appears on the status line. Since the file already exists on the disk, the editor asks if it should be replaced. Press **y** for yes so that the old file is replaced by the corrected document.

You now have a corrected source file and can compile it again. Exit the EDITOR and call the compiler. The inputs are the same as described above.

```
source file name:sample.c [RETURN]
link file name:1.l [RETURN]
```

If a link file with the same name exists on the disk, it will be overwritten. This time the COMPILER runs through the source text without error. The concluding message this time is:

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

The link file is now complete. You can proceed to the linking. To do this, exit the COMPILER with the **x** key and return to the LOADER menu.

5.3 Linking (Binding)

Since formatted input and output are used in the C program, you must bind the linkfile `stdio2.l` with the linkfile `1.l`. The functions for formatted input and output are defined in `stdio2.l`.

Before you start the binding, check to see that all of the files you want to bind are on the diskette. If the link file `stdio2.l` is not on the work diskette on which the file `1.l` is stored, load C-COPY and copy `stdio2.l` to the diskette in question.

Once the two link files `1.l` and `stdio2.l` are on one disk, load the C-LINKER (insert master disk). The linker responds with the title line and the link file input. Insert your work diskette and then enter the two link files `1.l` and `stdio2.l`. The input of `stdio2.l` is a default entry, so you need only press [RETURN].

If you want to bind `stdio1.l`, delete the last three characters of `stdio2.l` with [DEL] and enter `1.l`. As in the compiler, only three control keys are active for the filename input: [DEL] (delete), [CLR] (SHIFT+HOME), and [RETURN]. [DEL] deletes the character last entered, [CLR] erases the entire input field, and [RETURN] ends the input.

The order of the link files is not important. You can change the order as you like, the same C program always results.

```
link file stdio2.l [RETURN]
link file 1.l [RETURN]
link file [RETURN]
```

After the two link files, just press [RETURN]. This ends the link file input. The C-LINKER then requires the name of the program file in which the executable C program will be saved. Enter the name sample:

```
program file sample [RETURN]
```

Following this set the upper boundary of the C program storage. If you do not need any memory for your own applications, you can use the default boundary. Then the maximum C program memory (50K) is available.

```
memory top page $d0 [RETURN]
```

The next input concerns the linker option. Here you can accept the default letter l. This means that the C program can be started only from the LOADER (as a user program). The linker option b means that a C program will be created which can only be executed like a BASIC program. We will designate a C program as L-version when it is to be executed from the LOADER and B-version when it is to be started like a BASIC program. The L-version has the advantage that one need not leave the C system to run the C program. The B-version has the advantage that one need not load the C-LOADER in order to execute the program.

```
linker option
(l=loader/b=basic)  l  [RETURN]
```

After this input the linkler starts to bind the link files. Status messages are printed in grey, errors in red. The linker requires two passes through the link files. The start and end of each pass is displayed by the linker. In addition, the current link file is displayed in yellow type.

```
pass 1
link file stdio2.l
link file  1.l
end of pass 1

pass 2
link file stdio2.l
link file 1.l
end of pass 2
```

After an error-free binding, the concluding message is:

```
linking finished

press r to restart, x to quit
```

Exit the C-LINKER with the **x** key. With **r** you can restart the linker without having to load it again.

5.4 Executing

You find yourself back in the C-LOADER menu and the file **sample** contains the finished C program. Select the menu option **u** and entered the desired C program:

```
filename: sample [RETURN]
```

The LOADER loads the C program **sample** and starts it automatically.

As soon as the program is started it clears the screen and waits for you to press keys. It then displays the text `Character:` and behind this the `char` constant of the key pressed as they are noted in C. In the line below, the ASC value of the key is printed in decimal, hexadecimal, and octal. These numbers are represented in their C notation.

The hexadecimal numbers have a leading 0x (or 0X) and the octal numbers have a leading zero. The decimal numbers are written as usual, but may not have a leading zero or they will be interpreted as octal numbers.

Since the program is in an infinite loop, you can stop it only by creating an NMI [STOP+RESTORE]. NMI stans for non-maskable interrupt. The following message then appears:

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

You can end the program with **x**, let it continue running with **c**, or restart it with **r**. You can try out these three choices.

Note that the two options **r** and **c** have some peculiarities which are explained in the System Guide, Section 6 and 7.2. You can use them without concern here, however.

You will find a listing of this C program in the appendix.

6.0 Introduction to C

In the last chapter you were introduced to program development with the Super C language compiler. In this chapter you'll become better acquainted with the C language. We'll do this by means of sample programs which you can type in. All examples have been tested and will run without error.

As the title says, this is only intended to be an introduction. More technical and specific information can be found in the System Guide Section 8. This language description is intended as a reference guide.

Experience C programmers can skip this introduction and continue with the Systems Guide Section 1.

6.1 Overview

6.1.1 The first program

The first program which you should enter looks like this:

```
#include "stdio.c"

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

Compile this program and bind it to the link file `stdio2.l`. Execute the finished program and you should see:

```
Your first
program
```

This text remains on the screen until you press a key. Now you have seen what the program does.

The first line of the file contains an include command which allows the standard functions from the file `stdio2.l` to be used. If this line is in the program, you must always bind `stdio2.l` to the program.

The rest of the program is a function definition. As mentioned in the preface, a C program consists of functions. The function **main** represents the primary function. This function is called when starting a C program. The program comes to an end at the end of this function.

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

Within the function block are the statements which are executed when the function is called. In our case there are two statements in the block. In the first is a function named **printf** and in the second a function named **getchar**. Both functions are defined in **stdio2.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 character string. **printf** outputs this string to the screen.

You have probably noticed the peculiar characters **\n** within the string. This (\) is an escape character. Following the escape character (\) is a letter which together with the escape-code symbol represents one character. **\n** represents the carriage return character and causes printing to resume on the next line down.

Calling the function **getchar** does not require any arguments and causes the computer to wait for a keypress, meaning that the program does not return from **getchar** to the function **main** until a key is pressed. There, the end of the block is reached and the program is over.

Most statements are concluded 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 area used by a program. Data can be stored in this area. Such objects must first be created before they can be used. To do this you use declarations. A declaration which creates objects is called a definition. A type and storage class are assigned to the object through the definition. The most important thing is that the object receives a name through the definition. It can be referred to within the program with this name.

The type of object determines the length and the interpretation of its contents. The storage class determines the life time of an object. We will go into the various possibilities later.

```
#include "stdio.c"

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

This is a very simple C program which produces the following output:

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

The first lines of the program are familiar to you. `main() {..}` defines the primary function **main**. Within a block you can make declarations. These must always be at the start of the block:

```
double e,pi;
```

declares two objects of type ***double***. The two objects have the names **e** and **pi**. The type ***double*** means that floating-point numbers with double precision can be stored in this object, in this case with up to 16 digits of precision.

To define several objects of the same type, you can separate these with commas. A declaration is, like most instructions, concluded with a semicolon.

```
int a,b;
```

is a similar definition. Here two objects **a** and **b** are defined which have the type ***int***. Only whole numbers can be stored in objects of this type.

The next four program lines are statements in which values are assigned to the defined objects. The identifier of an object must always be on the left side of

the = character. Such an identifier is called as **L-value** (left value). On the right of the = character is the value which is to be stored in the object. All objects are assigned the right number value in the four program lines.

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

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

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

The characters up to the `%g` are printed. These are format instructions. They cause an additional 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 printing to move down to the next line.

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

getchar waits for a key before the program is ended.

You have become acquainted with objects. The objects in this example were all assigned a storage class. You will see later what kind of effect this has. We will only say that these objects exist only within the block in which they are defined.

The number values which appear 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 need to have constants on the right in an assignment. A name or a complex expression may be on the right side.

```
pi=e;
```

assigns `pi` the contents of `e`.

6.1.3 Loops

Up to now our example programs have been processed sequentially, meaning that the statements were always executed in order and the program is ended when the last statement is reached. This is not sufficient for solving all problems, so there are loop constructs which make it possible to repeat statements.

We will write a program which prints a table of Celsius and the equivalent Fahrenheit degrees.

```
#include "stdio.c"
/* Table from 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();
}
```

First, the compiler skips everything between `/*` and `*/`. This allows comments to be inserted in the program.

A set of objects are defined. **start** and **end** represent the first and last table numbers. **step** is the step size with which the Fahrenheit degrees are to be calculated. **celsius** represented the current Celsius value, **fahr** the current Fahrenheit value.

start, end, and **step** are assigned the requires values. **celsius** must be assigned the value of **start**. This gives **celsius** the value for the first conversion.

Now we encounter a **while** statement. After **while** must come a condition enclosed in parentheses. In this case we compare to see if **celsius** is less than or equal to **end**. If this condition is true, the loop body is executed. In this case the loop body is a block. Within the block are the statements which

are to be executed as long as the condition is true. In this case the loop is executed until **celsius** is larger than **end**. Then the end of the table is reached.

At the start of the loop the Fahrenheit value is calculated from the Celsius value. On the right side of the assignment is a complex expression which performs a calculation. The Celsius and Fahrenheit values are printed next to each other with the `printf` function.

The Celsius value is incremented by the step size at the end of the loop. In the program it is then tested 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 beyond the loop block.

Now to some of the program details.

```
celsius=lower;
```

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

The division 9.0/5.0 strikes your eye in the conversion formula. Here `double` constants are used. If one were to use `int` constants and write: 9/5, the result would be 1 because an integer division would be performed. If one wants the result of a division to be a `double` value, at least one of the operators must be of this type.

The format statement of `printf` is altered somewhat. `%f` means that the double number will be printed without exponent. Number inputs can be given between the `%` and the `f`. `%4.0f` means that the double number is printed with a string of at least four characters and zero places after the decimal. The decimal point and sign must be figured into the string width. Numbers with up to two digits can be printed with this format instruction. If the numbers are larger, the field will become larger than 4 characters and the formatting is destroyed. If the number is smaller, the string is padded with blanks until it is 4 characters wide. The instruction `%7.1f` means that a double number without exponent with an output width of at least seven characters and one place after the decimal will be printed.

6.1.4 Symbolic constants

The previous conversion program can easily be altered for different table values. Imagine a considerably more complex program. Changing all of the constants would be a lot of work and would be highly susceptible to error in case a constant were missed. To avoid this, modern programming languages have symbolic constants. A name is defined as a constant. Wherever this name occurs in the program it is replaced with the constant.

```
#include "stdio.c"
#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();
}
```

The program yields the same results but it looks quite different. The variables **start**, **end**, and **step** are gone. Constants have been defined for these. This is done with the **#define** directive. This directive must appear at the start of a line. After such a definition the given name can be used in place of the constant value.

The **while** statement is replaced by the **for** statement. After **for** are three expressions enclosed in parentheses. The first expression is the initialization of the loop, the second represents the loop condition, and the third the continuation of the loop. This continuation is always executed when the loop body is ended and before the condition is tested.

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

celsius+=STEP is similiar to celsius=celsius+STEP

This way celsius need be evaluated only once, meaning that the assignment is executed faster.

6.1.5 Arrays

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

```
#include "stdio.c"

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:%dtimes\n",i,numbers[i]);
        getchar();
}
```

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

In C, array elements are counted from zero on, meaning that a ten-element array has elements 0 through 9. We'll store the number of times the digit keys 0 to 9 are pressed.

An ***int*** object **i** and a ***char*** object **c** are also defined. The type ***char*** creates objects which can assume a character value from the character set.

The first statement is a **for** loop. In it the variable **i** ranges from 0 to 49. **i++** is the continuation of the loop. This expression is the equivalent of **i+=1**, incrementing **i** by one (i=i+1, in familiar BASIC).

In the loop body is a block with two statements. First the function **getchar** is called which waits for a key to be pressed. It not only waits, it returns the code of the key pressed as the result. This value is represented by calling the function. Here the value is assigned to the object **c**.

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

'0' and '9' are character constants whose value is the same as the code of the enclosed character. This code can vary from computer to computer. The C-64 has a modified ASCII character set, a table of which can be found in the appendix. The digits are always coded in ascending order in every character set, however.

The `if` condition checks to see if the character read in is less than or equal to the character '9' and if the code is greater than or equal to the character '0'. The two conditions are combined with a `&&` operator which makes the entire condition true only when the two individual conditions are satisfied.

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

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

is executed. **c-'0'** yields the digit as a value, for '0' the value 0 and for '9' the value 9. The array **numbers** is indexed with this value, meaning that the element is selected by the number **c-'0'**. This element is incremented by **++**. The corresponding array element value is incremented each time a digit key is pressed.

The `for` loop is executed 50 times, meaning that you must press 50 keys before the loop is terminated.

The next statement is again a `for` loop which prints a list. **i** ranges in value from 0 to 9 and the elements of the array are printed, for example:

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

The last `getchar` waits for a key to be pressed and displays the table on the screen until one is pressed. After you start the program, you must press keys, 0 - 9. After 50 keys the table appears, indicating how often you pressed each digit key.

6.1.6 Character arrays

You have already become acquainted with character strings. Up to now you have been working only with string constants, strings with a set sequence of characters. Variable strings do not exist as a type in C. Strings are set up as arrays of type ***char***. This means that the length of the string is limited by the length of the array, but not only by this. The end of the string is denoted by a zero in the array. This end of string indicator (null character) is created by the compiler for string constants.

```
#include  "stdio.c"

main()
{         char  name[41];

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

In this program a character array with 41 elements is defined. Since a character is required for the end of the string, you can use a maximum of 40 characters for storage. **gets** is a standard function for reading strings. The first argument is an array name in which the string is to be stored. The second argument gives the maximum number of characters to be read in. The third argument tells the function where to read the string from. **STDIO** is a value from the standard module which indicates that the standard input or output should be used, the keyboard or screen. The function causes a cursor to appear on the screen and allows you to enter a string. The input is done as in BASIC, that is, you must end it with [RETURN].

In the function **printf** is the control character **%s** which expects an array name as an argument. The string (***char*** array) is printed as text.

getchar again waits for a key so that you can view the output.

6.2 Expressions and declarations

6.2.1 Names

The names which are tied to objects through declarations may not be the same as any of the C keywords. These are reserved names which have a predetermined meaning in the program text, such as **int** or **char**.

A name must begin with a letter. After the initial letter may come digits. The underscore character _ counts as a letter.

One should chose variable names which suggest the purpose and contents of the variable and sufficiently unique so that a small typing error does not result in another, valid, variable name.

6.2.2 Types

There are a number of so-called simple data types in C. The simple types differ from the more complex data types such as arrays.

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

float is a floating-point type like **double** but with lower accuracy. On the C-64 this type has an accuracy of about 6 digits.

short int, also abbreviated to **short** can store only integers like **int**.

long int, also abbreviated **long**, can also store only integers. The three integer types are differentiated only by their value ranges, the size of the largest integer which can be represented. The value range of **short** is guaranteed to be less than or equal to that of **int** and the range of the **long** is guaranteed to be greater than or equal to that of **int**. In the Super C language compiler, **short** and **int** have the same size while **long** requires twice as much memory.

The memory required is also called SIZE.

All integral types (including ***char***) can be represented without sign by placing ***unsigned*** in front of the type name. The contents of such an object is always interpreted as positive. You can also write just ***unsigned*** for ***unsigned int***.

6.2.3 Constants

You have already used ***int*** and ***double*** constants. The compiler interprets a numerical constant as ***double*** if it finds a decimal point or an exponent in the number. An exponent is denoted 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, the number can no longer be stored in an object of type ***int*** (this can be different with other compilers). In this case the constant becomes type ***long***. If one wants to make an integer constant always `long`, one can place an `l` or `L` after it.

```
15L     21     0     40000
```

Integer constants which have a leading zero are evaluated as octal.

`077 = 63` (decimal)

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

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

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

We have also already used the character constants. They contain a character enclosed in single quotes:

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

The value of such a constant is the code of the character in the character set.

This value is converted to type **int** so that calculations can be performed on it. Combinations with the escape sequence can also be used as characters, of course. For example, '\n' represents the code for [RETURN].

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

'\0' is the escape sequence which is used as the end character for strings. Up to three digits may follow the \ which are then interpreted as an octal number. The value of this octal number is then the code of the character:

```
'\101'   =   'A'   =   65
```

Another constant is the string. The character string is placed in memory. At the end stands the end character:

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

A string constant can be used like an array name. Two string 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 (null character) at the end while 'a' is the value of the code of the letter a.

6.2.4 Storage classes

Up to now objects have been defined only within function blocks. If no storage class is given, the storage class **auto** is assumed. This storage class has the effect that the objects are available only within the block and are discarded when the block is left.

Objects defined within a block are called *local*. Local objects with the storage class **static** are also available only within that block. But they retain their value throughout the entire program and remain intact when 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 defined. These are defined outside of a function. If no storage class is given, the object is defined throughout the entire program.

The storage class **`static`** can also be used in global definitions. Regarding the program in which such objects are defined, this has no other effect than if the storage class were omitted. But when several separately compiled C programs are bound together, static global variables from one file cannot be accessed from another.

As a general rule, all objects in C must be declared before they can be used. Names which the compiler does not recognize from declarations are assumed to be global and of type **`int`** or as a function with result type **`int`**. As a result, such variables do not have to be declared explicitly.

6.2.5 Arithmetic Operators

Arithmetic operators are the basic types of calculations + - * /. The meaning of the operators should be clear in that they cause two numbers to be added, subtracted, multiplied, or divided, respectively. The type of the result is important in C. Standard type conversions are used when different types of operands are combined with these operators.

1. **`char`** or **`short`** operands are converted to **`int`**, **`float`**, or **`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 and the result become **`long`**.

4. If one of the operands is **`unsigned`**, the other operand and the result are made **`unsigned`**.

5. If both operators are of type **`int`**, the result is also **`int`**.

% also belongs to the arithmetic operators. The result of this is the remainder after the division. The standard conversions are also applied to this operator. Only integral 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.

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

These operators mean: less than, less than or equal, greater than, greater than or equal, not equal, and equal.

All simple types may be compared to each other. Type conversions are made according to the usual rules.

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

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

The ! operator can be used on all types. The operand is checked to see if it is zero. Then the result is 1 (true) else 0 (false).

Two conditions can be combined with `&&` or `||`. The operands may not be conditions, however. They are only compared with zero and then result in a value-true or false.

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

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

These two operators are evaluated from left to right. The second operand is not evaluated if the result can be determined from the first operand. If the first operand of && is 0, or if the first operand of || is not zero, then the result of the operator is already known.

6.2.7 Type conversions

Type conversions are automatically performed in some cases, such as the standard type conversions. The arguments of a function call also result in type conversions. ***char*** and ***short*** are converted to ***int*** and ***float*** to ***double***.

Type conversions can also be forced, however. This is done through a CAST. In parentheses is the type name of the result. This cast is placed before the value to be converted.

```
(char) pi
```

The value of the object **pi** is converted to type ***char***. The conversion is usually done such that the values with "smaller" types are converted to "larger" types without changing the value. A conversion in the other direction can change the value if it does not fit into the value range of the destination type.

6.2.8 Increment and decrement

C has increment and decrement commands ++ and -- to increment and decrement a value by one.

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

increments the object **i**. Both expressions have the same effect. In C however every expression has a value, including assignments and increment or decrement operations. In the first case the expression has the value of **i** before it is incremented, and in the second case the value after the increment. The -- (subtract 1) operator can be used like the ++ operator.

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

For both of these operators you must remember that they may have a side effect if they are in the same expression twice.

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

The above statement indexes the array numbers with the value **i**. But **i** is also incremented. The result is dependent on whether **i++** or **+i** is evaluated first. The order of the operations is not defined by the C language. It's possible that the compiler will reverse the order of the evaluation.

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

The side effect now yields a different result because the first **i** is not incremented first. So watch for the side effects! In general aviod them. Objects which are altered by side effects should be used only once in an expression.

The increment and decrement operators are faster than a corresponding assignment for integral types.

6.2.9 Bit operations

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

First are the operators which combine two values bit by bit. The usual type converions apply.

`&` bit-wise AND operation:
result bit 1 if both operator bits are 1, else 0

`|` bit-wise OR operation:
result bit 0 if both operand bits are 0, else 1

`^` bit-wise exclusive OR operation:
result bit 1 if both operand bits equal, else 0

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

```
1<<2
```

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

```
4>>2
```

The >> operator shifts the bit pattern right. The result here is 1. The operation corresponds to an integer division by 2. If the left operand is unsigned, zero-bits are shifted in on the left side. If the operand is not unsigned, however, the sign bit is shifted in so that `-4>>2` yields -1. This is the case in this compiler, but not necessarily in others which always shift in zero-bits. This means that the result can vary from machine to machine.

6.2.10 Assignments

The assignment with the = has already been used often and is a fundemental part of every program. The assignment causes the right operand to be assigned to the object denoted by the left operand. The left operand must denote an object--it must be an L-value. 1+2 not an L-value according to this.

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

```
#include "stdio.c"

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

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

Note that the assignment must be enclosed in parentheses or the compiler with first carry out the comparison and then assign its result to the variable **c**.

There are abbreviated forms for assignments in the event that the L-value is to combined with the value being assigned. `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 precendence of operators does not apply. The entire right operand will be combined. The L-value is evaluated only once in these short forms.

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

result in **i** being incremented only once.

If the value of an assignment is used, the side effects of increment and decrement must be noted.

6.2.11 Conditional evaluation

An interesting feature of C is conditional evaluation. It consists of three parts and two operators:

```
a ? b : c
```

The value of the expression **a** determines whether **b** will be evaluated or **c** will be evaluated. **b** is evaluated if **a** is not zero, else **c**. The value of the entire expression is the value of the expression thus evaluated.

```
1 ? 2 : 0
```

always yields the value 2.

```
i ? 2 : 0
```

returns the value 0 if **i** is zero, else 2.

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

If **x** is equal to zero, **j** is incremented. The value of **j** before the incrementation is the value of the expression. Otherwise **i** is incremented and the value of the expression is the value of **i** before the increment. Only one of the two objects is incremented.

If the result types of the two possible result expressions are different, the usual type conversions are made in order to get the same result type in both possible cases.

6.2.12 Precedence and order of operators

You know that multiplication and division operations are performed before addition and subtraction. But all of the other operators also have a set precedence which determines which operator is executed first. If several operators of the same precedence are in a row, the order of the operator determines whether they are evaluated from left to right or right to left.

In the following table, all of the operators on a row have the same precedence. The first row has the highest precendence. The last row has the lowest precedence.

Operators	Order
() [] . ->	from left
++ -- * & - ! ~ (CAST) size of	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 **+ * ^ | &** are determined by the compiler. You cannot prevent this even by using parentheses. For the operators, the order is not set (whether left-most or right-most operand is evaluated first). The **&&** and **||** operators are exceptions. Their operands are guaranteed to be evaluated from left to right.

6.2.13 Additional operators

Operators in the above list which you do not yet recognize will be discussed later in the User's Guide. You'll find an exact description in the Systems Guide.

One operator should be mentioned here. An expression can be divided into two parts which are both executed with the , operator. The value of the expression is the value of the right part. This is very useful when you have only one expression available:

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

The condition of the **if** statement is only **s+1**. **t** is first set to zero.

6.2.14 Program text

In principle a program can be entered format-free. The compiler 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);
}
```

makes no difference as far as the machine language program which results. However you can get a clean, easy-to-read program by following 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
- use color

In the C-EDITOR you have the ability to change the color of program sections. Use this, but don't overuse it because a rainbow-colored program is also hard to read. Use the supplied programs as examples.

6.3 Control structures

A C program is composed of one or more functions. Statements which the program executes are written in the function blocks. Such statements are executed sequentially, one after the other. A programming language offers control structures to alter the order of execution. Control structures are themselves statements. They contain more statements whose execution is not necessarily sequential. They may be repeated or skipped altogether (loops, branches).

This alteration is conditional, meaning that the order of execution is made based on the value of certain data.

6.3.1 Block

A block is a group of statements within braces {}. Local variables can be defined at the start of the block and can be used only in this block. A block is itself an statement, so that blocks can be nested.

A block serves to group statements together. In a loop, for instance, only one statement can be repeated. If a loop is to contain more than one statement, you can place them together in a block and execute them as the object of the loop.

The block is an exception to the other statements because it is not concluded with a semicolon but with a brace }.

6.3.2 The if statement

An **if** statement is executed only if a certain condition is satisfied.

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

Only when the condition c=='a' is true, will the statements following it be executed. This statement may also be a block.

The expression in the parentheses that follow **if** need not be a condition. The expression is evaluated to see if its value is zero (false) or non-zero (true).

The **if** statement can also be extended with an **else** section.

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

The statement behind **else** is always executed if the condition is false. Either the statement following the **if** or the statement following the **else** is executed.

You can program a branch to one of two different statements with **if...else**. You can also nest the **if-else**'s by placing an **if** statement in the **else** portion of the previous **if**.

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

Note that this is considered one statement, although it consists of several nested sub-statement. In order to increase the readability, you can eliminate the usual indentation:

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

if-else nesting has the disadvantage of requiring a good deal of writing. The conditions must be somewhat different in each **if** statement but must be reprogrammed. Furthermore all compilers place a limit on the number of nested statements. For this reason, most higher-level programming languages have other ways of handling multiple branches.

6.3.3 switch statement

A **switch** statement can branch to one of up to 43 other statement. The branch is made based on 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 the object **c**. Its value is the basis for the branch. After **switch**(..) is a block with various statements.

If a jump is to be made to a specific statement based on a specific result, a

If a jump is to be made to a specific statement based on a specific result, a **case** label must be placed in front of the statement:

```
case 'a' :
```

Behind **case** is a constant. If the result of the expression agrees with the constant, the statement is executed. Not only the following statement, but all following statements. In some cases this can be quite useful. To prevent it, however, you can place a **break** statement following. This causes a jump to the end of the block.

Note that you can place several **case** labels in a row. One peculiarity is the **default** label. If this is placed before a statement in the block, this statement is 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 statements are executed if a matching **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 are probably wondering about the last **break** statement in the block. This is in fact superfluous since the block is ended even without this statement. But we've placed this **break** statement here because the danger exists that it will be forgotten if a new case statement is inserted after it. The statements of this new **case** would then be executed along with those for label which is currently last.

6.3.4 while statement

The **while** statement is a loop. It repeats the following statement as long as the condition is true.

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

The condition is enclosed in parentheses as for the **if** statement. Then comes the statement which is repeated as long as the condition is true (the expression in the parentheses is not zero).

With loops make sure that the loop condition will become false, otherwise it will never end.

Inifinitely repeating loops, called infinite loops, can be programmed by omitting the condition.

```
while()
    statement
```

There is generally little application for such infinite loops.

6.3.5 for statement

The **for** statement is a special **while** statement. Not only is there a loop condition, but an intialization expression and a continuation expression are also specified.

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

This is also used as in BASIC or Pascal, meaning that a variable is set to a starting value. (initialization: **i=0**). The statements are preformed with the control variable ranging to the loop condition: **i<10**. The variable is incremented at the end (continuation: **i++**). But a **for** statement is not tied to one variable. You can use three expressions for the initialization, condition, and continuation in many more ways.

```
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. It waits for a key and stores the key's code. **i** will be set to 0. The condition checks to see if **i** is less then 10. The continuation again consists of two parts. The key pressed is printed and **i** is incremented. The statement of the loop is only a semicolon. This is the `empty` statement.

The entire statement waits for a key and prints the key code ten times.

You can see that much more complex **for** loops can be programmed in C than in other lanuages.

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

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

is the shortest **for** loop (an infinite loop).

6.3.6 do statement

The **do** statement forms a new type of loop. The condition in the **while** and **for** loops is always checked at the start of each pass through the loop, including the first time. It may occur that the loop is not executed at all.

The **do** statement does not cause the condition to be checked at the start of the loop. Such loops are used when the loop body is to be executed at least once.

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

The loop begins with the key word **do**. Next comes the loop statement, here a block with two statements. At the end of the loop is the loop condition following **while**. The statement prints "Input :" and then reads a string with a maximum of 20 characters into the array string, which naturally must have been previously declared with 21 elements. The loop condition checks to see 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 other characters were read. Then the loop and with it the input is repeated. Otherwise the loop is ended.

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

If the loop condition is omitted, you get an infinite loop.

6.3.7 break statement

This statement has already been discussed in context with **switch**. It causes a loop or a **switch** statement to be interrupted. The **break** statement functions only inside of loops or **switch** blocks. It always affects the last loop or **switch** statement if several are nested.

```
break;
```

The **break** statement causes the loop to be interrupted immediately and execution to continue after the entire loop. In a **switch** statement, the block is left.

```
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 keypress and assigns the key to the ith element of the character array ***string***. When the element read in is a [RETURN] character, a **break** statement is executed. This exits the loop.

A maximum of 20 character are read in, but no more that the first [RETURN] character. In the last statement, a '\0' is appended to the string.

You could combine the test for the [RETURN] key in the loop condition. This would tend to make the program harder to read, however.

6.3.8 continue statement

The **continue** statement is seldom used. It applys only to loops. Execution is directed to the end of the loop statement when **continue** is executed.

With a **while** or **do** loop, execution immediately continues with the loop condition and eventually the loop statement is repeated. In a **for** loop the continuation is executed and the loop condition is then checked.

The `continue` statement is mostly used to skip complex instructions within the loop.

```
for(i=0; i<20, i++)
{  c=getchar();
   if( c=='\n')
      continue;
   /* complex calculation */
}
```

The complex calculation indicated with a comment will be skipped if the key read was [RETURN].

6.3.9 goto statement and labels

Labels may be placed in front of any statement. They consist of a name and a colon:

```
name : statement
```

The name is defined through the label. You can jump to such a label with a `goto` statement, meaning that the execution of the program will continue behind the label.

```
goto name;
```

Such jump statements should be used sparingly, however. There are no cases in which a jump cannot be replaced by one of the available control structures. Jumps are strictly to be avoided in so-called structured programming in order to preserve the readability of the program. Only in rare cases can a jump be useful, such as when an error occurs within several nested loops. Only the innermost loop can be exited with `break`. `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, these 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**. But programs usually consist of more than one function. As you already know, arguments can be passed to functions which can then process them. A function can also return a resulting value. If a function does not return a result, it is called it a procedure. You can define procedures 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 is given in the definition, as here for **main**, the compiler adds the type ***int*** and assumes that the end result has this type. **main** does not return a result because the end of **main** also means the end of the program. The type ***void*** is often left off of **main** because of sheer laziness.

Functions can return any simple type. Take a look at the following function **power** which requires two arguments. The first argument should be a ***double*** value **x** and the second an ***int*** value **y**. The function **power** calculates the value of **x** to the power **y** for its result.

```
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 next. This is done in the usual manner, but without storage class. Note that no ; is allowed between the first and second lines while a semicolon must follow each parameter declaration.

The parameter declarations must be made in the order in which they occur in the parameter list.

What significance do these parameters have? The parameters are local variables exactly as those which are defined within the block. They receive the values of the arguments when the function is called. 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 statements of **power** step by step. When the function is called, the value of the arguments are placed in the parameters. If we want to calculate 5 to the power 2, we call the function **power**.

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

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

What happens in **power**? **x** has the value 5.0, **y** the value 2. If **y** were zero, the result of **power** would be 1. This is established through a **return** statement. It can be anywhere in the function. If it is executed, the function execution is stopped. An expression may appear behind **return** which calculates the result of the function. Here the result is 1. This time we do not have to write 1.0 because the result type is converted automatically.

In this case **y** was not equal to zero. A check is now made to see if **y** is negative, that is, if the exponent is negative. In this case the function is broken off. **1/power(x,-y)** calculates the same thing as the expression **power(x,y)**, but now the exponent is positive. The function power calls itself. This is called

recursion. x and -y are here arguments whose values wil be assigned to x and y in the new call. Like **auto** objects, the parameters are recreated each time the function is called so they cannot disturb each other.

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

```
5.0*power(5.0,1)
```

This is a correct result. The function **power** is recursive and is called with a lower but certainly positve result. In this call x has the value 5.0 and y the value 1. Note the result of the new call. It is:

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

power is called one more time. In this call however, power returns the value 1 so that the value fo the second call is:

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

The recursion ends. The result of the second call is incorporated into the first.

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

25.0 is now 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)** represents the result **5.0**.

Recursion is often easier to read than linear programs. This allows the function **power** to implement with few statements.

An expression need not always follow **return**. If there is no expression following it then the result is not defined. A procedure with type **void** should also be exited with an expressionless **return**. A function is also ended after the last statement in the function block is ended. You know this from **main**. You can also end a C program with a **return** statement in **main**.

6.4.2 Argument

There are some characteristics of pasing parameters in C which you should be aware of. The argument is always evaluated. If the type is ***char*** or ***short***, it is converted to ***int***, and ***float*** is converted to ***double***.

The types of the arguments must agree with the types of the parameters. This is not checked by the compiler. It is the responsibility of the programmer. If the types do not agree, you can force agreement with CAST. If you fail to check types, the function is almost certain to yield erroneous results.

Parameters which are declared with the type ***char***, ***short***, ***float*** are converted to ***int*** or ***double***.

Passing arguments is done only by value in C, which means that the value of the argument is assigned to the parameter. This parameter can then 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 even though the corresponding parameters may be changed.

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

Structures and unions, as well as arrays cannot be passed as arguments.

6.4.3 Global definitions

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

```
{   ...
    i=maximal+1;
    ...
}
```

maximal is a global object with the type ***int***. A storage class may not be given. A global object retains its value throughout the whole program and is not discarded. The name of such an object can be used throughout the program. If a local object is defined with the same name, the global object is "covered up" by the local object, meaning that the local object will always be accessed.

Global objects are used for storing data which multiple functions are to have access to. They can be used to reduce the number of parameters passed or to allow multiple results from a function. Data can be exchanged between functions with global objects.

Global objects, defined without storage class, can be used from other separately compiled programs when both link files are bound together. If you wish to prevent this use from outside, the storage class static can be placed in front of the definition. The object still remains valid throughout the entire 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 defined as ***static***, meaning they are usuable only within the file. To do this, ***static*** is placed in from of the function header. **main** may not be defined as ***static***.

6.4.4 Declarations

If you want to use global objects from another separately compiled program segment, the objects must be declared so that the compiler knows what type they are. No memory space is reserved by a declaration, however. The compiler codes the declaration in the compiled program so that the declared object is bound with its definition when the binding takes place.

Declarations can be made global or local, where the declaration applies to the whole program in both cases. They are designated with the storage class ***extern***.

The module with the standard functions represents nothing more than a separately compiled program. These functions must be declared before they can be used. This is done by the program line:

```
#include "stdio.c"
```

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

```
extern void printf();
extern int gets();
```

The functions **printf** and **gets** have been declared in this manner. You can see that no parameter list and no function block is given. Such declarations must always come before their first use.

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

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

Objects other than functions can be declared. With functions, declarations inside a source file are also important. If you have two functions which call each other, one function must logically come first. When the first function is compiled, the compiler does not yet know the type of the second. This must then be declared:

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

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

It would be better programming techniques to, in both functions, declare the other. In conclusion: If an object is accessed via 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 an object of type **int** and global storage class (not **static**).

If the name is used as a function, meaning that a function with an unknown name is called, the compiler assumes that the result type is **int**. 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 of type **int** also need not be declared. It is recommended that you do so to preserve the readability of your programs.

6.4.5 Local definitions

The local definitions are found within a block. If no storage class is given, the compiler defaults to **auto**. Objects of this storage class are generated at the start of the block and are deleted again at the end. If the block is called recursively, new objects are generated which have only the name in common with the old.

The static local objects are handled differently. These are defined with the storage class **static**. The object remains intact throughout the entire program but is accessible only within 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. The compiler tries to place these objects in special processor registers so that they can be accessed faster. Unfortunately, the CBM-64 has no such registers. If such definitions can no longer be placed in registers, because they are all used, or there are none, these definitions are handled like **auto**.

6.4.6 Initializations

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

Initializations are made by placing an = character behind the declarator (the name being declared) followed by the appropriate value.

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

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

Local ***auto*** objects are not automatically set to zero. ***auto*** objects can be initialized with entire expressions so that the above initialization is possible. **minimal** 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 at each new call of the block. Global and all static objects are initialized all together at the start of the program. Their value is therefore initialized only once.

Declarations of the storage class ***extern*** cannot be initialized.

Arrays can be initialized element by element. ***auto*** arrays cannot be initialized.

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

A list of constants 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 in with zero. There is a shorter form of the above intialization. Character arrays

can be initialized with strings, meaning that the elements of the string are placed in the character array in order.

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

performs the same initialization.

The specification of the dimension can also be omitted from the definition of an initialized array. The dimension is then automatically the number of intialized 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 through 12 contain the number days in the months 1 through 12. Element 0 is not used but must appear in the list.

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

The array **name** is here dimensioned with **five** elements. Remember the \0 character at the end of the string.

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 meaning that the elements of the array are arrays, whose elements are of type `int`. The initialization is done recursively--with a list of two elements. These are again lists, whose elements are initialized.

For multi-dimension 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. 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 the value 29.

If fewer elements than necessary are given in a sublist, the rest are again filled with zero. If you want to initialize all of the elements of the list, you can make the initialization in a 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 text as the constant. This concept is not limited to just constants. You can assign a name to any desired piece of text. Overall, wherever this name occurs in the source text, the text string is inserted instead. The replacement string must by separated from the the name by a space.

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

In the following program you can use the following expression:

```
NL;
```

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

Note that the # character must be at the beginning of the line. Directives that begin with # belong to the preprocessor. It does not belong directly to C but operates only on the source text. A preprocessor directive can occur in the middle of the program text, but requires its own line.

A name which you can define with **`#define`** is called a macro. Such macros can be used similarly to functions, meaning that they can be passed arguments.

```
#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. In the argument, the value of the argument is not used, rather the argument text. This is inserted in the replacement string wherever the parameter name is. If the macro is called in the following ways:

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

then the calls will be replaced by:

```
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 be typed directly behind the macro name or it will be recognized as a normal replacement string. The parameter names must be chosen so that the same name never occurs as an argument.

Macro names are often written in upper case so that they can be easily recognized as such. This is a matter of choice and can be done differently from programmer to programmer.

A defined macro name remains valid up to the end of the source text. The name can no longer be declared because it will also be replaced in the declaration. If a name is already declared before a macro definition with the same name, the name will always be interpreted as a macro.

```
#undef  PRINT
```

This is a preprocessor directive which erases a defined macro again. From this line up to the end of the file, the macro defined with **PRINT** is no longer available.

6.4.8 File chaining

Multiple source files can 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 directive. This directive has already been used to insert the file **stdio.c**. All of the functions in the standard module are declared in this file and various macros are defined.

Additional **#include** directives may appear with the first **#include** file. Such chained files generate only one file for the compiler because the preprocessor affects only the source text. Chained files are not to be confused with 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, as a rule, points to another object. You can access such an object via this address without using the name of the object.

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

6.5.1 Pointers

You declare a pointer by placing an asterisk (*) before 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 is placed before an L-value and returns the address of the object in question as its result. **&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 L-value of the object in question. ***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 have written **a**.

Additional consequences:

***&a** corresponds to **a**
(the * and & operators are evaluated from right to left, ***(&a)**)

&*p corresponds to **p**
(**&*p** is not an L-value but only and address)

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 L-value.

The & operator requires an L-value and returns the constant address of the object.

6.5.2 Address arithmetic

Computations can be performed with address or 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 **p** is assigned the address of array element 4. The pointer is then incremented by one. This does not increment the address by 1 but by 2. The pointer arithmetic operates such that the summand 1 is multiplied by the SIZE of the object concerned, in this case two, before the addition is performed. This addition has the result that **p+1** yields an address which points to exactly one object beyond the current position, or to **array[5]** in this case. ***p=5** has the same effect as **array[5]=5**. The addition of 1 is independent of the type of the pointer. If the elements were of type ***double***, 8 would have been added to the address because the SIZE of the type ***double*** is 8. This addition makes sense only when the new address still points within the same array since it is only guaranteed that the objects follow each other precisely.

Instead of **p=p+1** we could also have written **p+=1** or even **p++** or **++p**. Furthermore, the last two lines could be rewritten as follows:

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

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

You can use subtraction exactly as the 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 address point within the same array.

```
&array[5]-&array[1]
```

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

The indexing of array elements is derived from this pointer arithmetic. 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 L-value. You can add the number of the desired element to it, however, and receive its address based on the pointer arithmetic. The expression becomes an expression as a result of the * operator so that ***(a+(b))** has exactly the expected effect of **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)
```

Additional consequences result from the indexing through []. Since these brackets are converted to addition according to the above scheme, their use is not limited to arrays.

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

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

6.5.3 Pointer 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. If you want to pass an array to a function, you pass only its address.

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

This fact has already been used in a previous example program. The function **gets** receives as argument the address of the array **name**. **name** alone represents this address. The corresponding parameter declaration of **gets** would have to look like this:

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

The specification of the dimension is unimportant and can be omitted here. In actuality **string** does not represent an array because the object to which the address of the array will be assigned is a pointer which can be used like an array within the function block. The parameter declaration could just as easily be:

```
char  *string;
```

The transmission of arrays is therefore done via the address (call by reference). This procedure has the result that the array can be altered within the function being called. This stands in contrast to the transmission of other types where only the value is passed.

If you want to be able to change other objects through the call, 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;
}
```

The call of the function **swap** passes the address of **a** and **b** so that **swap** can exchange the contents of the the objects.

6.5.4 Declarations, more complex

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 up to now look like:

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

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

At first the declarators appear to be chosen somewhat randomly. But it applies for all declarators that when you use them as in the declaration, a result is returned, the type of which is the type of the declaration. Proceeding from this rule, complex declarations can be constructed:

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

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

Note the three declarators. First the * operator is applied to **alpha** which has the result that **alpha** is a pointer. Then the index parentheses 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 parentheses 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 yields an `int` value as its result.

You see that arbitarily complex declarators can be declared. These are, however, rarely 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 from the array with an index and then they can be used as pointers.

It is interesting to note the use of pointer arrays of type ***char***. It has been mentioned that string constants can be used like array names. Now you can say that a string constant is a constant address to the given character string and can therefore be used like an address. You can, for example, initialize a pointer array of type ***char*** with strings.

```
#include "stdio.c"

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

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

In this program, the array **strings** is initialized with the names of the months. In reality the compiler places the character strings somewhere in the program and initializes the address to the given string. The element 0 is assigned **NIL**. This is the address to "nothing." You must be careful with **NIL**. In no case may an object by accessed via **NIL**. **NIL** serves only to indicate that such an access is not allowed.

The program passes the address of the ith character string to the function **puts** (put string) which then prints this on the screen (**STDIO**). **getchar** waits for a key so that the output is not immediately cleared again.

Keep this initialization separate from the initialization of character arrays through string constants. Here only the address is initialized. With character arrays the content of the character string is 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-dimension array and in the second a pointer array. The beginner will probably find it difficult to keep both constructions apart. They 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 5 objects. These are all pointers, however. **beta** generates no individual ***int*** objects. An element of **beta** can only point to an ***int*** object.

The advantage of pointer arrays is that a pointer can point to a limitlessly long sub-vector and this can be accessed like a two-dimensional array. The various pointers can point to arrays or various lengths while the number of elements in a two-dimensional array is set. The disadvantage is that the sub-vectors must be declared in addition and the whole construction requires more memory space because the pointer objects must be included.

We have seen 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 interpreted as subvectors.

```
string[12][0];
```

accesses the letter **D** of the month **December**. The second dimension is variable and depends on the initialization.

6.6 Structures and unions (struct/union)

Structures are found 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 among these subobjects as you can select an array element. The difference from an array is that an array contains subobjects of the same type.

6.6.1 Declarations of structures

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

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

This whole structure 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. Such subobjects are called components. The component declarations are called the **struct** specifier.

You have several several in order to declare such an object. In the previous example no object was declared. A **struct** name was defined. This name is assigned the specifier so that the specifier can be omitted in further declarations:

```
struct date birthday
```

birthday is an object that consists of the above components.

You could have defined this object along with the definition of **date**:

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

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

```
y;
 ... } 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 consisting of 50 structures of type **date** are defined.

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

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

If fewer elements are given, the structure is padded with zeros.

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

This list can be nested again. The sublists can always be omitted when all of the subobjects are initialized. The compiler then assigns the elements of the list to the array elements and components in order.

6.6.2 Access to components

Components are accessed in C with the . operator.

```
birthday.day
```

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

```
birthday.monthname
```

is naturally not a L-value but an address to a character array with a maximum of 4 characters.

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

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

*p must be parenthesized because the . operator has precedence. There is a separate operator for this construction , ->, since it occurs quite frequently.

```
p->year
```

has the same effect.

If you have an array of structures, the 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 to functions as arguments, but the addresses of structures can. The & operator can be used on structures for this purpose. A function cannot return a structure as a result, but by the same token 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 list of months is used to determine 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.

The complex condition of 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 above condition is true it returns 1, otherwise 0. It returns as the result exactly what is needed in the calculation.

The function monthdays can be called as follows:

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

6.6.4 Recursive structures

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

You cannot use the same structure currently being defined as a substructure. It is permissable to declare pointers to this 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 trees are used to keep names in alphabetical order, for instance.

6.6.5 Unions

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

Unions are used where you want to store objects of varying types and you require an object which is always the same size. If we want to define an object which can store a C constant, for example:

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

we define a union. It can store either an ***int***, a ***long***, a ***double*** constant or an address to ***char***.

The union is accessed just like a structure. The component 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 therefore easily portable. You must ensure that the union is read as it was stored. If the components are changed at the access, the result is no longer defined.

Unions can also be declared in structures and vice versa. On the C-64, however, the declaration of a specifier within a specifier is not possible. Specifiers of substructures or subvariants must be defined outside with their own names.

A union can also be denoted as a structure whose components are all stored at the start of the object or posses the relative address 0. Unions, like structures, cannot be passed to functions as arguments, and cannot be the result of a function. Unions cannot be initialized.

6.6.6 Type definitions

You can also define new types in C. These are not entirely new, but are combinations of the available types.

One gives the "storage class" ***typedef*** for such a definition. This tells the compiler that an object is not being declared, but a type. A name is declared which can be then be used as a type name. It represents the type with which it was defined.

```
typedef int length;
```

length can now be used 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 is in reality a structure and must be used as such.

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

6.7 Programming environment

6.7.1 Files

You know how files are opened from BASIC. 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 included, but here it is an empty string. You no doubt noticed that the logical file number is missing. This is not necessary in C. A similar instrument is the file descriptor. The file descriptor is used exactly like the file number in order to access the file. A file descriptor is an object that 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 else the file can neither be used not closed again.

The file is closed with:

```
close (fchannel);
```

The type **file** is defined with typedef in the file **stdio.c** and is not otherwise usuable in C.

You have already become acquainted with the functions **puts** and **gets**:

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

sends a format command to the drive which then formats the disk.

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

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

6.7.2 EOI

The EOI flag, End Of Information, is used to recognize the end of the file. This flag is realized through a macro which has the value 64 if EOI is encountered, else 0 if not. In order to get the error message from the disk drive, for instance, one reads characters until an EOI occurs.

```
#include "stdio.c"

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

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

The EOI value must be stored in a temporary value because it can be changed by other input and output functions such as **putchar**.

6.7.3 STDIO

STDIO is a special file descriptor. It can be used wherever a file descriptor is required as an argument. No open call is necessary for STDIO. Output is directed to the screen with STDIO, inputs are read from the keyboard:

```
#include "stdio.c"

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

    gets(command,38,STDIO);
    puts(command,fchannel);
    close(fchannel);
};
```

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

6.7.4 Additional functions

The standard module contains a number of other functions whose complete descriptions can be found in the system section.

Important and useful are the functions **printf** and **scanf** with which formatted output and input are possible. These two functions are relatively

large. They are therefore contained only in the module **stdio2.l**. Otherwise the modules **stdio1.l** and **stdio2.l** contain the same functions.

The declaration file **stdio.c** can be used for both modules. The declaration of `printf` or `scanf` when using **stdio1.l** does not create an error, as long as the functions are not actually called.

6.7.5 Error handling

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

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

You can interrupt the program with the **x** key, restart it with **r**, and continute execution wth **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 off and on as desired with the procedures **erroff()** and **erron()**. **erron()** is the initial condition.

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 often useful to exit an infinite loop. The message:

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

appears. You have the same options here as for error messages. The same caution is recommended. The NMI interrupt can also occur during an input or output operation. It can also be that undesired side effects may occur later in the program if the execution is continued 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

1.0 C-LOADER

You can load the C-LOADER in BASIC and start it with RUN.

```
LOAD"C-LOADER",8
....
RUN
```

The LOADER menu will then appear:

```
X: to basic

a: c-copy          c: c-compiler
b: c-editor        d: c-linker

u: user file:
```

The individual functions of the LOADER are designated by letters. The menu item just selected is displayed in reverse text.

You can move the menu selection forwards and backwards with the cursor up and down keys. The menu item **X: back to basic** cannot be selected with these keys. It can be chosen only by pressing [SHIFT]+x. A selected menu item is executed by pressing the [RETURN] key.

You can also select and directly execute the menu functions by pressing the letter key corresponding to the individual item.

The meaning of the individual menu options:

X: ([SHIFT]+x) You exit the editor and return to BASIC

u: Execute a compiled and bound C program. Enter the filename of the user file at the cursor.

```
u: user file:_.
```

During this input you can use the following control keys: [DEL] (delete), erases the last character entered, [CLR] ([SHIFT+HOME]),erases the entire input field, and [RETURN], ends the input.

With this command you can enter files which the LOADER then loads and starts. You must enter the filename without quotation marks and without spaces in front of the name. If you do not enter a file, you will be returned to the menu selection.

The files which you can enter are compiled C programs, note that you can start only those programs which were put together by the linker as Loader-versions. Loader-version means that this C program is intended to be started from this menu. If you load a C program which was compiled as a B-version, intended to be started from BASIC, you will see the message `B-version!` on the screen when you try to load it. Press a key after this message and you find yourself back in the menu selection of the LOADER.

`a:` c-copy is loaded

`b:` c-editor is loaded

`c:` c-compiler is loaded

`d:` c-linker is loaded

The loading can be interrupted with the [STOP] key. You are then returned to the menu and can select another function.

Before a given program is to be loaded, you should naturally insert the proper diskette. If the LOADER does not find the program or encounters an error while loading it, the loading is stopped and you can again choose a function from the menu. Don't worry if the disk error light continues to blink.

2.0 C-COPY

The commands of C-COPY consist of command characters and parameters. A command character need not stand at the start of a line. The parameters of the individual commands must follow the command character immediately. If you have entered a command which consists of command characters and parameters, press the [RETURN] key and the command will be executed.

2.1 Command characters

@ displays the error message from the disk drive. This command needs no additional parameters. The characters behind the @ character are arbitrary.

. *text* send the *text* immediately following the period to the disk drive (for example, .r:c prog=test).

/ *text* displays the directory of the diskette. You can enter a specifier directly after the / character (such as /0, /0:test*, /*=seq,test*=usr). More about the syntax of the specifier can be found in the disk drive manual. You can slow down the screen scrolling with the [CTRL] key. You can halt the display with [STOP].

< *filename* copy the file *filename* from diskette into working memory. The file can be of type SEQ, USR, or PRG, but not REL. You can interrupt the loading process with [STOP].

> *filename* save the file with the name *filename* to diskette. You can interrupt the saving with [STOP], but NOTE: The file will then be closed correctly, but will contain only part of the original file.

*devnum* set the current disk drive device number *devnum*. You can then copy files from one device to another, for instance, by changing the device number between loading and saving. The device number 8 is the default.

x This permits you to exit C-COPY and you returns to the LOADER.

2.2 Messages

i/o error	an input/output error was encountered. The operation is halted.
break	loading or saving was interrupted with [STOP].
missing command	no characters were found behind the . command .
no device number	the digit 8 or 9 was not found behind the # character.
no filename	the filename was not found after < (load).
no file previously loaded	attempt to save a file without first loading one.
file loaded	the file was loaded correctly.
file saved	the file was saved correctly.
File too large	the file is larger than the working memory (48K).

3.0 C-EDITOR

The editor has two text areas, a **file** area in which you edit the C source text, and an **extra text** area in which text is stored temporarily. 43K of text storage is available for both text areas.

In contrast to BASIC, the cursor in the C-EDITOR does not blink and the repeat function works with all keys. The key assignment has been slightly changed from that of BASIC so that C-specific characters can be entered. The editor can represent, in addition to the C character set, the CBM lower case character set. The character sets can be switched with [SHIFT+CBM].

The key layout below applies for the whole C system, even when a C program is running.

Keys	C	BASIC	ASC code	old ASC code
[SHIFT]+[0]	_	0	95	48
[SHIFT]+[+]	{	+	219	219
[SHIFT]+[-]	}	|	221	221
[SHIFT]+[=]	|	=	220	61
[SHIFT]+[↑]	~	π	222	222
[↑]	^	↑	94	94
[£]	\	£	92	92
[←]	TAB	←	8	95
[SHIFT]+[←]	SET	←	9	95

The C-EDITOR displays the cursor position on the first line (status line) on the right side of the screen. The first number is the column, the second the line number at which the cursor is found. Messages and errors are displayed in the status line. In addition, command inputs are confirmed on the status line.

The second screen line contains either the filename in **file** area or the message **extra text** when using the **extra text** area. The third line displays the tabs, * indicates that a tab is set.

The remaining lines, 4 through 25, contain the current text area. A file consists of individual lines which have a set maximum length (40-80 characters). If the line length is greater than 40 characters, the rest of the characters outside the screen are displayed by shifting the screen left or right. Each line may have 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 file cannot be written on. If you try to move the cursor beyond this line or write on it, the editor displays the message **`last line`**.

The text which you enter is stored in memory immediately. You do not have to press the [RETURN] key.

If the text becomes too long as the result of an operation, it no longer fits in the available memory, the operation will not be executed and the editor issues an overflow message.

Control characters and commands are available to you for editing the text. Control characters are available with only a keypress during the text input. Commands are more complex editor functions, and require additional parameters. Commands are signaled with the command key **F5**. After this key, you select a command by pressing the key corresponding to the command desired. Various parameter inputs can follow the command. There are five types of inputs of parameters. These five input types are described in the following sections.

3.1 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

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].

3.2 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.

3.2.1 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.

3.2.2 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.

3.2.3 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.

3.2.4 Block input

For this input the first screen line contains the message `marking out range`. In this input type you can detemine 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.

3.2.5 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.

3.3 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).

```
hunt:<search string>
```

r replace

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

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

e erase

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

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

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

t transfer

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

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

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

m move

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

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

c color

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

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

l load

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

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

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

s save

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

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

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

f filename

Change the name of the document in the file area. This command can be used in 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.

x exit

Exit from the editor and return to the LOADER.

```
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.

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.

3.4 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.

last line: You tried to write on the last line or you tried to move the cursor past it.

i/o error: A input/output error occurred or the device being accessed is not turned on.

4.0 C-COMPILER

4.1 Operation

After the compiler is loaded, the title is displayed and this message appears:

```
source file name:Δ
```

Enter the filename of the source file to be compiled. Do not use quotation marks or leading spaces. End the filename with [RETURN]. Next the following appears:

```
link file name:Δ
```

Enter the name of the link file that is to be created followed by [RETURN]. A link file already on the disk having the same name is overwritten. Then the compiler begins the compilation.

Only three control keys are valid here. The last character entered may be erased with [DEL]. The entire input field is erased with [CLR]. [RETURN] ends the input.

It is recommended that you name your source files with a suffix of **.c**. In this case the compiler will supply the same name with a suffix of **.l** at the end of the link file name, so that you only need press [RETURN].

To stop the compilation for some reason, press [STOP+RESTORE]. The compiler then responds ?NMI INTERRUPT. This interruption is handled by the compiler like an error.

The files are erased and the message below is displayed.

```
?nmi interrupt in xxx
compiling finished
linkfile not available
press x to quit, r to restart
```

The compiler can be restarted with **r**.

The compiler prints the names of any function definitions in yellow. The names of the included files appear in grey. A grey **#** character indicates that a file has been read.

Errors encountered during the compilation are printed in red. The error messages are also written to an error file with the name **error-c**. This file can be read by the C-EDITOR and contains other status text in addition to the error messages. It should not be difficult to assign the errors to the various files.

The compiler prints a version of the following message at the conclusion of the compilation:

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

Whether the link file is available or not depends on whether any errors were encountered during compilation. Pressing **x** returns you to the menu of C-LOADER. With **r** you can restart the compiler and it is ready to compile another source file.

4.2 Compiler error messages

The compiler outputs error messages to both the screen and the error file **error-c**. This file can be read by the editor to help you find the causes of the errors.

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

Following the error message is the number of the line in the source file in which the error occurred. Often on error causes subsequent compilation errors. These will disappears once the original error is corrected.

Some errors cause the compilation to stop. These are called "fatal" errors. Such errors are not written to the error file because writing to the error file could lead to more errors (such as with bus errors).

?FLOPPY ERROR (FATAL ERROR)

FLOPPY ERROR stands for an error message of the disk. In any case the compilation will be stopped. The cause of the error is determinable from the error text.

?DEVICE NOT PRESENT (FATAL ERROR)
Output device is not accessable.

?ILLEGAL COMMAND
- no preprocessor recognized
- no string follows `#include`

?RUN END OF LINE
Terminating " character is missing from a string.

?STRING TOO LONG
- string constant contains more than 254 characters
- macro definition longer than 254 characters
- argument of a macro call longer than 254 characters

?TOO MANY CONCATS
More than six file chainings with `#include`

?EXPECTING IDENTIFIER
- no name follows `#ifdef`, `#ifndef`, `#define`
- parameter of a macro definition is not a name
- a `struct`, `union`, `enum` name is expected for a struct/union component whose type is `struct`, `union`, or `enum`
- Names expected in `enum` specifier
- a struct/union name expected in 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 error
- the expression evaluation is interrupted through `#if`. `#if` is possible only outside of an expression or a constant.

?RUN END OF FILE
- end of program although pre-processor command `#if` not closed with `#endif` yet
- end of program but the arguments of a macro call are still expected
- declaration not closed
- block structure still open

?MACRO EXISTS
The macro to be defined already exists

?STACK OVERFLOW (FATAL ERROR)
- no space for a new macro with `#define`
- no space for the entry of a new declaration
- recursion by intialization too large (about 40)
- Too many string constants within an initializer
- Constant buffer exceeded, cannot be emptied

?MACRO NOT DEFINED
`#undef` was used on a non-existent macro

?ILLEGAL NOTATION
- improper `char` constant (not exactly one character)
- more than one decimal point or exponent in a `double` number
- exponent is incomplete

?ILLEGAL MACRO CALL
- the call needs parameters
- the call has too many or too few arguments

?ILLEGAL OPERATOR
A character was recognized which cannot be evaluated as an operator

?OVERFLOW ERROR
- an oversized `double` constant was read
- an `enum` constant is too large
- constant evaluation caused overflow

?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 contains no 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 was recognized as a declaration only because of the previous error, it is skipped. The compiler reponds 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 the declarator
- no type given for a struct/union component
- no } as the end of a `enum` specifier
- neither type nor storage class of a parameter definition or local declaration

?DECLARATION SEMANTIC ERROR

- `auto` or `register` as the storage class of 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 unions
- an attempt was made to define a component as a structure or union whose type agrees with the type of the structure or union being defined (recursion)
- declaration of a local function

?INDENTIFIER ALREADY DEFINED

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

?EXCEPTION ERROR

?IDENTIFIER NOT DEFINED

- given `struct, union, enum` name is not defined, although no specifier is given
- the `struct, union` name in a component (in struct/union sub-declaration) is not defined

?DECLARATION INCOMPATIBLE

name is defined, but is not a `struct, union` name

?EXPECTING SPECIFIER

neither a specifier nor a specifier class was given

?TYPE CONFLICT

- no `int` convertible type or address in `enum`
- `char` cannot be initialized with addresses
- `auto` addresses are not allowed as constants
- `switch` expression is not of integral type
- general illegal type with operator: for instance `double` with %

?INITIALIZER TOO LONG

- more elements or components than possible were intialized in an array or structure initialization
- a string is longer than the `char` array

?PARAMETER MISMATCH

- the declarations of the parameters not do agree with the order in the parameter list
- occurs as a secondary error if a parameter declaration is erroneous

?TOO MANY STATEMENTS NESTED

more than 16 instructions (blocks are also instructions) were nested

?TOO MANY BLOCKS NESTED

More than 8 blocks were nested (functions blocks count)

?STATEMENT SYNTAX ERROR

- occurs as a secondary error when errors are found in instructions concluding blocks
- no name is found behind goto
- no ; behind break or continue
- conditions are not parenthesized (for if, for, while, switch)
- expressions in for are not separated by ; (two ; needed)

- a block must follow switch
- no `while(...);` follows do

?LABEL NOT DEFINED
the name behind goto is not defined as a label

?EXPRESSION SYNTAX ERROR
- general syntax error: improper parenthesization, multiple constants or names in a row (when ; is forgotten)
- use of a keyword in an expression (only SIZEOF allowed)
- `CAST` not parenthesized
- improper parenthesization withina `CAST` declarator
- a name was used which denoted no object
- 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 inside a switch block

?CASE WITHOUT SWITCH
a `case` or default label was defined outside of a `switch` block

?EXPRESSION SEMANTIC ERROR
- only one `CAST` is allowed per simple expression
- unary operator * may not be applied to functions

?EXPRESSION OVERFLOW
- expression consists of more than 63 elements
- expression consists of more than 28 names or constants
- `CAST` storage exceeded

?NO CONSTANT ERROR
the expression does not return a constant although this is required

?EXPECTING L-VALUE
- the first operand of an assignment is not an L-value
- the first operand in front of . is not an L-value
- an L-value is expected for the unary operator &
- ++, -- may only be used on L-values

?EXPECTING ADDRESS
an address is expected for the unary operator *

?NMI INTERRUPT
creates an interruption; all files are closed

4.3 Looking for errors

When errors occur during compilation, the error file generated can be loaded and examined to aid in finding the causes. Load the file **error-c** like a source text file from the C-EDITOR and copy it to the extra text with the **transfer** (t) command. By using the status messages between the error messages, you can find the text file in which the errors occur when several files are nested with **#include**.

Load the appropriate file into the main text area of the editor. Following the error message is the line number in the source file in which the error occurred. You can jump directly to this line with the **goto** (g) command.

In most cases it is possible to localize the error with the help of the error list. The most command error is the omission of semicolons. This error is usualy indicated by an EXPRESSION or DECLARATION SYNTAX ERROR. Sometimes, however, other error messages can appear if the compiler recognizes another structure as a result of the missing semicolon.

If an error message is absolutely incomprehensible to you, the error may be a secondary error as the result of some previous error. Errors in declarations and complex instructions like *while, for,* etc., result in a number of secondary errors. In all of these cases you should correct all of the obvious errors and recompile the text.

Once the source is compiled without error, it must be bound. Additional errors can occur in the C-LINKER. One common error is:

```
?no reference to NAME
```

NAME is an identifier to which no object is assigned. This error can have several causes:

1. The name is declared `extern`, but never defined
2. The name is not declared at all and is declared by the compiler as `int` object or as a function of `int`. The name is not defined, however.

The second case occurs whenever you misspell a name. The compiler does not recognize the name (because of misspelling) and declares it as an `int` object or as a function of `int`. It is the linker that first discovers that this object is never defined. In this case you must start from the beginning. You can search for the incorrect name with the editor (this is given in the linker error message) and change it.

If your program is bound but it does not yield the expected results, there is a logical error in it somewhere. Note that almost no checks of the program being compiled are built into the C compiler. C is very different from Pascal in this regard.

The following are not checked:

1. the agreement of the types of arguments and parameters
2. the boundaries of arrays (including strings)
3. if a structure or variant stands to the left of a . operator. Any L-value is accepted and handled like a structure or variant.
4. the component names behind . and -> are not bound to the structures or variants in which they were defined.

You must check each of these cases yourself. Above all, improper types in the current parameters lead to strange results.

Checking the boundaries of arrays is very important. This can lead to a program crash if not checked. Note that when an array is defined with `int array[10]`, for example, element 10 is already outside of the array. This array is comprised of the elements 0 through 9. With arrays of char (strings), the array must be large enough to include the terminating zero. A string variable which is to accept strings of up to 10 characters must be dimensioned with 11 elements.

Pointers which point to "nothing," which have the value NIL, point to memory location 0. Be careful not to make any write accesses via pointers with this value. This will cause memory locations 0 and 1, the processor port, to be overwritten. In most cases this causes the computer to crash.

5.0 C-LINKER

The C-LINKER converts compiled source files, link files, into an executable machine language program. You can bind together up to seven separately compiled source files into one program file which then contains the executable C program.

The order of the link files is irrelevant. As long as you enter the appropriate link files, the same C program results.

Note, however that the C-LINKER makes no declaration checks. You can, for example, define something 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 binds references to both objects without recognizing their different declarations.

All link files which you wish to bind must be on the same disk. You can use C-COPY to copy files.

5.1 Operation

All inputs to the LINKER are limited to a certain number of characters. Input is ended with [RETURN]. You can erase the last character entered with [DEL]. The key [CLR] ([SHIFT+HOME]) erases the entire input field.

```
link-file  stdio2.l_
```

First enter the individual link files. The name of the standard library `stdio2.l` is already printed as a default since this file is usually bound.

You can enter a maximum of seven files. The LINKER automatically ends this after the seventh file is entered. If you want to bind fewer than seven files, you can end this by entering nothing following the link-file by simply pressing [RETURN]. You must have at least one link file, however, or the LINKER will ignore the ending of the link file input.

```
program file _
```

The program file input designates the file name of your executable C program will be stored. An already-existing file with this name will be overwritten.

```
memory top page $d0_
```

The default value is given for the input of the top memory page. You will use this value in most of your C programs. This input is done in hexadecimal and requires exactly two digits. Numbers from $20 to $d0 are the legal input values. The question is repeated if you make an incorrect response.

The top memory page indicates the first page (one page = 256 bytes) which is no longer available for C program storage. With the default value of $d0 the upper boundry of the C program storage is $d000. This means that your C program storage extends from $0801 to $cfff (50K bytes).

By specifying a top memory page, you have the option of protecting memory from other programs. You can then use the memory from the top page to $cfff for your own applications, such as graphics storage.

```
linker option:
(l=loader/b=basic)  l_
```

A default value is also given for the linker option which you can accept by pressing [RETURN]. The option **l** means that your C program cab be started only from the C-LOADER. The C program is then designated as an user-version. With option **b** your C program can be started like a BASIC a program nd is designated as a B-version.

Now the LINKER starts to bind the link files. The LINKER can be stopped at any time by pressing [STOP+RESTORE] (=NMI). You then see the message:

```
? nmi interrupt in xxxx
linkfile not available
press x to quit, r to restart
```

With the **x** key you exit the LINKER and return to the LOADER. With **r** you restart the LINKER. If you have entered an incorrect parameter, you can issue an NMI [STOP+RESTORE] and chose **r** for restart. You can then repeat the parameter inputs.

5.2 Error messages

The following errors and messages can occur during the binding:

pass 1/2
The linker is starting the first/second pass through the link file.

end of pass 1/2
The first/second pass is done.

link file .....
The LINKER is currently reading out of the given link file.

linking finished
The binding ended without error.

linking aborted
The binding cannot be continued because of an error.

incorrect linking
Errors occurred during the linking.

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

no reference to .....
The given name is not defined in any link file.

no clear reference to .....
The given name is defined in at least two link files.

no extern declaration of .....
The given name is not declared as extern. The error does not prevent the correct creation of a C program.

no linkfile format
The format of the file read is not correct. The file in question is probably not a link file.

i/o error
Error on the bus (time out, device not present, etc.).

overflow in symbol table

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

statics out of range

The C program does not fit in the available C program memory because the static variable area exceeds the top of memory (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.

exception error ?

This error does not occur in normal circumstances.

6.0 C programs

C programs can be started from the menu with the USER command or like a BASIC program (the B-version). You choose between the two versions when you use the LINKER. C programs run in the same environment whether they are started from BASIC or from the USER command. The memory layout and usage of this environment is described in Section 6.2. The memory layout is of importance to you as the programmer only if you want to access the memory directly. Direct access means that you explicitly assign a pointer a value and then access the memory via this pointer. If you do not assign pointers you don't need to have knowledge of the environment in which the C programs run.

If a C program ends normally, the **main** block is ended and you go back to the LOADER in the USER version or back to BASIC in the B-version. The C program can be restarted only by loading again.

Run-time errors can occur in a C program. A run-time error consists of an error message and an error number. The C program reacts differently to the run-time error depending on the operating mode and the error number.

The operating modes are ERRON, ERROFF, NMION, NMIOFF which can all be set with the standard I/O commands with the same names. The table below clarifies the relationship between error number and operating mode. The reactions to the error, halt and (no) interruption are shown in the table. NMIxxx stands for NMION or NMIOFF, and ERRxxx stands for ERRON or ERROFF.

Error no.	Operating mode	Reaction
0	ERRxxx,NMIxxx	no reaction / no error
1 - 63	ERRON, NMIxxx	interruption
	ERROFF,NMIxxx	no interruption
64 - 127	ERRxxx,NMION	interruption
	ERRxxx,NMIOF	no interruption
126 - 255	ERRxxx,NMIxxx	halt

no interruption means that the C program continues although a run-time error was encountered. The error number is entered in an error register. This

register can later be read with the standard I/O command `qerror()`. This allows you to implement your own error handling routines in a C program.

If a halt occurs, the error message of the run-time error is printed on the standard output (screen) and a key is expected. As soon as a key is pressed, the C program is ended and you are returned to the LOADER or to BASIC.

interruption means that the C program is stopped and the error message and the interruption message are printed on the standard output device (screen).

```
?.....(error message)

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

The program is ended with the **x** key. You are returned to BASIC or to the LOADER. The program is continued with **c**. The error register is loaded with the error number and can be read as if *no interruption* had taken place. The C program is restarted with **r**. Note however that the values of all variables are not changed and initializations are not made. Static variables also do not have their defined initial value of zero.

The use of [STOP+RESTORE] (NMI) key combination is handled like a run-time error, the error message reads **nmi interrupt** and the error number is 127. [STOP+RESTORE] leads to an interruption of the running C proram when the operating mode **NMION** is enabled. If the mode **NMIOFF** is active, the error register is not loaded with 127, in contrast to a normal run-time error. You should never stop a C program with [STOP+RESTORE] during an I/O operation.

6.1 Run-time errors

Run-time errors and error numbers (in parentheses) of the run-time system:

?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 value range. After **c** (continue) the expression has the value *zero*.

?stack overflow (129)
The run-time stack exceeded the available C program memory.
Run-time errors and error numbers of the standard library:

?too many files (1)
No more than 10 file descriptors may be allocated at the same time. This run-time error occurs on the 11th **open**.

?illegal file descriptor (3)
The file descriptor used has never been allocated.

?device not present (5)
A device is no longer accessible on the bus (as in BASIC).

?illegal device number (7)
The device number used is not valid (as in BASIC).

?illegal dev. nr. (RS-232) (9)
The device number used is not valid (as in BASIC).

?i/o#2 (2)
Corresponds to the BASIC error **file open**. Normally cannot occur in C.

?i/o#4 (4)
Corresponds to the BASIC error **file not found** and cannot normally occur in C.

?i/o#6 (6)
Corresponds to the BASIC error **not input file** and cannot normally occur in C.

?i/o#8 (8)
Corresponds to the BASIC error **missing filename** and cannot normally occur in C.

?run eoi (10)
The input through a **scanf** function is ended through an EOI signal, but more data is expected.

?illegal format (11)
The data being read with a **scanf** function does not agree with the expected format.

6.2 Memory layout

The numbers with the $ characters are hexadecimal numbers. In the parentheses are the corresponding numbers in decimal.

$0000-$00ff (0-255)
Zero page; used by the run-time system and the operating system. These registers may not be changed.

$0200-$0258 (512-600)
Buffer for C program.

$0400-$07ff (1024-2047)
Reserved for C system.

$0800-$xx00 (2048-x)
This is the C work storage. **$xx** represents the variable top of memory which is set in the C-LINKER. The work storage for a C program includes the machine code, the memory for the variables, and the run-time stack. **$xx** can vary from $20 to $d0 (decimal: 8192<= x <=53248 and x is variable in steps of 256).

$xx00-$cfff (x-53247)
This is the user area. You can modify it using direct access via pointers. You could set a graphics storage area here, for instance. The control of this memory is your responsibility alone. If **$xx** has the value $d0, no user memory is available.

$d000-$d7ff (53248-55297)
I/O range

$d800-$dfff (55298-57343)
Color RAM

$d000-$dfff (53248-58367)
The character set for C, namely the C set at $d000 and the BASIC lower-case set at $d800, is located here in RAM under $dxxx. You cannot access this memory in the usual ways. It is covered by the I/O pages and the color RAM. It is possible to access it with the standard I/O function **move**, (seetion 8).

$e000-$e3ff (57344-58367)
Video RAM

$e400-$ffff (58368-65535)
Operating system

The exact register layout and terms, such as operating system, color RAM, video RAM, etc., cannot be fully explained here. Explanations of the terms can be found in books about the C-64 such as The Anatomy of the Commodore 64, from Abacus Software.

7.0 The standard library

The standard library contains pre-programmed functions which can perform certain tasks for you. All of the input and output functions belong to this library.

Two standard libraries belong to this program package. The first, *stdio1.l* contains only simple input/output functions and string operations. It requires little memory. The second library, *stdio2.l* contains all the functions in the first plus two more for formatted input and output.

Each library consists of two files. The first is the compiled code of the library which is bound with the LINKER *(stdio1.l and stdio2.l)*. The second file is chained to the source file with `#include "(stdio.c)"`.

All external declarations for the standard functions are made in this file (***stdio.c***). In addition, this file contains the definitions of important constants. These files must be copied from the master disk to your work disk which contains the link files or source files which are to be linked or compiled with the standard library. Use the C-COPY program to copy these files. After you have copied the file *stdio.c* from the master disk to a work disk, you are naturally free to change it. You can write all of the constants, currently in upper case, in lower case, or insert new constants.

7.1 "stdio.c"

You will find a listing of this file in the appendix. *stdio.c* is the standard file inserted into the source file and it contains the declarations for both link files *stdio1.l* and *stdio2.l*. In addition to the declarations of the functions contained in these link files, *stdio.c* also contains the following contants defined with `#define`.

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 above constants are all of type `int` and can be printed with **putc()**, for example.

file	is the type of the file descriptor (see 7.2)
screen	as address of the screen
color	as address of the color RAM
charram1	as address of character set 1 (C character set)
charram2	as address of character set 2 (CBM set)
EOI	as End Of Information character
MAXINT	as the largest number storable in int
MAXLONG	as the largest number storable in long

The function **inkey(fd)** is also defined in *stdio.c*. This function returns a character read from the file with the file descriptor **fd**. Characters are read until a non-zero character is encountered.

7.2 "stdio1.l"

This is the simple library. It is bound to C programs with the LINKER.

The functions are annotated as they are defined in C. But since they were actually created with an assembler which creates linkable code, the function block is not formulated in C, but does contain a comment which clarifies the function.

```
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 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 funtion 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 LOADER or to BASIC.
      */ }

/*    The following input/output functions make errors
      known in the ERROFF mode with certain results. In
      the ERRON mode an interruption is created as a
      result of the error. If the exeution is continued with
      c, the function has the same result as in the ERROFF mode.
*/

file open(prim, sec, name)
   int prim, sec;
   char *name;
      { /*  prim is the primary address (device number), sec is
            the secondary address (an operating mode of the
            corresponding device). name is a pointer to the
            filename. Only the first 256 characters of the
            filename are valid. If you do not want to send a
            filename, give an empty string as the argument.
            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.c.
            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 close(fd)
  file fd;
      { /*  fd is the file descriptor of the file to be closed.
            close() returns the file descriptor fd as the result. If
            an error occurs, the value zero is returned in the
            operating mode ERROFF.
      */ }
```

```
int putc(c, fd)
  char c;
  file fd;
      { /*  putc() outputs the character variable c to the file fd.
            The result will be 1 unless an error occurs in the
            ERROFF mode, in which case the result will be 0.
      */ }

char getc(fd)
  file fd;
      { /*  getc() returns a character from the file fd. In case
            of an error and ERROFF mode the getc function
            returns -1. The bus signal EOI can be read with the
            macro EOI. The macro EOI returns 0 as long as no
            EOI (End Of Information) is encountered, otherwise
            64. The EOI macro is used like the variable st in
            BASIC.
      /* }

int puts(line, fd)
    char *line;
    file fd;
      { /*  The string to which line points is printed with the
            the terminating 0 in the file fd. puts() returns the
            number of characters printed as the result.
      */ }

char *gets(line, max, fd)
    char *line;
    int max;
    file fd;
      { /*  Reads all of the characters to 0 or $0d (=carriage
            return CR) from the file fd. At most max characters
            will be read. A 0 character will then be placed at the
            (max+1)st character in line. The CR character is
            saved, in contrast to standard C, and a 0 is
            appended.
            The result of the gets function is the number of
            characters actually read.
      */ }
```

```
intfputf(obj, n, fd)
    char *obj;
    int n;
    file fd;
       { /*   Writes the first n characters of the object to which
              obj points in the file with the file descriptor fd.
              putf() always returns the number of the characters
              actually written as an integer value.
       */ }

intfgetf(obj, n, fd)
    char *obj;
    int n;
    file fd;
       { /*   getf() reads the next n character from the file fd
              and writes them at obj. getf() always returns the
              number of characters actually read as an int value.
              This value is smaller than n if the reading process
              was interrupted by an error or an EOI.
       */ }
```

For **putc(), getc(), puts(), gets(), putf(),** and **getf(), fd** can equal **STDIO**=0. Here the input is taken from the keyboard or output is directed to the screen. The UNIX functions **putchar()** and **getchar()** can be defined as follows:

```
#define  putchar(xx)  putc(xx,STDIO)
#define  getchar()    getc(STDIO)
```

These two macro definitions are contained in ***stdio.c***. This concludes the I/O functions. The following functions are contained in stdio1.l:

```
void move(pos, n, field, mem)
    char *pos;
    int n;
    char *field;
    int mem;
       { /*   The first n characters at the memory location field
              are copied to pos. pos can also lie within the
              memory range to be copied. mem gives the memory
              configuration of the CBM 64. The processor register
              (address 1) is loaded the contents of mem during the
              copying. After move() the processor register
              contains its original value again. The meaning of the
              individual bits of this register can be found in the
              appropriate manuals. mem=52, for example, means that the
```

memory configuration 64K RAM is enabled during the copy. This way you can also change the character sets in RAM at $dxxx (see example program char-set.c in the appendix). mem=53 is the memory configuration during a C program (see also 2.6 C programs). Note: if the move routine is located in a section of memory which can be covered with BASIC ($8000-$9fff or $a000-$bfff), it can occur that it will be overlaid with an appropriate choice of **mem**. The causes the computer to crash.

```
        */ }

    void cursor(line, pos)
        int line, pos;
            { /*    Sets the cursor to the given position. line is the line
                    (0-24), pos is the column (0-39). The standard
                    input/output continues printing at this position or
                    starts an input position.
            */ }

    int strlen(line)
        char *line;
            { /*    Returns the length of the string to which line points.
                    The length is the number of characters up to the end
                    of the character 0.
            */ }

char *
    void strcpy(s, t)
        char *s, *t;
            { /*    The string to which s points is transferred to t
                    (string assignment).
            */ }

    int strcmp(s, t)
        char *s, *t;
            { /*    Lexical comparison of the two strings. strcmp() returns:
                            -1 for s<t, 0 for s=t or +1 for s>t
            */ }

char *
    void strcat(s, t)
        char *s, *t;
            { /*    The string t is appended to the string s.
            */ }
```

also strncmp(

```
char *alloc(size)
    int size;
       { /*   alloc() allocates memory space for objects. These
              objects possess no names an can be only accessed
              via pointer values. They can serve for list
              management or for temporary storage, for instance.
              The length of the object required is passed as the
              argument. alloc() returns the pointer value to this
              object. This pointer value is defined as a pointer to
              char. if other types are required, the address must be
              converted to another type with CAST.
              The argument size may only be a positive value
              from 0 to 32767 or an overflow error will result. If
              larger objects are required, two alloc() calls are
              required whereby the second call returns the basis
              address of the whole object.
              If there is not enough space for an object, a stack
              overflow error is generated. Note that the alloc
              function reduces the space available for the C stack
              This stack contains local variables and the data for
              function calls.
       */ }
void free(size)
    int size;
       { /*   free() represents the opposite function of alloc().
              It frees objects defined with alloc(). The objects
              must be freed in the opposite order in which they
              were generated. The argument size may not be
              larger than 32767 or the run-time error overflow
              error will be created.
       */ }
```

7.3 "stdio2.l"

The functions in the link file *stdio1.l* are also contained in *stdio2.l*. In addition, *stdio2.l* contains the following functions:

7.3.1 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 nonsence.

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 ths 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 the 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   :
```

7.3.2 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 charater 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 supressed 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, 7.3.2.1.

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 EOI signal, the input is repeated until all arguments are served. The **sscanf** funtion creates an EOI signal if it encounters the end of the input string.

The **scanf** functions return the number of correctly-read data as the result.

7.3.2.1 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 EOI 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 EOI signal, the input is stopped only by the boundary character.

7.3.2.2 Error messages

If the input is ended by an EOI signal although the ***scanf*** function expects more data, the error ?RUN EOI (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.

7.3.2.3 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;
```

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 tarsnported 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 are skipped during the compilation. They serves 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 prefined 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, as in most compilers.

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
\\	\	
\'	'	
\"	"	
\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 definiton.

8.3.1 Storage classses

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 separatelty 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.

One 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 operands, 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. The rules for Super C are found in Section 6.

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 funtion 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 mentione as a warning before use. **A<B<C** does not yield the expected result. Th comparison **A<B** returns the result 0 for false, 1 for true. Than a comparison 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 paretheses. Thus the folllowing 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 that with which it 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. Paretheses 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 delare 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 applys 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 leat 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. If the second expression is ommitted, 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** instructionas 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 distinctior 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 outisde 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 inm 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 exsists 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 =op instead of op=. 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.

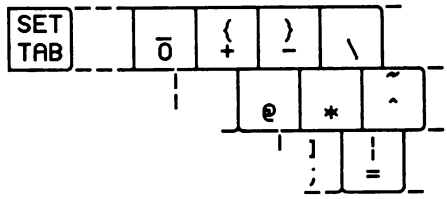
Super C also offers possibilities which other systems do not offer. Do not use these in programs which are to transported to other machines.

Addresses can be given as `case` constants. The specification of a boundary character is possible when reading strings with `scanf`, `sscanf`, and `fscanf`.

PART III. Appendix

1. Keyboard layout

Keys	Name	Number	Keys	Name	Number
CTRL+1	black	0	CBM+1	orange	8
CTRL+2	white	1	CBM+2	brown	9
CTRL+3	red	2	CBM+3	light red	10
CTRL+4	cyan	3	CBM+4	dark grey	11
CTRL+5	purple	4	CBM+5	grey	12
CTRL+6	green	5	CBM+6	light green	13
CTRL+7	blue	6	CBM+7	light blue	14
CTRL+8	yellow	7	CBM+8	light grey	15



2. Keyboard chart

dec	hex	oct	0	16	32	48	64	80	96	112	128	144	160	176	192	208	224	240	dec	hex	oct
0	0	0	nul		spc	0	@	P	spc	P		black	spc	┌	spc	P	spc	┌	0	0	0
1	1	1		crsr→	!	1	a	q	A	Q	orange	crsr↑	▌	┴	A	Q	▌	┴	1	1	1
2	2	2		rvs on	"	2	b	r	B	R		rvsoff	▐	┬	B	R	▐	┬	2	2	2
3	3	3		home	#	3	c	s	C	S		clr	—	┤	C	S	—	┤	3	3	3
4	4	4		delete	$	4	d	t	D	T		insert	_	├	D	T	_	├	4	4	4
5	5	5	white		%	5	e	u	E	U	F1	brown	_	▄	E	U	_	▄	5	5	5
6	6	6			&	6	f	v	F	V	F3	pink	✻	▌	F	V	✻	▌	6	6	6
7	7	7			'	7	g	w	G	W	F5	d grey	▌	─	G	W	▌	─	7	7	7
8	8	10	TAB		(	8	h	x	H	X	F7	grey	▐	▐	H	X	▐	▐	8	8	10
9	9	11	SET		)	9	i	y	I	Y	F2	l grn	▚	┼	I	Y	▚	─	9	9	11
10	A	12			*	:	j	z	J	Z	F4	l blue	▌	╱	J	Z	▌	┘	10	A	12
11	B	13			+	;	k	[	K	(	F6	l grey	┼	▪	K	(	┼	▪	11	B	13
12	C	14		red	,	<	l	\	L	¦	F8	purple	▪	▄	L	¦	▪	▄	12	C	14
13	D	15	return	crsr→	-	=	m	]	M	)	sh ret	crsr↓	┘	┐	M	)	┘	┐	13	D	15
14	E	16	CBMset	green	.	>	n	^	N	~	C set	yellow	┐	▪	N	~	┐	▪	14	E	16
15	F	17		blue	/	?	o	_	O	▒		cyan	—	▖	O	▒	—	~	15	F	17

TAB : Tabulator C-set \ ` { | } ~
SET : Tabulator set CBM-set £ ↑ ← + ▒ ▌ ✻
CBMset : Standard Character set
C set : C Character set

3. listing "stdio.c"

```
PAGE:   1       stdio.c
DATE: 6-11-85

    1
    2 typedef int file;
    3
    4 extern void erron(),erroff(),nmion(),nmi
off;
    5 extern void error(),exit(),move(),cursor
();
    6 extern void strcpy(),strcat(),free();
    7 extern int  qerror(),putc(),puts(),gets(
),putf();
    8 extern int  getf(),strlen(),strcmp();
    9 extern char getc(),*alloc();
   10 extern file open(),close();
   11
   12 extern void printf(),sprintf(),fprintf()
;
   13 extern int  scanf(),sscanf(),fscanf();
   14
   15 #define STDIO      0
   16 #define NULL       '\0'
   17 #define CR         '\n'
   18 #define CRSUP      '\221'
   19 #define CRSDOWN    '\21'
   20 #define CRSRIGHT   '\35'
   21 #define CRSLEFT    '\235'
   22 #define HOME       '\23'
   23 #define CLR        '\223'
   24 #define REVERSON   '\22'
   25 #define REVERSOFF  '\222'
   26 #define NIL        0
   27 #define EMPTY      ""
   28 #define MAXINT     32767
   29 #define MAXLONG    2147483647
   30
   31 #define EOI        (*(char *)0x90 & 0x40)
```

```
PAGE:   2       stdio.c
DATE: 6-11-85

  32
  33 #define putchar(X0) putc(X0,STDIO)
  34 #define getchar()   inkey(STDIO)
  35 #define CMOVE(X1,X2,X3) move(X1,X2,X3,0x
35)
  36
  37
  38
  39 char (*screen)[40] = 0xe000;
  40 char (*color )[40] = 0xd800;
  41 char (*charram1)[8]= 0xd000;
  42 char (*charram2)[8]= 0xd800;
  43
  44 char inkey(fd)
  45 file fd;
  46 {  char c;
  47    while((c=getc(fd))==0);
  48    return c;
  49 }
  50
  51
```

4. Listing "sample.c"

```
PAGE:   1        sample.c
DATE: 6-11-85

    1 #include "stdio.c"
    2 #define  CASE(Z) case '\Z': printf("'\\Z
     ");break
    3
    4 main()
    5 {  char c;
    6
    7    putc(CLR,STDIO);
    8    puts("Display the values for Key pres
sed\n",STDIO);
    9
   10    while()
   11    {
   12        c=getchar();
   13        printf("Character: ");
   14
   15        if((c & 0x7f) >= 0x20)
   16            if(c=='\\' || c=='\'' || c=='
\"')
   17                printf("'\\%c'    ",c);
   18            else
   19                printf("'%c'    ",c);
   20        else
   21            switch(c)
   22            {
   23                CASE(n);
   24                CASE(t);
   25                CASE(f);
   26                CASE(r);
   27                CASE(b);
   28                default: printf("'\\%o' "
,c);break;
   29            }
   30
   31
```

```
PAGE:   2        sample.c
DATE: 6-11-85

  32          printf("\nASC-Code: %3d    0X%02x
  0%-3o\n\n",c,c,c);
  33      }
  34 }
  35
  36
```

5. Listing "text.c"

```
PAGE:   1       text.c
DATE: 6-11-85

    1 We can display 16 colors
    2 on the screen.  The source text does
    3 not produce them.  The colors
    4 are used to high-light the most importan
t
    5 lines of the source text.
    6
    7
    8
    9 Color list:
   10 CBM+1  black
   11 CBM+2  white          ^^black^^
   12 CBM+3  red
   13 CBM+4  cyan
   14 CBM+5  violet
   15 CBM+6  green
   16 CBM+7  blue
   17 CBM+8  yellow
   18 CTRL+1 orange
   19 CTRL+2 brown
   20 CTRL+3 light red
   21 CTRL+4 dark gray
   22 CTRL+5 gray
   23 CTRL+6 light green
   24 CTRL+7 light blue
   25 CTRL+8 light gray
   26
   27
   28
   29 The Editor can display two character set
s:
   30
   31 the BASIC-character set and a
```

```
PAGE:   2        text.c
DATE: 6-11-85

  32 special  C-character set
  33
  34  !"#$%&'()*+,-./0123456789:;<=>?
  35 @abcdefghijklmnopqrstuvwxyz[\]^_
  36 `ABCDEFGHIJKLMNOPQRSTUVWXYZ{|}~)
  37 ---and  2 x 32 Graphic characters----
  38
  39 With [SHIFT]+[CBM] it is possible to swi
tch
  40 between the two character sets.
  41
  42 The character set includes the following
  43 special C-characters: \ ^ _ { | } ~
  44
  45 When in the CBM-character mode, it will
  46 be difficult to find the special C
  47 characters, please use the C- character
mode.
  48
  49 The Characters: _ is done with [SHIFT]+[
0] (zero)
  50 and the Left-arrow key is the
  51 Editor TAB key. TAB SET and release is
  52 done by using [SHIFT]+[Left-arrow].
  53 To obtain the character | use the
  54 [SHIFT]+[=].
  55
  56
  57 These lines stand at the start of our go
al.
  58 Our goal is to mark a block once.
  59 Set   the  goal line  of the  previous l
ine.
  60
  61
  62
```

```
PAGE:   3       text.c
DATE: 6-11-85

  63 This line is the end of the Block ******
**********************************
  64 This is the beginning of the text block
that we will erase
  65
  66
  67                          GREEN
  68
  69 Block to be erased        only
  70
  71                          has
  72
  73                           one
  74
  75                          Number
  76           1
  77            2
  78             3
  79              4
  80               5
  81                6
  82                 7
  83                  8
  84                   9
  85 Block to be         10
  86     erased           11
  87                       12
  88                        13
  89                         14
  90                          15
  91                           16
  92                            17
  93                             18
```

```
PAGE:   4       text.c
DATE: 6-11-85

  94 These are the last lines of the block
  95 that we will erase with the erase comman
d  >>> line 95 <<<
  96 This Text is after the block we  *******
**************************************
  97 wish to erase.
  98
  99
 100 This line is before the blue  text block
 101 Here begins the  Block for moving.
 102
 103        ┌┐ ┐ ┌┐┌┐┌┐
 104        ├┘ │ ││└ ├┘
 105        ┴└┘┴ └┘└┘┴ └
 106
 107
 108 This is the last line of the blue block.
 109 This line is not in the blue text block
 110
 111 This is the next line to the 'last line'
 112
 113
 114
 115
 116
```

6. Listing "char-set.c"

The source file of this example program **char-set.c**, as well as the compiled C program **char-set**, is found on your master disk.

The function **BASICset()** transfers a character set from the character ROM to the underlying RAM. The argument **loc** designates the character set. You can pass the constants **charram1** and **charram2**, defined in *stdio.c*, to **BASICset()** as the address of the character sets.

First temporary space is prepared for the character set with the function **alloc()**. **move()** moves the character set from the ROM to the temporary storage. The character ROM is brought in at $dxxx with the memory configuration **0x31** and can then be read.

The second call to **move()** moves the character set from the temporary storage to the RAM under $dxxx. The value **0x34** switched the memory configuration to 64K RAM. The RAM under $dxxx can then be written.

The temporary space reserved for the character set is freed again with **free()**.

readset() reads a character set from the disk. The filename is passed in the variable **name**. **loc** again denotes the address of the character set as with **BASICset()**.

The character set is expected in program file format. This means that first two bytes of the file will be skipped because they represent the program start address. The following 0x800 bytes are read into a temporary storage area. From there the character set is stored in the character generator.

The primary function **main()** serves only to test the other two functions. Input from the screen is called with **gets()**. In this input you can enter characters and examine them. As soon as you enter read[RETURN] at the start of a line, **main()** prints **name:** and requests the filename of the character set to be read. The **quadrato** on the master disk is such a character set file.

```
PAGE:   1        char-set.c
DATE: 6-12-85

    1 #include "stdio.c"
    2
    3 char buffer[41];
    4
    5 main()
    6 {
    7     putc(CLR,STDIO);
    8     BASICset(charram1);
    9     while()
   10     {
   11         do{
   12             gets(buffer,40,STDIO);
   13             putc(CR,STDIO);
   14
   15           }while(strcmp(buffer,"read\n")
!= 0);
   16
   17         puts("\nname:",STDIO);
   18         gets(buffer,40,STDIO);
   19         putc(CR,STDIO);
   20         readset(buffer,charram1);
   21     }
   22 }
   23
   24 void readset(name,loc)
   25     char *name;
   26     unsigned int loc;
   27
   28 {  file setf;
   29     char *help;
   30
   31     if(setf=open(8,2,name) != 0)
```

```
PAGE:    2        char-set.c
DATE: 6-12-85

  32      {    help=alloc(0x800);
  33           getc(setf);
  34           getc(setf);
  35           getf(help,0x800,setf);
  36           move(loc,0x800,help,0x34);
  37           free(0x800);
  38      }
  39      close(setf);
  40  }
  41
  42  void BASICset (loc)
  43      unsigned int loc;
  44
  45  {
  46      char *help;
  47
  48      help=alloc(0x800);
  49      move(help,0x800,loc,0x31);
  50      move(loc,0x800,help,0x34);
  51      free(0x800);
  52  }
  53
```

7. Index

- M -

- N -

- O -

- P -

- Q -

- R -

- S -

FIRST SOFTWARE LIMITED